本文主要照搬此 **链接**,加入了部分资料令其更为自洽,此外进行了些许勘误.

摘要

Raft是一个用来管理多副本日志（replicated log）的 共识算法 。其作用与（multi-）Paxos相同、效率与Paxos想用，但结构与Paxos不同；这让Raft比Paxos更容易理解，且Raft为构建实用的系统提供了更扎实的基础。为了提高可理解性，Raft将共识的关键元素分离为： 领导选举、日志复制、和安全性 ；且其增强了连贯性（coherency）译注1， 以减少必须考虑的状态数 。用户学习结果表明，对于学生来说，Raft比Paxos更容易学习。Raft还包括一个用于变更集群成员的新机制，其使用重叠的大多数来保证安全性。

> 译注1：本文的连贯性指 *coherency* ，在很多翻译中将其翻译成了一致性，这样容易与 *consistency* 混淆，二者间存在一定差异。

一致性算法的目标就是保证集群上所有节点的状态一致,节点要执行的指令可以分为两种,读与写。只有写指令会改变节点状态,因此为了保证集群各个节点状态的一致,那就必须将写指令同步给所有节点。

理想状态下,我们期望**任意节点**发生写命令都会**立即**的在其他节点上变更状态,这其中没有任何时延,所有节点都好像是单机一样被变更状态。

网络延迟要远远慢于内存操作, 写入命令不可能被同时执行,因此如果在不同节点发生不同的写命令,那么在其他节点上这些写命令被应用的顺序很可能完全不同。

如果我们不要求所有节点的写命令立即被执行,而仅仅是保证所有的写命令在所有的节点上按同样的顺序最终被执行呢? 第一, 仅仅允许一个节点处理写命令,第二,所有的节点维护一份顺序一致的日志。

每个节点上的状态机按照自己的节奏,逐条应用日志上的写命令来变更状态。

1. 引言

共识算法让一组机器能像一个连贯组一样工作,并且可以容忍一些成员故障。因为这一点，它们在构建可靠的大规模软件系统中扮演者关键角色。Paxos[15, 16]在过去的十年中主导了共识算法的讨论：大多数共识的实现都基于Paxos或受其影响，且Paxos成为了用来教授学生有关共识知识的主要工具。

不幸的是，Paxos相当难以理解，尽管有很多使其更易接受的尝试。另外，其架构需要复杂的修改以支持实用的系统。其结果是，系统构建者和学生都很受Paxos困扰。

在我们自己饱受Paxos困扰后，我们开始寻找一个能够为系统构建和教育提供更好的基础的新的共识算法。我们的方法不太寻常，因为我们的主要目标是 可理解性 ：我们能否定义一个为实用系统设计的共识算法，并用一个比Paxos更容易学习的方式描述它？此外，我们希望算法能够让开发更加直观，这对系统构建者来说是很重要的。重要的不光是算法，还有为什么算法能工作。

这项工作的成果是一个被称为**Raft**的共识算法。在设计Raft时，我们使用了特殊的方法来提高可理解性，包括 算法分解（Raft将领导选举、日志复制、和安全性分离开来）和减少状态空间（与Paxos相比，Raft减少了不确定性，且该方法允许服务器相互不一致） 。对两个大学的43个学生的研究表明，Raft比Paxos更好理解得多：在学习这两种算法后，这些学生中的33名能够更好得回答有关Raft的问题。

Raft与现有的共识算法在很多方面都很相似（最明显的时候，Oki和Liskov的Viewstamped Replication[29, 22]），但Raft有很多新特性：

- **强leader：** 与其它共识算法相比，Raft使用了更强的领导权形式。例如，日志条目(log entry)仅从leader流向其它服务器。这简化了对多副本日志的管理，并使Raft更容易理解。
- **领导选举：** Raft使用随机计时器来选举leader。这仅在任何共识算法都需要的心跳机制上增加了很小的机制，但能够简单又快速地解决冲突。
- **成员变更：** Raft用来变更集群中服务器集合的机制使用了一个新的 *联合共识（joint consensus）* 方法，其两个不同配置中的大多数服务器会在切换间有重叠。这让集群能够在配置变更时正常地继续操作。

我们认为，无论为了教育目的还是作为实现的基础，Raft都比Paxos和其它共识算法更优秀；Raft的描述足够完整，能够满足使用系统的需求；Raft有很多开源实现并已经被一些公司使用；Raft的安全性性质已经被形式化定义并证明；Raft的效率与其它算法相似。

本文的剩余部分介绍了多副本状态机问题（**第二章**），讨论了Paxos的优势与劣势（**第三章**），描述了我们为了可理解性使用的通用方法（**第四章**），给出了Raft共识算法（**第5~8章**），评估了Raft（**第九章**），并讨论了相关工作（**第十章**）。

2. 多副本状态机

共识算法通常在 *多副本状态机问题（replicated state machine problem）* [37]的上下文中出现。通过这种方法，在一系列服务器上的状态机会计算相同状态的相同副本，且即使在一些服务器宕机是也可以继续操作。多副本状态机被用来解决分布式系统中各式各样的容错问题。例如，有单集群leader的大型系统（如GFS[8]、HDFS[38]、和RAMCloud[33]）通常使用独立的多副本状态机来管理领导选举并存储必须能在leader崩溃时幸存的配置信息。多副本状态机的例子还包括Chubby[2]和ZooKeeper[11]。

> "Replicated state machine problem"（复制状态机问题）是指在分布式系统中，如何确保多个副本（replica）的状态保持一致的问题。在分布式系统中，由于网络延迟、故障和并发操作等因素，不同的副本可能会出现状态的不一致性，即使它们最初具有相同的初始状态。

多副本状态机通常使用多副本日志实现，如**图1**所示。每个服务器存储一个包含一系列指令的日志，状态机会按照顺序执行日志。每个日志包含相同顺序的相同指令，因此每个状态机会处理相同的指令序列。因为状态机是确定的，每个状态机都会计算出相同的状态并得出相同的输出序列。

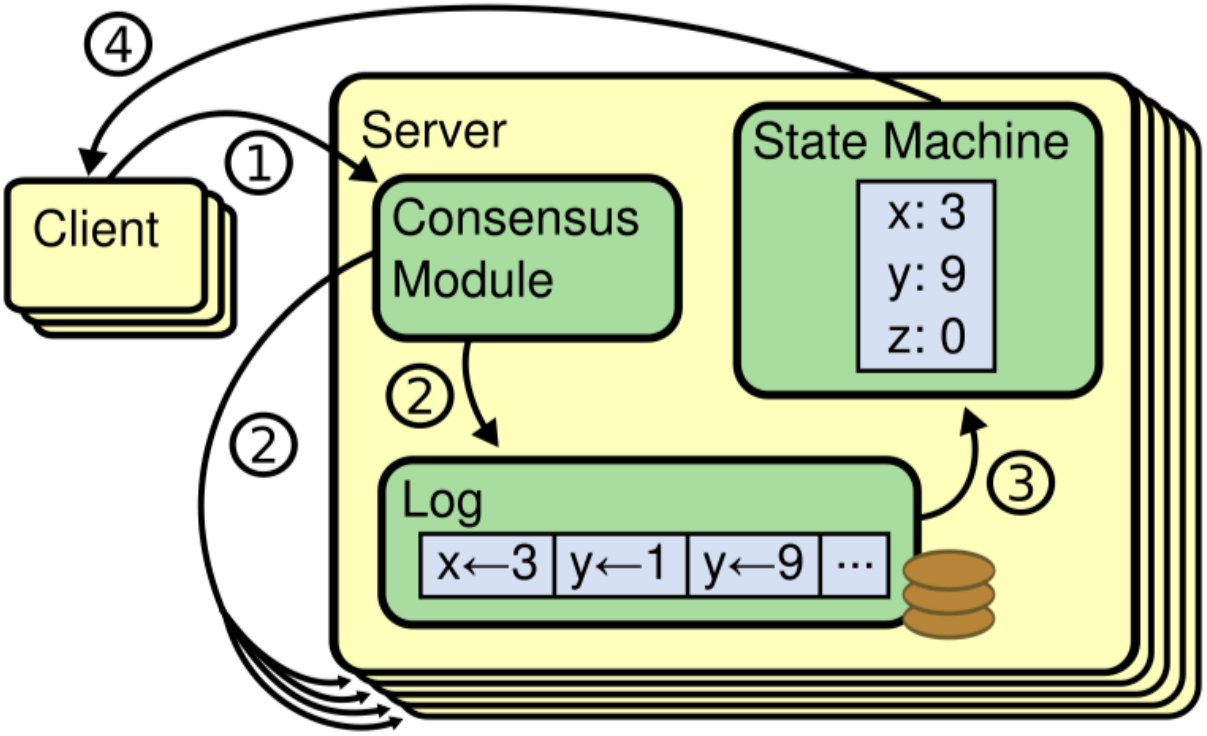

图1 多副本状态机架构。共识算法管理由来自不同客户端的状态及指令组成的多副本日志。状态机按照日志处理相同的指令序列，因此它们会产生相同的输出。

保持多副本日志的一致性是共识算法的任务。服务器上的共识模块会接收来自客户端的指令，并将其添加到它的日志中。它与其它服务器上的共识模块通信来确保每个日志最终包含相同顺序的相同请求，即使一些服务器故障也是如此。一旦指令被恰当地多副本化，每个服务器的状态机就可以按日志顺序处理它们，并将输出返回给客户端。这样，所有服务器对外会表现为单个高可靠性的状态机。

为实用系统设计的共识算法通常有如下属性：

- 它们确保所有非拜占庭条件下的安全性（永远不会返回错误结果），需要处理的问题包括网络延迟、分区、丢包、重复、和乱序。

拜占庭容错系统要解决的正是分布式**系统中存在恶意节点（即拜占庭节点）时，系统的一致性、正确性等问题。**

> Byzantine fault: Fault in a computer system that presents different symptoms to different observers
>
> - **C1.**所有忠诚的接收命令的将军遵守相同的命令
> - **C2.**如果发送命令的将军是忠诚的，那么所有忠诚的接收命令的将军遵守所接收的命令
>
> 拜占庭容错系统要解决的正是分布式**系统中存在恶意节点（即拜占庭节点）时，系统的一致性、正确性等问题（即达成上述C1和C2条件）。**
>
> **假设分布式系统拥有n台节点，并假设整个系统拜占庭节点不超过m台（$n \geq 3m + 1$）**，拜占庭容错系统需要满足如下两个条件：
>
> - **所有非拜占庭节点使用相同的输入信息，产生同样的结果。**在区块链系统中，可以理解为，随机数相同、区块算法相同、原账本相同的时候，计算结果相同。

