

Raft 论文

一、Abstract

分布式一致性是构建容错系统的基础，它使得一些机器可以构成集群工作，并容许其中一些节点失效。不幸的是，最通用的一致性算法--Paxos，被普遍认为是难以理解和正确实现的。

这篇论文提出了一个新的共识算法--Raft，它就是为易于理解而设计的。Raft 首先会选举出一台机器作为 Leader，然后让系统所有的决定都由 Leader 来处理，这两步操作是相当独立的，相比于 Paxos 形成了一个更好的结构，Paxos 的各个组件是很难分离的。Raft 通过投票和随机超时来选举 Leader，选举保证 Leader 存有所有必要的信息，因此数据只能从 Leader 流向其他节点。相比于其他基于 Leader 的算法，这使得行为流程更简单。一旦 Leader 被选举出来，就由它来管理复制日志。

Raft 也更适合于在工程上实现。它在实际部署中表现良好，它解决了部署完整系统所需要的一切，包括怎样管理客户端的交互、怎样管理集群成员关系、如何压缩日志等。为了可以改变集群成员，Raft 允许增加和删除集群中的节点，并且保证集群在这个过程中可以不中断服务。

我们相信 Raft 优于 Paxos 和其他的一致性算法。许多实现都可以证明，它的 leader 选举算法适合于各种各样的环境，并有着和 Multi-Paxos 同样的性能。

可以在这个可视化网站上查看动态的 Raft，方便理解：[可视化 Raft](#)

二、Introduction

现在的数据中心和应用都运行在高动态的环境中，可以通过增加或减少机器来进行横向扩展和收缩。机器和网络经常会面临故障，每年大约 2-4% 的磁盘驱动损坏，机器也经常宕机，而且在现代的数据中心中每天都会有数十条网络连接中断。因此系统在正常操作的时候可以处理机器加入或离开集群的情况，必须能够在很短时间内响应这些变化，对用户不造成影响。在当前系统中，这是一个主要的挑战。故障处理、协作、服务发现、配置管理在这种动态环境中都是很难做的。

幸运的是分布式一致性可以帮助我们处理这些挑战。一致性允许一些机器组成一个集群工作，可以容忍其中一些节点失效。在一个一致性的集群中，节点失效可以被正确的处理，保证系统的高可用，其他一些系统组件可以以一致性集群作为基础来实现容错。因此分布式一致性协议在构建高可用的大规模软件系统中起着重要的作用。

许多分布式一致性的实现都依赖于 Paxos 算法，在过去的二十年里，Paxos 算法在分布式一致性领域处于主导地位，大部分的实现要么基于 Paxos，要么会受它的影响，而且 Paxos 也是教授学生一致性的主要算法。然而 Paxos 太难以理解，尽管已经做了很多努力来使它看起来简单一点。而且为了支撑实际系统，它的架构需要作出复杂的改变，而且需要作出许多原文没有提到的扩展，难以工程实现。

因此我着手来实现一个更易理解并且易于实现的算法，称为 Raft。在设计 Raft 算法的过程中，我使用了一些技术来提高算法的可理解性，包括：分解(分为 Leader 选举、日志复制、安全)和减少状态空间。我们也解决了在构建一个完整的分布式系统中会遇到的问题。我们仔细的设计了每一个过程，使得它易于理解和实现。

我们也做了一个 Raft 的开源实现，称为 LogCabin 在第十章中会有更多的描述。

三、Motivation

一致性是容错系统中的基本问题，在许多提供高可用和一致性要求高的环境中会遇到这种问题，一致性算法通常用在一致的大规模存储系统中。

1、Replicated State Machine

一致性算法通常会在 Replicated State Machine 的上下文中来描述，即多个机器拥有状态的多份 copy，并能在一些机器故障时不中断的提供服务。replicated state machine 用于解决分布式系统中的各种容错问题。replicated state machine 的一些例子比如 Chubby 和 Zookeeper，提供了少量数据的 KV 存储，除了基本的 Put/Get 操作之外，还加入了 CAS 等操作用于安全的处理并发问题。

Replicated state machine 通常使用 replicated log 来实现，如图：

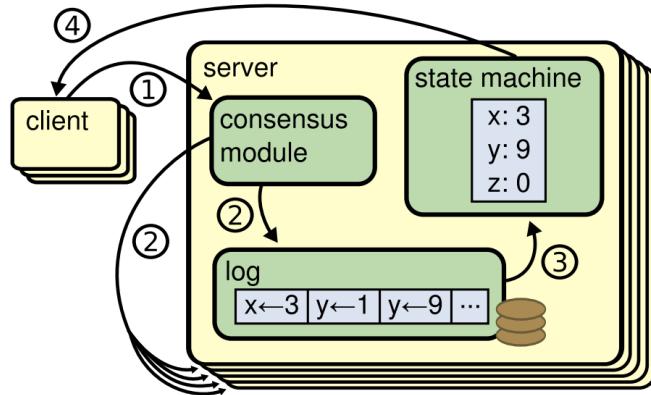


Figure 2.1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

每一个 server 都有一个日志保存了一系列的指令，state machine 会顺序执行这些指令。每一个日志都以相同顺序保存着相同的指令，因此每一个 state machine 处理相同的指令，state machine 是一样的，所以最终会达到相同的状态及输出。

保证 replicated log 的一致是一致性算法的任务。server 中的一致性模块接收客户端传来的指令并添加到自己的日志中，它也可以和其他 server 中的一致性模块沟通来确保每一条 log 都能有相同的内容和顺序，即使其中一些 server 崩机。一旦指令被正确复制，就可以称作 **committed**。每一个 server 中的状态机按日志顺序处理 committed 指令，并将输出返回客户端。

实际系统中的一致性算法通常有如下特性：

- 在任何情况下能够确保安全(不返回错误的结果)，包括网络延迟、分区、丢包、重复、失序
- 只要大多数节点可以正常工作和通信就能够保证完整的可用性。因此，一个典型的拥有 5 台机器的集群可以容忍两台机器的故障。当 server 恢复后可以读取持久化的状态并重新加入集群中。
- 不依赖于时间来确保日志的一致。错误的时钟和极端的消息延迟会导致不可用，因此它是异步的方式来保证安全性，消息以任意的速度来执行。
- 通常情况下，只要大多数节点对对 RPC 调用进行了响应就可看作命令执行完成，一小部分速度慢的 server 也不会拖慢整个系统的性能。

2、Replicated state machine 的常用场景

Replicated state machine 可以以各种方式来使用，这一节会讨论它的各种使用模式。

通常会使用三或五台机器来部署一个 replicated state machine 集群，其他的 server 可以使用状态机来协调他们的活动，如图，这些系统通常使用 replicated state machine 来提供 group membership、配置管理、锁等。比如一个具体的例子，replicated state machine 可以提供一个容错的工作队列，其他 server 可以利用它来协调任务分配。

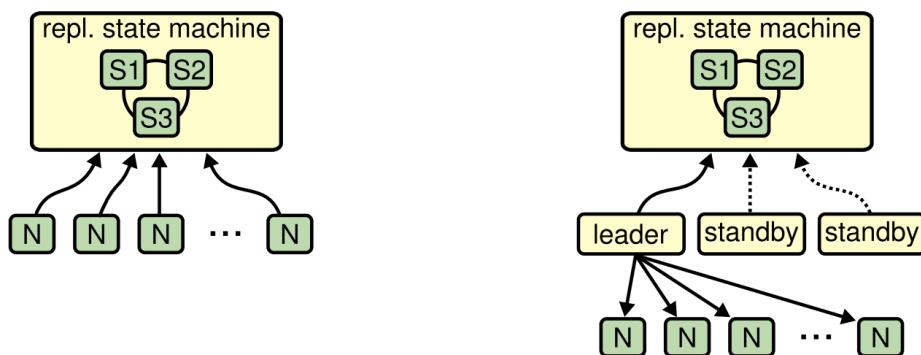


Figure 2.2: Common patterns for using a single replicated state machine.

一个更简化的场景如图 b，一个 server 作为 leader，管理其他的 server，leader 将一些重要的数据存储在一致性系统中，如果 leader 故障，会从其他 server 竞争选出新的 leader，一旦成功就可以继续使用一致性系统中的数据继续操作。许多大规模的存储系统都是这么做的，如 GFS、HDFS 等。

有时候会用来复制超大的数据，如图 2.3，大规模存储系统中，将数据分散到集群的多个 server 上，将数据划分成许多 replicated state machine，使用 two-phase commit protocol 来管理一致性。

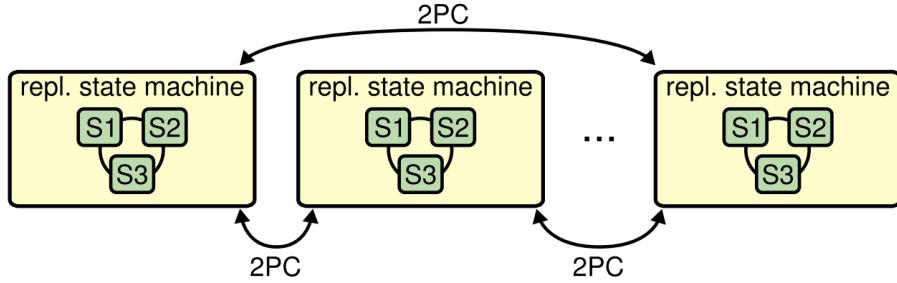


Figure 2.3: Partitioned large-scale storage system using consensus. For scale, data is partitioned across many replicated state machines. Operations that span partitions use a two-phase commit protocol.

