Raft

摘要

Raft 是用来管理复制日志（replicated log）的一致性协议。它跟 multi-Paxos 作用相同，效率也相当，但是它的组织结构跟 Paxos 不同。这使得 Raft 比 Paxos 更容易理解并且更容易在工程实践中实现。为了使 Raft 协议更易懂，Raft 将一致性的关键元素分开，如 leader 选举、日志复制和安全性，并且它实施更强的一致性以减少必须考虑的状态的数量。用户研究的结果表明，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，它使用重叠的大多数（overlapping majorities）来保证安全性。

1 介绍

一致性算法允许多台机器作为一个集群协同工作，并且在其中的某几台机器出故障时集群仍然能正常工作。 正因为如此，一致性算法在建立可靠的大规模软件系统方面发挥了关键作用。 在过去十年中，Paxos [15,16] 主导了关于一致性算法的讨论：大多数一致性的实现都是基于 Paxos 或受其影响，Paxos 已成为用于教授学生一致性相关知识的主要工具。

不幸的是，Paxos 实在是太难以理解，尽管许多人一直在努力尝试使其更易懂。 此外，其架构需要复杂的改变来支持实际系统。 结果是，系统开发者和学生都在与 Paxos 斗争。

在我们自己与 Paxos 斗争之后，我们开始着手寻找一个新的一致性算法，可以为系统开发和教学提供更好的基础。 我们的方法是不寻常的，因为我们的主要目标是可理解性：我们可以为实际系统定义一个一致性算法，并以比 Paxos 更容易学习的方式描述它吗？在该算法的设计过程中，重要的不仅是如何让该算法起作用，还有清晰地知道该算法为什么会起作用。

这项工作的结果是一个称为 Raft 的一致性算法。 在设计 Raft 时，我们使用了特定的技术来提高可理解性，包括分解（Raft 分离 leader 选举，日志复制和安全）和状态空间减少（相对于 Paxos ，Raft 减少了不确定性程度和服务器之间彼此不一致的方式 ）。 一项针对两个大学的 43 名学生的用户研究表明，Raft 比 Paxos 更容易理解：在学习两种算法后，其中 33 名学生能够更好地回答关于 Raft 的问题。

Raft 在许多方面类似于现有的一致性算法（尤其是 Oki 和 Liskov 的 Viewstamped Replication [29,22]），但它有几个新特性：

- **Strong leader**：在 Raft 中，日志条目（log entries）只从 leader 流向其他服务器。 这简化了复制日志的管理，使得 raft 更容易理解。
- **Leader 选举**：Raft 使用随机计时器进行 leader 选举。 这只需在任何一致性算法都需要的心跳（heartbeats）上增加少量机制，同时能够简单快速地解决冲突。
- **成员变更**：Raft 使用了一种新的联合一致性方法，其中两个不同配置的大多数在过渡期间重叠。这允许集群在配置更改期间继续正常运行。

我们认为，Raft 优于 Paxos 和其他一致性算法，不仅在教学方面，在工程实现方面也是。 它比其他算法更简单且更易于理解; 它被描述得十分详细足以满足实际系统的需要; 它有多个开源实现，并被多家公司使用; 它的安全性已被正式规定和验证; 它的效率与其他算法相当。

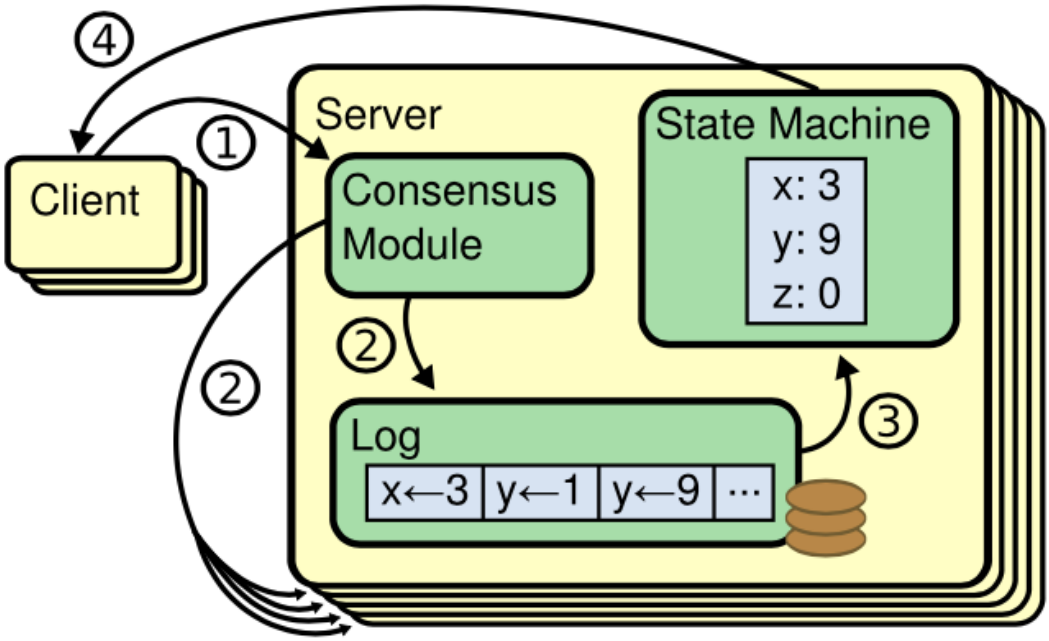
本文的剩余部分介绍了复制状态机问题（第 2 节），讨论了 Paxos 的优点和缺点（第3节），描述了我们实现易理解性的方法（第 4 节），提出了 Raft 一致性算法（第 5-8 节），评估 Raft（第 9 节），并讨论了相关工作（第 10 节）。

2 复制状态机

一致性算法是在复制状态机[37]的背景下产生的。 在这种方法中，一组服务器上的状态机计算相同状态的相同副本，并且即使某些服务器宕机，也可以继续运行。

复制状态机用于解决分布式系统中的各种容错问题。 例如，具有单个 leader 的大规模系统，如 GFS [8]，HDFS [38] 和 RAMCloud [33]，通常使用单独的复制状态机来进行 leader 选举和存储 leader 崩溃后重新选举需要的配置信息。Chubby [2] 和 ZooKeeper [11] 都是复制状态机。

复制状态机通常使用复制日志实现，如图 1 所示。每个服务器存储一个包含一系列命令的日志，其状态机按顺序执行日志中的命令。 每个日志中命令都相同并且顺序也一样，因此每个状态机处理相同的命令序列。 这样就能得到相同的状态和相同的输出序列。



一致性算法的工作就是保证复制日志的一致性。 每台服务器上的一致性模块接收来自客户端的命令，并将它们添加到其日志中。 它与其他服务器上的一致性模块通信，以确保每个日志最终以相同的顺序包含相同的命令，即使有一些服务器失败。 一旦命令被正确复制，每个服务器上的状态机按日志顺序处理它们，并将输出返回给客户端。 这样就形成了高可用的复制状态机。

实际系统中的一致性算法通常具有以下属性：

- 它们确保在所有非拜占庭条件下（包括网络延迟，分区和数据包丢失，重复和乱序）的安全性（不会返回不正确的结果）。
- 只要任何大多数（过半）服务器都可以运行，并且可以相互通信和与客户通信，一致性算法就可用。 因此，五台服务器的典型集群可以容忍任何两台服务器的故障。 假设服务器突然宕机，它们可以稍后从状态恢复并重新加入群集。
- 它们不依赖于时序来确保日志的一致性：错误的时钟和极端消息延迟在最坏的情况下会导致可用性问题（译者注：言外之意是可以保证一致性）。
- 在通常情况下，只要集群的大部分（过半服务器）已经响应了单轮远程过程调用，命令就可以完成;少数（一半以下）慢服务器不会影响整个系统性能。

3 Paxos 存在的问题

在过去十年里，Leslie Lamport 的 Paxos 协议[15]几乎成为一致性的同义词：它是课堂上教授最多的一致性协议，并且大多数一致性的实现也以它为起点。 Paxos 首先定义了能够在单个决策（例如单个复制日志条目）上达成一致的协议。 我们将这个子集称为 single-decree Paxos。 然后 Paxos 组合该协议的多个实例以促进一系列决策，例如日志（multi-Paxos）。 Paxos能够确保安全性和活性，并且支持集群成员的变更。它的正确性已被证明，并且在正常情况下是高效的。

不幸的是，Paxos 有两个显著的缺点。 第一个缺点是 Paxos 非常难以理解。 Paxos 的描述晦涩难懂，臭名昭著（译者注：《The Part-time Parliament》比较晦涩难懂，但是《Paxos Made Simple》就比较容易理解）；很少有人成功地理解它，即使能理解也必须付出巨大的努力。 因此，已有几个尝试用更简单的方式来描述 Paxos [16,20,21]。 这些描述集中在 single-degree Paxos ， 但它们仍然具有挑战