> - **如果输入的信息正确，那么所有非拜占庭节点必须接收这个消息，并计算相应的结果。**在区块链系统中，可以理解为，非拜占庭节点需要对客户的请求进行计算并生成区块。

- 只要大多数服务器可以操作那么其所有功能都可用，且能够相互通信或与客户端通信。因此，通常使用的由5个服务器组成的集群能够容忍任意2个服务器故障。服务器被假设可能宕机停止；它们也可能在随后从稳定存储中恢复并重新加入集群。
- 它们不依赖定时来保证日志的一致性：在最坏的情况下，时钟故障和极端的消息延迟会导致可用性问题。
- 在通常情况下，一条指令能在集群的大多数响应一轮远程过程调用（RPC）后完成；少数的较慢的服务器不会影响整个系统的性能。

3. Paxos有什么问题?

在过去十年中，Leslie Lamport的Paxos协议[15]几乎和共识成了同义词：Paxos是在课程中最常被教授的协议，也是大多数共识实现的起点。Poxos首先定义了一个能够对单个决策达成一致的协议，如单个多副本日志条目。我们称这个子集为 *单决策Paxos（single-decree Paxos）* 。Paxos接着将该协议的多个实例结合，以实现一系列的决策，例如一个日志（multi-Paxos）。Paxos同时确保了安全性和活性（liveness），且它支持集群中成员的变更。它的正确性已经被证明，且Paxos在一般场景下很高效。

不幸的是，Paxos有两个显著的劣势。第一个劣势是Paxos非常难以理解。众所周知，Paxos的完整解释[15]非常隐晦；只有很少的人在付出很大努力后才能成功理解它。因此，出现了很多试图通过更简单的方式解释Paxos的尝试[16, 20, 21]。这些解释着手于单决策Paxos这一子集，尽管这仍很有挑战。在对NSDI2012出席者的非正式调查中，我们发现尽管在经验丰富的研究者中，也几乎没有人觉得Paxos容易。我们自己就受Paxos困扰，直到阅读了一些简化的解释后我们才理解了完整的协议，所以我们设计了自己的替代的协议，这一过程花了差不多一年时间。

我们假设Paxos的隐晦性来自于其选择了单决策子集作为其基础。单决策Paxos很冗杂且隐晦：单决策Paxos被分为两个阶段，其没有简单的直观解释也不能被单独理解。因此，人们很难对单决策协议为什么可行建立一个直观认识。Multi-Paxos的规则由增加了很大的额外的复杂性和隐晦性。我们认为达到多决策共识（例如，一个日志而不是单个日志条目）的整个问题可被分解为更直观更显然的其他方式。

Paxos的第二个问题是它没有为构建实用的实现提供良好的基础。其原因之一是人们对multi-Paxos算法没有广泛的一致意见。Lamport的描述几乎都关于单决策Paxos；他概括了multi-Paxos的可能的方法，但缺少许多细节。后来出现了很多试图具体化并优化Paxos的尝试，如[26, 39, 13]，但这些方法互相之间都不一样且与Lamport的蓝图也不通。像Chubby[4]这样的系统实现了类Paxos算法，但在大多数条件下的细节都没有发表。

此外，Paxos架构对构建实用系统来说很弱；这时将算法分解为单决策的另一个后果。例如，单独选取一组日志中的每个条目并将它们合并为一个顺序日志并没有什么还出；这样做只会增加复杂性。设计一个围绕系统的日志更加简单且高效，其中新日志条目可以按照受约束的顺序被依次添加。另一个问题是，Paxos的核心使用了一个对称的（symmetric）对等(peer-to-peer)方法（尽管其最后提出了一个弱领导权方法作为性能优化）。这在仅需要做一个决策的简单的世界中是有意义的，但是对于使用这种方法的实用系统来说意义不大。如果必须做出一系列决策，那么先选举出一个leader更简单却更快，随后让该leader协调决策。

因此，使用系统与Paxos相似之处很少。每个实用系统的实现首先都会从Paxos开始，然后发现很难实现它，接着开发了一个非常复杂的架构。这很消耗时间且容易出错，且Paxos难以理解加剧了这一问题。Paxos的表达形式可能对于证明其理论正确性来说很好，但是因为其真实实现与Paxos太过不同，这样理论证明就失去了价值。来自Chubby的实现者的评论尤为典型：

> 引用

现实系统的需求的与Paxos算法的描述之间有很大的隔阂。为了构建现实的系统，专家需要使用分散在各种文献中的许多思想，并作出一些较小的协议扩展。这些不断累积的扩展会非常多，最后系统会基于一个未被证明的协议。[4]

因为这些问题，我们总结出Paxos没有为系统构建和教学提供良好的基础。考虑到共识对大型软件系统的重要性，我们决定尝试我们能否构建出一个能替代Paxos的且比Paxos的属性更好的共识算法。Raft就是这一实验的成果。

4. 为可理解性做出的设计

我们在设计Raft时有许多目标：它必须为系统构建提供完整且实用的基础，这样就能大量减少开发者所需的设计工作；它必须在所有条件下都安全，在典型的操作条件下可用；它必须能在通用操作中保持高效。但我们最重要的目标是可理解性，这也是最难的挑战。它必须能被大量读者容易地理解。另外，它必须能够建立对算法的直观直觉，这样系统构建者可以对其进行扩展，这在真实实现中是不可避免的。

在我们设计Raft时，有很多要点不得不选择替代方法。在这些场景中，我们基于可理解性对这些替代方法进行了评估：解释每个替代方法有多难？（例如：其状态空间多复杂？其有没有难以捉摸的实现？）读者完整理解该方法并实现它有多简单？

我们意识到这样的分析有很大的主观性；尽管如此，我们还是使用了两种能使其更容易被大家接受的方法。第一个方法是众做周知的问题分解方法：我们尽可能地将问题划分为能被解决、解释并理解的相对独立的子问题。例如，在Raft中，我们将其分为领导选举、日志复制、安全性和成员变更。

我们的第二个方法是通过减少需要考虑的状态数来简化状态空间，使系统具有连贯性并尽可能消除不确定性。特别是，日志不允许有“洞”（译注：这里的“洞”指日志间的空隙，见Ceph的论文。），且Raft限制了日志变得与其它不一致的方式。尽管在大多数情况下，我们试图消除不确定性，但是有些情况下不确定性实际上可以提高可理解性。在实践中，随机化的方法引入了不确定性，但是它们通常会通过用相同的方法解决所有可能的选择，因此减少了状态空间。我们使用的随机化的方法简化了Raft的领导选举算法。

5. Raft共识算法

Raft是一种用来管理 **第二章** 中描述的形式的多副本日志的算法。**图2**以浓缩的形式总结了算法以供参考，**图3**列出可算法的关键性质；这些图中元素将在本章剩下的部分分条讨论。

State

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)

AppendEntries RPC

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)

RequestVote RPC

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)

Rules for Servers

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).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.

图2 对Raft共识算法的浓缩总结（不包括成员变更和日志压缩)。服务器的行为在左上角的的格子中被作为一系列独立且可重复的触发器规则描述。像“§5.2”这样的章节号表示某个特征将在哪一节中讨论。正式定义[31]将会更精确地描述算法。

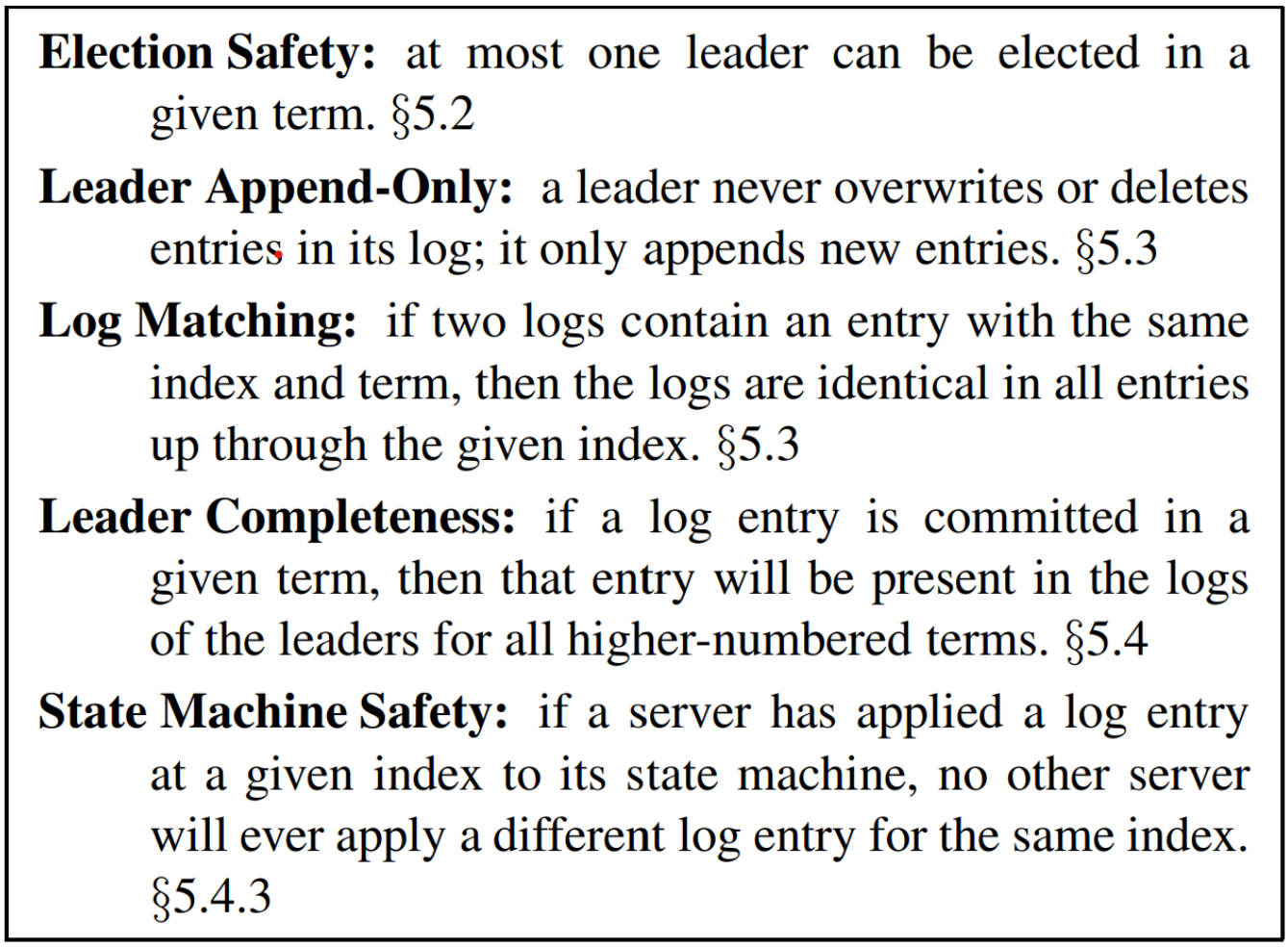

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

图3 Raft在所有时刻都保证这些性质的每一条都成立。章节号表示每条性质将在哪一节中讨论。

Raft通过先`选举一个高级leader`,然后给予该leader管理分布式日志的所有责任来确保副本的一致性。该leader接收来自客户端的日志条目，将它们复制到其它服务器上，然后告诉服务器什么时候可以安全地将这些日志条目应用到它们的状态机中。使用leader可以简化对分布式日志的管理。例如，leader可以在不询问其它服务器的情况下决定将日志的新条目放在哪儿，且数据流仅简单地从leader流向其它服务器。`leader可能发生故障也可能在其它服务器中失去对其的连接，在这种情况下，会有新的leader被选举出来。`

