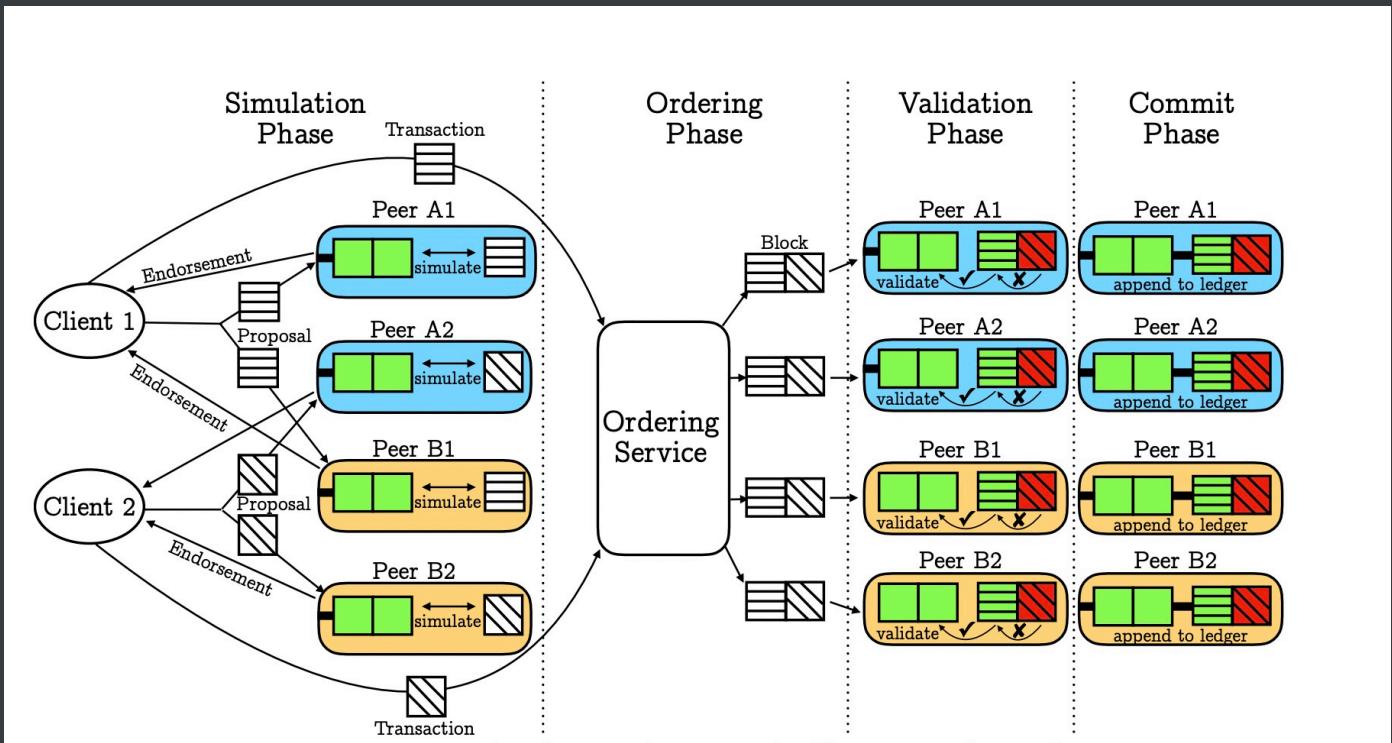


Hyperledger Fabric



Background

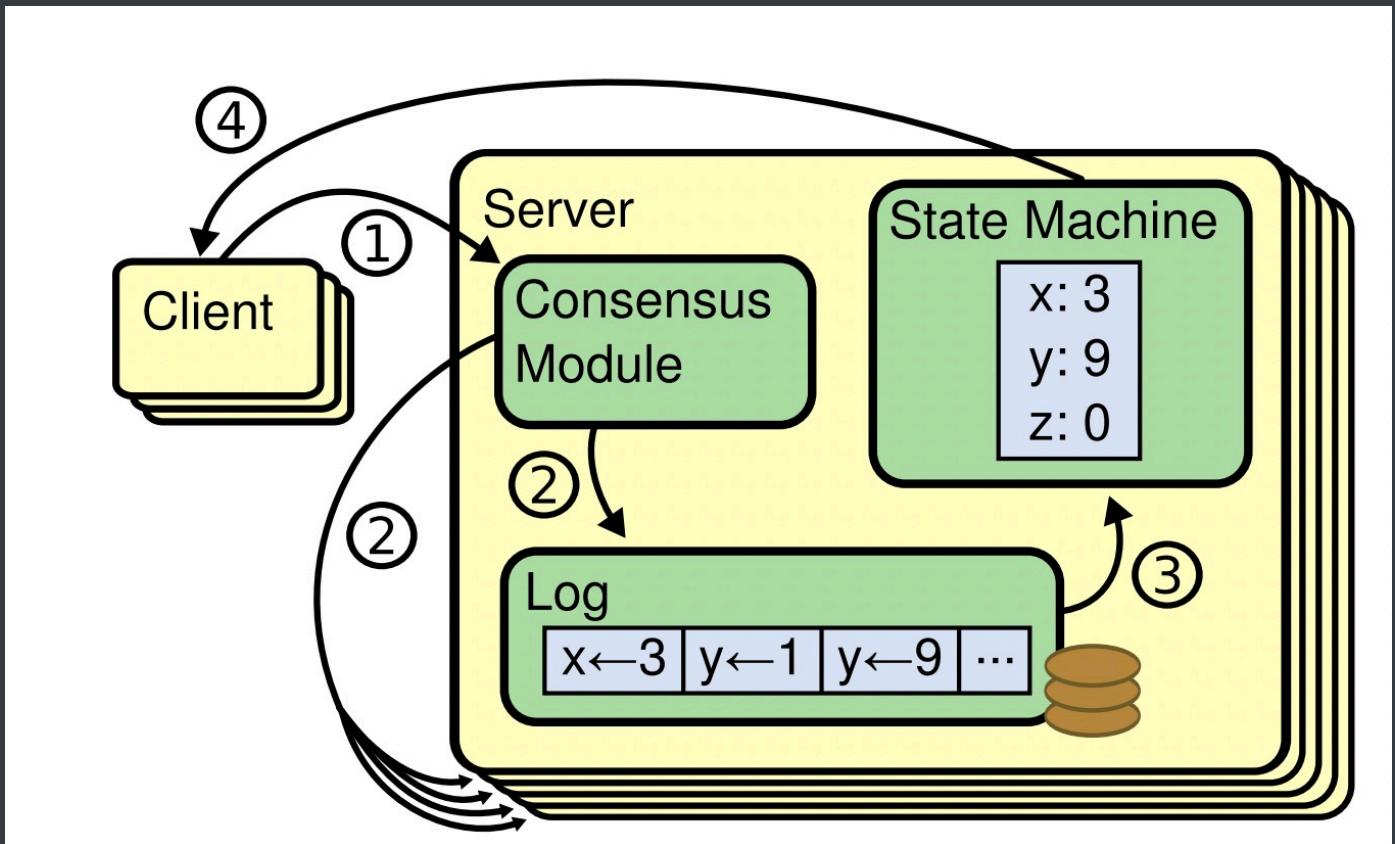
参考文献：

1. [Stanford - 《In Search of an Understandable Consensus Algorithm》](#)
2. [raft.github.io](#)
3. [Raft 动画演示](#)
4. [Raft 脑裂 - leader lease 机制](#)
5. [Raft 应对各种Failover的处理方式](#)
6. [Raft-blog - DaiDai](#)
7. [Jraft - Github](#)

为了令进程实现高可用，可以对进程进行备份，而实现进程的主从备份有两种方法：

- **State Transfer**（状态转移）：主服务器将状态的所有变化都传输给备份服务器
- **Replicated State Machine**（备份状态机）：将需要备份的服务器视为一个确定性状态机 —— 主备以相同的状态启动，以相同顺序导入相同的输入，最后它们就会进入相同的状态、给出相同的输出