性。 在对 NSDI 2012 参会者的非正式调查中，我们发现很少有人喜欢 Paxos，即使是经验丰富的研究人员。 我们自己也跟 Paxos 进行了艰苦的斗争; 我们也无法完全理解整个协议，直到阅读了几个更简单的描述和自己设计替代 Paxos 的协议，整个过程花了将近一年。

Paxos 晦涩难懂的原因是作者选择了single-degree Paxos作为基础。Single-decree Paxos 分成两个阶段，这两个阶段没有简单直观的说明，并且不能被单独理解。因此，很难理解为什么该算法能起作用。Multi-Paxos 的合成规则又增加了许多复杂性。我们相信，对多个决定（日志而不是单个日志条目）达成一致的总体问题可以用其他更直接和更明显的方式进行分解。

Paxos的第二个问题是它不能为构建实际的实现提供良好的基础。 一个原因是没有针对 multi-Paxos 的广泛同意的算法。 Lamport的描述主要是关于 single-decree Paxos; 他描述了 multi-Paxos 的可能方法，但缺少许多细节。 已经有几个尝试来具体化和优化 Paxos，例如[26]，[39]和[13]，但这些彼此各不相同并且跟 Lamport 描述的也不同。 像Chubby [4] 这样的系统已经实现了类 Paxos（Paxos-like）算法，但大多数情况下，它们的细节并没有公布。

此外，Paxos 的架构对于构建实际系统来说是一个糟糕的设计，这是 single-decree 分解的另一个结果。 例如，独立地选择日志条目集合，然后再将它们合并到顺序日志中几乎没有任何好处，这只会增加复杂性。 围绕日志设计系统是更简单和有效的方法，新日志条目按照约束顺序地添加到日志中。 Paxos 的做法适用于只需要做一次决策的情况，如果需要做一系列决策，更简单和快速的方法是先选择一个 leader，然后让该 leader 协调这些决策。

因此，实际的系统跟 Paxos 相差很大。几乎所有的实现都是从 Paxos 开始，然后发现很多实现上的难题，接着就开发了一种和 Paxos 完全不一样的架构。这样既费时又容易出错，而且 Paxos 本身晦涩难懂使得该问题更加严重。Paxos 的公式可能可以很好地证明它的正确性，但是现实的系统和 Paxos 差别是如此之大，以至于这些证明并没有什么太大的价值。下面来自 Chubby 作者的评论非常典型：

> 在Paxos算法描述和实现现实系统之间有着巨大的鸿沟。最终的系统往往建立在一个还未被证明的协议之上。

由于以上问题，我们得出的结论是 Paxos 算法没有为系统实践和教学提供一个良好的基础。考虑到一致性问题在大规模软件系统中的重要性，我们决定尝试设计一个能够替代 Paxos 并且具有更好特性的一致性算法。Raft算法就是这次实验的结果。

4 为可理解性而设计

在设计 Raft 算法过程中我们有几个目标：它必须提供一个完整的实际的系统实现基础，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典型的应用条件下是可用的；并且在正常情况下是高效的。但是我们最重要的目标也是最大的挑战是可理解性。它必须保证能够被大多数人容易地理解。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现实中进行扩展。

在设计 Raft 算法的时候，很多情况下我们需要在多个备选方案中进行选择。在这种情况下，我们基于可理解性来评估备选方案：解释各个备选方案的难道有多大（例如，Raft 的状态空间有多复杂，是否有微妙的含义）？对于一个读者而言，完全理解这个方案和含义是否容易？

我们意识到这样的分析具有高度的主观性；但是我们使用了两种通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，可被解决的、可解释的和可理解的子问题。例如，Raft 算法被我们分成 leader 选举，日志复制，安全性和成员变更几个部分。

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 限制了使日志之间不一致的方式。尽管在大多数情况下我们都试图去消除不确定性，但是在某些情况下不确定性可以提高可理解性。特别是，随机化方法虽然引入了不确定性，但是他们往往能够通过使用相近的方法处理可能的选择来减少状态空间。我们使用随机化来简化 Raft 中的 leader 选举算法。

5 Raft 一致性算法

Raft 是一种用来管理第 2 节中描述的复制日志的算法。图 2 是该算法的浓缩，可用作参考，图 3 列举了该算法的一些关键特性。图中的这些内容将在剩下的章节中逐一介绍。

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).

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Raft 通过首先选举一个 distinguished leader，然后让它全权负责管理复制日志来实现一致性。Leader 从客户端接收日志条目，把日志条目复制到其他服务器上，并且在保证安全性的时候通知其他服务器将日志条目应用到他们的状态机中。拥有一个 leader 大大简化了对复制日志的管理。例如，leader 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，并且数据都是从 leader 流向其他服务器。leader 可能宕机，也可能和其他服务器断开连接，这时一个新的 leader 会被选举出来。

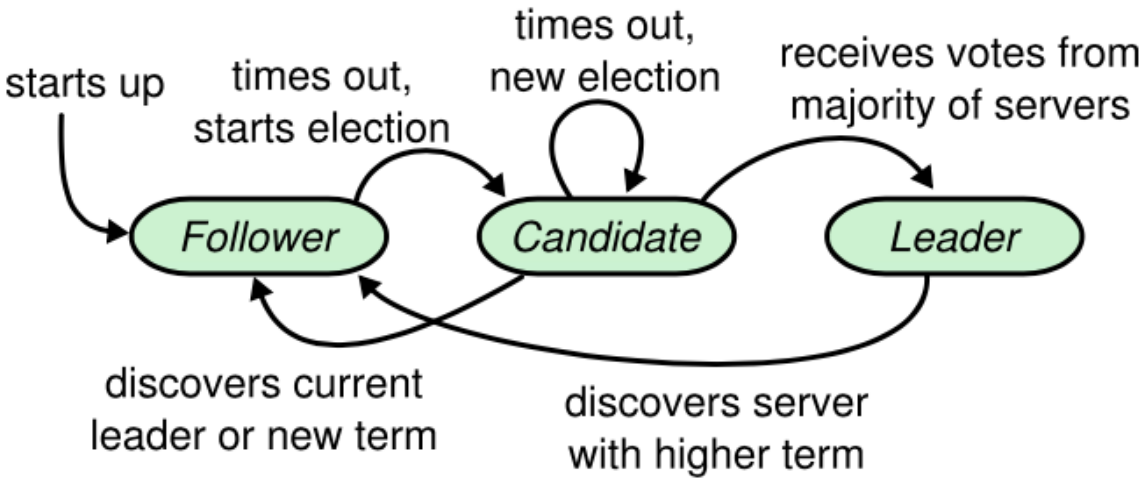
通过选举一个 leader 的方式，Raft 将一致性问题分解成了三个相对独立的子问题，这些问题将会在接下来的子章节中进行讨论：

- **Leader 选举**：当前的 leader 宕机时，一个新的 leader 必须被选举出来。（5.2 节）
- **日志复制**：Leader 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和自己的保持一致。
- **安全性**：Raft 中安全性的关键是图 3 中状态机的安全性：如果有任何的服务器节点已经应用了一个特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条不同的指令。章节 5.4 阐述了 Raft 算法是如何保证这个特性的；该解决方案在选举机制（5.2 节）上增加了额外的限制。

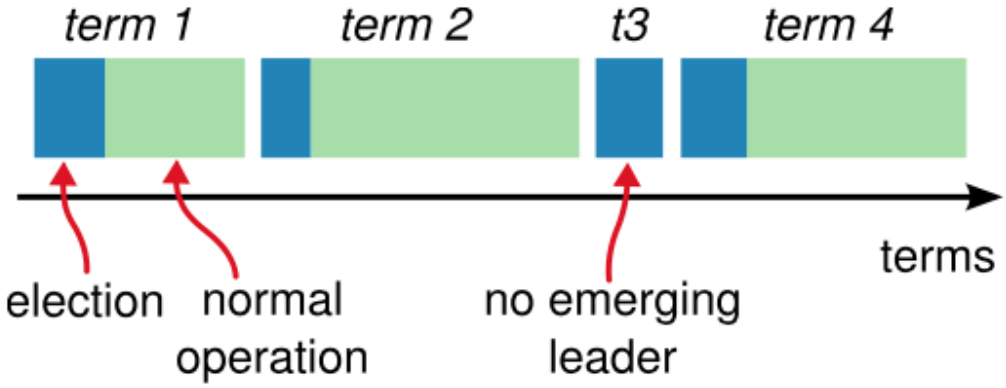
在展示一致性算法之后，本章节将讨论可用性的一些问题以及时序在系统中的作用。

5.1 Raft 基础

一个 Raft 集群包含若干个服务器节点；通常是 5 个，这样的系统可以容忍 2 个节点的失效。在任何时刻，每一个服务器节点都处于这三个状态之一：leader、follower 或者 candidate 。在正常情况下，集群中只有一个 leader 并且其他的节点全部都是 follower 。Follower 都是被动的：他们不会发送任何请求，只是简单的响应来自 leader 和 candidate 的请求。Leader 处理所有的客户端请求（如果一个客户端和 follower 通信，follower 会将请求重定向给 leader）。第三种状态，candidate ，是用来选举一个新的 leader（章节 5.2）。图 4 展示了这些状态和他们之间的转换关系；这些转换关系在接下来会进行讨论。