考虑leader的方法，Raft将共识问题分解成三个相对独立的子问题，接下来的小节会对这些问题进行讨论：

- **领导选举：** 当已有的leader故障时必须有新的leader被选举出来（章节5.2）。
- **日志复制：** leader必须接收来自客户端的日志条目，并将它们复制到集群中，强制其它日志对它自己的日志达成一致（章节5.3）。

- **安全性：** Raft中关键的安全性是**图3**中的状态机安全性（State Machine Safety Property）：如果任意服务器将一个特定的日志条目应用到了其状态机中，那么不会有应用了有相同index（索引）的其他指令的日志条目的服务器。章节5.4描述了Raft如何确保这一性质；其解决方案包括对章节5.2中描述的选举机制的一个额外的约束。

在给出共识算法后，本章讨论了可用性问题和定时在本系统中的角色。

5.1 Raft基础

一个Raft集群包括多个服务器；通常数量是5，这可以让系统能够容忍2个服务器故障。在给定时间内，每个服务器处于以下三个状态之一：leader、follower、或candidate。在正常的操作中，会有恰好一个leader，所有其它服务器都是follower。follower是被动的：它们不会自己提出请求，而仅响应来自leader和candidate的请求。leader处理所有的客户端请求（如果客户端联系了一个follower，该follower会将其重定向到leader）。第三个状态candidate，在章节5.2中描述的选举新leader时使用。**图4**展示了状态和状态间的转移；状态转移将在后文中讨论。

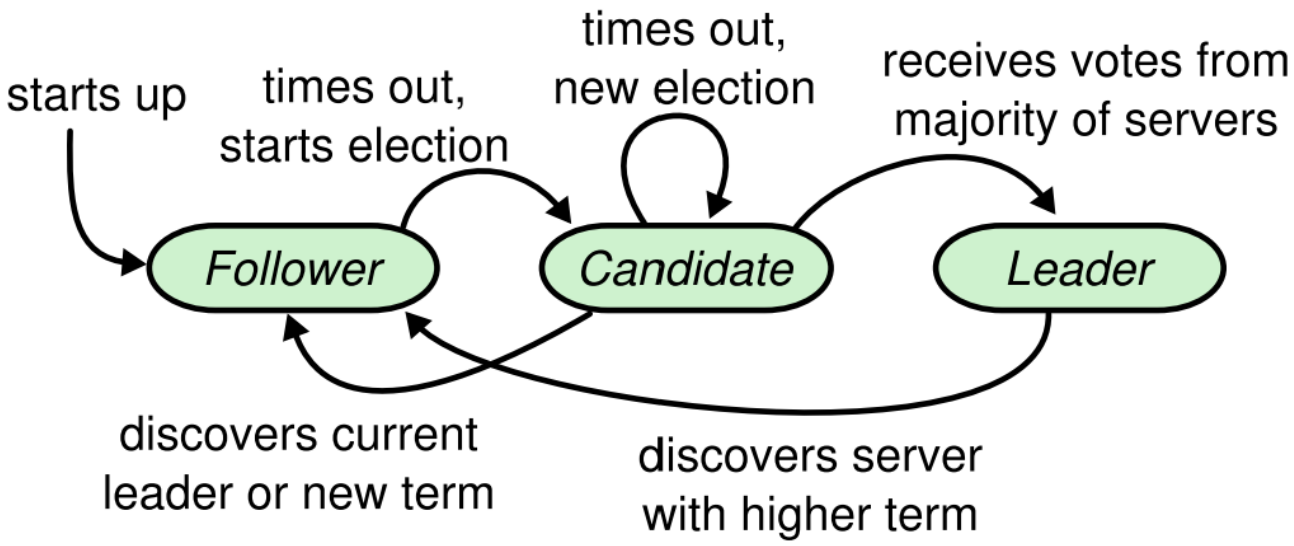

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

图4 服务器状态。follower仅响应来自其它服务器的请求。如果follower没有收到通信，那么它会变为candidate并开始一次选举。收到了来自整个集群中大多数节点投票的candidate会成为新的leader。leader通常会持续到其故障。

Raft将时间划分为任意长度的term（任期），如**图5**所示。term被编号为连续的整数。每个term从选举（election）开始，在选举中一个或多个candidate会试图成为leader，就像章节5.2中描述的那样。如果candidate赢得选举，那么它将在该term余下的时间了作为leader提供服务。在某些情况下，一次选举可能导致投票决裂，此时该term最终可能没有leader，那么很快会开始一个新的term（伴随一次新的选举）。Raft确保一个给

定的term中最多只会有一个leader。

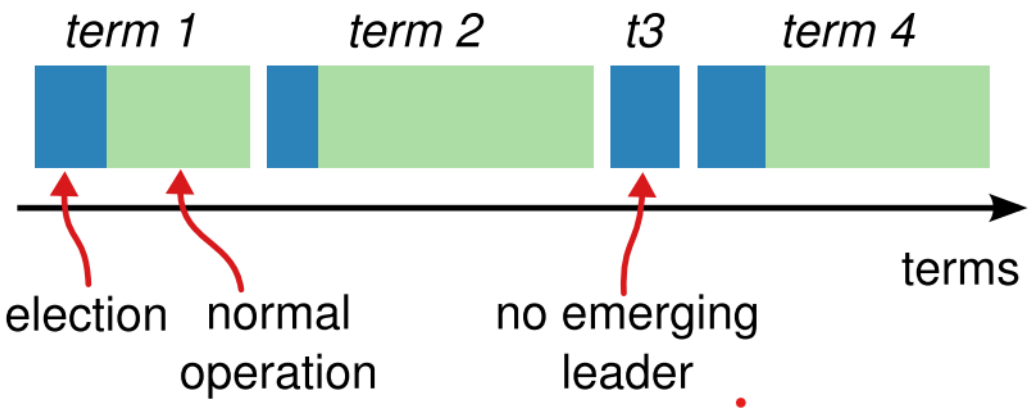

Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

图5 时间被划分为term，每个term从选举开始。在一次成功选举后，单个leader会管理集群，直到该term结束。有些选举会失败，在这种情况下，term会不选择leader就结束。term之间的转换可以在不同服务器上的不同时间被观测到。

不同的服务器可能在不同时间观测到term的转换，且在一些情况下，服务器可能没有观测到选举甚至没观测到整个`term`。term在Raft扮演`逻辑时钟`[14]的角色，且term能让服务器检测到过时的（obsolete）信息，如陈旧的（stale）leader。每个服务器会存储当前的term号，其随时间单调递增。当前的term号在任何服务器通信时都会被交换，`若该服务器当前的term小于其它服务器的，那么它会更新其term号到到较大值`。如果candidate或leader发现它的term过期了，它会立刻转到follower状态。如果服务器收到了有陈旧的term号的请求，它会拒绝该请求。

Raft服务器使用远程过程调用（remote procedure call，RPC）通信，且基本的共识算法仅需要两种RPC。`RequestVote` RPC在选举时由candidate发起（**章节5.2**），而`AppendEntries` RPC被leader发起，用来复制日志条目和提供心跳（**章节5.3**）。**第七章**加入了第三个RPC，用来在`服务器间传输快照`。如果服务器没有及时收到响应，它们会重试RPC，且它们会并行地发起RPC以获得最佳性能。

5.2 领导选举

Raft使用心跳机制来触发领导选举。当服务器启动时，它们按照如下方式开始。只要服务器能收到来自leader或candidate的合法的RPC，它就会保持follower状态。leader会定期发送心跳（不携带任何日志条目的AppendEntries RPC）给所有follower，以维护其权威。如果follower在超过一段时间后（这段时间被称为选举超时时间，election timeout）仍没受到通信，那么它会假设当前没有可行的leader，并开始一次选举以选择一个新leader。

为了开始一次选举，follower会 增大其当前的term ，并 转换到candidate状态 。其随后为自己投票，并并行地给集群中每个其它的服务器发起RequestVote RPC。candidate会保持其状态，知道以下三件事情之一发生：（a）它赢得了选举；（b）另一个服务器成为了leader；（c）一段时就过后仍没有胜者。这些后果将在后文中分别讨论。

如果candidate收到了整个集群中大多数相同term服务器的投票，那么它会赢得选举。在一个给定的term中，每个服务器会按 先到先得 （first-come-first-served）的方式 给最多一个candidate投票 （注意： 章节5.4 对投票增加了一个额外约束）。“大多数”规则确保了在特定的term中最多只有一个candidate能赢得选举（选举安全性如**图3**所示）。一旦candidate赢得选举，它会变成leader。随后它会向所有其他服务器发送心跳消息以建立起权威，并防止新选举发生。

在 等待投票时 ，candidate可能收到来自其它服务器声明自己是leader的AppendEntries RPC。如果leader的term（包括在其RPC中） 至少与该candidate当前的term一样大 ，那么这个candidate会视其为合法的leader，并返回到follower状态。 如果RPC中的term比该candidate当前的term小 ，那么该candidate会拒绝RPC并继续以candidate状态运行。

第三种可能的结果是，candidate既没有赢得选举也没有输： 如果许多follower在同时变成了candidate，投票可能决裂 ，这样可能没有candidate获取大多数的投票。当这种情况发生时， 每个candidate都将会超时并通过增大其term和开始另一轮RequestVote RPC来开始新的一轮选举 。然而，如果不采取额外措施， 投票决裂可能会无限反复。