其中，Replicated State Machine 是较为常用的主从备份实现方式。



Replicated State Machine 主从复制的流程：

1. 客户端向服务发起请求，执行指定操作
2. 共识模块将该操作以日志的形式备份到其他备份实例上
3. 当日志安全备份后，指定操作被应用于上层状态机
4. 服务返回操作结果至客户端

在 Replicated State Machine 中，分布式共识算法的职责就是按照固定的顺序将指定的日志内容备份到集群的其他实例上。

Overview

角色

一个 Raft 集群由若干个节点组成。节点可能处于以下三种角色的其中之一：

- Leader 负责从客户端处接收新的日志记录，备份到其他服务器上，并在日志安全备份后通知其他服务器将该日志记录应用到位于其上层的状态机上
 - Follower 总是处于被动状态，接收来自 Leader 和 Candidate 的请求，而自身不会发出任何请求
 - Candidate 会在 Leader 选举时负责投票选出 Leader
-

关键问题

在 Leader-Follower 架构的语境下，Raft 将其需要解决的共识问题拆分为了以下 3 个问题：

- Leader 选举：已有 Leader 失效后需要选举出一个新的 Leader
 - 日志备份：Leader 从客户端处接收日志记录，备份到其他服务器上
 - 安全性：如果某个服务器为其上层状态机应用了某个日志记录，那么其他服务器在该 index 值处则不能应用其他不同的日志记录
-

性质

Raft 共识算法的的性质：

- Election Safety（选举安全）：在任意给定的 Term 中，至多一个节点会被选举为 Leader
- Leader Append-Only（Leader 只追加）：Leader 绝不会覆写或删除其所记录的日志，只会追加日志
- Log Matching（日志匹配）：若两份日志在给定 Term 及给定 index 值处有相同的记录，那么两份日志在该位置及之前的所有内容完全一致
- Leader Completeness（Leader 完整性）：若给定日志记录在某一个 Term 中已经被提交（后续会解释何为“提交”），那么后续所有 Term 的 Leader 都将包含该日志记录
- State Machine Safety（状态机安全性）：如果一个服务器在给定 index 值处将某个日志记录应用于其上层状态机，那么其他服务器在该 index 值处都只会应用相同的日志记录

基础结构

在所有服务器上持久存储的(响应RPC之前稳定存储的)

currentTerm	服务器最后知道的任期号(从0开始递增)
votedFor	在当前任期内收到选票的候选人Id(如果没有就为null)
log[]	日志条目, 每个条目包含状态机要执行的命令以及从Leader收到日志时的任期号

在所有服务器上不稳定存在的

commitIndex	已知被提交的最大日志条目索引
lastApplied	已被状态机执行的最大日志条目索引

在Leader服务器上不稳定存在的

nextIndex[]	对于每一个follower, 记录需要发给他的下一条日志条目的索引
matchIndex[]	对于每一个follower, 记录已经复制完成的最大日志条目索引

在 Raft 集群中, 节点间的交互主要由两种 RPC 调用构成。

1 首先是用于日志备份的 AppendEntries: 由leader通过RPC向follower复制日志, 也会用作 heartbeat

入参

term	Leader任期号
leaderId	Leader id, 为了能帮助客户端重定向到Leader服务器
prevLogIndex	在正在备份的日志记录之前日志记录的 index 值
prevLogTerm	在正在备份的日志记录之前日志记录的 Term ID
entries[]	将要存储的日志条目列表(为空时代表heartbeat, 有时候为了效率会发送超过一条)
leaderCommit	Leader 已经提交的最后一条日志记录的 index 值

返回值

term	当前的任期号, 用于 leader 更新自己的任期号
success	如果其他follower包含能够匹配上prevLogIndex和prevLogTerm的日志, 那么为真

接收方follower在接收到该 RPC 后会进行以下操作:

1. 若 `term < currentTerm`, 不接受日志并返回false
2. 如果索引 `prevLogIndex` 处的日志的任期号与 `prevLogTerm` 不匹配, 不接受日志并返回false
3. 如果一条已存在的日志与新的冲突(index相同但是term不同), 则删除已经存在的日志条目和他之后所有的日志条目
4. 在保存的日志后追加新的日志记录
5. 如果 `leaderCommit > commitIndex`, 则设置 `commitIndex = min(leaderCommit, index of last new entry)`

2 而后是用于 Leader 选举的 RequestVote:

入参

term	候选人的任期号
candidateId	发起投票请求的候选人id
lastLogIndex	候选人最新的日志条目索引
lastLogTerm	候选人最新日志条目对应的任期号

返回值

term	目前的任期号, 用于候选人更新自己
voteGranted	如果候选人收到选票, 那么为true

接收方在接收到该 RPC 后会进行以下操作：

1. 如果 `term < currentTerm`, 那么拒绝投票并返回false
2. 如果 `votedFor=null` 或者 `votedFor=candidateId`, 并且候选人的日志和自己一样新或者更新, 那么就给候选人投票并返回true

最后, Raft 集群的节点还需要遵循以下规则:

对于所有节点:

- 若 `commitIndex > lastApplied`, 则对 `lastApplied` 加 1, 并将 `log[lastApplied]` 应用至上层状态机
- 若 RPC 请求或相应内容中携带的 `term > currentTerm`, 则令 `currentTerm = term`, 且 Leader 降级为 Follower

对于 Follower:

- 无条件响应来自candidate和leader的RPC
- 如果在选举超时之前, 没收到任何来自leader的AppendEntries RPC或 RequestVote RPC, 那么自己转换状态为candidate

对于 Candidate:

- 在进入 Candidate 角色时，发起 Leader 选举：
 1. `currentTerm` 加 1
 2. 将选票投给自己
 3. 重置 Election Timeout 计时器
 4. 发送 RequestVote RPC 至其他所有节点
- 如果接收到来自其他大多数节点的选票，则进入 Leader 角色
- 若接收到来自其他 Leader 的 AppendEntries RPC，则进入 Follower 角色
- 若再次 Election Timeout，那么重新发起选举

对于 Leader：

- 在空闲时周期地向 Follower 发起空白的 AppendEntries RPC（作为心跳信息），以避免 Follower 发起选举
- 如果收到来自客户端的请求，向本地日志追加条目并向所有服务器发送 AppendEntries RPC，在收到大多数响应后将该条目应用到状态机并回复响应给客户端
- 如果 Leader 最新一条日志记录的 `index` 值大于等于某个 Follower 的 `nextIndex` 值，则通过 AppendEntries RPC 发送在该 `nextIndex` 值之后的所有日志记录：
 1. 如果备份成功，那么就更新该 Follower 对应的 `nextIndex` 和 `matchIndex` 值
 2. 如果备份失败，对 `nextIndex` 减 1 并重试
- 如果存在一个值 `N`，使得 `N > commitIndex`，且大多数的 `matchIndex[i] >= N`，且 `log[N].term == currentTerm`，令 `commitIndex = N`

Leader Election

Term 定义

Raft 算法在运行时会把时间分为任意长度的 Term

- Term 由单调递增的 Term ID 所标识，每个节点都会在内存中保存当前 Term 的 ID。
- 每个 Term 的开头都会包含一次 Leader 选举，在选举中胜出的节点会担当该 Term 的 Leader。

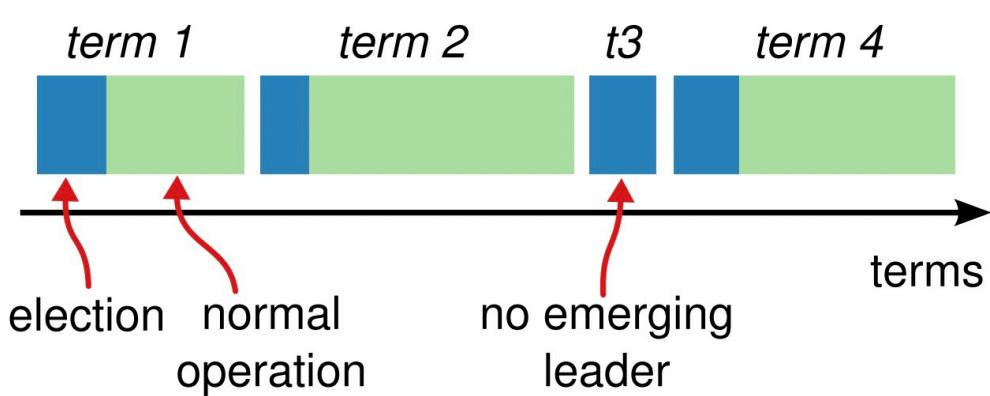


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different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