四、Basic Raft Algorithm

这一章会讲述 Raft 算法。我们尽可能简单的设计 Raft 算法，第一部分描述了我们的设计思路，接下来的部分描述算法本身以及一些示例来帮助我们更好的理解。

1. Designing for understandability

在设计 Raft 算法的时候有几个目标：它必须可以支持实现一个完整的系统，需要极大的减少开发者的设计工作；它必须在任何条件下都能保证安全和可用性；必须效率足够高。但是我们最重要的目标也是最大的挑战---可理解性，它必须能够让更多的人可以理解，而且便于工程实现。

我们使用了两种技术来简化我们的算法。第一个就是众所周知的问题分解方法，在任何可能的地方将问题划分成几块来解决，可以独立的分析和理解，比如我们在 Raft 中我们设计了 Leader 选举、日志复制、和安全三部分。我们的第二个方法就是通过减少状态数来简化状态空间，尽可能消除系统中的不确定性，比如 Raft 限制 log 的使用方式来减少这种不确定性，当然有时候引入一些不确定性也会便于我们的理解，我们会使用随机化来简化 Leader 的选举过程。

2、Raft 简介

Raft 是一个管理 replicated log 的算法，图 3.1 总结了算法的核心内容便于参考，图 3.2 列出来了算法的一些关键特性，图中的内容我们会在后面的文章中详细的进行分析。

Raft 首先选举出一个 server 作为 Leader，然后赋予它管理日志的全部责任。Leader 从客户端接收日志条目，复制给其他 server，并告诉他们什么时候可以安全的将日志条目应用到自

己的状态机上。拥有一个 Leader 可以简化 replicated log 的管理。例如，leader 可以决定将新的日志条目放在什么位置，而无需询问其他节点，数据总是简单的从 leader 流向其他节点。Leader 可能失败或者断开连接，这种情况下会选出一个新的 leader。

通过 leader，Raft 将一致性问题分解成三个相当独立的子问题：

- Leader Election：当集群启动或者 leader 失效时必须选出一个新的 leader。
- Log Replication：leader 必须接收客户端提交的日志，并将其复制到集群中的其他节点，强制其他节点的日志与 leader 一样。
- Safety：最关键的安全点就是图 3.2 中的 State Machine Safety Property。如果任何一个 server 已经在它的状态机 apply 了一条日志，其他的 server 不可能在相同的 index 处 apply 其他不同的日志条目。后面将会讲述 raft 如何实现这一点。

State		RequestVote RPC
Persistent state on all servers: (Updated on stable storage before responding to RPCs)		Invoked by candidates to gather votes (§3.4).
currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)	Arguments: term candidate's term
votedFor	candidateId that received vote in current term (or null if none)	candidateId candidate requesting vote
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)	lastLogIndex index of candidate's last log entry (§3.6) lastLogTerm term of candidate's last log entry (§3.6)
Volatile state on all servers:		Results: term currentTerm, for candidate to update itself voteGranted true means candidate received vote
commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)	Receiver implementation:
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)	1. Reply false if term < currentTerm (§3.3) 2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§3.4, §3.6)
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)	Rules for Servers
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)	All Servers:
AppendEntries RPC		
Invoked by leader to replicate log entries (§3.5); also used as heartbeat (§3.4).		
Arguments:		Followers (§3.4):
term	leader's term	• Respond to RPCs from candidates and leaders
leaderId	so follower can redirect clients	• If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate
prevLogIndex	index of log entry immediately preceding new ones	Candidates (§3.4):
prevLogTerm	term of prevLogIndex entry	• On conversion to candidate, start election:
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)	<ul style="list-style-type: none"> • 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
leaderCommit	leader's commitIndex	Leaders:
Results:		<ul style="list-style-type: none"> • Upon election: send initial empty AppendEntries RPC (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§3.4) • If command received from client: append entry to local log, respond after entry applied to state machine (§3.5) • If last log index \geq nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex <ul style="list-style-type: none"> • If successful: update nextIndex and matchIndex for follower (§3.5) • If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§3.5) • If there exists an N such that $N > commitIndex$, a majority of $matchIndex[i] \geq N$, and $log[N].term == currentTerm$: set $commitIndex = N$ (§3.5, §3.6).
Receiver implementation:		
1. Reply false if term < currentTerm (§3.3) 2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5) 3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5) 4. Append any new entries not already in the log 5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)		

Figure 3.1: A condensed summary of the Raft consensus algorithm (excluding membership changes, log compaction, and client interaction). The server behavior in the lower-right box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §3.4 indicate where particular features are discussed. The formal specification in Appendix B describes the algorithm more precisely.

Election Safety

At most one leader can be elected in a given term. §3.4

Leader Append-Only

A leader never overwrites or deletes entries in its log; it only appends new entries. §3.5

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. §3.5

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. §3.6

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. §3.6.3

Figure 3.2: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

3、Raft 基础

一个 Raft 集群会包含数个 server，5 是一个典型值，可以容忍两个节点失效。在任何时候每个 server 都会处于 Leader、Candidate、Follower 三种状态中的一种。在正常情况下会只有一个 leader，其他节点都是 follower，follower 是消极的，他们不会主动发出请求而仅仅对来自 leader 和 candidate 的请求作出回应。leader 处理所有来自客户端的请求(如果客户端访问 follower，会把请求重定向到 leader)。Candidate 状态用来选举出一个 leader。如图：

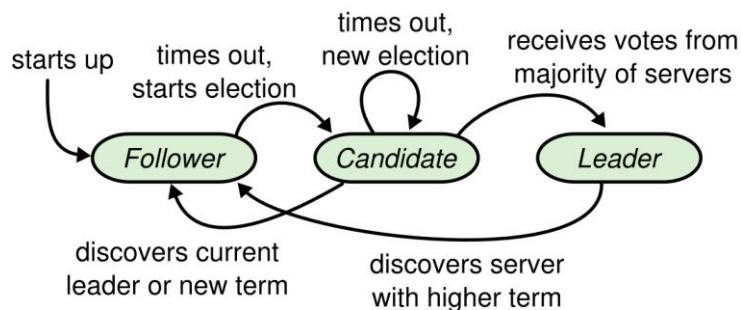


Figure 3.3: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

Raft 将时间划分为任意长度的 **term**，用连续整数编号。每一个 term 都从选举开始，一个或多个 candidate 想要成为 leader，如果一个 candidate 赢得选举，它将会在剩余的 term 中

作为 leader。在一些情况下选票可能会被瓜分，导致没有 leader 产生，这个 term 将会以没有 leader 结束，一个新的 term 将会很快产生。Raft 确保每个 term 至多有一个 leader。Term 在 Raft 中起到了逻辑时钟的作用，它可以帮助 server 检测过期信息比如过期的 leader。每一个 server 都存储有 current term 字段，会自动随时间增加。当 server 间通信的时候，会交换 current term，如果一个节点的 current term 比另一个小，它会自动将其更新为较大者。如果 candidate 或者 leader 发现了自己的 term 过期了，它会立刻转为 follower 状态。如果一个节点收到了一个含有过期的 term 的请求，它会拒绝该请求。

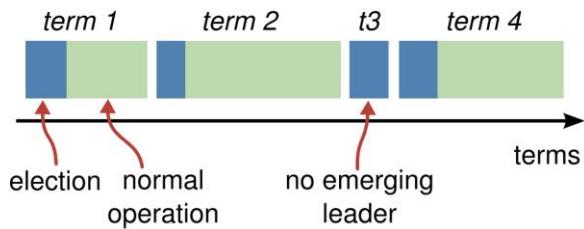


Figure 3.4: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

Raft 节点之间通过 RPC 进行通信，基本的一致性算法仅仅需要两种 RPC。RequestVote RPC 由 candidate 在选举过程中发出，AppendEntries RPC 由 leader 发出，用于复制日志和提供心跳。每一个请求类型都有对应的 response，Raft 假定 request 和 response 都可能会丢失，因此要求请求者有超时重试的能力。为了性能，RPC 请求会并行发出，而且不保证 RPC 的到达顺序。

4、Leader election

Raft 使用心跳机制来触发 leader 选举。当 server 启动的时候是处于 follower 状态，当它可以收到来自 leader 或者 candidate 的有效 RPC 请求时就会保持 follower 的状态。Leader 发送周期性的心跳(不含日志的 AppendEntries RPC)给所有的 follower 来确保自己的权威。如果一个 follower 一段时间(称为 election timeout)没有收到消息，它就会假定 leader 失效并开始新的选举。

为了开始新一轮选举，follower 会提高自己当前的 term 并转为 candidate 状态。它会先给自己投一票然后并行向集群中的其他 server 发出 RequestVote RPC，candidate 会保持这

个状态，直到下面三种事情之一发生：

- (a) 赢得选举。当 candidate 收到了集群中相同 term 的多数节点的赞成票时就会选举成功，每一个 server 在给定的 term 内至多只能投票给一个 candidate，先到先得。收到多数节点的选票可以确保在一个 term 内至多只能有一个 leader 选出。一旦一个 candidate 赢得选举，它就会成为 leader。它之后会发送心跳消息来建立自己的权威，并阻止新的选举。
- (b) 另一个 server 被确定为 leader。在等待投票的过程中，candidate 可能收到来自其他 server 的 AppendEntries RPC，声明它才是 leader。如果 RPC 中的 term 大于等于 candidate 的 current term，candidate 就会认为这个 leader 是合法的并转为 follower 状态。如果 RPC 中的 term 比自己当前的小，将会拒绝这个请求并保持 candidate 状态。
- (c) 没有获胜者产生，等待选举超时。candidate 没有选举成功或者失败，如果许多 follower 同时变成 candidate，选票就会被瓜分，形不成多数派。这种情况发生时，candidate 将会超时并触发新一轮的选举，提高 term 并发送新的 RequestVote RPC。然而如果不采取其他措施的话，选票将可能会被再次瓜分。

Raft 使用随机选举超时来确保选票被瓜分的情况很少出现而且出现了也可以被很快解决。election timeout 的值会在一个固定区间内随机的选取(比如 150-300ms)。这使得在大部分情况下仅有一个 server 会超时，它将在其他节点超时前赢得选举并发送心跳。candidate 在发起选举前也会重置自己的随机 election timeout，也可以帮助减少新的选举轮次内选票瓜分的情况。

5、Log Replication

一旦一个 leader 被选举出来，它开始为客户端请求服务。每一个客户端请求都包含着一个待状态机执行的命令，leader 会将这个命令作为新的一条日志追加到自己的日志中，然后并向其他 server 发出 AppendEntries RPC 来复制日志。当日志被安全的复制之后，leader 可以将日志 apply 到自己的状态机，并将执行结果返回给客户端。如果 follower 宕机或运行很慢，甚至丢包，leader 会无限的重试 RPC(即使已经将结果报告给了客户端)，直到所有的 follower 最终都存储了相同日志。

日志按如下图的方式进行组织，每一条日志储存了一条命令和 leader 接收到该指令时的 term 序号。日志中的 term 序号可以用来检测不一致的情况，每一条日志也拥有一个整数索引用于定位。

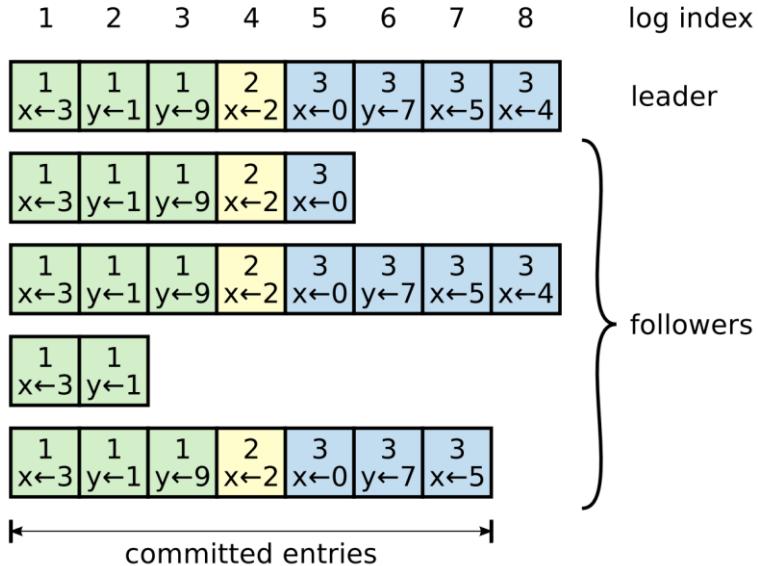


Figure 3.5: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

leader 会决定何时 apply 一条日志是安全的，这被称为 committed。Raft 确保 committed 日志是持久化的并最终被所有的状态机执行。一旦 leader 把日志复制到了大多数节点上，就会 committed，这也意味着在此之前的所有日志都被 commit 了，包括之前其他 leader 创建的日志。图 3.6 展示了一些在 leader 改变后执行这些规则可能产生的一些情况，它也表明了 commit 的定义是安全的。leader 会追踪已经 committed 的最高的日志索引，并将这个索引放入之后的 AppendEntries RPC，以便于其他节点可以最终发现，一旦一个 follower 意识到一条日志被 committed 了，它就会将其 apply 到自己的状态机。