Raft使用了 随机额选举超时时间 以确保投票决裂很少发生，且投票决裂可以被快速解决。为了在初次选举时防止投票决裂， 选举超时时间会从一个固定的时间段（例如，150~300ms）中随机选取 。这在所有服务器上实现，这样大多数情况下只有单个服务器会超时，它会在任何其它服务器超时前赢得选举并发送心跳。同样的机制还在 处理投票决裂 时使用（译注：之前介绍的是**防止**投票决裂，接下来是**处理**投票决裂）。每个candidate在开始选举时重置其随机选举超时时间的计时器，且它会在下一次选举开始前等待该超时时间流逝，这减少了新的选举中再次发生投票决裂的可能性。 章节9.3 说明这种方法能快速地选出leader。

选举是可理解性如何指导我们在备选设计中做出选择的一个实例。最初我们计划使用一个排名（ranking）系统：假设每个candidate有一个唯一的排名（rank），这个排名会在选取竞争中的candidate时使用。如果candidate发现了有更高排名的另一个candidate，它会返回follower状态，这样有更高排名的candidate能够更容易地赢得下一次选举。我们发现这种方法制造了有关可用性的隐晦的问题（如果排名较高的服务器故障，排名较低的服务器可能需要等待超时才能再次成为candidate，但是如果它再次成为candidate过早，它可能会重置领导选举的进度）。我们对该算法做了很多次调整，但是在每次调整后都会出现新的小问题。最终我们总结出，随机重试的方法更显而易见且易于理解。

5.3 日志复制

一旦leader被选举出来，它会开始为客户端的请求提供服务。每个客户端请求包含一条需要被多副本状态机执行的指令。leader将该指令作为新的条目追加到其日志中，随后它会并行地向每台其他的服务器发起AppendEntries RPC复制该条目。 当该条目被安全地复制时（正如后文描述的那样），leader会将该条目应用到其状态机中 ，并将执行的结果返回给客户端。如果follower崩溃或运行缓慢，或者如果网络包丢失，leader会无限重试AppendEntries RPC（即使它已经响应了客户端），直到所有的follower最终存储了所有的日志条目。

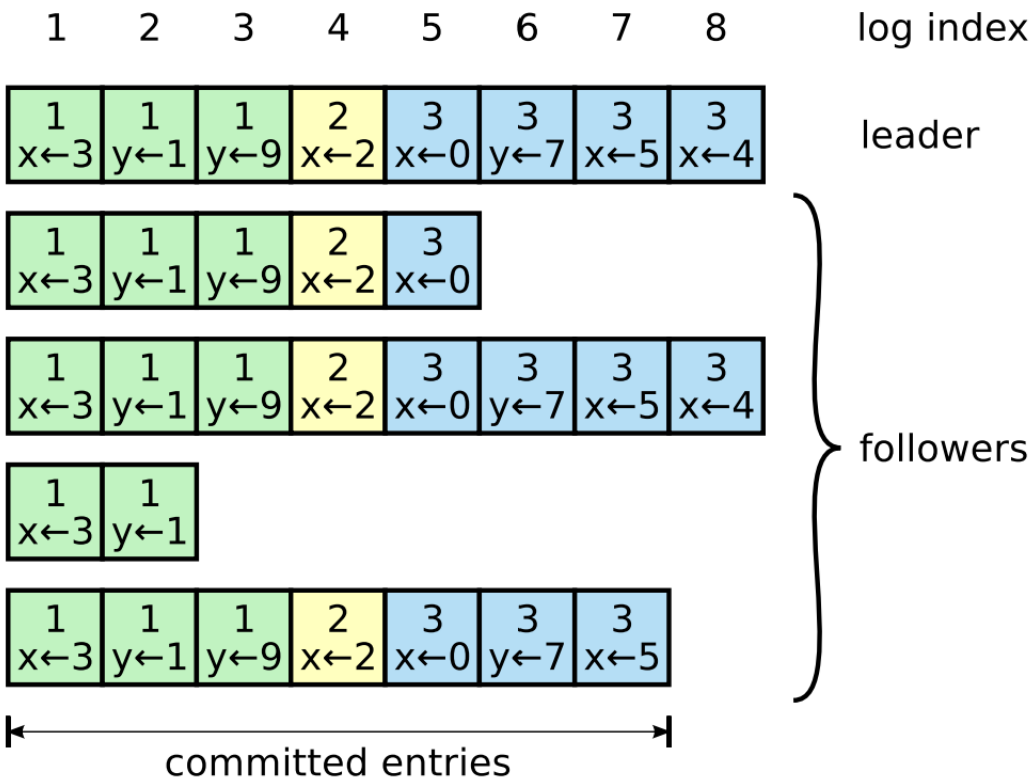

Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

图6 日志由条目组成，日志条目被顺序编号。每个日志条目都包含它被创建时的term号（每个方框中的数字）和一条给状态机的指令。如果一个日志条目能被安全地应用到状态机中，它会被视为committed。

日志被按照如**图6**的方式组织。每条日志都保存了一条状态机指令和leader收到该条目时的term号。日志条目中的term号被用作检测日志间的不一致性并确保**图3**中的一些性质。每个日志条目还有一个标识它在日志中的位置的整数index（索引）。

leader会决定什么时候能够安全地将日志条目应用到状态机，这种条目被称为committed（已提交）的。Raft保证committed的条目是持久性的，且最终将会被所有可用的状态机执行。leader ***一旦发现大部分副本*** 已经记录了enrty, 那么该条目会变成committed的（例如，**图6**中的条目7）。这还会提交在leader的日志中所有之前的条目。**章节5.4** 讨论了在leader变更后应用这一规则的一些微妙的情况，其还展示了“提交是安全的”的定义。leader会跟踪它知道的被提交的最高的index，且它会在之后的AppendEntries RPC中包含这个index，这样其它server最终会发现它。一旦follower得知一个日志条目被提交，它会将该条目（按日志顺序）应用到它本地的状态机中。

我们设计的Raft日志机制能维护不同服务器间高级别的连贯性。这不但简化了系统的行为，还使系统更可预测，这时确保安全性的重要部分。Raft维护的如下的性质共同构成了**图3**中的“日志匹配性质（Log Matching Property）”：

- 如果不同日志的两个条目有相同的index和term，那么它们存储了相同的指令。
- 如果不同日志的两个条目有相同的index和term，那么日志中之前的所有条目都是相同的。

第一条性质基于“leader在给定term期间只会创建一条有给定index的日志条目，且日志条目在日志中的位置永远不会改变”的事实。第二条性质由AppendEntries提供的一个简单的一致性校验保证。当发送AppendEntries RPC时，leader会包含其日志中紧随新条目之前的index和term。如果follower在其日志中没有找到有相同index和term的条目，那么它会拒绝该新条目。一致性检验会以归纳的步骤执行：日志最初的空状态满足“日志匹配性质”，且每当日志被扩展时一致性检验会保证“日志匹配性质”。因此，每当AppendEntries成功返回，leader会得知follower的日志和它自己的日志直到新条目的位置都是相同的。

在正常的操作中，leader的日志会和follower的日志保持一致，所以AppendEntries的一致性检验永远不会失败。然而，leader崩溃会使日志不一致（旧的leader可能没有完全复制它的日志中的所有条目）。这些不一致会随着一系列leader和follower的故障加剧。**图7**给出了follower可能与新leader不同的原因。follower可能丢失了leader有的一些条目，follower可能多出leader没有的一些条目，或者二者都有。日志中丢失或多出的条目可能垮了多个term。

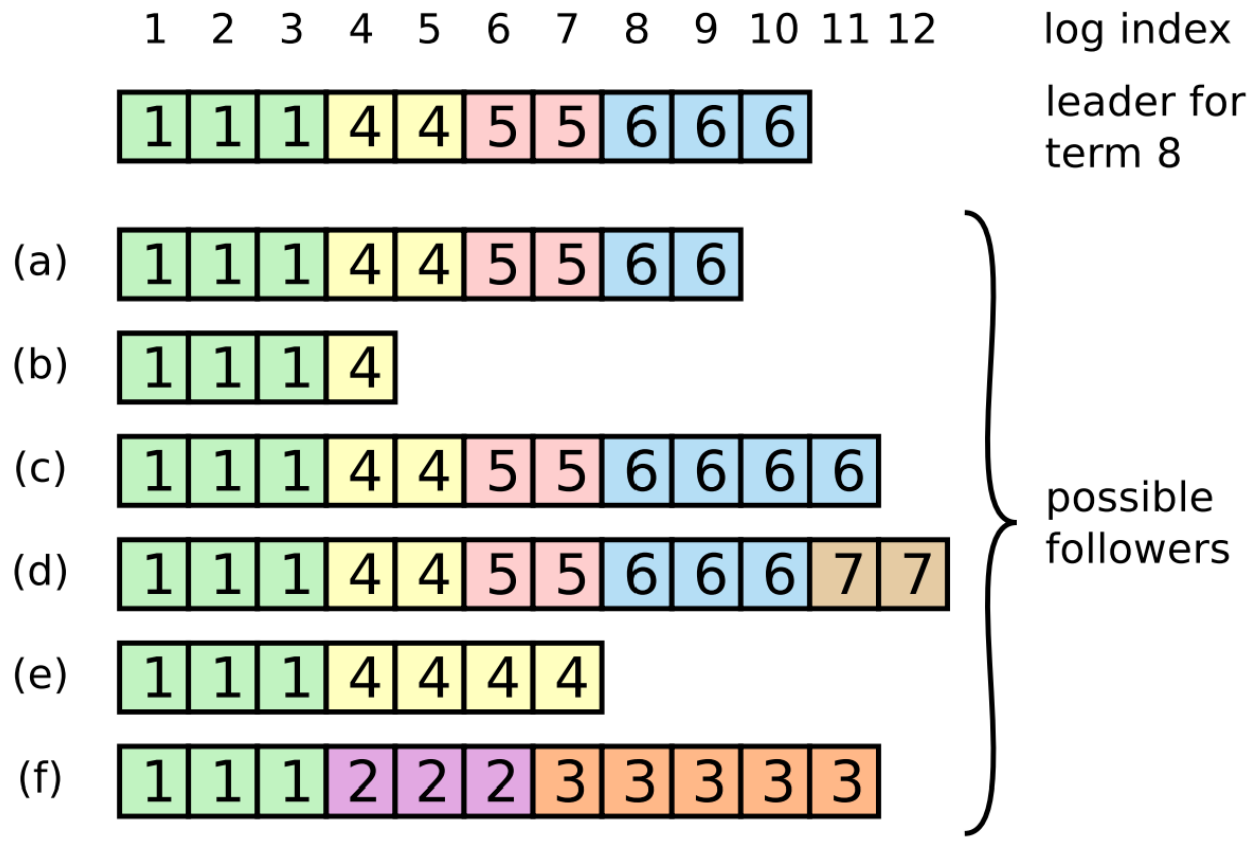

Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

图7 当最上面的leader开始当权时，任意情况(a-f)都可能在follower中发生。每个方框表示一个日志条目，方框中的条目是它的term。follower可能丢失条目（a-b），可能有额外的未被提交的条目（c-d），或者二者都有（e~f）。例如，在情况f中，该服务器曾经是term 2的leader，其将一些条目添加到了它的日志中，然后它在将任意这些条目提交前崩溃了；该服务器快速地重启了并成了了term 3的leader，并又将更多条目添加到了它的日志中；在这些term 2和term 3的条目被提交前，该服务器再次崩溃，并在几个term期间保持离线状态。