Raft 把时间分割成任意长度的任期（term），如图 5 所示。任期用连续的整数标记。每一段任期从一次选举开始，一个或者多个 candidate 尝试成为 leader 。如果一个 candidate 赢得选举，然后他就在该任期剩下的时间里充当 leader 。在某些情况下，一次选举无法选出 leader 。在这种情况下，这一任期会以没有 leader 结束；一个新的任期（包含一次新的选举）会很快重新开始。Raft 保证了在任意一个任期内，最多只有一个 leader 。



不同的服务器节点观察到的任期转换的次数可能不同，在某些情况下，一个服务器节点可能没有看到 leader 选举过程或者甚至整个任期全程。任期在 Raft 算法中充当逻辑时钟的作用，这使得服务器节点可以发现一些过期的信息比如过时的 leader 。每一个服务器节点存储一个当前任期号，该编号随着时间单调递增。服务器之间通信的时候会交换当前任期号；如果一个服务器的当前任期号比其他的小，该服务器会将自己的任期号更新为较大的那个值。如果一个 candidate 或者 leader 发现自己的任期号过期了，它会立即回到 follower 状态。如果一个节点接收到一个包含过期的任期号的请求，它会直接拒绝这个请求。

Raft 算法中服务器节点之间使用 RPC 进行通信，并且基本的一致性算法只需要两种类型的 RPC。请求投票（RequestVote） RPC 由 candidate 在选举期间发起（章节 5.2），追加条目（AppendEntries）RPC 由 leader 发起，用来复制日志和提供一种心跳机制（章节 5.3）。第 7 节为了在服务器之间传输快照增加了第三种 RPC。当服务器没有及时的收到 RPC 的响应时，会进行重试， 并且他们能够并行的发起 RPC 来获得最佳的性能。

5.2 Leader 选举

Raft 使用一种心跳机制来触发 leader 选举。当服务器程序启动时，他们都是 follower 。一个服务器节点只要能从 leader 或 candidate 处接收到有效的 RPC 就一直保持 follower 状态。Leader 周期性地向所有 follower 发送心跳（不包含日志条目的 AppendEntries RPC）来维持自己的地位。如果一个 follower 在一段选举超时时间内没有接收到任何消息，它就假设系统中没有可用的 leader ，然后开始进行选举以选出新的 leader 。

要开始一次选举过程，follower 先增加自己的当前任期号并且转换到 candidate 状态。然后投票给自己并且并行地向集群中的其他服务器节点发送 RequestVote RPC（让其他服务器节点投票给它）。Candidate 会一直保持当前状态直到以下三件事情之一发生：(a) 它自己赢得了这次的选举（收到过半的投票），(b) 其他的服务器节点成为 leader，(c) 一段时间之后没有任何获胜者。这些结果会在下面的章节里分别讨论。

当一个 candidate 获得集群中过半服务器节点针对同一个任期的投票，它就赢得了这次选举并成为 leader 。对于同一个任期，每个服务器节点只会投给一个 candidate，按照先来先服务（first-come-first-served）的原则（注意：5.4 节在投票上增加了额外的限制）。要求获得过半投票的规则确保了最多只有一个 candidate 赢得此次选举（图 3 中的选举安全性）。一旦 candidate 赢得选举，就立即成为 leader 。然后它会向其他的服务器节点发送心跳消息来确定自己的地位并阻止新的选举。

在等待投票期间，candidate 可能会收到另一个声称自己是 leader 的服务器节点发来的 AppendEntries RPC 。如果这个 leader 的任期号（包含在RPC中）不小于 candidate 当前的任期号，那么 candidate 会承认该 leader 的合法地位并回到 follower 状态。 如果 RPC 中的任期号比自己的小，那么 candidate 就会拒绝这次的 RPC 并且继续保持 candidate 状态。

第三种可能的结果是 candidate 既没有赢得选举也没有输：如果有多个 follower 同时成为 candidate，那么选票可能会被瓜分以至于没有 candidate 赢得过半的投票。当这种情况发生时，每一个候选人都会超时，然后通过增加当前任期号来开始一轮新的选举。然而，如果没有其他机制的话，该情况可能会无限重复。

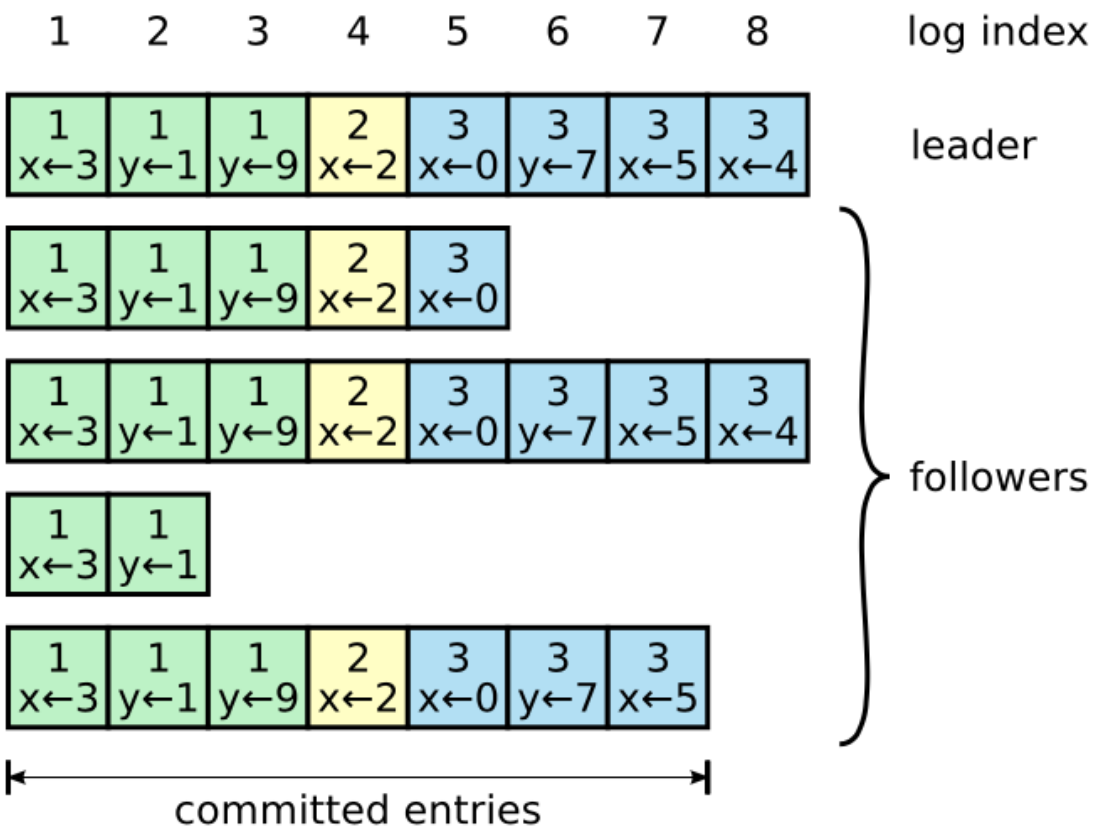
Raft 算法使用随机选举超时时间的方法来确保很少发生选票瓜分的情况，就算发生也能很快地解决。为了阻止选票一开始就被瓜分，选举超时时间是从一个固定的区间（例如 150-300 毫秒）随机选择。这样可以把服务器都分散开以至于在大多数情况下只有一个服务器会选举超时；然后该服务器赢得选举并在其他服务器超时之前发送心跳。同样的机制被用来解决选票被瓜分的情况。每个 candidate 在开始一次选举的时候会重置一个随机的选举超时时间，然后一直等待直到选举超时；这样减小了在新的选举中再次发生选票瓜分情况的可能性。9.3 节展示了该方案能够快速地选出一个 leader 。

选举的例子可以很好地展示可理解性是如何指导我们选择设计方案的。起初我们打算使用一种等级系统（ranking system）：每一个 candidate 都被赋予一个唯一的等级（rank），等级用来在竞争的 candidate 之间进行选择。如果一个 candidate 发现另一个 candidate 拥有更高的等级，它就会回到 follower 状态，这样高等级的 candidate 能够更加容易地赢得下一次选举。但是我们发现这种方法在可用性方面会有一下小问题。我们对该算法进行了多次调整，但是每次调整之后都会有新的小问题。最终我们认为随机重试的方法更加显然且易于理解。

5.3 日志复制

Leader 一旦被选举出来，就开始为客户端请求提供服务。客户端的每一个请求都包含一条将被复制状态机执行的指令。Leader 把该指令作为一个新的条目追加到日志中去，然后并行的发起 AppendEntries RPC 给其他的服务器，让它们复制该条目。当该条目被安全地复制（下面会介绍），leader 会应用该条目到它的状态机中（状态机执行该指令）然后把执行的结果返回给客户端。如果 follower 崩溃或者运行缓慢，或者网络丢包，leader 会不断地重试 AppendEntries RPC（即使已经回复了客户端）直到所有的 follower 最终都存储了所有的日志条目。