Raft 日志机制可以保证不同 server 上的日志具有很高的一致性。这不仅仅简化了系统和增强了可预测性，而且这也是确保安全的一个重要组件，Raft 通过下列特性构建了图 3.2 中描述的 Log Matching Property：

- 如果不同日志中的两条记录有相同的 index 和 term，他们会存有相同的命令。
- 如果不同日志中的两条记录有相同的 index 和 term，那么他们之前的记录也是完全相同的。

leader 在给定的 index 和 term 处至多只会创建一条记录，并且新的记录不会改变之前的位置，因此可以确保第一条。第二条是通过 AppendEntries 的一致性检查实现的。当发送 AppendEntries RPC 的时候，leader 会将之前最新日志的 term 和 index 包含在其中，如果 follower 没有在自己的日志中找到相同的 index 和 term，它就会拒绝这条请求。累加效果，因此只要 AppendEntries RPC 返回成功，leader 就会知道 follower 的日志直到最新的这条都和自己一样。

在正常操作中，leader 和 follower 的日志是完全一致的，因此 AppendEntries 的一致性检查不会失败。然而，leader 失效可能会造成不一致的情况，这种不一致可能是多次形成的。图 3.6 展示了一些 follower 和 leader 日志不一样的情况。follower 可能会缺少一些日志，也可能会比当前 leader 额外多出一些日志，或者两者兼有，而且可能涉及到几个 term。

在 Raft 中，leader 通过强制 follower 复制自己的日志来解决这种不一致的情况，意味着 follower 和 leader 产生冲突的部分日志会以 leader 为准进行重写。

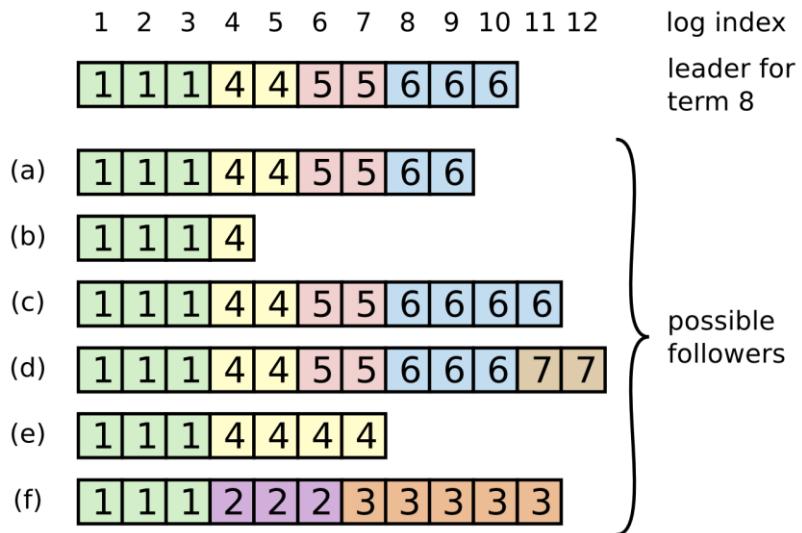


Figure 3.6: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, 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.

为了使得 follower 的日志和 leader 的日志一致，leader 必须找到自己和 follower 最后一致的日志索引，然后删掉在那之后 follower 的日志，并将 leader 在那之后的日志全部发送给

follower。所有的这些操作都发生在 AppendEntries RPC 的一致性检查中。leader 持有针对每一个 follower 的 nextIndex 索引，代表下一条要发送给对应 follower 的日志索引。当 leader 刚上任时，它会初始化所有的 nextIndex 值为最后一条日志的下一个索引，如图 3.6 中的 11。如果 follower 的日志和 leader 的不一致，下一次 AppendEntries 的一致性检查就会失败。在遭到拒绝后，leader 就会降低该 follower 的 nextIndex 并进行重试。最终 nextIndex 会到达 leader 和 follower 一致的位置。这条 AppendEntries RPC 会执行成功，并覆盖 follower 在这之后原有的日志，之后 follower 的日志会保持和 leader 一致，直到这个 term 结束。

如果 leader 发现自己的日志和 follower 是完全匹配的，leader 就可以发送不带有日志数据的 AppendEntries(心跳)来节省带宽。一旦 matchIndex 追上了 nextIndex，leader 应当开始发送日志数据。

当然上面的寻找 index 的过程可以优化减少 AppendEntries RPC。例如，当拒绝 AppendEntries 请求时，follower 可以返回发生冲突的 entry 所在的 term 以及该 term 的第一个 index，通过这些信息，leader 可以直接跳过这个 term 中的全部 index。另外 leader 可以使用二分搜索来查找和 follower 第一个不同的日志。实际上，这些优化不是很必要的，因为故障不很频繁，而且不太可能有太多不一致的日志。Leader 从不需要重写或者删除自己的日志。

6、Safety

前面的章节讲述了 raft 如何进行 leader 选举和复制日志，然而到目前为止所描述的这些机制是不能充分确保每一个状态机都以相同的顺序执行相同的指令。例如，在 leader 提交了几条日志的过程中，某个 follower 始终不可用，之后该 follower 被选举为 leader 并重写了原来 leader 的这些日志，结果导致不同的状态机可能执行了不同的命令。

这一小节将会通过给选举过程增加一些限制来完成 Raft 算法，这个限制可以确保 leader 在给定的 term 都含有全部已提交的日志。通过选举限制使得日志提交的规则更加精确。

(1) Election restriction

在任何基于 leader 的一致性算法中，leader 必须最终存有全部 committed 日志。在一些一致性算法（如 Viewstamped Replication），节点 即使不包含全部 committed 日志也会被选举为 leader，这些算法通过其他的机制来定位缺失的日志，并将其转移给新的 leader。然而这增加了系统的复杂度，raft 使用了更加简单的方法来确保所有 committed 的日志存在于每个

新选举出来的 leader，不需要转移日志。因此日志只需要从 leader 流向 follower 即可，而且不需要重写自己的日志。

Raft 使用投票过程来确保选举成为 leader 的 candidate 一定包含全部 committed 的日志。Candidate 必须联系大部分节点以赢得选举，也就意味着每一个 committed 日志至少存在于其中一个节点上(复制超过一半节点才会 commit)，如果 candidate 的日志至少和这一大部分节点的日志一样新，它就会含有全部已提交的日志。RequestVote RPC 实现了这一限制：**RPC 请求包含着 candidate 的日志信息，其他节点如果发现自己的日志比 candidate 的日志更新，就回拒绝该请求。**

Raft 通过比较最后一条日志的 index 和 term 来决定谁更新一些。如果 term 不一致则拥有更大的 term 日志更新，如果 term 一样，则 index 更大的日志更新。

(2) Committing entries from previous terms

如果一条日志成功复制到大多数节点上，leader 就知道可以 commit 了。如果 leader 在 commit 之前崩溃了，新的 leader 将会尝试完成复制这条日志。然而一个 leader 不可能立刻推导出之前 term 的 entry 已经 commit 了。图 3.7 展示了这样的情景，已经复制到大多数节点的日志可能会被覆盖。为了消除这种可能，Raft 不会通过计算已经复制的数目来提交以前 term 的日志，它只会提交当前 term 中的日志。一旦当前 term 的日志 committed，那么之前的 term 也会被间接提交。这样做更简单。

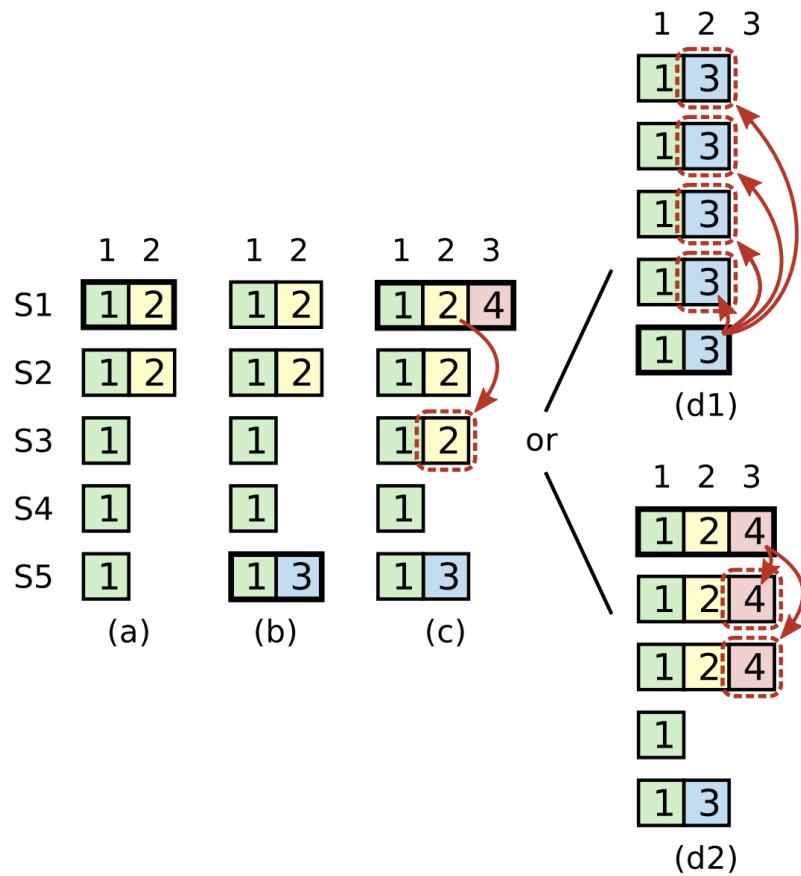


Figure 3.7: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry 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 (d1), 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 (d2), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

7、Follower and candidate crashes

Follower 和 candidate 的崩溃处理相比于 leader 简单许多，而且可以用相同的方式处理。如果 follower 或者 candidate 崩溃，之后发给他们的 RPC 请求会失败。Raft 会进行无限重试，如果重启了，RPC 请求将会执行成功。如果 server 完成了 RPC 请求，但是响应丢失了，Leader 也会重新发送 RPC 请求，节点收到重复请求是没有影响的，比如收到了重复的 log entry，它会忽略这些日志。