在Raft中，`leader会通过强制follower复制其日志的方式处理不一致`。这意味着follower日志中不一致的条目会被覆盖为leader日志中的条目。**章节5.4**将展示当结合一个额外的约束后，这会是安全的。

为了使follower的日志和leader自己的日志一致，leader必须找到二者的日志中最新的一致的日志条目，删除follower日志中该点后的所有条目，并将leader日志中该点后的所有条目发送给follower。所有这些操作都在响应AppendEntries RPC的一致性检验时发生。leader会为每个follower维护一个nextIndex，它是leader将发送给该follower的下一条日志的index。当leader首次掌权时，它会将所有的nextIndex值初始化为其日志的最后一个条目的下一个index（在**图7**中该值为11）。如果follower的日志与leader的不一致，下一次AppendEntries RPC中的一致性检验会失败。当RPC被拒绝后，`leader会减小该nextIndex并重试AppendEntries RPC`。最终，nextIndex会达到leader和follower的日志匹配的点。这时，AppendEntries会成功，它会删除follower日志中任何冲突的条目，并将日志条目从leader的日志中追加到follower的日志中（如果有的话）。一旦AppendEntries成功，follower的日志就与leader的一致，并在该term的余下的时间里保持这样。

如果需要的话，`协议可被优化以减少拒绝AppendEntries RPC的次数`。例如，当follower拒绝AppendEntries请求时，`它可以在响应中带上冲突条目的term和它存储的该term下第一个条目的index`。有了这一信息，leader可以减少相应的nextIndex来绕过该term中所有冲突的条目，这样每个term的日志条目仅需要一次AppendEntries RPC，而不是每个条目需要一次RPC。`在实际中，我们对这一优化是否必要持怀疑态度，因为故障不会频繁发生，且不太可能出现许多不一致的条目。`

通过这种机制，leader在掌权时不需要采取特殊的行为来恢复日志一致性。它只需要开始正常的操作，然后日志会在响应AppendEntries一致性检验故障时自动恢复。leader永远都不会覆写或删除它自己的日志条目（**图3**中的领导只增性质，Leader Append-Only Property）。

这种日志复制机制体现了**第二章**描述的理想的共识性质：只要大多数服务器在线，Raft就可以接受、复制、并应用新日志条目；在正常情况下新日志条目只需要一轮RPC就可以被复制到集群中大多数节点上；单个缓慢的follower不会影响性能。

5.4 安全性

前面的章节描述了Raft如何选举领导和复制日志。然而，这些讨论过的机制目前还不能充分确保每个状态机会精确地按照相同的顺序执行相同的指令。例如，`follower可能在leader提交一些日志条目时不可用，然后它可能被选举为leader并将这些条目覆盖为新的条目`；这样，不同的状态机可能执行不同的指令序列。

本章通过加入了`一个对哪些服务器可以被选举为leader的约束`完善了Raft算法。这一约束确保了在任意给定的term中，`leader包含所有之前term中提交了的日志条目`（**图3**中的领导完整性性质，Leader Completeness Property）。通过选举约束，我们可以让提交的规则更加精确。最后，我们给出了领导完整性性质的简要证明，并展示了它如何让多副本状态机的行为正确。

5.4.1 选举约束

在任何基于leader的共识算法中，leaer最终都必须保存所有已提交的日志条目。在一些共识算法中（像Viewstamped Replication[22]），即使一个服务器最初没有还包含所有已提交的日志条目，它也可以被选举为leader。这些算法有额外的机制来识别确实的日志条目，并在选举时或选举完成后的很短的时间里，将它们传输给新的leader。不幸的是，这需要引入相当多的额外的机制，大大增加了复杂性。Raft使用了一种更简单的方法，该方法会保证在过去的term中已被提交的条目在选举之初就在每个新的leader上，而不需要将这些条目再传输给leader。`这意味着日志条目仅单向流动：从leader到follower，且leader永远不会覆写它的日志中已存在的条目。`

Raft通过投票的过程确保只含有所有已提交日志条目的candidate才能赢得选举。candidate必须联系集群中大多数的才有可能被选举，这意味着每个被提交过的条目一定出现在这些服务器中的至少一个上。如果该candidate的日志至少与这大多数的服务器的日志一样“新（up-to-date）”（up-to-date在后文中有精确的定义），那么它就持有所有已提交的日志条目。`RequestVote RPC实现了这一约束：RPC包括有关candidate的日志的信息，且如果投票者（voter）自己的日志比该candidate的日志更新，那么该投票者会拒绝给这个candidate投票`。

Raft通过比较日志最后一个条目的index和term来确定哪个日志更新。如果日志最后个条目的term不同，那么有更新的term的日志更新。如果两个日志最后的term相同，那么更长的日志更新。

5.4.2 提交之前term的日志条目

正如 **章节5.3** 中描述的那样，一条当前term的日志，一旦它被存储到大多数的服务器上，那么leader会得知这个条目被提交了。如果leader在提交一个条目时崩溃，之后的leader会试图完成这个条目的复制。然而，当之前term的条目被存储到大多数服务器上时，leader无法立刻断定它被提交了。**图8**阐述了一种情况，在该情况下，就的日志条目被存储到了大多数服务器中，但仍会被之后的leader覆盖。

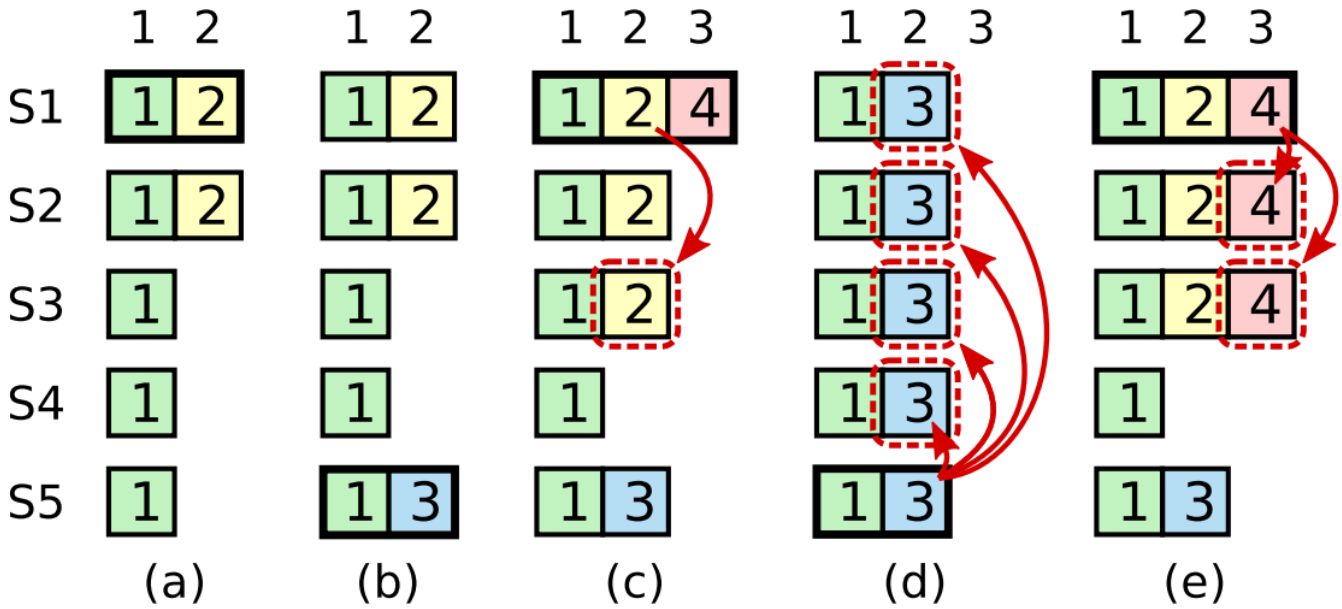

Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

图8 展示为什么leader不能通过之前term的日志条目判断提交的时间序列。在（a）中，S1是leader，它部分复制了index为2的日志条目。在（b）中，S1崩溃，S5收到来自S3、S4和它自己的投票而被选举为term 3的leader，并接受了不同的日志条目作为index 2的条目。在（c）中，S5崩溃，S1重启且被选举为leader，并继续复制。此时，term 2的日志已经被复制到了大多数服务器上，但是没有被提交。如果此时S1像（d）中的情况一样发生崩溃，那么S5将会被选举为leader（S2、S3和S4会为其投票），且会用它自己的term 3的日志条目覆盖index 2中的条目。然而，如果S1在崩溃前复制了其当前term的条目到大多数服务器上，正如（e）中所示，那么这个条目会被提交（S5无法赢得选举)。此时，日志中所有之前的条目都同样被提交了。

为了消除**图8**中的这样的问题，Raft永远不会通过对副本数计数的方式提交之前term的条目。只有leader当前term的日志条目才能通过对副本数计数的方式被提交。而一旦当前term的日志被以这种方式提交后，所有在其之前的日志条目会因“日志匹配性质”被间接地提交。虽然在某些情况下，leader可以安全断定之前的日志条目被提交（例如，该条目被存储到了每台服务器上），但是Raft为了简单起见而采取了更保守的方法。

因为当leader复制来自之前term的日志条目时，日志条目会保持其原来的term号，所以Raft在提交规则中引入了额外的复杂性。在其它共识算法中，如果新的leader要重新复制来自更高“term”的条目，它必须用它的新“term号”来做。Raft的方法会使日志条目的推导更简单，因为无论何时它们都能在日志间维护相同的term号。另外，Raft中的新的leader发送的来自之前term的日志条目比其它算+法更少（其它算法必须发送冗余日志条目，对其重新编号，然后才能提交）。

5.4.3 安全性参数

有了完整的Raft算法，现在我们可以更精确地证明“领导完整性性质”成立（该证明基于安全性的证明，见 **章节 9.2**）。我们假设“领导完整性性质”不成立，那么我们会得出一个矛盾。假设term TTT的leader（leaderTleader_TleaderT）提交了一个该term的日志条目，但是该日志条目没被之后的某个term的leader保存。考虑该leader没有保存该条目的最小term UUU（U>TU>TU>T）。