日志以图 6 展示的方式组织。每个**日志条目**存储一条**状态机指令**和 leader 收到该指令时的**任期号**。任期号用来检测多个日志副本之间的不一致情况，同时也用来保证图 3 中的某些性质。每个日志条目都有一个整数索引值来表明它在日志中的位置。



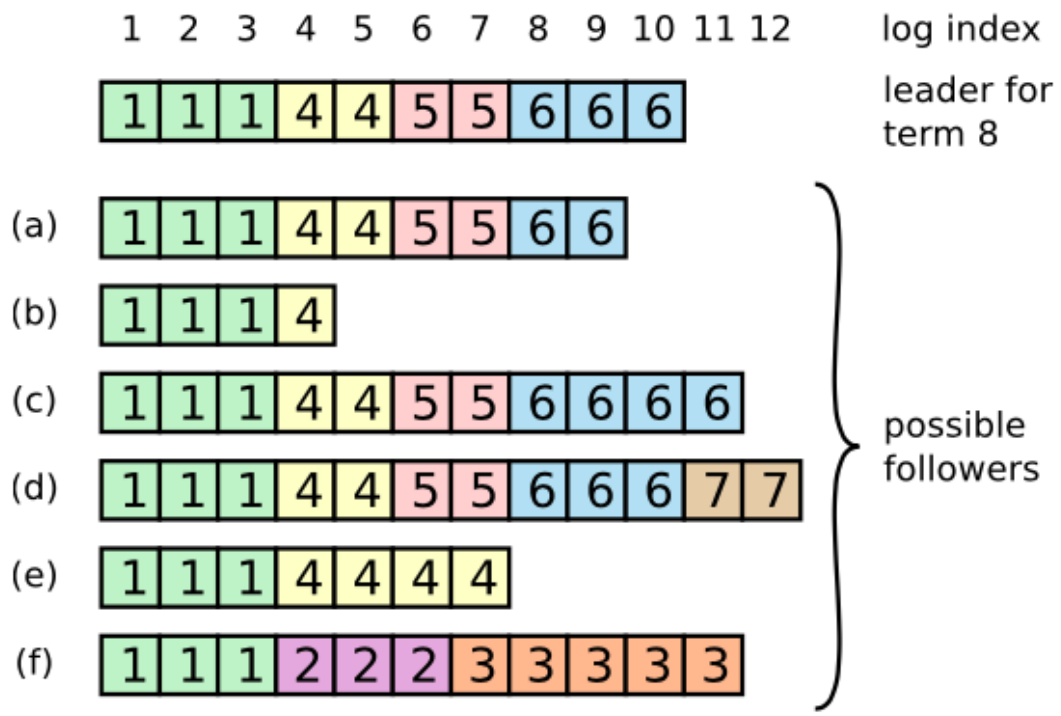
Leader 决定什么时候把日志条目应用到状态机中是安全的；这种日志条目被称为**已提交的**。Raft 算法保证所有已提交的日志条目都是持久化的并且最终会被所有可用的状态机执行。一旦创建该日志条目的 leader 将它复制到过半的服务器上，该日志条目就会被提交（例如在图 6 中的条目 7）。同时，leader 日志中该日志条目之前的所有日志条目也都会被提交，包括由其他 leader 创建的条目。5.4 节讨论在 leader 变更之后应用该规则的一些细节，并且证明了这种提交的规则是安全的。Leader 追踪将会被提交的日志条目的最大索引，未来的所有 AppendEntries RPC 都会包含该索引，这样其他的服务器才能最终知道哪些日志条目需要被提交。Follower 一旦知道某个日志条目已经被提交就会将该日志条目应用到自己的本地状态机中（按照日志的顺序）。

我们设计了 Raft 日志机制来维持不同服务器之间日志高层次的一致性。这么做不仅简化了系统的行为也使得系统行为更加可预测，同时该机制也是保证安全性的重要组成部分。Raft 维护着以下特性，这些同时也构成了图 3 中的日志匹配特性：

- 如果不同日志中的两个条目拥有相同的索引和任期号，那么他们存储了相同的指令。
- 如果不同日志中的两个条目拥有相同的索引和任期号，那么他们之前的所有日志条目也都相同。

Leader 在特定的任期号内的一个日志索引处最多创建一个日志条目，同时日志条目在日志中的位置也从来不会改变。该点保证了上面的第一条特性。第二个特性是由 AppendEntries RPC 执行一个简单的一致性检查所保证的。在发送 AppendEntries RPC 的时候，leader 会将前一个日志条目的索引位置和任期号包含在里面。如果 follower 在它的日志中找不到包含相同索引位置和任期号的条目，那么他就会拒绝该新的日志条目。一致性检查就像一个归纳步骤：一开始空的日志状态肯定是满足 Log Matching Property（日志匹配特性） 的，然后一致性检查保证了日志扩展时的日志匹配特性。因此，每当 AppendEntries RPC 返回成功时，leader 就知道 follower 的日志一定和自己相同（从第一个日志条目到最新条目）。

正常操作期间，leader 和 follower 的日志保持一致，所以 AppendEntries RPC 的一致性检查从来不会失败。然而，leader 崩溃的情况会使日志处于不一致的状态（老的 leader 可能还没有完全复制它日志里的所有条目）。这种不一致会在一系列的 leader 和 follower 崩溃的情况下加剧。图 7 展示了在什么情况下 follower 的日志可能和新的 leader 的日志不同。Follower 可能缺少一些在新 leader 中有的日志条目，也可能拥有一些新 leader 没有的日志条目，或者同时发生。缺失或多出日志条目的情况可能会涉及到多个任期。



> 图 7：当一个 leader 成功当选时（最上面那条日志），follower 可能是（a-f）中的任何情况。每一个盒子表示一个日志条目；里面的数字表示任期号。Follower 可能会缺少一些日志条目（a-b），可能会有一些未被提交的日志条目（c-d），或者两种情况都存在（e-f）。例如，场景 f 可能这样发生，f 对应的服务器在任期 2 的时候是 leader，追加了一些日志条目到自己的日志中，一条都还没提交（commit）就崩溃了；该服务器很快重启，在任期 3 重新被选为 leader，又追加了一些日志条目到自己的日志中；在这些任期 2 和任期 3 中的日志都还没被提交之前，该服务器又宕机了，并且在接下来的几个任期里一直处于宕机状态。

在 Raft 算法中，leader 通过强制 follower 复制它的日志来解决不一致的问题。这意味着 follower 中跟 leader 冲突的日志条目会被 leader 的日志条目覆盖。5.4 节会证明通过增加一个限制可以保证安全性。

要使得 follower 的日志跟自己一致，leader 必须找到两者达成一致的最大的日志条目（索引最大），删除 follower 日志中从那个点之后的所有日志条目，并且将自己从那个点之后的所有日志条目发送给 follower 。所有的这些操作都发生在对 AppendEntries RPCs 中一致性检查的回复中。Leader 针对每一个 follower 都维护了一个 nextIndex，表示 leader 要发送给 follower 的下一个日志条目的索引。当选出一个新 leader 时，该 leader 将所有 nextIndex 的值都初始化为自己最后一个日志条目的 index 加1（图 7 中的 11）。如果 follower 的日志和 leader 的不一致，那么下一次 AppendEntries RPC 中的一致性检查就会失败。在被 follower 拒绝之后，leaer 就会减小 nextIndex 值并重试 AppendEntries RPC 。最终 nextIndex 会在某个位置使得 leader 和 follower 的日志达成一致。此时，AppendEntries RPC 就会成功，将 follower 中跟 leader 冲突的日志条目全部删除然后追加 leader 中的日志条目（如果有需要追加的日志条目的话）。一旦 AppendEntries RPC 成功，follower 的日志就和 leader 一致，并且在该任期接下来的时间里保持一致。

> 如果想要的话，该协议可以被优化来减少被拒绝的 AppendEntries RPC 的个数。例如，当拒绝一个 AppendEntries RPC 的请求的时候，follower 可以包含冲突条目的任期号和自己存储的那个任期的第一个 index 。借助这些信息，leader 可以跳过那个任期内所有冲突的日志条目来减小 nextIndex；这样就变成每个有冲突日志条目的任期需要一个 AppendEntries RPC 而不是每个条目一次。在实践中，我们认为这种优化是没有必要的，因为失败不经常发生并且也不可能有很多不一致的日志条目。

通过这种机制，leader 在当权之后就不需要任何特殊的操作来使日志恢复到一致状态。Leader 只需要进行正常的操作，然后日志就能在回复 AppendEntries 一致性检查失败的时候自动趋于一致。Leader 从来不会覆盖或者删除自己的日志条目（图 3 的 Leader Append-Only 属性）。

这样的日志复制机制展示了第 2 节中描述的一致性特性：只要过半的服务器能正常运行，Raft 就能够接受，复制并应用新的日志条目；在正常情况下，新的日志条目可以在一个 RPC 来回中被复制给集群中的过半机器；并且单个运行慢的 follower 不会影响整体的性能。