节点角色转变

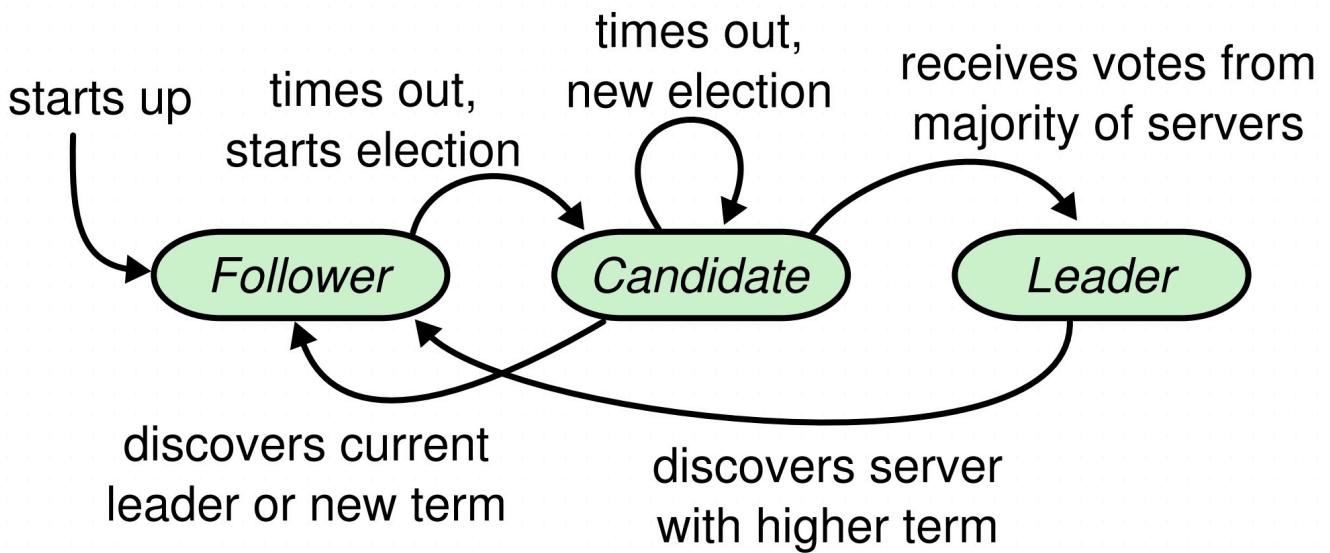


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

Raft 节点角色变化：

1. 在初次启动时，节点首先会进入 Follower 角色。只要它能够一直收到来自其他 Leader 节点发来的 RPC 请求，它就会一直处于 Follower 状态。
2. 如果接收不到来自 Leader 的通信，Follower 会等待一个称为 Election Timeout（选举超时）的超时时间，然后便会开始发起新一轮选举。Follower 发起选举时会对自己存储的 Term ID 进行自增，并进入 Candidate 状态。随后，它会将自己的一票投给自己，并向其他节点并行地发出 RequestVote RPC 请求。
3. 其他节点在接收到该类 RPC 请求时，会以先到先得的原则投出自己在该 Term 中的一票。
4. 当 Candidate 在某个 Term 接收到来自集群中大多数节点发来的投票时，它便会成为 Leader，然后它便会向其他节点进行通信，确保其他节点知悉它是 Leader 而不会发起新一轮投票。
5. 每个节点在指定 Term 内只会投出一票，而只有接收到大多数节点发来的投票才能成为 Leader 的性质确保了在任意 Term 内都至多会有一个 Leader。由此我们实现了前面提及的 Eleaction Safety 性质。
6. Candidate 在投票过程中也有可能收到来自其他 Leader 的 AppendEntries RPC 调用，这意味着有其他节点成为了该 Term 的 Leader。如果该 RPC 中携带的 Term ID 大于等于