1. 被提交的条目在leaderUleader_UleaderU被选举时必须不在其日志中（leader从未删除或覆写日志条目）。
2. leaderTleader_TleaderT将该条目复制到了集群中大多数服务器上，且leaderUleader_UleaderU收到了来自集群大多数的投票。因此，至少一个服务器（“投票者”）既从leaderTleader_TleaderT接受了该条目，又为leaderUleader_UleaderU投了票，如**图9所示**。该投票者是达成矛盾的关键。
3. 投票者必须在为leaderUleader_UleaderU投票前接受来自leaderTleader_TleaderT的已提交的条目；否则，它会拒绝来自leaderTleader_TleaderT的AppendEntries请求（它当前的term将会比T高）。
4. 投票者在为leaderUleader_UleaderU投票时仍保存着该条目，因为每个中间leader都包括该条目（基于假设），leader永远不会删除条目，且follower仅在条目与leader冲突时才会删除它们。
5. 投票者将它的选票投给了leaderUleader_UleaderU，因此leaderUleader_UleaderU的日志必须至少于该投票者一样新。这导致两个矛盾之一。
6. 首先，如果该投票者和leaderUleader_UleaderU的最后一条日志条目的term相同，那么leaderUleader_UleaderU的日志必须至少与投票者一样长，所以它的日志包括投票者日志中的每一个条目。这有一个矛盾，因为投票者包含了该已提交的条目而我们假设了leaderUleader_UleaderU没有包括该条目。
7. 否则，leaderUleader_UleaderU的最后一条日志条目的term必须比投票者大。此外，该term还大于TTT，因为投票者的最后一条日志的term至少为TTT（它包括来自term TTT的被提交的日志条目）。创建了leaderUleader_UleaderU的最后一个日志条目的之前的leader的日志必须包含了该被提交的条目（基于假设）。那么，根据“日志匹配性质”，leaderUleader_UleaderU的日志必须同样要包含该被提交的日志，这是一个矛盾。
8. 这样，就产生了完整的矛盾。因此，所有term大于TTT的leader必须包括所有在term TTT中提交的条目。
9. “日志匹配性质”简介地确保了之后的leader也包含了被提交的条目，如**图8（d）**中的index 2一样。

图9 如果S1（term TTT的leader）提交了来自它的term的新的日志条目，且S5在之后的term UUU中被选举成为leader，那么必须有至少一个服务器（S3），其接受了该日志条目，并给S5投了票。

有了“领导完整性性质”，我们可以证明**图3**中的“状态机安全性质”，即：如果一个服务器将给定index的日志条目应用到了其状态机中，那么不会有另一台应用了index相同但内容不同的日志条目的服务器。当服务器将一个日志条目应用到其状态机时，它直到该条目的日志必须与leader的日志相同，且该条目必须是被提交的。现在考虑任何服务器都应用了给定index的日志条目的最小的term；“日志完整性性质”确保了所有term更高的leader都保存了相同的这个条目，所以在之后的term中应用了该条目的服务器将会应用相同的值。因此，“状态机安全性性质”成立。

最后，Raft要求服务器按照日志index的顺序应用条目。结合“状态机安全性质”，这意味着服务器会精确地按照相同的顺序将相同的日志条目应用到它们的状态机中。

5.5 follower和candidate崩溃

目前我们都专注于leader故障的情况。处理follwer和candidate崩溃比处理leader崩溃简单得多，且这二者的故障可以用相同的方式处理。如果一个follower或candidate故障，那么之后发送到它的RequestVote和AppendEntries RPC会失败。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RPC将会成功完成。如果服务器在完成RPC后但在响应前故障，那么在它重启后会收到相同的RPC。Raft的RPC是幂等的，所以这不会造成影响。例如，如果一个follower收到了一个包含了已经在它日志中的条目的AppendEntries，它会忽略新请求中的那些条目。

5.6 定时和可用性

我们对Raft的要求之一是安全性决不能依赖定时：系统绝不能因某个时间发生的比预期的更快或更慢而产生错误的结果。然而，可用性（系统能够及时响应客户端的能力）必须不可避免地依赖定时。例如，如果崩溃服务器间的消息交换消耗了比通常更长的时间，candidate将不会保持足够的时间以赢得选举。没有稳定的leader，Raft不能进展（make progress，译注：本文中指可以继续提交新值）。

领导选举是Raft中定时最重要的一个方面。只要系统满足如下的 *定时要求（timing requirement）* ，那么Raft就能够选举并维护一个稳定的leader：

broadcastTime≪electionTimeout≪MTBF broadcastTime \ll electionTimeout \ll MTBF
broadcastTime≪electionTimeout≪MTBF

在这个不等式中，broadcatTimebroadcatTimebroadcatTime是服务器将RPC并行地发送给集群中的每个服务器并收到它们的响应的平均时间，electionTimeoutelectionTimeoutelectionTimeout是 **章节5.2** 中描述的选举超时，MTBFMTBFMTBF是单个服务器发生两次故障间的平均时间。broadcastTimebroadcastTimebroadcastTime应该比electionTimeoutelectionTimeoutelectionTimeout小一个数量级，这样leader可以可靠的发送心跳消息以阻止选举发生；因为exectionTimeoutexectionTimeoutexectionTimeout采用了随机方法，这一不等性也让投票决裂不太可能发生。electionTimeoutelectionTimeoutelectionTimeout应该比MTBFMTBFMTBF小几个数量级，这样系统能够取得稳定的进展。当leader崩溃时，系统将会在大概electionTimeelectionTimeelectionTime的时间内不可用，我们想让这一时间仅占总时间的很小的比例。

broadcastTimebroadcastTimebroadcastTime和MTBFMTBFMTBF是下层系统的属性，而electionTimeoutelectionTimeoutelectionTimeout是必须由我们选取的。Raft的RPC通常需要收件人将信息持久化到稳定存储中，所以broadcastTimebroadcastTimebroadcastTime可能在0.5ms到20ms不等，这取决于存储技术。因此，electionTimeoutelectionTimeoutelectionTimeout可能在10ms到500ms之间。通常服务器的MTBF为几个月或更长时间，这可以轻松满足定时要求。

6. 集群成员变更

到目前为止，我们都假设集群 *配置（configuration）*（参与共识算法的服务器集合）是固定的。在实践中，偶尔变更配置是必要的，例如当服务器故障时替换它们，或修改副本数。尽管这可以通过该将整个集群下线、更新配置文件并重启集群来实现，但是这会让集群在变更期间不可用。另外，任何人工参与的步骤都有操作错误的风险。为了避免这些问题，我们决定将配置变更自动化，并将它整合到Raft共识算法中。

为了使配置变更机制是安全的，在切换期间决不能有同时出现两个被选举出的leader的情况。不幸的是，任何让服务器直接从就配置转换到新配置的方法都是不安全的。立刻对所有服务器进行切换是不可能的，因此集群在切换中可能被分为两个相互独立的“大多数”（见**图10**）。

图10 直接从一个配置切换到另一个配置是不安全的，因为不同的服务器会在不同时间切换。在本例中，集群从3个服务器增长到5个服务器。不幸的是，有一个时间点可能出现在同一个term中有两个被选举出来的leade的情况，其中一个leader有旧配置（ColdC *{old}Cold）的大多数，另一个有新配置（CnewC* {new}Cnew）的大多数。

为了确保安全，配置变更必须使用两阶段的方法。实现这两个阶段有很多方式。例如，有些系统（例如[22]）在第一阶段禁用旧配置，这样它就不能处理客户端请求，然后在第二阶段应用新配置。在Raft中，集群首先会切换到过渡配置，我们称之为 *联合共识（joint consensus）*；一旦联合共识被提交，接着系统会切换到新配置。联合共识结合了旧配置和新配置：

- 日志条目会被复制到两个配置的所有服务器上。
- 任一个配置的任何服务器都可能成为leader。
- 一致（用来选举和日志条目提交）需要在旧配置和新配置中的大多数分别达成。

联合共识让每个服务器能在不同时间切换配置而不需要做出安全性妥协。另外，联合共识让集群能够在配置变更时继续为客户端请求提供服务。

图11 配置变更的时间线。虚线表示已被创建但还没被提交的配置条目，实现表示最新的已被提交的配置条目。leader首先在它的日志中创建了配置条目Cold,newC *{old,new}Cold,new，并将其提交到了Cold,newC* {old,new}Cold,new中（ColdC *{old}Cold中的大多数和CnewC* {new}Cnew中的大多数）。然后它创建CnewC *{new}Cnew条目并将其提交到CnewC* {new}Cnew的大多数。不存在ColdC *{old}Cold和CnewC* {new}Cnew能独立作出决策的时间。

集群配置的存储和通信使用了多副本日志中的特殊条目。**图11**阐释了配置变更的过程。当leader收到将配置从ColdC *{old}Cold变更为CnewC* {new}Cnew的请求时，它会将该配置的联合共识（图中的Cold,newC *{old,new}Cold,new）作为一个日志条目存储，并将该条目使用之前描述的机制复制。一旦某个服务器将新配置的条目加入到了它的日志中，它会在所有之后的决策中应用该配置（服务器总是使用它的日志中的最新的配置，无论该条目是否被提交了）。这意味着leader会使用Cold,newC* {old,new}Cold,new的规则来决定什么时候Cold,newC *{old,new}Cold,new的日志条目被提交。如果leader崩溃，新的leader可能在ColdC* {old}Cold或Cold,newC *{old,new}Cold,new下被选举出，这取决于赢得选举的candidate有没有收到Cold,newC* {old,new}Cold,new。这期间的任何情况下，CnewC_{new}Cnew都不能单独做决策。

一旦Cold,newC *{old,new}Cold,new已经被提交，无论ColdC* {old}Cold还是CnewC *{new}Cnew都不能在没有对方认同的情况下做决策，而“领导完整性特性”确保了只有Cold,newC* {old,new}Cold,new的日志条目的服务器才可能被选举为leader。现在leader可以安全地创建描述CnewC *{new}Cnew的日志条目并将其复制到集群中。第二次也是一样，配置一被服务器看到就会生效。当新的配置已在CnewC* {new}Cnew的规则下被提交时，旧的配置就无关紧要了，且不在新配置下的服务器可以被关机。如**图11**所示，在任何时间ColdC *{old}Cold和CnewC* {new}Cnew都不能单独做决策，这确保了安全性。