5.4 安全性

前面的章节里描述了 Raft 算法是如何进行 leader 选举和日志复制的。然而，到目前为止描述的机制并不能充分地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 follower 可能会进入不可用状态，在此期间，leader 可能提交了若干的日志条目，然后这个 follower 可能会被选举为 leader 并且用新的日志条目覆盖这些日志条目；结果，不同的状态机可能会执行不同的指令序列。

这节通过对 leader 选举增加一个限制来完善 Raft 算法。这一限制保证了对于给定的任意任期号，leader 都包含了之前各个任期所有被提交的日志条目（图 3 中的 Leader Completeness 性质）。有了这一 leader 选举的限制，我们也使得提交规则更加清晰。最后，我们展示了对于 Leader Completeness 性质的简要证明并且说明该性质是如何领导复制状态机执行正确的行为的。

5.4.1 选举限制

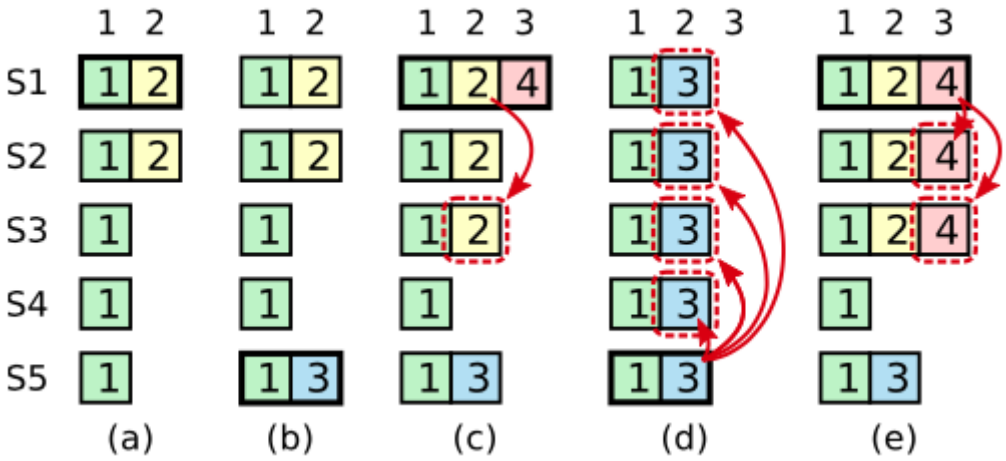
在任何基于 leader 的一致性算法中，leader 最终都必须存储所有已经提交的日志条目。在某些一致性算法中，例如 Viewstamped Replication[22]，一开始并没有包含所有已经提交的日志条目的服务器也可能被选为 leader 。这种算法包含一些额外的机制来识别丢失的日志条目并将它们传送给新的 leader ，要么是在选举阶段要么在之后很快进行。不幸的是，这种方法会导致相当大的额外的机制和复杂性。Raft 使用了一种更加简单的方法，它可以保证新 leader 在当选时就包含了之前所有任期号中已经提交的日志条目，不需要再传送这些日志条目给新 leader 。这意味着日志条目的传送是单向的，只从 leader 到 follower，并且 leader 从不会覆盖本地日志中已经存在的条目。

Raft 使用投票的方式来阻止 candidate 赢得选举除非该 candidate 包含了所有已经提交的日志条目。候选人为了赢得选举必须与集群中的过半节点通信，这意味着至少其中一个服务器节点包含了所有已提交的日志条目。如果 candidate 的日志至少和过半的服务器节点一样新（接下来会精确地定义“新”），那么他一定包含了所有已经提交的日志条目。RequestVote RPC 执行了这样的限制： RPC 中包含了 candidate 的日志信息，如果投票者自己的日志比 candidate 的还新，它会拒绝掉该投票请求。

Raft 通过比较两份日志中最后一条日志条目的索引值和任期号来定义谁的日志比较新。如果两份日志最后条目的任期号不同，那么任期号大的日志更新。如果两份日志最后条目的任期号相同，那么日志较长的那个更新。

5.4.2 提交之前任期内的日志条目

如同 5.3 节描述的那样，一旦当前任期内的某个日志条目已经存储到过半的服务器节点上，leader 就知道该日志条目已经被提交了。如果某个 leader 在提交某个日志条目之前崩溃了，以后的 leader 会试图完成该日志条目的复制。然而，如果是之前任期内的某个日志条目已经存储到过半的服务器节点上，leader 也无法立即断定该日志条目已经被提交了。图 8 展示了一种情况，一个已经被存储到过半节点上的老日志条目，仍然有可能会被未来的 leader 覆盖掉。



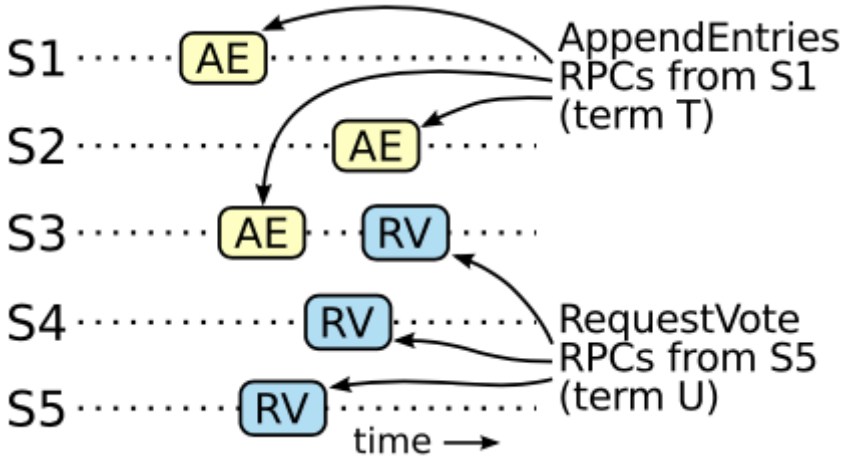
> 图 8：如图的时间序列展示了为什么 leader 无法判断老的任期号内的日志是否已经被提交。在 (a) 中，S1 是 leader，部分地复制了索引位置 2 的日志条目。在 (b) 中，S1 崩溃了，然后 S5 在任期 3 中通过 S3、S4 和自己的选票赢得选举，然后从客户端接收了一条不一样的日志条目放在了索引 2 处。然后到 (c)，S5 又崩溃了；S1 重新启动，选举成功，继续复制日志。此时，来自任期 2 的那条日志已经被复制到了集群中的大多数机器上，但是还没有被提交。如果 S1 在 (d) 中又崩溃了，S5 可以重新被选举成功（通过来自 S2，S3 和 S4 的选票），然后覆盖了他们在索引 2 处的日志。但是，在崩溃之前，如果 S1 在自己的任期里复制了日志条目到大多数机器上，如 (e) 中，然后这个条目就会被提交（S5 就不可能选举成功）。 在这种情况下，之前的所有日志也被提交了。

为了消除图 8 中描述的问题，Raft 永远不会通过计算副本数目的方式来提交之前任期内的日志条目。只有 leader 当前任期内的日志条目才通过计算副本数目的方式来提交；一旦当前任期的某个日志条目以这种方式被提交，那么由于日志匹配特性，之前的所有日志条目也都会被间接地提交。在某些情况下，领导人可以安全地断定一个老的日志条目已经被提交（例如，如果该条目已经存储到所有服务器上），但是 Raft 为了简化问题使用了一种更加保守的方法。

Raft 会在提交规则上增加额外的复杂性是因为当 leader 复制之前任期内的日志条目时，这些日志条目都保留原来的任期号。在其他的一致性算法中，如果一个新的 leader 要重新复制之前的任期里的日志时，它必须使用当前新的任期号。Raft 的做法使得更加容易推导出（reason about）日志条目，因为他们自始至终都使用同一个任期号。另外，和其他的算法相比，Raft 中的新 leader 只需要发送更少的日志条目（其他算法中必须在它们被提交之前发送更多的冗余日志条目来给它们重新编号）。

5.4.3 安全性论证

在给出了完整的 Raft 算法之后，我们现在可以更加精确的讨论 leader 完整性特性（Leader Completeness Prop-erty）（这一讨论基于 9.2 节的安全性证明）。我们假设 leader 完整性特性是不满足的，然后我们推出矛盾来。假设任期 T 的 leader（leader T）在任期内提交了一个日志条目，但是该日志条目没有被存储到未来某些任期的 leader 中。假设 U 是大于 T 的没有存储该日志条目的最小任期号。



> 图 9：如果 S1 （任期 T 的 leader）在它的任期里提交了一个新的日志条目，然后 S5 在之后的任期 U 里被选举为 leader ，那么肯定至少会有一个节点，如 S3，既接收了来自 S1 的日志条目，也给 S5 投票了。