8、Persisted state and server restarts

Raft server 必须将一些必要的数据持久化以便重启后数据不丢失。首先需要持久化它当前的 term 和投出的选票，这可以防止节点在一个 term 内投出两次选票。每一个 server 也需要在 commit 前持久化最新的日志，可以防止 committed 日志丢失。

其他的变量是可以丢失的，因为可以重新产生。一个例子就是 commitIndex，在重启的时候可以安全的重置为 0.即使每一个 server 都在相同的时候重启，commitIndex 也仅仅是短暂的落后于它的真实值。一旦 leader 被选举出来并提交新的 entry，它的 commitIndex 就会很快的提高上来，并立刻传播给其他的 server。

对于状态机来说，它可以是持久化的也可以是易失的，易失性的状态机在重启后必须重新 apply 它的日志，而持久化的状态机已经 apply 过了，为了避免重复 apply，它需要持久化保存最后 apply 的 index。

如果一个 server 丢失了全部的持久化的状态，这时候它不能以之前的身份安全的加入到集群中。这个节点可以当作一个新的节点，通过触发集群的 membership change 来加入集群。如果大部分集群节点都丢失了持久化的数据，这时候就需要集群管理员出马人工干预了。

9、Timing and availability

我们对 Raft 安全性的一项要求是它不依赖于精确的时间。然而，可用性(系统在规定时间内响应客户端)却不可避免的依赖于时间。Leader 选举是 Raft 对时间最敏感的部分。当满足下面的式子时，raft 可以选出并保持一个稳定的 leader：

broadcastTime << electionTimeout << MTBF

broadcastTime 代表节点并行向其他所有节点并行发送 RPC 请求并获得响应的平均时间；electionTimeout 是选举超时；MTBF 是单个节点两次宕机间隔的平均时间。broadcastTime 应当远小于 electionTimeout，从而 leader 可以依靠心跳来阻止 follower 发起选举；给 electionTimeout 引入随机性可以避免选票瓜分；electionTimeout 应远小于 MTBF，从而使得系统可以平稳工作。当 leader 宕机时，系统将仅会在 electionTimeout 的时间内不可用。

broadcastTime 和 MTBF 都受到底层系统的影响，但是 electionTimeout 是我们必须谨慎选择的。Raft 的 RPC 请求一般会需要持久化存储，因此 broadcastTime 一般需要 0.5-20ms，

依赖于持久化技术。因此 electionTimeout 最好选择在 10-500ms 之间。典型的 MTBF 值一般是几个月，它很容易就满足要求。

10、Leadership transfer extension

这一节描述了 Raft 的一个可选扩展，它允许一个节点转移自己的 leader 权给其他节点，在下面两种情况下会很有用：

- 1、有时候 leader 需要主动下线。比如，它可能需要重启或者移出集群。当它下线的时候，集群在 electionTimeout 的时间内处于闲置状态，直到有一台机器成为 leader，这种不可用的情况可以通过主动转移 leader 权来避免。
- 2、在一些情况下，其他的节点可能更适合于担当 leader。比如 Raft 的 leader 节点承担了全部的客户端负载，当负载很高时会影响系统的性能，这时候 leader 可以周期性的检查集群中的 follower 是否有更适合成为 leader 的，然后将领导权转移给他。

为了转移领导权，当前 leader 会把自己的日志发送给目标节点，然后目标节点提前触发一轮选举。当前 leader 确保了目标节点拥有全部 committed 日志。下面是详细步骤：

1. 当前 leader 停止接收客户端请求
2. 当前 leader 通过复制日志将目标节点的日志更新为和自己完全一样
3. 当前 leader 发送一个 TimeoutNow 请求给目标节点。这个请求会使得目标节点立刻触发超时并开启新一轮选举。它有极大可能在其他节点超时之前赢得选举，它的下一条消息将会包含新的 term 编号，导致 leader 自动下线。leader 转移完成。

如果转移失败，之前的 leader 必须中断转移过程，并重新开始接收客户端的请求。

五、Cluster membership changes

到目前为止我们都假定集群的配置信息（如 server 列表）是不变的，实际上有时候需要改变集群的配置，比如某台 server 宕机时将其替换或者增加 follower，这可以通过两种方式来进行手动操作：

- 可以使集群下线一段时间来更新配置文件，然后重启集群。这会使得在配置更新期间集群不可用。

- 一个新的 server 可以通过询问待下线节点的地址来进行替换。管理员必须确保被替换掉的节点不会再回来，否则系统将会丧失安全性。

上面的方法都有很大的缺点，而且一旦引入手动操作，将会增加操作的风险。为了避免这些问题，我们决定引入自动更新配置机制到 Raft 算法中。Raft 允许集群在配置更新的时候可以继续工作，并且仅仅给基本的 Raft 算法增加一些扩展就可以实现。图 4.1 总结了会用到的 RPC，将会在后面的章节中进行讨论。

AddServer RPC	RemoveServer RPC
Invoked by admin to add a server to the cluster configuration.	Invoked by admin to remove a server from the cluster configuration.
Arguments:	Arguments:
newServer address of server to add to configuration	oldServer address of server to remove from configuration
Results:	Results:
status OK if server was added successfully	status OK if server was removed successfully
leaderHint address of recent leader, if known	leaderHint address of recent leader, if known
Receiver implementation:	Receiver implementation:
1. Reply NOT LEADER if not leader (§6.2)	1. Reply NOT LEADER if not leader (§6.2)
2. Catch up new server for fixed number of rounds. Reply TIMEOUT if new server does not make progress for an election timeout or if the last round takes longer than the election timeout. (§4.2.1)	2. Wait until previous configuration in log is committed (§4.1)
3. Wait until previous configuration in log is committed (§4.1)	3. Append new configuration entry to log (old configuration without oldServer), commit it using majority of new configuration (§4.1)
4. Append new configuration entry to log (old configuration plus newServer), commit it using majority of new configuration (§4.1)	4. Reply OK and, if this server was removed, step down (§4.2.2)
5. Reply OK	

Figure 4.1: RPCs used to change cluster membership. The AddServer RPC is used to add a new server to the current configuration, and the RemoveServer RPC is used to remove a server from the current configuration. Section numbers such as §4.1 indicate where particular features are discussed. Section 4.4 discusses ways to use these RPCs in a complete system.

1、Safety

保证配置更新期间的安全性是面临的第一个挑战，必须确保在同一个 term 中不会有两个 leader 被选举出来。如果直接更新配置，增加或者删除集群中的 servers，直接将集群从旧的配置更新到新的配置是不安全的，可能会出现两个 leader。不可能一次性自动的完成所有节点的转换，因此可以在转换期间将集群划分为两个独立的部分，如图 4.2。

大部分配置更新的算法都引入了额外的机制来解决问题，这也是 Raft 一开始所做的，但是后来找到了一种更好的方法，我们对配置更新的操作进行了限制：集群在某一时刻只能添加或删除一台机器。复杂的多台机器的配置更新可以通过逐个执行上述操作来实现。这一章的大部分内容都在讲述单节点更新方法，它更容易理解。本章第三节描述了我们原来的操作，它带来

了额外的复杂性。

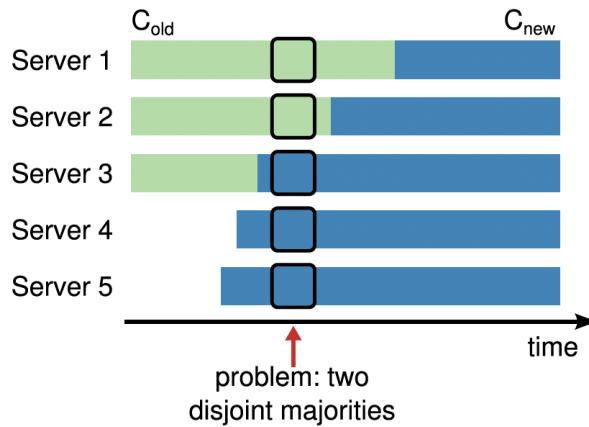
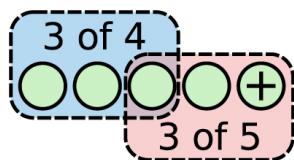
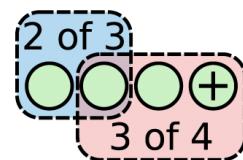


Figure 4.2: Switching directly from one configuration to another can be unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

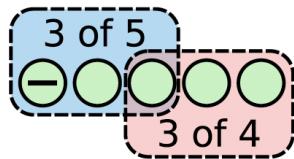
当向集群添加或移除一个节点时，如图 4.3，老的集群节点形成的多数派和新集群节点形成的多数派，必然会发生重叠，也就是说必然只能形成一个多数派，这个重叠避免了脑裂。因此只添加或删除一个节点时，可以直接更新配置。Raft 利用此属性，几乎不需要其他机制即可安全的更改集群成员的身份。



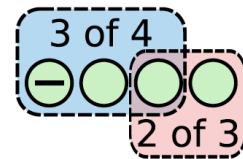
(a) Adding one server to a 4-server cluster.



(b) Adding one server to a 3-server cluster.



(c) Removing one server from a 5-server cluster.



(d) Removing one server from a 4-server cluster.

Figure 4.3: The addition and removal of a single server from an even- and an odd-sized cluster. In each figure, the blue rectangle shows a majority of the old cluster, and the red rectangle shows a majority of the new cluster. In every single-server membership change, an overlap between any majority of the old cluster and any majority of the new cluster is preserved, as needed for safety. For example in (b), a majority of the old cluster must include two of the left three servers, and a majority of the new cluster must include three of the servers in the new cluster, of which at least two must come from the old cluster.