重配置有另外三个问题待解决。第一个问题是，新的服务器最初可能没有存储任何日志条目。如果它们在这种状态下加入集群，它可能需要很长时间来追赶，这段时间内它可能不能提交新的日志条目。为了避免可用性有间隔，Raft在配置变更前引入了一个额外的阶段，在这一阶段中，新的服务器作为不投票的成员加入集群（leader会将日志条目复制给它们，但它们不在关于“大多数”的考虑范围内）。一旦新服务器追赶上了集群中其余的服务器，重配置可按之前描述的那样继续。

第二个问题是，集群的leader可能不是新的配置的一部分。在这种情况下，一旦leader提交了CnewC *{new}Cnew的日志条目，它就会下台（返回为follower状态）。这意味着，在一段时间内（当leader提交CnewC* {new}Cnew时），leader会管理不包括它自己在内的集群。它将日志复制到不包括自己在内的大多数中。leader在CnewC *{new}Cnew被提交时发生切换，因为这是新配置可以单独操作的第一个时间点（它将总是可以从CnewC* {new}Cnew中选择一个新leader）。在这个时间点之前，可能处于只有ColdC_{old}Cold中的服务器才能被选举为leader的情况下。

第三个问题是，被移除的服务器（那些不在CnewC_{new}Cnew中的服务器）可能会扰乱集群。这些服务器将不会收到心跳，所以它们会超时并开始新的选举。接着，他们会使用新的term号发送RequestVote RPC，这会导致当前的leader转换成follower状态。最终，新的leader会被选举出来，但是被移除的服务器将再次超时，这一过程会重复，这会导致很弱的可用性。

为了防止这一问题，当服务器认为当前有leader存在时，它们会忽略RequestVote RPC。具体地说，如果一个服务器在当前leader的最小选举超时内收到了RequestVote RPC，它不会更新其term或投票。这不会影响正常的选举，在正常的选举中，每个服务器都在开始选举前等待了最小选举超时时间。这可以帮助免受被移除的服务器的打扰：如果leader能够给其集群发送心跳，那么它不会被更大的term号废除。

7. 日志压缩

Raft的日志会随着正常的操作增长，以合并更多的客户端请求。但是在实用系统中，它不能不受限地增长。随着日志越来越长，它会占用越来越的的空间，并需要更长的时间来重放。最终，如果没有机制来丢弃日志中积累的过时的信息，这会导致可用性问题。

快照策略是最简单的压缩方式。在快照策略中，当前的整个系统状态会被写入到稳定存储上的一个快照中，然后直到这一点的所有日志条目会被丢弃。快照策略被使用在Chubby和Zookeeper中，本章其余部分会描述Raft中的快照策略。

用来压缩的增量方法（如日志清理（log cleaning）[36]和日志结构合并树[30， 5]）也是可行的。这些操作每次会处理一定比例的数据，所以它们能够更均匀地分摊压缩的负载。首先，它们会选取一个已经积累了许多被删除或被覆写的对象的数据区域；然后，它们将这个区域中存活的对象更紧凑地重新，并释放该区域。这与快照策略相比，需要很多额外机制与复杂性，快照策略通过总是对整个数据集操作的方式简化了这一问题。日志清理还需要对Raft进行修改，而状态机能使用与快照策略相同的接口实现LSM树。

图12 服务器使用新的快照取代它日志中已提交的条目（index 1~5），快照只保存了当前状态（在本例中为变量xxx和yyy）。该快照包括的最后的index和term用来安置紧随其后的日志条目6

图12展示了Raft的快照的基本想法。每个服务器独立地创建快照，快照仅覆盖服务器日志中已提交的条目。大部分的工作由状态机将其当前状态写入到快照组成。Raft还在快照中包括了少量的元数据： *last included index* 是快照替换的最后一个条目（状态机应用的最后一个条目）的index， *last included term* 是这一条目的term。这些被保存的信息用来支持对紧随快照后面的第一个条目的AppendEntries的一致性检验，因为该条目需要前一条日志的index和term。为了启用集群成员变更（见 **第六章**），快照还包括日志中直到last included index的最新的配置。一旦服务器完成了快照写入，它会删除直到last included index的所有日志条目和任何之前的快照。

尽管正常情况下服务器独立地创建快照，leader偶尔必须将快照发送给落后的follower。这会在leader已经丢弃了它需要发送给follower的下一条日志条目是发生。幸运的是，这种情况不太可能在正常操作中出现：在正常情况下，跟上了leader的follower中已经有了这一条目。然而，异常缓慢的follower或新加入集群中的服务器（见 **第六章**）中没有这一条目。对leader来说，让这样的follower追上新状态的方法就是通过网络向其发送快照。

图13 InstallSnapshot RPC总结。快照被分割成块来传输，这可以通过给follower发送每个块时告知其存活，这样follower可以重设选举定时器。

leader使用一种被称为InstallSnapshot的新的RPC像落后太多的follower发送快照，如**图13**所示。当follower通过这个RPC收到一个快照时，它必须决定如何处理它已有的日志条目。通常，快照会包含在接收者的日志中还没有的新信息。在这种情况下，follower会丢弃它全部日志，因为这些日志都可以被快照取代，且日志中可能含有与快照冲突的未提交的日志条目。相反，如果follower收到的快照描述的是它日志的前面一部分（由于重传或错误），那么被快照覆盖的日志条目会被删除，但是快照之后的条目仍有效且必须被保留。

这种快照策略与Raft的强领导原则相驳，因为follower可以不通过leader的知识来创建快照。然而，我们认为这一偏差是合理的。尽管通过leader可以避免达成共识时的决策冲突，但是由于共识已经在快照中被达成过，所以不会有决策冲突。数据仍仅从leader流向follower，只是现在只有follower认识它们的数据。

我们考虑了另一种基于leader的方法，在这个方法中只哟iuileader会创建一个快照，然后它会将快照发给它的每个follower。然而，这有两个缺点。首先，向每个follower发送快照会浪费网络带宽，且会让快照创建的进程变慢。每个follower已经有了创建它自己的快照所需的信息，且通常服务器根据其本地状态创建快照比通过网络发送和接受快照开销低很多。第二，leader的实现会变得更复杂。例如，leader会需要并行地执行向follower发送快照的操作和给它们复制日志条目的操作，这样才能不阻塞新的客户端请求。

影响快照性能的问题还有另外两个。第一，服务器必须决定什么时候创建快照。如果服务器创建快照太过频繁，它会浪费磁盘带宽和能量；如果创建快照太少，它会面临好近其存储容量的风险，且会增加重启时重放日志的时间。一个简单的策略时当日志大小达到固定字节数时创建快照。如果设置的大小明显比期望的快照大小大很多，那么创建快照时的额外磁盘带宽开销会很小。

第二个性能问题是写入快照可能需要特别多的时间，且我们不想让这耽搁正常的操作。其解决方案是使用写入时复制（copu-on-write）技术，这样新的更新可以在不影响快照写入的情况下被接受。例如，由功能性数据结构构造的状态机本身就支持这一点。另外，操作系统的写入时复制支持（例如Linux的fork）可被用作对整个状态机创建内存式快照（我们的实现使用了这一方法）。

8. 客户端交互

本章描述了客户端如何与Raft交互，包括客户端如何找到集群leader和Raft如何支持线性语义[10]。这一问题存在于所有的基于共识的系统，且Raft的解决方案与其它系统类似。

Raft的客户端会将所有的请求发送给leader。当客户端首次启动时，它会随机选择一个服务器连接。如果客户端首次选择的服务器不是leader，该服务器会拒绝该客户端的请求，并为其提供它所知道的最新的leader的相关信息（AppendEntries请求需要leader的网络地址）。如果leader崩溃，客户端的请求会超时，客户端会再次随机挑选一个服务器重试。

我们对Raft的目标是实现线性语义（每个操作看上去是在它的调用和响应期间的某一时刻被瞬时（instantaneously）、至少一次执行（exactly once）的）。然而，目前我们描述的Raft算法可能多次执行统一指令：例如，如果leader在提交日志条目但是没有响应客户端的时候崩溃，客户端会向新的leader重试该指令，这导致该指令会被二次执行。其解决方案是，让客户端为每个指令分配一个唯一的序列号。然后，状态机会记录每个客户端执行过的最新的序列号和相应的响应。如果它收到的请求的序列号已经被执行过，那么它不会执行请求，并立即发出响应。

处理只读操作不需要向日志写入任何东西。然而，如果不引入额外的方法，其会有返回陈旧数据的风险，因为响应该请求的leader可能已经被一个它未知的新leader取代。线性读取严禁返回陈旧的数据，Raft需要两个不使用日志的额外的预防措施来确保这一点。第一，leader必须有关于哪些条目被提交了的最新信息。“领导完整性性质”确保了leader有所有已提交的条目，但是在它的term开始时，它可能不知道哪些条目是被提交了的。为了得知哪些条目已被提交，它必须提交一个来自于它的term的条目。Raft通过让每个leader在它的term开始时向日志提交一个空的 *no-op* 条目的方式处理这一问题。第二，leader必须在处理只读请求前检查其是否被废除（如果有更新的leader被选举出来，之前的leader的信息可能是陈旧的）。Raft通过让leader在响应只读请求前与集群的大多数节点交换心跳消息来处理这一问题。另外，leader可以依赖心跳机制来提供租约（lease）[9]，但这需要依赖定时来保证安全性（其假时钟偏斜（clock skew）是有界的）。

9. 实现和评估

我们已经实现了Raft算法，其作为RAMCloudRAMCloud中保存配置信息的多副本状态机的一部分，并为RAMCloud协调器的故障转移提供帮助。该Rafte实现包含了约2000行C++代码，但不包括测试、注释、或空白行。该源码可以随意访问[23]。此外，还有约25个Raft的第三方开源实现[34]被用在不同的开发领域，它们基于本论文的草稿。另外，很多公司都部署了基于Raft的系统[34]。

本章其余部分将从三个角度来评估Raft：可理解性、正确性、和性能。

9.1 可理解性

为了衡量Raft的可理解性并与Paxos对比，我们在Stanford University的Advanced Operating System课程和U.C.Berkeley的Distributed Computing课程的高年级本科生和研究生中开展了一项实验。我们分别录制了有关Raft和Paxos的课程，并编写了相关的小测验。Raft的课程覆盖了本文除了日志复制的内容；Paxos覆盖了能创建等效的多副本状态机的内容，包括单决策Paxos、多决策Paxos、重配置、和实践中需要的较少的一些优化（例如领导选举）。小测验检测了对算法的基本理解，且要求学生推理极端情况。每个学生都先看一个视频，做相关的测验，然后看第二个视频，再做第二个测验。为了避免学习的先后顺序对表现和经验的影响，大概半数的参与者先做了Paxos的部分，另一半先做了Raft的部分。我们通过比较参与者在两个测验中的分数来判断参与者是否对Raft有更好的理解。