1. U 一定在刚成为 leader 的时候就没有那条被提交的日志条目了（leader 从不会删除或者覆盖任何条目）。
2. Leader T 复制该日志条目给集群中的过半节点，同时，leader U 从集群中的过半节点赢得了选票。因此，至少有一个节点（投票者）同时接受了来自 leader T 的日志条目和给 leader U 投票了，如图 9。该投票者是产生矛盾的关键。
3. 该投票者必须在给 leader U 投票之前先接受了从 leader T 发来的已经被提交的日志条目；否则它就会拒绝来自 leader T 的 AppendEntries 请求（因为此时它的任期号会比 T 大）。
4. 该投票者在给 leader U 投票时依然保有这该日志条目，因为任何 U 、T 之间的 leader 都包含该日志条目（根据上述的假设），leader 从不会删除条目，并且 follower 只有跟 leader 冲突的时候才会删除条目。
5. 该投票者把自己选票投给 leader U 时，leader U 的日志必须至少和投票者的一样新。这就导致了以下两个矛盾之一。
6. 首先，如果该投票者和 leader U 的最后一个日志条目的任期号相同，那么 leader U 的日志至少和该投票者的一样长，所以 leader U 的日志一定包含该投票者日志中的所有日志条目。这是一个矛盾，因为该投票者包含了该已被提交的日志条目，但是在上述的假设里，leader U 是不包含的。
7. 否则，leader U 的最后一个日志条目的任期号就必须比该投票者的大了。此外，该任期号也比 T 大，因为该投票者的最后一个日志条目的任期号至少和 T 一样大（它包含了来自任期 T 的已提交的日志）。创建了 leader U 最后一个日志条目的之前的 leader 一定已经包含了该已被提交的日志条目（根据上述假设，leader U 是第一个不包含该日志条目的 leader）。所以，根据日志匹配特性，leader U 一定也包含该已被提交的日志条目，这里产生了矛盾。
8. 因此，所有比 T 大的任期的 leader 一定都包含了任期 T 中提交的所有日志条目。
9. 日志匹配特性保证了未来的 leader 也会包含被间接提交的日志条目，例如图 8 (d) 中的索引 2。

通过 Leader 完整性特性，我们就能证明图 3 中的状态机安全特性，即如果某个服务器已经将某个给定的索引处的日志条目应用到自己的状态机里了，那么其他的服务器就不会在相同的索引处应用一个不同的日志条目。在一个服务器应用一个日志条目到自己的状态机中时，它的日志和 leader 的日志从开始到该日志条目都相同，并且该日志条目必须被提交。现在考虑如下最小任期号：某服务器在该任期号中某个特定的索引处应用了一个日志条目；日志完整性特性保证拥有更高任期号的 leader 会存储相同的日志条目，所以之后任期里服务器应用该索引处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 要求服务器按照日志索引顺序应用日志条目。再加上状态机安全特性，这就意味着所有的服务器都会按照相同的顺序应用相同的日志条目到自己的状态机中。

5.5 Follower 和 candidate 崩溃

到目前为止，我们只关注了 leader 崩溃的情况。Follower 和 candidate 崩溃后的处理方式比 leader 崩溃要简单的多，并且两者的处理方式是相同的。如果 follower 或者 candidate 崩溃了，那么后续发送给他们的 RequestVote 和 AppendEntries RPCs 都会失败。Raft 通过无限的重试来处理这种失败；如果崩溃的机器重启了，那么这些 RPC 就会成功地完成。如果一个服务器在完成了一个 RPC，但是还没有响应的时候崩溃了，那么在它重启之后就会再次收到同样的请求。Raft 的 RPCs 都是幂等的，所以这样的重试不会造成任何伤害。例如，一个 follower 如果收到 AppendEntries 请求但是它的日志中已经包含了这些日志条目，它就会直接忽略这个新的请求中的这些日志条目。

5.6 定时（timing）和可用性

Raft 的要求之一就是安全性不能依赖定时：整个系统不能因为某些事件运行得比预期快一点或者慢一点就产生错误的结果。但是，可用性（系统能够及时响应客户端）不可避免的要依赖于定时。例如，当有服务器崩溃时，消息交换的时间就会比正常情况下长，candidate 将不会等待太长的时间来赢得选举；没有一个稳定的 leader ，Raft 将无法工作。

Leader 选举是 Raft 中定时最为关键的方面。 只要整个系统满足下面的时间要求，Raft 就可以选举出并维持一个稳定的 leader：

> 广播时间（broadcastTime） << 选举超时时间（electionTimeout） << 平均故障间隔时间（MTBF）

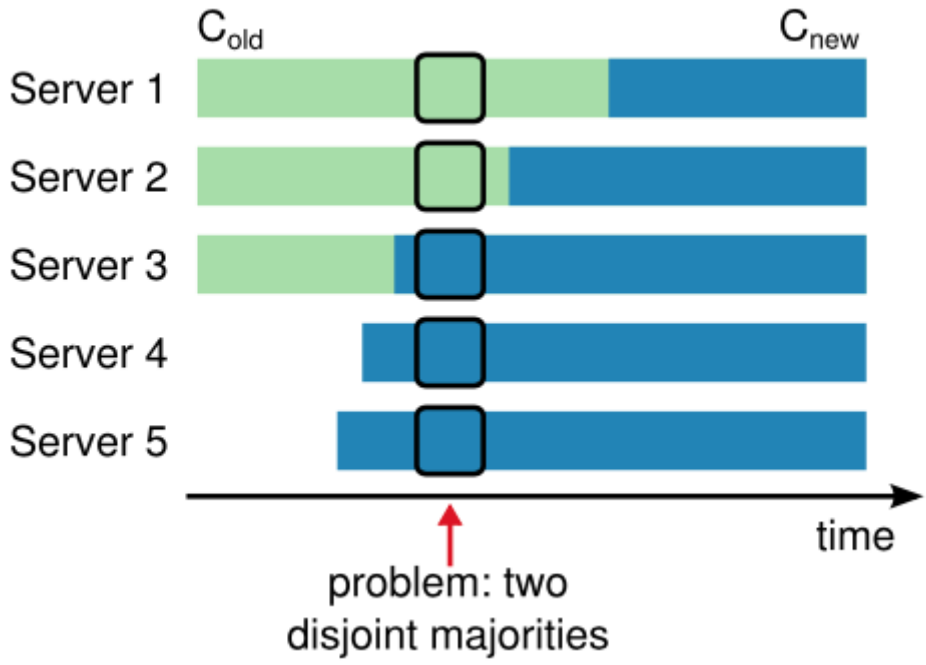
在这个不等式中，广播时间指的是一个服务器并行地发送 RPCs 给集群中所有的其他服务器并接收到响应的平均时间；选举超时时间就是在 5.2 节中介绍的选举超时时间；平均故障间隔时间就是对于一台服务器而言，两次故障间隔时间的平均值。广播时间必须比选举超时时间小一个量级，这样 leader 才能够可靠地发送心跳消息来阻止 follower 开始进入选举状态；再加上随机化选举超时时间的方法，这个不等式也使得选票瓜分的情况变得不可能。选举超时时间需要比平均故障间隔时间小上几个数量级，这样整个系统才能稳定地运行。当 leader 崩溃后，整个系统会有大约选举超时时间不可用；我们希望该情况在整个时间里只占一小部分。

广播时间和平均故障间隔时间是由系统决定的，但是选举超时时间是我们自己选择的。Raft 的 RPCs 需要接收方将信息持久化地保存到稳定存储中去，所以广播时间大约是 0.5 毫秒到 20 毫秒之间，取决于存储的技术。因此，选举超时时间可能需要在 10 毫秒到 500 毫秒之间。大多数的服务器的平均故障间隔时间都在几个月甚至更长，很容易满足时间的要求。

6 集群成员变更

到目前为止，我们都假设集群的配置（参与一致性算法的服务器集合）是固定不变的。但是在实践中，偶尔会改变集群的配置的，例如替换那些宕机的机器或者改变复制程度。尽管可以通过使整个集群下线，更新所有配置，然后重启整个集群的方式来实现，但是在更改期间集群会不可用。另外，如果存在手工操作步骤，那么就会有操作失误的风险。为了避免这样的问题，我们决定将配置变更自动化并将其纳入到 Raft 一致性算法中来。

为了使配置变更机制能够安全，在转换的过程中不能够存在任何时间点使得同一个任期里可能选出两个 leader 。不幸的是，任何服务器直接从旧的配置转换到新的配置的方案都是不安全的。一次性自动地转换所有服务器是不可能的，所以在转换期间整个集群可能划分成两个独立的大多数（见图 10）。



> 图 10：直接从一种配置转到另一种配置是不安全的，因为各个机器会在不同的时候进行转换。在这个例子中，集群从 3 台机器变成了 5 台。不幸的是，存在这样的一个时间点，同一个任期里两个不同的 leader 会被选出。一个获得旧配置里过半机器的投票，一个获得新配置里过半机器的投票。

为了保证安全性，配置变更必须采用一种两阶段方法。目前有很多种两阶段的实现。例如，有些系统（比如，[22]）在第一阶段停掉旧的配置所以不能处理客户端请求；然后在第二阶段在启用新的配置。在 Raft 中，集群先切换到一个过渡的配置，我们称之为联合一致（joint consensus）；一旦联合一致已经被提交了，那么系统就切换到新的配置上。联合一致结合了老配置和新配置：