集群配置信息可以作为特殊的 log entry 来存储和交流，这可以利用 Raft 复制和持久化日志的现有机制。通过在配置更新和客户端请求之间添加顺序，它也可以允许集群在配置更新的过程中持续为客户端提供服务。

当 leader 收到一个让其从现有集群配置(Cold)增加或删除一个节点的请求时，它向其日志中追加一条含有新的配置信息 Cnew 的 entry，并使用正常的 Raft 机制将其复制给其他节点，一旦该 entry 被追加到节点的日志中 新的日志就开始生效 Cnew entry 会被复制到 Cnew Server 上，是否大部分节点应用了新配置来决定是否 commit。这意味着节点不用等待新配置 commit，而是总会使用最新的 config。

当 Cnew 被 commit 之后配置变更完成，此时 leader 知道大部分的 server 已经应用了新的配置，它也知道没有更新配置的 server 不可能再组成多数派，也就不可能选出 leader。新配置的 commit 可以使得下面的三件事继续：

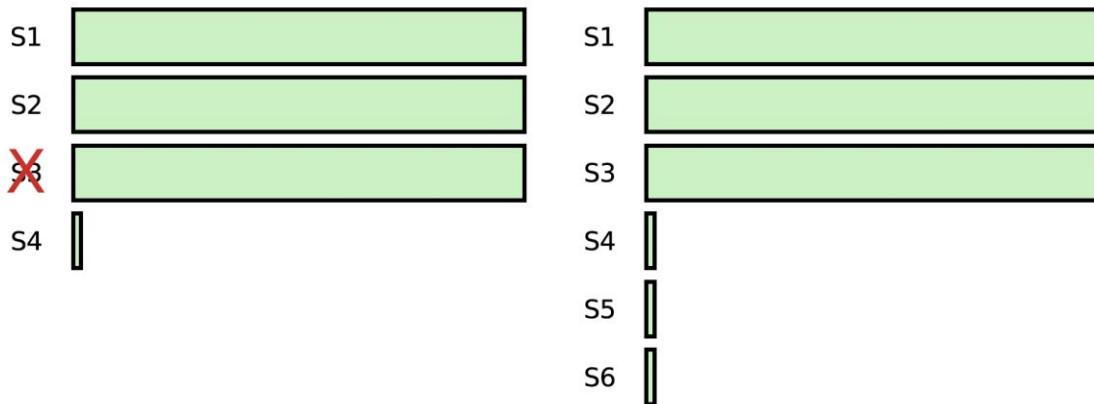
1. leader 可以知道成功完成了配置更新
2. 如果新配置要移除一个节点，它就可以下线了。
3. 可以开始后续的配置更新。在此之前，重叠的配置更新可能会导致不安全的情况。

如上所述，server 总是使用日志中的最新配置信息，而不需关注它是否 committed。如果只有当新配置 committed 之后才可以应用到节点，leader 将会很难搞清楚什么时候大部分节点使用了新配置。

2、Availability

(1) Catching up new servers

当一个新的 server 被添加到集群中时，它通常是没有存储任何日志信息的，它的日志通常需要很长时间才能追上 leader 的日志进度，这期间集群更容易出现不可用。例如，一个三节点的集群可以容忍单节点失效，然而如果带有空日志的第四个 server 被添加到这个集群中，并且原来的一个节点出现宕机，这期间集群将不能够 commit 日志，如图 4.4.另一种可能发生的情况是如果多个新节点被连续添加到集群，要想形成多数派的话就需要这些新的节点。在这些情况下，直到新节点的日志追上 leader，否则集群会一直不可用。



(a) Failure of S3 while adding S4.

(b) Adding S3–S6 in quick succession.

Figure 4.4: Examples of how adding servers with empty logs can put availability at risk. The figures show the servers' logs in two different clusters. Each cluster starts out with three servers, S1–S3. In (a), S4 is added, then S3 fails. The cluster should be able to operate normally after one failure, but it loses availability: it needs three of the four servers to commit a new entry, but S3 has failed and S4's log is too far behind to append new entries. In (b), S4–S6 are added in quick succession. Committing the configuration entry that adds S6 (the third new server) requires four servers' logs to store that entry, but S4–S6 have logs that are far behind. Neither cluster will be available until the new servers' logs are caught up.

为了避免这种不可用的情况，Raft 在配置更新之前引入了一个额外的阶段，期间新加入的节点不占有投票权，leader 会将日志复制给它，但是选举或者 commit 时统计多数选票都不会计入它。一旦新节点的日志追上来，配置便可以开始更新了（这种机制在其他场景下也会非常有用，比如可以用在向大部分节点复制状态，对于一致性不那么高的只读请求可以通过读取 follower 节点来实现）。

leader 需要知道新节点什么时候真正的追上了 leader 的日志，从而可以进行后续的配置更新。如果新节点刚刚添加就开始更新配置可能会导致不可用，我们的目标是将不可用时间保持在 election timeout 之内，因为正常情况下也会出现 election timeout 时间的不可用。而且，我们希望尽可能的让新节点的日志接近于 leader 的日志，最小化不可用时间。

如果新节点不可用或者速度太慢不可能追上 leader 的日志，leader 应该终止这个更新。事实上，我们的第一次配置更新的请求需要包含网络端口号，系统可以正确的终止更新并返回错误。

我们建议使用下面的算法来确保新节点的日志可以充分追上 leader：将对新节点的复制日志过程划分为几轮，如图 4.5，每一轮复制从开始到现在的所有日志给新节点，由于在复制日志的过程中，leader 会继续接收新的日志，这些新加的日志会在下一轮进行复制，随着不断的进

行，每一轮次复制的持续时间会不断缩短，当算法等待几个轮次(比如 10)的复制之后，如果最后一轮的持续时间短于 election timeout，leader 就可以将新节点正式加入集群，否则 leader 就会中断配置更新并返回错误，调用者之后可以进行重试（下次重新更有可能成功，因为该节点的日志已经部分赶了上来）。

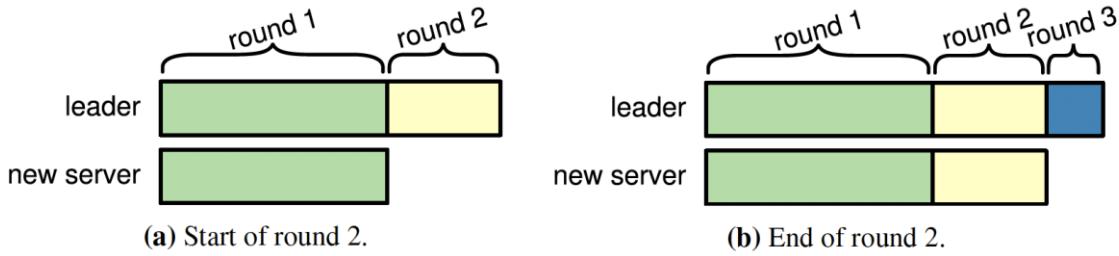


Figure 4.5: To catch up a new server, the replication of entries to the new server is split into rounds. Each round completes once the new server has all of the entries that the leader had in its log at the start of the round. By then, however, the leader may have received new entries; these are replicated in the next round.

作为日志追上来的第一步，leader 必须去发现新节点的日志是空的。对新节点的 AppendEntries 一致性检查会一直失败，直到 nextIndex 减为 1 (之后可以将全部日志传输过去)，这个过程可能是影响节点加入性能的主要因素。虽然有各种的方法优化这一过程，但是最简单的方式就是新节点在收到 AppendEntriesRPC 之后能在 response 中返回自己的日志长度，从而 leader 可以快速确定它的 nextIndex。

(2) Removing the current leader

如果当前 leader 需要被从集群中移除，它必须在某个时间点下线。一个直接的方法就是使用之前讲过的领导权主动转移的扩展，leader 将领导权转移给其他节点，之后可以执行正常的配置更新。

如果没有实现这个扩展，我们可以采用另外一种方式。在这种方式中，要被移除的 leader 会在新配置 Cnew 被 commit 之后下线。如果下线早于这个时间点，可能会触发超时并重新成为 leader，从而使得下线进程延后。在一个极端场景下，移除两节点集群中的 leader，这个 leader 可能必须再次成为 leader，以使得集群可以正常运行，如图 4.6。因此 leader 等到新配置 commit 之后再下线。这是新配置可以确定在不需要移除的 leader 参与的情况下正常运行的第一个时间点：因为这允许新配置下的节点可以在他们中选出一个新的 leader。在原来 leader 下线之后，新配置下的一个 server 会超时触发选举。这个短暂的不可用应该是可以容忍的。

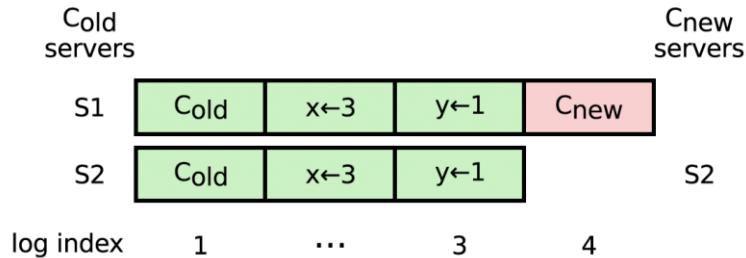


Figure 4.6: Until the C_{new} entry has been committed, a removed server may need to lead the cluster to make progress. The figure shows the removal of S1 from a two-server cluster. S1 is currently leader. S1 should not step down quite yet; it is still needed as leader. S2 cannot become leader until it receives the C_{new} entry from S1 (since S2 still needs S1's vote to form a majority of C_{old} , and S1 won't grant its vote to S2 because S2's log is less up-to-date).

这种方法可能会有两个虽然不严重但需要关注的问题。第一，将会有一段时间（正在 commit 新配置）leader 在管理集群，但它却不是集群成员，它复制日志但在统计多数派时却不计入自己。第二，节点可能在不持有最新配置的情况下触发选举，因为在新配置 commit 之前还需要它，在选举的时候不会统计它的选票，直到它也获得了最新配置。

(3) Disruptive servers

不处于新配置的节点可能会毁坏整个集群，一旦 leader 创建了一个新配置，不在新配置中的节点将不会收到心跳信息，因此它可能会超时并开始新的选举，而且它将不会接收到新配置和新配置 commit 的消息，它也就将不会知道自己被移出集群。它将会以新的 term 发送 RequestVote RPC，并使得其他的 leader 被迫转为 follower。处于新配置下的一个节点最终会成为 leader，但是这个待下线节点将会再次超时并形成恶性循环。如果要下线的节点更多，情况将会更严重。

Raft 的解决办法是使用心跳来判断是否有一个有效的 leader 存在，如果 leader 可以持续的发送心跳那就认为 leader 是活跃的。因此节点可以收到心跳时是不可以打断 leader 的。我们修改一下 RequestVote RPC：如果节点在心跳超时之前收到了 RequestVote 请求，它不会提升自己的 term。它可以丢弃该请求或者返回一个拒绝投票的请求，或者推迟这个请求，结果都是相同的。这不会影响正常的选举，这却保证了 leader 不会被迫下线。

这和前面描述的领导权转移扩展冲突，因为那需要在超时之前就触发选举，即使仍然有 leader 存在也需要去处理 RequestVote 请求。可以通过给 RequestVote 请求加个标志来解决这个冲突。

3、Availability argument

这一节将会提出一个更复杂的方法来处理针对一次性任意数量节点(添加、删除)的配置更新。例如，两个节点可以一次性加入到一个集群中，或者一个 5 节点集群的 server 需要全部替换。当然这个方法会引入额外的复杂性，我们还是建议使用单节点的更新方式，任意数量的同时更改通常只是在论文理论分析中会用，我不认为在实际系统中需要这样做，因为一系列的单节点更改已经足够满足各种各样的情景了。