我们尽可能公平地在Paxos和Raft间挫比较。实验对Paxos有两方面偏向：43名参与中的15个称他们之前对Paxos有一定经验，且Paxos的视频比Raft的视频长了14%。如**表1**所示，我们采取了一些措施来减轻潜在的偏见。我们所有的材料都可供审查[28, 31]。

表1 学习中可能对Paxos存在的偏见、为每种情况采取的措施、可用的其他材料。

平均来看，参与者在Raft的测验中比Paxos的测验高了4.9分（总分为60分，Raft的平均分是25.7，Paxos的平均分是20.8）。**图14**展示了他们每个人的分数。ttt-检验表明，在95%的置信度下，Raft分数的真实分布比Paxos的至少高2.5分。

图14 比较43名参与者在Raft和Paxos的测验中的表现的散点图。斜线以上的的点（33个）表示Raft分更高的参与者。

我们还基于三个因素创建了一个用来预测新学生的测验分数的线性回归模型，这三个因素是：他们做了哪个测验、他们之前对Paxos的理解程度、和他们学习算法的顺序。根据该模型的预测，测验的选择偏向Raft 12.5分。这比实际看到的4.9分的差异高很多，因为许多学生实际上之前都有Paxos的经验，这对理解Paxos很有帮助，而对理解Raft的帮助相对较少。奇怪的是，该模型还预测先参加过Paxos测验的参与者比先参加Raft测验的参与者低6.3分，虽然我们不知道为什么，但这在统计上确实很重要。

我们在参与者做完测验后，我们还对他们进行的调研，以看他们觉得那种算法更容易实现或解释，这一结果如**图15**所示。超过大多数的参与者报告称Raft更容易实现与解释（对每个问题都有41人中的33个这样表示）。然而，这些自己报告的感觉可能没有参与者的测试分数那么可靠，且参与者可能因为我们对Raft的“更易理解”的假设产生偏见。

关于Raft在用户学习方面的讨论可见[31]。

图15 参与者通过5个程度衡量哪个算法更容易在有价值、正确、高效的系统中实现（左）和哪个对CS的研究生来说更容易解释（右）。

9.2 正确性

我们已经为 **第五章** 描述的共识机制进行了形式化规范与证明。形式化的规范[31]使用了TLA+规范语言[17]能让**图2**中总结的信息完全准确。其有大概400行长，并可以作为证明。它对任何实现Raft的人来说也很有用。我们通过TLA证明系统[7]机械化地证明了“日志完整性性质”。然而，该证明依赖了还没被机械化检验的不变式（日历，我们还没有证明规范的类型安全性）。此外，我们为“状态机安全性性质”编写了完整（它仅依赖规范本身）且相对准确的非形式化的证明[31]（其大概有3500个词那么长）。

9.3 性能

Raft的性能与其他共识算法（如Paxos）相似。对性能来说，最重要是一个已建立的leader复制新日志条目的时候。Raft使用了最少的消息数量来实现这一点（从leader到集群中的半数只需要一轮）。Raft的性能还可以进一步提升。例如，Raft可以轻松支持分批（batching）和流水线处理（pipelining）请求，以获得高吞吐量和低延迟。在其它算法的文献中，已经有各种优化方法被提出。这些优化中，许多都可以被用在Raft中，但我们将此留给了后续工作。

我们使用我们的Raft实现测量了Raft领导选举算法的性能，并回答了两个问题。第一，领导选举过程能快速收敛吗？第二，领导崩溃后可实现的最小停机时间是多少？

图16 检测与替换崩溃的leader的时间。上图中设置了不同的electionTimeout随机范围，下图中设置了不同的最小\electionTimeout。每条线代表每个不同的electionTime的范围的1000次考察（除了“150-150ms”，其只有100次）；例如“150-150ms”表示对electionTimeout在150ms到155ms间随机且均匀分布的实验的考察。这些测量值是在5个服务器组成的集群得出的，该集群广播时间大概在15ms左右。9个服务器组成的集群实验结果相似。

为了测量领导选举，我们反复地让5个服务器组成的leader崩溃，并测量集群多长时间能检测到崩溃并选举出新的leader（见图**16**）。为了保证最坏情况，每次考察中的日志长度都不同，所以一些candidate没有成为leader的资格。另外，为了提高投票决裂的产生，我们的测试脚本会在杀死leader的进程前触发一个同步的心跳RPC广播（这近似于leader在复制了新条目后崩溃的情况）。leader会在心跳时间内均匀且随机地崩溃，对于所有测试，心跳时间是最小\electionTimeout的一半。因此，可能存在的最小停机事件大概是electionTimeout的一半。

图16的上图表明，electionTimeout只要小范围的随机化就足以在选举中避免投票决裂。在我们的测试中，如果没有随机化，领导选举会持续超过10秒，这是因为产生了许多的投票决裂。仅增加5ms的随机化也能显著改善这一问题，其平均停机时间为287ms。使用更大的随机范围能够改善最坏情况的表现：（在超过1000次实验中）50ms的随机范围的最坏完成时间为513ms。

图16的下图表明减小\electionTimeout可以减少停机时间。当electionTimeout为12_{24ms}时，选举leader的平均时间仅为35ms（最长的一次为152ms）。然而，降低该超时时间可能会违背Raft的定时要求：leader在其它服务器开始新的选举前很难将心跳广播出去。这可能导致不必要的领导变更并降低整个系统的可用性。我们推荐使用保守的electionTimeout，如150^{300ms}；这样的超时不太可能造成不必要的领导变更，且仍能提供良好的可用性。

10. 相关工作

共识算法相关的出版物有很多，这些出版物大多都属于如下类别之一：

- Lamport对Paxos的原始描述[15]，和将其解释地更清晰的尝试[16, 20, 21]。
- Paxos的详述，其填补了算法的缺失，并对算法做出修改以为实现提供更好的基础[26, 39, 13]。
- 实现了共识算法的系统，如Chubby[2, 4]、ZooKeeper[11, 12]，和Spanner[6]。Chubby和Spanner的算法没有详细地公开发表，但二者都声称它们基于Paxos。Zookeeper的算法发表得更详细一些，但它与Paxos很不同。
- 能应用到Paxos的性能优化[18, 19, 3, 25, 1, 27]。

- Oki和Liskov的Viewstamped Replication（VR），这是另一个实现共识的方法，它和Paxos大概在相同的时间被开发出。它的初始描述[29]非常复杂，其使用了一个分布式事务协议，但是核心共识协议在最近的更新[22]中被分离了出来。VR使用了基于leader的方法，它与Raft有很多相似之处。

Raft和Paxos最大的不同在于Raft的强领导权：Raft的领导选举是共识协议必要的部分，且Raft尽可能多地将功能集中到了leader中。这使算法更简单，且更容易理解。例如，在Paxos中，领导选举与基本的共识协议是独立的：它仅作为性能优化，而实现共识并不需要它。然而，这导致需要额外的机制：Paxos基本共识包括两段协议和用来领导选举的独立的机制。相反，Raft直接讲领导选举合并到了算法中，并用它作为两段共识的第一段。这使Raft的机制比Paxos更少。

像Raft一样，VR和Zookeeper也基于leader，因此它们和Paxos相比，有很多与Raft相同的优势。然而，Raft比VR或Zookeeper的机制更少，因为它减少了非leader的功能。例如，Raft中的日志条目仅单向流动：通过AppendEntries RPC从leader外流。在VR中，日志条目有两个流动方向（leader可以在选举过程中接收日志条目），这需要额外的机制和复杂性。Zookeeper发表的描述中，日志条目同样能传输给leader也能从leader传输，但是它的实现似乎更像Raft[35]。

Raft为达成共识所需的消息类型比其他我们知道的基于日志复制的算法更少。例如，我们数了VR和Zookeeper用作基本共识和成员变更的消息类型数（不包括日志复制和成员交互，因为它们几乎与算法相互独立）。VR和Zookeeper都定义了10中不同的消息类型，而Raft仅有4中消息类型（两种RPC请求和它们的响应）。Raft的消息比其它算法的更浓缩一点，但是总体上更简单。另外，在VR和Zookeeper的描述中，它们在leader变更时会传输整个日志，在实际情况下，将需要额外的消息类型来优化这些机制。

Raft的强领导权方法简化了算法，但它也妨碍了一些性能优化。例如，Egalitarian Paxos（ePaxos）可以在一些条件下通过无leader的方法实现更好的性能[27]。EPaxos利用了状态机指令的可交换性。任何服务器，在有其它指令被提议并与它发生指令交换时，可以仅用一轮通信就能提交一个指令。然而，如果被并发提议的指令没有互相发生交换，EPaxos就需要额外一轮通信。因为任何服务器都可能提交指令，EPaxos在服务器间的负载均衡更好，且在广域网配置下能实现比Raft更低的延迟。然而，它比Paxos又增添了相当的复杂性。

在其它工作中，有很多不同的用作集群成员变更的方法被提出或实现，包括Lamport最初的提议[15]、VR[22]、和SMART[24]。我们为Raft选择了联合共识的方法，因为它利用了其余的共识协议，因此只需要很少的额外机制来实现成员变更。Raft没有选择Lamport的α\alphaα方法，因为它假设共识的达成不需要leader。相比VR和SMART，Raft的重配置算法的优势是，成员变更时不需要限制对正常请求的处理；相反，VR在配置变更期间会停止所有正常的处理，SMART对未完成的请求数量有类α\alphaα的限制。Raft的方法还比VR或SMART加入了更少的额外机制。

11. 结论

算法通常是以正确、效率、和/或简洁为主要目标设计的。尽管这些都是很有价值的目标，但我们认为可理解性和这些目标一样重要。在开发者将算法转化为实用的实现之前，其它目标都无法实现，且实用的实现不可避免地会脱离或扩展算法发表的形式。除非开发者对算法有很深的理解并能建立有关该算法的直觉，否则他们很难在实现过程中保留算法期望的性质。

本文中，我们解决了分布式共识的问题。该领域有一个被广泛接受但是难以理解的算法Paxos，它已经挑战了学生和开发者很多年。我们开发了一个新算法Raft，我们已经表明它比Paxos更好理解。我们还认为Raft为系统构建提供了更好的基础。把可理解性作为设计的主要目标改变我我们设计Raft的方法。在设计的过程中，我们发现我们反复地使用了一些技术，如分解问题和简化状态空间。这些技术不但提高了Raft的可理解性，还让我们能更好地理解它的正确性