- 日志条目被复制给集群中新、老配置的所有服务器。
- 新、旧配置的服务器都可以成为 leader 。
- 达成一致（针对选举和提交）需要分别在两种配置上获得过半的支持。

联合一致允许独立的服务器在不妥协安全性的前提下，在不同的时刻进行配置转换过程。此外，联合一致允许集群在配置变更期间依然响应客户端请求。

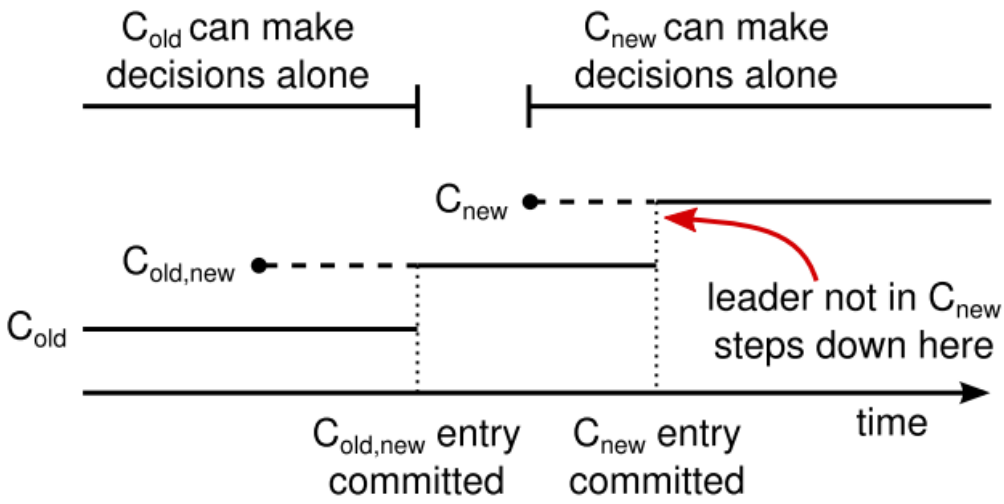


Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a majority of C_{old} and a majority of C_{new}). Then it creates the C_{new} entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.

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 展示了配置变更过程。当一个 leader 接收到一个改变配置从 C-old 到 C-new 的请求，它就为联合一致将该配置（图中的 C-old,new）存储为一个日志条目，并以前面描述的方式复制该条目。一旦某个服务器将该新配置日志条目增加到自己的日志中，它就会用该配置来做出未来所有的决策（服务器总是使用它日志中最新的配置，无论该配置日志是否已经被提交）。这就意味着 leader 会使用 C-old,new 的规则来决定 C-old,new 的日志条目是什么时候被提交的。如果 leader 崩溃了，新 leader 可能是在 C-old 配置也可能是在 C-old,new 配置下选出来的，这取决于赢得选举的 candidate 是否已经接收到了 C-old,new 配置。在任何情况下， C-new 在这一时期都不能做出单方面决定。

一旦 C-old,new 被提交，那么 C-old 和 C-new 都不能在没有得到对方认可的情况下做出决定，并且 leader 完整性特性保证了只有拥有 C-old,new 日志条目的服务器才能被选举为 leader 。现在 leader 创建一个描述 C-new 配置的日志条目并复制到集群其他节点就是安全的了。此外，新的配置被服务器收到后就会立即生效。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，同时不使用新配置的服务器就可以被关闭了。如图 11 所示，任何时刻 C-old 和 C-new 都不能单方面做出决定；这保证了安全性。

在关于配置变更还有三个问题需要解决。第一个问题是，新的服务器开始时可能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，它们需要一段时间来更新来赶上其他服务器，这段它们无法提交新的日志条目。为了避免因此而造成的系统短时间的不可用，Raft 在配置变更前引入了一个额外的阶段，在该阶段，新的服务器以没有投票权身份加入到集群中来（leader 也复制日志给它们，但是考虑过半的时候不用考虑它们）。一旦该新的服务器追赶上了集群中的其他机器，配置变更就可以按上面描述的方式进行。

第二个问题是，集群的 leader 可能不是新配置中的一员。在这种情况下，leader 一旦提交了 C-new 日志条目就会退位（回到 follower 状态）。这意味着有这样的一段时间（leader 提交 C-new 期间），leader 管理着一个不包括自己的集群；它复制着日志但不把自己算在过半里面。Leader 转换发生在 C-new 被提交的时候，因为这是新配置可以独立运转的最早时刻（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，那些被移除的服务器（不在 C-new 中）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，它们就会进行新的选举过程。它们会发送带有新任期号的 RequestVote RPCs，这样会导致当前的 leader 回到 follower 状态。新的 leader 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，导致系统可用性很差。

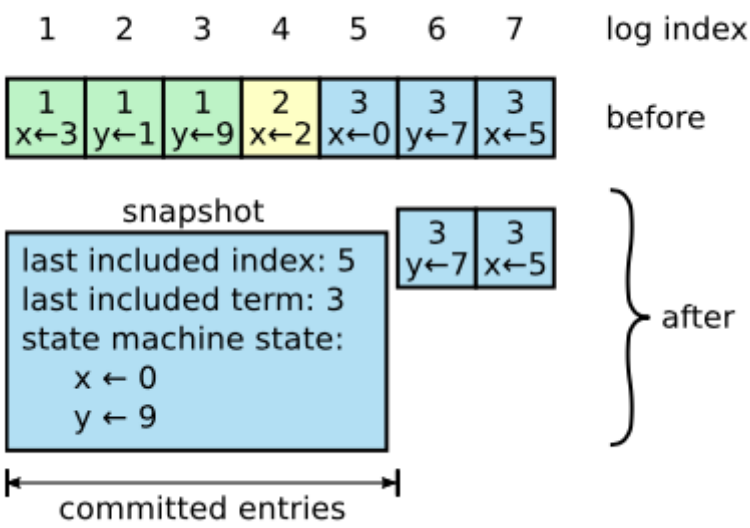
为了防止这种问题，当服务器认为当前 leader 存在时，服务器会忽略RequestVote RPCs 。特别的，当服务器在最小选举超时时间内收到一个 RequestVote RPC，它不会更新任期号或者投票。这不会影响正常的选举，每个服务器在开始一次选举之前，至少等待最小选举超时时间。相反，这有利于避免被移除的服务器的扰乱：如果 leader 能够发送心跳给集群，那它就不会被更大的任期号废黜。

7 日志压缩

Raft 的日志在正常操作中随着包含更多的客户端请求不断地增长，但是在实际的系统中，日志不能无限制地增长。随着日志越来越长，它会占用越来越多的空间，并且需要花更多的时间来回放。如果没有一定的机制来清除日志中积累的过期的信息，最终就会带来可用性问题。

快照技术是日志压缩最简单的方法。在快照技术中，整个当前系统的状态都以快照的形式持久化到稳定的存储中，该时间点之前的日志全部丢弃。快照技术被使用在 Chubby 和 ZooKeeper 中，接下来的章节会介绍 Raft 中的快照技术。

增量压缩方法，例如日志清理或者日志结构合并树（log-structured merge trees，LSM 树），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，它们先选择一个积累了大量已经被删除或者被覆盖的对象的数据区域，然后重写该区域还活着的对象，之后释放该区域。和快照技术相比，它们需要大量额外的机制和复杂性，快照技术通过操作整个数据集来简化该问题。状态机可以用和快照技术相同的接口来实现 LSM 树，但是日志清除方法就需要修改 Raft 了。