为了确保任意数量节点配置更新时的安全性，集群首先需要转移的一个过渡状态称为 joint consensus，一旦这个状态被提交，系统之后就可以过渡到新配置。joint consensus 结合了新老配置：

- 日志被复制到新老配置中的所有节点
- 来自新老配置的 server 都有可能成为 leader
- Agreement 需要来自新老配置各自的多数选票。例如，从三节点过渡到 9 节点时，agreement 需要原来三节点配置的 2 个 server 和新配置下的 5 个 server 的赞成。

joint consensus 允许 server 各自过渡到新配置而不需要作出安全性的保证，而且允许集群在整个变更过程中持续提供服务。这个方法通过中间日志 entry 来扩展单节点的配置更新算法。如图 4.8 所示，当 leader 收到需要将配置从 C_{old} 过渡到 C_{new} 的请求时，它将 joint consensus 需要的配置 C_{old,new} 存为日志 entry 并通过正常的 Raft 机制复制。。。。。。

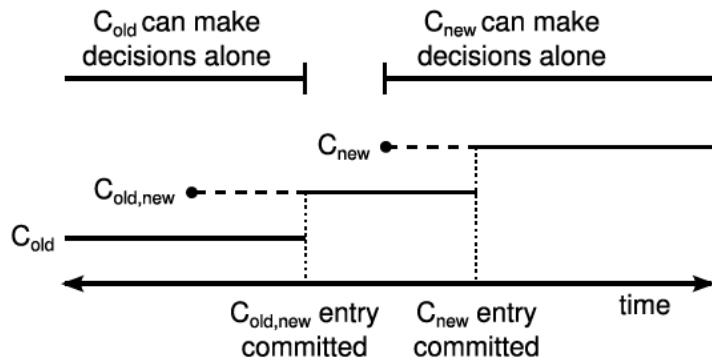


Figure 4.8: Timeline for a configuration change using joint consensus. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits 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.

4. System integration

不同的 Raft 实现可能会以不同的方式来实现本章中描述的配置更新机制。比如 `AddServer` 和 `RemoveServer` 的 RPC 方法可能直接由管理员触发，也可能通过脚本来触发一系列的单节点配置更新。

在一些情况下，比如机器宕机，可能需要自动触发配置更新，这应当在合理的策略下才可以进行。比如，自动移除宕机的节点对集群来说可能是很危险的，因为可能会剩下很少的节点，不能满足系统的持久化和容错要求。一种有效的方式是让那个管理员配置一个要求的集群节点数量，在这个限制下，可以用可用的节点自动代替下线的节点。

当需要处理多个节点时，更倾向于先增加节点后删除节点。例如，为了替换一个三节点集群中的一台机器，先增后删可以使得系统在任何时候都可以容错一台机器。然而如果先删后增，系统可能会遇到短暂的不可用状态。

配置更新操作也导致了不同的集群启动方式。在不支持配置更新时，每一个 server 只需要又一个静态配置文件就可以。在可以动态配置更新的情况下，就不需要这个静态配置文件了，因为系统通过 Raft 日志来管理配置。但是我们推荐，在集群最开始创建的时候每一个 server 都有一个配置文件作为它的第一条日志，这个配置仅列出该 server 自己的信息，它独自组成了它的配置中描述的多数派。其他的 server 应当初始化时只有 0 条日志，他们被添加到集群并获取

当前配置信息。

六、Log compaction

Raft 日志会随着不断处理客户端的请求而变大，会占用越来越多的空间而且重放日志也需要更多的时间。如果不采取一些压缩日志的方式，最终会造成一些可用性的问题：机器可能会耗尽空间，或者话花费太多的时间来重启。因此对于实际系统来说日志压缩是必须的。

压缩日志的通常的思路是过期的日志占用了太多无用信息，可以丢弃掉。例如，一个操作 set $x=2$ ，之后又一个操作 set $x=3$ ，那么前一个日志就是无用的。一旦日志被 committed 并 apply 到状态机，用来到达当前状态的中间状态和操作就是不需要的了，可以被压缩掉。

与核心 Raft 算法和成员配置更新不一样，对于不同的系统有不同的日志压缩需要，没有一个放之四海而皆准的方法。这一章的目标是讨论一些日志压缩的方法，在每一个方法中，大部分日志压缩的责任都落在状态机上，raft 状态机可以有不同的实现方式，如图 5.1 所示：

Memory-Based Snapshots (§6.1)

Apply entry:

Mutate in-memory data structure

Service read:

Look up result in in-memory data structure

Take snapshot:

When Raft log size in bytes reaches 4x previous snapshot size:

1. Fork the state machine's memory
 - In parent, continue processing requests
 - In child, serialize in-memory data structure to new snapshot file on disk
2. Discard previous snapshot file on disk
3. Discard Raft log up through child's last applied index

State to transfer to slow follower:

Latest snapshot file (immutable)

Disk-Based Snapshots (§6.2)

Apply entry:

1. Mutate on-disk data structure
2. Discard Raft log up through last applied index

Service read:

Look up result in on-disk data structure

State to transfer to slow follower:

Copy-on-write snapshot of on-disk data structure

Log-Structured Merge Trees (§6.3)

Apply entry:

Add entry to in-memory tree

Service read:

1. Search for key in in-memory tree
2. Search all level 0 runs (any might contain the key)
3. For each level counting up from 1 in order, search the single run that might contain the key

Create new run:

When in-memory tree reaches 1 MB:

1. Serialize in-memory tree into new sorted level 0 run on disk
2. Reset in-memory tree
3. Discard Raft log up through last applied index

Compact runs:

When there are 4 runs at level 0:

1. Merge all level 0 runs with all level 1 runs, producing new non-overlapping level 1 runs split at 2 MB boundaries
2. Discard merged runs

When the total size of all runs at level L exceeds 10^L MB:

1. Merge one level L run (chosen round-robin) with all overlapping level L+1 runs, producing new non-overlapping level L+1 runs split at 2 MB boundaries
2. Discard merged runs

State to transfer to slow follower:

All runs on disk (immutable)

Very Small Leader-Based Snapshots (§6.4)

Apply entry:

Mutate in-memory data structure

Service read:

Look up result in in-memory data structure

Take snapshot:

When Raft log size in bytes reaches 1 MB:

1. Stop accepting client requests
2. Wait until last applied index reaches end of log
3. Serialize data structure, append to new snapshot entry in log
4. Resume processing client requests
5. As each server learns the snapshot entry is committed, it discards its Raft log entries up to that entry

State to transfer to slow follower:

Raft log (no additional state)

Raft State for Compaction

Persisted before discarding log entries. Also sent from leader to slow followers when transmitting state.

prevIndex	index of last discarded entry (initialized to 0 on first boot)
prevTerm	term of last discarded entry (initialized to 0 on first boot)
prevConfig	latest cluster membership configuration up through prevIndex

Figure 5.1: The figure shows how various approaches to log compaction can be used in Raft. Details for log-structured merge trees in the figure are based on LevelDB [63], and details for log cleaning are based on RAMCloud [98]; rules for managing deletions are omitted.

各种日志压缩方法都有一些通用的概念。首先，每一个 server 都独自压缩自己的已 committed 的日志，而不是将日志压缩任务集中在 leader，这避免了通过网络传输大量的冗余数据给 follower。它也有助于模块化，大部分的日志压缩操作集中于状态机而不是 raft 本身，

这可以使得整个系统的复杂度最小。第四节也会讲述基于 leader 的日志压缩，但是适合于很小数据量的日志压缩。第二，状态机和 raft 之间的交互都需要将日志从 raft 传输到状态机。在 apply 这些日志 entry 后，状态机会将这些日志存到磁盘，便于以后恢复系统状态。一旦状态机完成，就会通知 raft 把这部分日志丢弃掉，在丢弃之前，它必须保留一些描述丢弃掉的日志的状态信息。通常 raft 会保留丢弃掉的最后一条日志的 index 和 term，从而可以使得 AppendEntries 的一致性检查继续工作，raft 也需要保留丢弃的日志中的最新的配置信息来支持配置更新。第三，一旦 raft 丢弃了之前的日志，状态机就会担负起另外两种新的责任。如果 server 重启，在状态机可以 apply raft 日志之前需要从磁盘加载这些数据，另外状态机也需要产生一个满足一致性的数据镜像，以便于可以将其发送给较慢的 follower。当 raft 监测到 AppendEntries 中需要发送的 next entry 已经被丢弃掉了，此时状态机必须提供提供这种镜像以便发送给 follower。

1、Snapshotting memory-based state machines

第一种 snapshot 压缩日志的方法适合于状态机数据保存在内存中的场景。这种方法在数据量在几 GB 或者几十 GB 的情况下是一个合适的选择，它操作完成很快，因为不需要从磁盘获取数据，它也容易编程实现，因为有丰富的数据结构可以使用并且操作不会被 IO 阻塞。

图 5.2 展示了 Raft snapshotting 的基本操作。每一个 server 单独拍摄快照，仅包括自己日志中 committed 部分。快照的大部分工作涉及到状态机当前状态的序列化，这具体到不同的状态机实现。一旦状态机完成了 snapshot，日志就可以被截断。Raft 首先会持久化重启所需要的状态，最后一条日志的 index、term 和所属的 config，然后丢弃在此 index 之前的所有日志。任何之前的快照也会被丢弃，因为没用了。

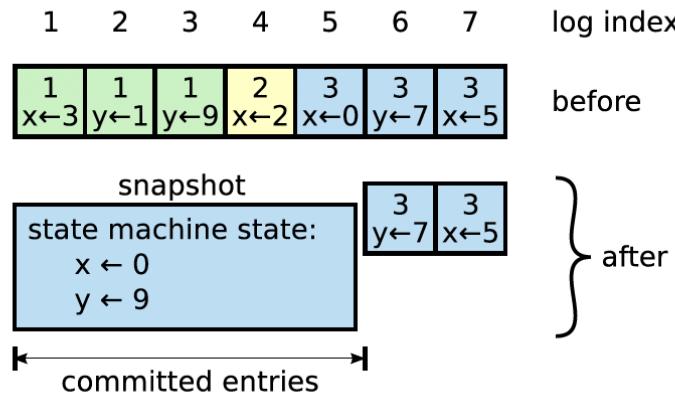


Figure 5.2: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables x and y in this example). Before discarding entries 1 though 5, Raft saves the snapshot's last included index (5) and term (3) to position the snapshot in the log preceding entry 6.