Candidate 当前保存的 Term ID，那么 Candidate 便会认可其为 Leader，并进入 Follower 状态，否则它会拒绝该 RPC 并继续保持其 Candidate 身份。

除此之外，选举也有可能发生平局的情况：若干节点在短时间内同时发起选举，导致集群中没有任何一个节点能够收到来自集群大多数节点的投票。

- 此时，节点同样会在等待 Election Timeout 后发起新一轮的选举，但如果加入额外的应对机制，这样的情况有可能持续发生。
- 为此，Raft 为 Election Timeout 的取值引入了随机机制：
 - 节点在进入新的 Term 时，会在一个固定的区间内（如 150~300ms）随机选取自己在该 Term 所使用的 Election Timeout。
 - 通过随机化来错开各个节点进入 Candidate 状态的时机便能有效避免这种情况的重复发生。

Raft 共识算法的性质 - 相关证明：

Election Safety (选举安全)：在任意给定的 Term 中，至多一个节点会被选举为 Leader

证明：

- term 任期是一个分布式逻辑时钟，一个 voter 在一个 term 下最多给一个 leader 投票，candidate 在收到过半选票时才会晋升 leader。
- 反证法：
 - 假设存在两个 leader
 - 则说明一个集群中至少一个节点，分别给两个 leader 进行了投票
 - 已知每个节点最多投一票
 - 与已知矛盾，多个 leader 同时存在不成立

Log Replication

在选举出一个 Leader 后，Leader 便能够开始响应来自客户端的请求了，客户端请求由需要状态机执行的命令所组成。

1. Leader 会将接收到的命令以日志记录的形式追加到自己的记录里，并通过 AppendEntries RPC 备份到其他节点上；
2. 当日志记录被安全备份后，Leader 就会将该命令应用于位于自己上层的状态机，并向客户端返回响应；
3. 无论 Leader 已响应客户端与否，Leader 都会不断重试 AppendEntries RPC 调用，直到所有节点都持有该日志记录的备份。