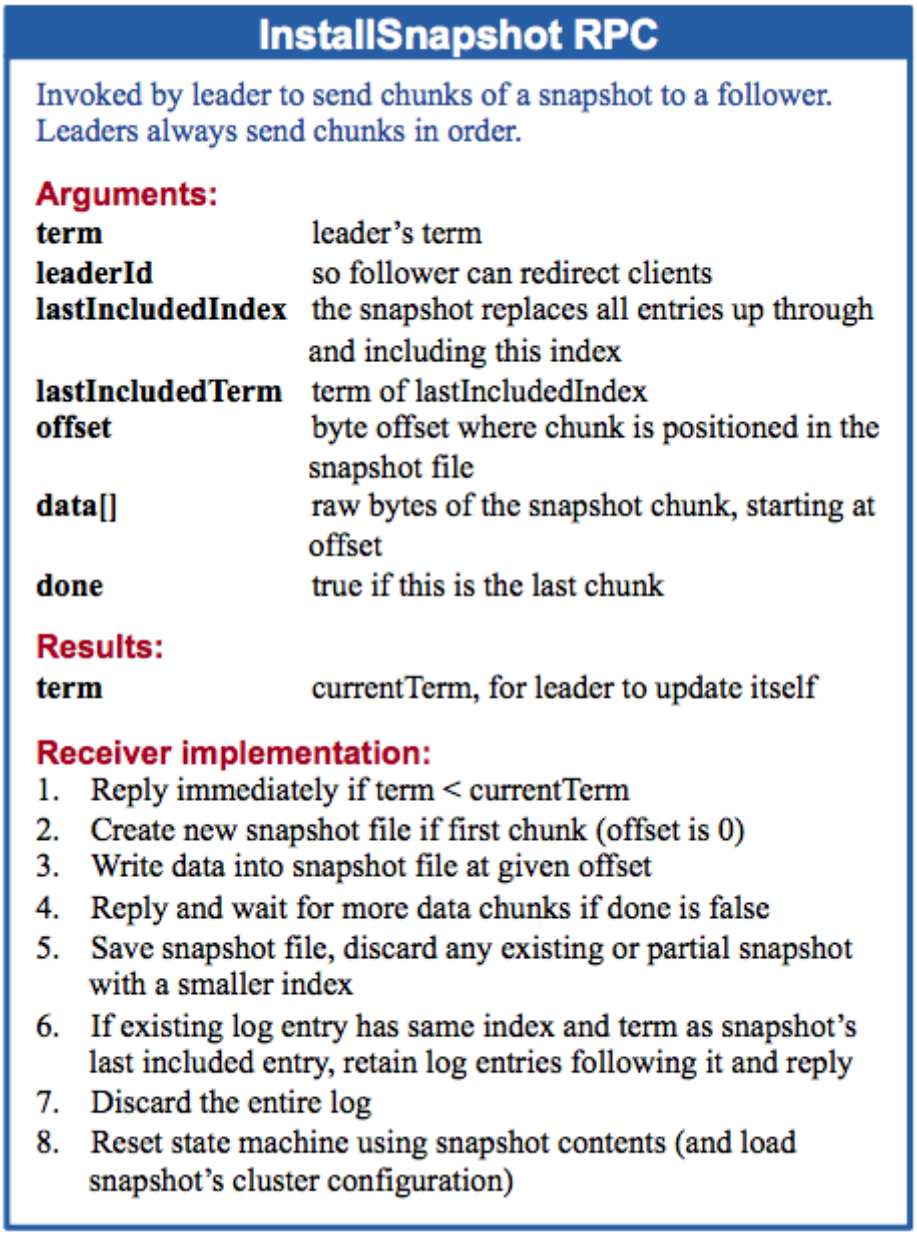


> 一台服务器用一个新快照替代了它日志中已经提交了的条目（索引 1 到 5），该快照只存储了当前的状态（变量 x 和 y 的值）。快照的 last included index 和 last included term 被保存来定位日志中条目 6 之前的快照

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，快照只包括自己日志中已经被提交的条目。主要的工作是状态机将自己的状态写入快照中。Raft 快照中也包含了少量的元数据：the last included index 指的是最后一个被快照取代的日志条目的索引值（状态机最后应用的日志条目），the last included term 是该条目的任期号。保留这些元数据是为了支持快照后第一个条目的 AppendEntries 一致性检查，因为该条目需要之前的索引值和任期号。为了支持集群成员变更（第 6 节），快照中也包括日志中最新的配置作为 last included index 。一旦服务器完成写快照，他就可以删除 last included index 之前的所有日志条目，包括之前的快照。

尽管通常服务器都是独立地创建快照，但是 leader 必须偶尔发送快照给一些落后的跟随者。这通常发生在 leader 已经丢弃了需要发送给 follower 的下一条日志条目的时候。幸运的是这种情况在常规操作中是不可能的：一个与 leader 保持同步的 follower 通常都会有该日志条目。然而一个例外的运行缓慢的 follower 或者新加入集群的服务器（第 6 节）将不会有这个条目。这时让该 follower 更新到最新的状态的方式就是通过网络把快照发送给它。

Leader 使用 InstallSnapshot RPC 来发送快照给太落后的 follower ； 见图 13。当 follower 收到带有这种 RPC 的快照时，它必须决定如何处理已经存在的日志条目。通常该快照会包含接收者日志中没有的信息。在这种情况下，follower 丢弃它所有的日志；这些会被该快照所取代，并且可能一些没有提交的条目会和该快照产生冲突。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，但是快照之后的条目仍然有用并保留。



这种快照的方式违反了 Raft 的 strong leader 原则，因为 follower 可以在不知道 leader 状态的情况下创建快照。但是我们认为这种违背是合乎情理的。Leader 的存在，是为了防止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，因此没有决策会冲突。数据依然只能从 leader 流到 follower ，只是 follower 可以重新组织它们的数据了。

我们考虑过一种可替代的基于 leader 的快照方案，在该方案中，只有leader 会创建快照，然后 leader 会发送它的快照给所有的 follower 。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过程。每个 follower 都已经拥有了创建自己的快照所需要的信息，而且很显然，follower 从本地的状态中创建快照远比通过网络接收别人发来的要来得经济。第二，leader 的实现会更加复杂。例如，leader 发送快照给 follower 的同时也要并行地将新的日志条目发送给它们，这样才不会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，服务器必须决定什么时候创建快照。如果快照创建过于频繁，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，就要承担耗尽存储容量的风险，同时也增加了重启时日志回放的时间。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。如果这个阈值设置得显著大于期望的快照的大小，那么快照的磁盘带宽负载就会很小。

第二个性能问题就是写入快照需要花费一段时间，并且我们不希望它影响到正常的操作。解决方案是通过写时复制的技术，这样新的更新就可以在不影响正在写的快照的情况下被接收。例如，具有泛函数据结构的状态机天然支持这样的功能。另外，操作系统对写时复制技术的支持（如 Linux 上的 fork）可以被用来创建整个状态机的内存快照（我们的实现用的就是这种方法）。

8 客户端交互

本节介绍客户端如何和 Raft 进行交互，包括客户端如何找到 leader 和 Raft 是如何支持线性化语义的。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 的客户端发送所有的请求给 leader 。当客户端第一次启动的时候，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 leader ，那么该服务器会拒绝客户端的请求并且提供关于它最近接收到的领导人的信息（AppendEntries 请求包含了 leader 的网络地址）。如果 leader 已经崩溃了，客户端请求就会超时；客户端之后会再次随机挑选服务器进行重试。

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，在它的调用和回复之间）。但是，如上述，Raft 可能执行同一条命令多次：例如，如果 leader 在提交了该日志条目之后，响应客户端之前崩溃了，那么客户端会和新的 leader 重试这条指令，导致这条命令被再次执行。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，状态机跟踪每个客户端已经处理的最新的序列号以及相关联的回复。如果接收到一条指令，该指令的序列号已经被执行过了，就立即返回结果，而不重新执行该请求。

只读的操作可以直接处理而不需要记录日志。但是，如果不采取任何其他措施，这么做可能会有返回过时数据（stale data）的风险，因为 leader 响应客户端请求时可能已经被新的 leader 替代了，但是它还不知道自己已经不是最新的 leader 了。线性化的读操作肯定不会返回过时数据，Raft 需要使用两个额外的预防措施来在不使用日志的情况下保证这一点。首先，leader 必须有关于哪些日志条目被提交了的最新信息。Leader 完整性特性保证了 leader 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个日志条目。Raft 通过让 leader 在任期开始的时候提交一个空的没有任何操作的日志条目到日志中来处理该问题。第二，leader 在处理只读请求之前必须检查自己是否已经被替代了（如果一个更新的 leader 被选举出来了，它的信息就是过时的了）。Raft 通过让 leader 在响应只读请求之前，先和集群中的过半节点交换一次心跳信息来处理该问题。另一种可选的方案，leader 可以依赖心跳机制来实现一种租约的形式，但是这种方法依赖 timing 来保证安全性（假设时间误差是有界的）。

参考资料

- [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, Sympo- sium 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 Sym- posium 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., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KAN- THAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-

distributed database. In Proc. OSDI'12, USENIX Conference on Operating Systems Design and Implemen- tation (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. Me´ry, Eds., vol. 7436 of Lec- ture 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.,ANDCHERITON,D.Leases:Anefficientfault- tolerant mechanism for distributed file cache consistency. In Proceedings of the 12th ACM Ssymposium on Operating Systems Principles (1989), pp. 202–210.
- [10] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. ACM Trans- actions on Programming Languages and Systems 12 (July 1990), 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 Con- ference (2010), USENIX, pp. 145–158.
- [12] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup sys- tems. In Proc. DSN'11, IEEE/IFIP Int'l Conf. on Depend- able 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 Transac- tions on Computer Systems 16, 2 (May 1998), 133–169.
- [16] LAMPORT, L. Paxos made simple. ACM SIGACT News 32, 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 replica- tion revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.
  17
- [23] LogCabin source code. logcabin/logcabin. http://github.com/
- [24] 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. Eu- roSys'06, ACM SIGOPS/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] MAZIE` RES, D. Paxos made practical. //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 Informat- ica 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., MAZIE`RES, D., MI- TRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. Com- munications 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 Com- puting 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 Sys- tems and Technologies (2010), IEEE Computer Society, pp. 1–10.
- [39] VAN RENESSE, R. Paxos made moderately complex. Tech. rep., Cornell University, 2012.