如上所述，leader 偶尔可能会需要将自己的数据发送给慢 follower 或者新加入集群的节点。leader 通过一个新的 RPC--InstallSnapshot 来发送最新的快照，如图 5.3.当 follower 收到 snapshot，它必须决定如何处理它当前的日志。通常快照会包含 follower 当前没有的新日志，这种情况下 follower 只需要全部丢弃。当然如果收到的 snapshot 数据不如 follower 自己的新，它会保留在这之后的日志，而把之前的用 snapshot 覆盖掉。

InstallSnapshot RPC	
Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.	
Arguments:	
term	leader's term
leaderId	so follower can redirect clients
lastIndex	the snapshot replaces all entries up through and including this index
lastTerm	term of lastIndex
lastConfig	latest cluster configuration as of lastIndex (include only with first chunk)
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk
Results:	
term	currentTerm, for leader to update itself
Receiver implementation:	
1.	Reply immediately if term < currentTerm
2.	Create new snapshot file if first chunk (offset is 0)
3.	Write data into snapshot file at given offset
4.	Reply and wait for more data chunks if done is false
5.	If lastIndex is larger than latest snapshot's, save snapshot file and Raft state (lastIndex, lastTerm, lastConfig). Discard any existing or partial snapshot.
6.	If existing log entry has same index and term as lastIndex and lastTerm, discard log up through lastIndex (but retain any following entries) and reply
7.	Discard the entire log
8.	Reset state machine using snapshot contents (and load lastConfig as cluster configuration)

Figure 5.3: Leaders invoke the InstallSnapshot RPC to send snapshots to slow followers. Leaders resort to sending a snapshot only when they have already discarded the next log entry needed to replicate entries to the follower with AppendEntries. They split the snapshot into chunks for transmission. Among other benefits, this gives the follower a sign of life with each chunk, so it can reset its election timer. Each chunk is sent in order, which simplifies writing the file to disk. The RPC includes the state needed for Raft to load the snapshot on a restart: the index and term of the last entry covered by the snapshot, and the latest configuration at that point.

(1) Snapshotting concurrently

创建一个快照可能需要很长时间，包括序列化状态和将数据写入磁盘。例如复制 10GB 的内存数据可能只需要 1 秒，但是将其序列化可能需要更长的时间，在写入磁盘的时候每秒大概只能写入 500M。因此序列化和写快照必须是和正常操作并发执行的，以免影响可用性。

幸运的是，copy-on-write 技术允许在不影响快照写入的情况下 apply 新的更新，有两种方式可以实现这一点：

- 可以使用不可变的数据结构来构建状态机。因为状态机命令不会原地修改状态，快照任务可以保存对之前状态的引用并将其一致的写入快照。

- 可以使用操作系统的 copy-on-write 技术。比如在 Linux 环境下，可以使用 fork 获得 server 当前地址空间的完整拷贝。子进程可以写入所有的数据，父进程可以持续处理客户端请求。

server 会需要额外的内存来支持并发快照，这是可以提前计划和管理的。状态机提供流式处理 snapshot 文件数据的接口是很重要的，从而快照可以不需要完全一次性载入内存。copy-on-write 技术也需要额外比例的内存支持。如果在快照的过程中内存耗尽了，server 应当停止接收新日志直到完成快照，这会短暂的耗尽可用性，但至少使得 server 可恢复。最好不要中断快照并重试，因为下次很可能会遇到相同的情况。

(2) When to snapshot

Server 必须决定什么时候进行快照。如果快照进行的太频繁，会浪费磁盘带宽和其他的资源，如果快照太少，将会使得存储空间耗尽的风险增大，也增加了重启后日志重放的耗时。一个简单的策略是当日志达到一个固定的内存使用值的时候进行快照，如果这个值被设置的明显大于 snapshot 的预期体积大小，则用于快照的磁盘带宽开销将很小，但是这会导致不必要的大的日志体积。

一个更好的方法是比较快照大小和日志的大小，如果快照比日志小许多倍，那此时进行快照将是值得的。但是在拍摄快照之前计算其体积大小是很困难的，并且为状态机增加了负担，甚至计算快照体积的工作量不亚于直接拍摄快照。压缩快照文件可以节省存储空间和带宽，但是也很难预测压缩后的文件大小。

幸运的是，我们可以使用上一次快照的体积。一旦当前日志的大小超过了上次快照体积的一定倍数，就需要进行快照了，这个倍数称为 expansion factor，是可配置的。expansion factor 需要在磁盘带宽和空间使用上寻求平衡，例如，expansion factor 为 4 的时候，快照会占用 20% 的磁盘带宽，以及 6 倍的磁盘空间，因为需要数据的一份拷贝（原来的快照+新的快照+4 倍大小的日志）。

快照也会导致 CPU 使用的剧增，会影响到性能。这可以通过增加额外的硬件来解决，比如额外的磁盘驱动。

(3) Implementation concerns

这一小节将会总结实现快照功能需要的组件并讨论实现时的一些难点：

- **Saving and loading snapshots**：保存快照涉及到序列化状态机的状态数据并将其写入到文件中，加载是其逆向过程。我们发现这其实很简单，虽然你将数据从原本的状态序列化成各种类型会有些繁琐。提供一个从状态机到磁盘文件的流式接口是很有用的，可以避免在内存中缓存整个的状态机状态数据，可能也会有助于压缩流并应用校验和。
- **Transferring snapshots**：传输快照涉及到实现 leader 和 follower 侧的 InstallSnapshot RPC 接口。这也是相当简单的，并且可以重用一些从磁盘保存和加载快照的代码。传输的性能通常不是很重要。
- **Eliminating unsafe log access and discarding log entries**：由于丢弃日志操作的存在，我们在访问日志的时候需要注意检查不要越界，当确保可以安全的访问日志的时候，丢弃前面的日志就显得很简单了。

2、Snapshotting disk-based state machines

这一节讨论针对于使用磁盘的大存储量的状态机，它的特点是总是在磁盘上保存有它的状态，Raft 日志中的每一个 entry 被 apply 到状态机后都会持久化到磁盘，相当于每次都进行快照。因此一旦一个 entry 被 apply，它就可以丢弃掉了。基于磁盘的状态机的主要问题是持久化数据会导致性能下降，如果没有写入缓冲，对于每条指令它都需要进行一或多次的随机磁盘访问，这可能会限制整体写入性能。

基于磁盘的状态机也必须可以提供一个一致性的快照以便于可以发送给慢 follower。虽然它总是在磁盘上留有快照，但是它在持续修改。因此仍然需要 copy-on-write 技术来保留一个一致性快照，以便于乐意在长时间内传输它。幸运的是磁盘格式几乎都会被划分成逻辑块，因此实现 copy-on-write 应当是很简单的。基于磁盘的状态机也可以依赖于系统的支持，比如 Linux 系统上的 LVM 可以用来创建整个磁盘分区的快照，并且最近的一些文件系统也支持针对单独的目录进行快照。

3、Alternative : leader-based approaches

基于 leader 复制的日志压缩方法，会由 leader 来压缩日志，并将快照发送给 follower，这是很浪费资源的。发送这些冗余的数据给每一个 follower 会浪费网络带宽并减慢日志压缩进程，而对于 leader 来说，带宽往往是它最重要的资源。

这种方式的一个好处就是不需要一些额外的复制和持久化方法，将其保存在日志中就可以了。

如图 5.5，leader 会将快照通过 AppendEntries RPC 发送给每一个 follower。

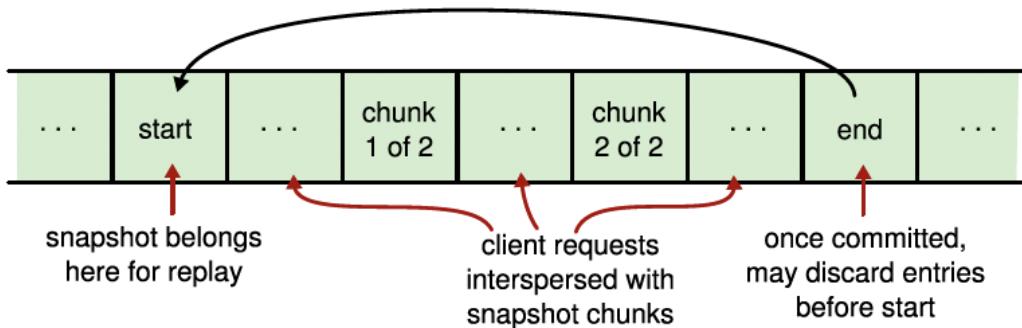


Figure 5.5: A leader-based approach that stores the snapshot in chunks in the log, interleaved with client requests. The snapshotting process is started at the *start* entry, and it completes by the *end* entry. The snapshot is stored in several log entries between *start* and *end*. So that client requests can proceed in parallel with snapshotting, each entry is limited in size, and the rate at which the entries are appended to the log is limited: the next snapshot chunk is only appended to the log when the leader learns that the previous snapshot chunk has been committed. Once each server learns that the *end* entry is committed, it can discard the entries in its log up to the corresponding *start* entry. Replaying the log requires a two pass algorithm: the last complete snapshot is applied first, then the client requests after the snapshot's *start* entry are applied.

这种方式适合于体积很小的快照，用处很少。

七、Client interaction

这一张讲述客户端是如何与基于 Raft 的复制状态机交互的。图 6.1 展示了所用的 RPC 方法，会在这一章进行详细讲述。

<p>ClientRequest RPC</p> <p>Invoked by clients to modify the replicated state.</p> <p>Arguments:</p> <table border="0"> <tr> <td>clientId</td> <td>client invoking request (§6.3)</td> </tr> <tr> <td>sequenceNum</td> <td>to eliminate duplicates (§6.4)</td> </tr> <tr> <td>command</td> <td>request for state machine, may affect state</td> </tr> </table> <p>Results:</p> <table border="0"> <tr> <td>status</td> <td>OK if state machine applied command</td> </tr> <tr> <td>response</td> <td>state machine output, if successful</td> </tr> <tr> <td>leaderHint</td> <td>address of recent leader, if known (§6.2)</td> </tr> </table> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply NOT LEADER if not leader, providing hint when available (§6.2) 2. Append command to log, replicate and commit it 3. Reply SESSION_EXPIRED if no record of clientId or if response for client's sequenceNum already discarded (§6.3) 4. If sequenceNum already processed from client, reply OK with stored response (§6.3) 5. Apply command in log order 6. Save state machine output with sequenceNum for client, discard any prior response for client (§6.3) 7. Reply OK with state machine output <p>Rules for Leaders</p> <ul style="list-style-type: none"> • Upon becoming leader, append <i>no-op</i> entry to log (§6.4) • If election timeout elapses without successful round of heartbeats to majority of servers, convert to follower (§6.2) 	clientId	client invoking request (§6.3)	sequenceNum	to eliminate duplicates (§6.4)	command	request for state machine, may affect state	status	OK if state machine applied command	response	state machine output, if successful	leaderHint	address of recent leader, if known (§6.2)	<p>RegisterClient RPC</p> <p>Invoked by new clients to open new session, used to eliminate duplicate requests. §6.3</p> <p>No arguments</p> <p>Results:</p> <table border="0"> <tr> <td>status</td> <td>OK if state machine registered client</td> </tr> <tr> <td>clientId</td> <td>unique identifier for client session</td> </tr> <tr> <td>leaderHint</td> <td>address of recent leader, if known</td> </tr> </table> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply NOT LEADER if not leader, providing hint when available (§6.2) 2. Append register command to log, replicate and commit it 3. Apply command in log order, allocating session for new client 4. Reply OK with unique client identifier (the log index of this register command can be used) <p>ClientQuery RPC</p> <p>Invoked by clients to query the replicated state (read-only commands). §6.4</p> <p>Arguments:</p> <table border="0"> <tr> <td>query</td> <td>request for state machine, read-only</td> </tr> </table> <p>Results:</p> <table border="0"> <tr> <td>status</td> <td>OK if state machine processed query</td> </tr> <tr> <td>response</td> <td>state machine output, if successful</td> </tr> <tr> <td>leaderHint</td> <td>address of recent leader, if known</td> </tr> </table> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply NOT LEADER if not leader, providing hint when available (§6.2) 2. Wait until last committed entry is from this leader's term 3. Save commitIndex as local variable readIndex (used below) 4. Send new round of heartbeats, and wait for reply from majority of servers 5. Wait for state machine to advance at least to the readIndex log entry 6. Process query 7. Reply OK with state machine output 	status	OK if state machine registered client	clientId	unique identifier for client session	leaderHint	address of recent leader, if known	query	request for state machine, read-only	status	OK if state machine processed query	response	state machine output, if successful	leaderHint	address of recent leader, if known
clientId	client invoking request (§6.3)																										
sequenceNum	to eliminate duplicates (§6.4)																										
command	request for state machine, may affect state																										
status	OK if state machine applied command																										
response	state machine output, if successful																										
leaderHint	address of recent leader, if known (§6.2)																										
status	OK if state machine registered client																										
clientId	unique identifier for client session																										
leaderHint	address of recent leader, if known																										
query	request for state machine, read-only																										
status	OK if state machine processed query																										
response	state machine output, if successful																										
leaderHint	address of recent leader, if known																										

Figure 6.1: Clients invoke the ClientRequest RPC to modify the replicated state; they invoke the ClientQuery RPC to query the replicated state. New clients receive their client identifier using a RegisterClient RPC, which helps identify when session information needed for linearizability has been discarded. In the figure, servers that are not leaders redirect clients to the leader, and read-only requests are serviced without relying on clocks for linearizability (the text presents alternatives). Section numbers such as §6.3 indicate where particular features are discussed.

1. Finding the cluster

当 raft 作为一个网络服务，客户端必须能够定位集群的地址。对于拥有固定成员的集群来说，这是简单的，比如他们的网络地址可以作为静态配置文件。然而，找到成员可变的集群的地址是一个挑战。通常有两种方法：

1. 客户端可以使用网络广播找到所有的集群节点，然而这仅仅在特殊的环境下才可以使用

2. 客户端可以通过额外的目录服务来发现集群节点，比如 DNS。在外部服务中保存的 server 列表需要满足一致性。客户端应当总是可以找到所有的集群节点，因此在集群配置发生改变的时候，这个外部服务的 server 列表应当在应当在配置更新之前进行修改，在集群配置更新完成之后再修改一次，移除那些不在集群中的节点，以便保持一致性。

2、Routing request to leader

Raft 集群中客户端的请求是由 leader 处理的，因此客户端需要一种方式来找到 leader。当客户端首次启动的时候，它随机选择集群的一个节点连接，如果不是 leader，节点会拒绝请求。再次重试的一个简单方式是再次随机选择一个节点，这种简单方式下，如果有 n 个节点，那么平均会尝试 $(n+1)/2$ 次，对于规模比较小的集群来说也还可以。

我们通过简单的优化就可以更快的找到 leader。集群中的节点通常知道当前集群中 leader 的地址，因为 AppendEntries 包含着 leader 的信息。当一个非 leader 节点收到客户端请求时，它可以做下面两件事：

1. 第一个选择，也是我们推荐的。节点拒绝该请求，如果知道 leader 地址，并将 leader 的地址返回给客户端。这使得客户端下次尝试可以直接连接 leader，速度更快。它也需要额外实现很少的代码，因为客户端也需要重联一个不同的节点，当 leader 失效的时候。
2. 节点也可以将客户端的请求代理到 leader。这可能在某些情况下更简单。比如，当一个客户端只需要读请求的时候，代理请求可以帮助客户端节省管理连接的成本。

Raft 必须避免过期的 leader 信息造成成客户端请求被无限延迟。过期的 leader 信息可能存在于整个系统：

- leaders：一个 leader 可能处于 leader 状态，但是它不是当前的 leader，这可能会导致客户端的请求延迟。例如一个 leader 出现了网络分区，它仍然可以和某个客户端进行通信。如果没有额外的机制，它可能会把客户端的请求无限延迟，因为它无法成功提交日志。同时可能在另一个分区里选出一个新的 leader，并成功提交客户端的请求。因此当在一个 election timeout 之内没有成功完成一轮心跳到大部分节点的话，它就应当下线了，这使得客户端可以向其他节点重试请求。
- followers：follower 保存着当前 leader 的信息。当开始一次新的选举或者 term 更新时，它必须丢弃当前的 leader 信息，比如可能会导致请求在两个节点之间来回发送，陷入

死循环

- clients : 如果 client 丢失了和 leader 的网络连接 , 它应当随机选择一个节点进行重试。如果坚持连接它最后一次知道的 leader 可能会导致不必要的延迟。

3、Implementing linearizable semantics

到目前为止 , raft 实现了对于客户端的请求至少执行一次的保证 , 但是可能会对相同的请求重复执行多次。例如 , 一个客户端向 leader 提交了一条命令 , 并且 leader 将这条命令追加到了自己的日志并提交成功 , 但是在向客户端返回响应的时候失败了 , 因为客户端没有收到响应 , 它会重试这条命令 , 从而作为新的 entry 被追加到日志中被提交 , 这时候请求被重复执行了。即使在没有客户端的参与下命令也可能被重复执行 , 比如网络可能产生重复请求。

这个问题不仅仅在 raft 中存在 , 大部分的分布式系统都会有这个问题。这种问题可能会造成不正确的返回结果和状态。如图 6.2 展示了一种情况 : 一个状态机用于提供锁机制 , 当客户端的请求没有收到相应的时候它会认为没有获得锁 , 但是实际上可能已经获得了锁。像 increment 操作的时候 , 可能会增加难以预料的值。网络的重排序和客户端并发请求甚至会带来更难以预料的情况。

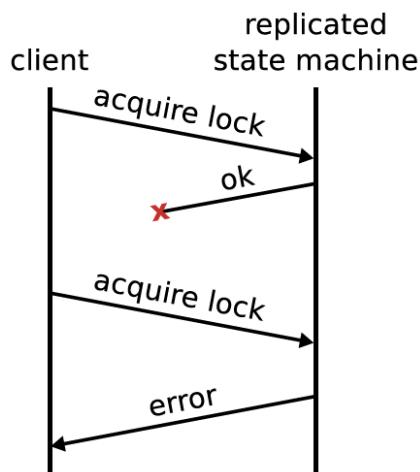


Figure 6.2: An example of an incorrect results that can arise from duplicated commands. A client submits a command to a replicated state machine to acquire a lock. The client's first command acquires the lock, but the client never receives the acknowledgment. When the client retries the request, it finds that the lock is already taken.

我们的目标是实现一种线性化的语义 , 从而可以避免上面的问题。在线性化机制下 , 每一个请求只能执行一次 , 对于客户端来说这是一种强一致性。为了实现线性化 , 客户端必须能够过滤

出重复请求。基本的思路是 server 保存客户端操作的执行结果，以便于收到相同请求时可以直接返回避免再次执行。为了实现这一点，每一个客户端会分配一个 UID，并且客户端给每一个命令分配一个序号。每一个 server 对每一个 client 维护一个 session，session 可以追溯到最新处理的客户端命令序号，以及对应的响应，如果收到的序号已经被执行过了，可以将响应直接返回。这种方式通常也适合于单客户端的并发访问，client 's session 包含着一系列的(请求序号，响应)对，对于每一个请求，客户端包含着尚未收到响应的最小的序号，并且状态机会丢掉比它更小的所有响应。

然而，由于空间是有限的，session 不可能一直保留，server 最后必须关闭 session，但是这产生了两个问题：server 怎样就何时关闭 session 达成一致，以及如何处理活跃客户端被频繁关闭 session 的问题？

servers 必须就何时关闭 session 达成一致，否则状态机就会出现状态不一致。

4、Process read-only queries more efficiently

只读的客户端命令只会查询复制状态机，不会改变状态。因此，我们自然会问，是否可以绕过日志呢？绕过日志具有吸引人的性能优势，只读请求在许多应用程序中很常见，并且追加日志条目的磁盘写操作是很耗时的。

然而如果没有其他的措施，绕过日志的只读查询会导致返回过期的查询结果。例如，一个 leader 可能被网络分区了，并且另一个分区的节点可能已经选出了新的 leader 并提交了日志，如果这个被分区的 leader 不咨询其他节点就对只读请求返回响应，它会返回过期的结果。线性化要求只读请求返回的结果能够反映的是只读请求触发之后的系统状态，每一个读请求至少应该返回最新 committed 的写操作的状态。幸运的是，我们可以实现对只读请求绕过日志并且实现线性化。为了实现这个目标，leader 需要采取下列步骤：

1. raft 让每个 leader 在其任期刚开始的时候提交一个空的日志条目，只要这个日志被提交了，leader 的 commitIndex 就可以保证是当前 term 内最大的 index。
2. leader 保存当前的 commitIndex 为一个 readIndex 变量，这会用来作为查询操作的版本的下限。
3. leader 需要确保它没有被不知道的新 leader 所取代。它会发起新一轮的心跳并等待来自集群大多数节点的响应。一旦这些响应到达，leader 就会知道在它发出心跳的时刻不会存

在另一个 term 更大的 leader。因此，readIndex 就是那时集群中最大的 commitIndex。

4. leader 等待状态机至少前进到 readIndex 处，这是满足线性化要求的。

5. 最后 leader 向状态机发起查询并将结果返回客户端。

这种方法相比于将只读操作提交到日志要更高效，因为它不需要同步磁盘写操作。为了进一步提高效率，leader 可以分摊确认自己是 leader 的开销，它可以对累计的任意数量的只读操作使用单次心跳。

Followers 也可以分担只读请求的压力。这将会提高系统的读吞吐量，它也会减少 leader 的负载压力，使得它可以处理更多的写操作。然而这样也面临着读取过期数据的风险。例如，一个被分区的 follower 可能很长一段时间内不会收到 leader 的新日志，甚至是 follower 可能收到了来自 leader 的心跳，但是 leader 已经过期了自己还不知道。为了可以安全的读取，follower 可以向刚刚请求当前 readIndex 的 leader 发起请求(leader 会执行步骤 1-3)，follower 接下来可以在自己的状态机执行 4 和 5.32