日志由若干日志记录组成：

- 每条记录中都包含一个需要在状态机上执行的命令，以及其对应的 index 值；
- 日志记录还会记录自己所属的 Term ID。

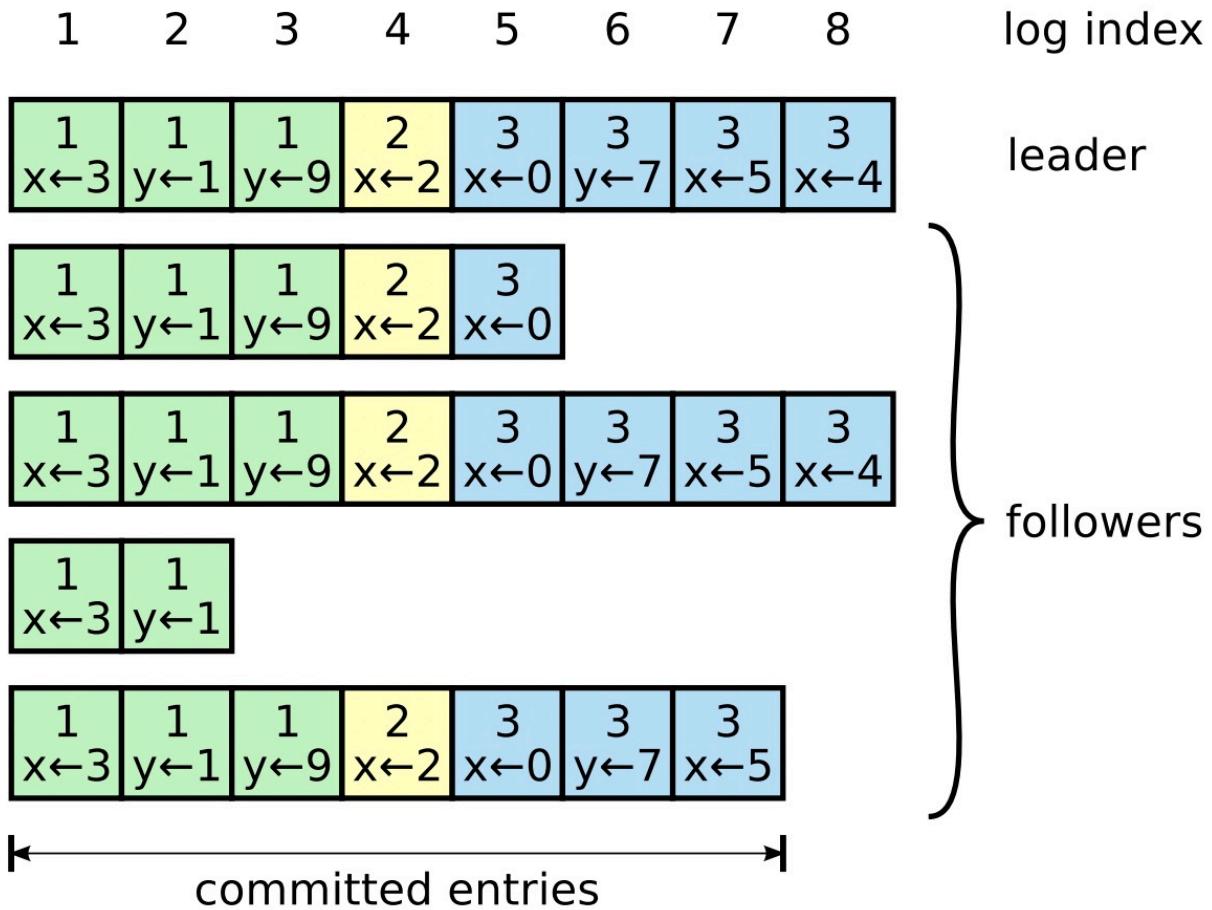


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

当某个日志记录顺利备份到集群大多数节点上后，Leader 便会认为该日志记录“已提交”（Committed），即该日志记录已可被安全的应用到上层状态机上。

- Raft 保证一个日志记录一旦被提交，那么它最终就会被所有仍可用的状态机所应用。
- 日志记录的提交也意味着位于其之前的所有日志记录也进入“已提交”状态。
- Leader 会保存其已知的最新的已提交日志的 index 值，并在每次进行 AppendEntries RPC 调用时附带该信息；
- Follower 在接收到该信息后即可将对应日志记录应用在位于其上层的状态机上。

Raft 共识算法的性质 - 相关证明：

Log Matching (日志匹配) : 若两份日志在给定 Term 及给定 index 值处有相同的记录，那么两份日志在该位置及之前的所有内容完全一致

证明：

- 对于两份日志中给定的 *index* 处，如果该处两个日志记录的 *Term ID* 相同，那么它们存储的状态机命令相同。
 - 考虑到 Leader 在一个 Term 中只会在一个 *index* 处创建一条日志记录，而且日志的位置不会发生改变。
- 如果两份日志中给定 *index* 处的日志记录有相同的 *Term ID* 值，那么位于它们之前的所有日志记录完全相同。
 - Leader 在进行 AppendEntries RPC 调用时，会同时携带在其自身的日志存储中位于该新日志记录之前的所有日志记录的 *index* 值及 *Term ID*；
 - 如果 Follower 在自己的日志存储中没有找到这条日志记录，那么 Follower 就会拒绝这条新记录。
 - 由此，每一次 AppendEntries RPC 调用的成功返回都意味着 Leader 可以确定该 Follower 存储的日志直到该 *index* 处均与自己所存储的日志相同。
 - 数学归纳法进行证明

AppendEntries RPC 的日志一致性检查是必要的，因为 Leader 的崩溃会导致新 Leader 存储的日志可能和 Follower 不一致。

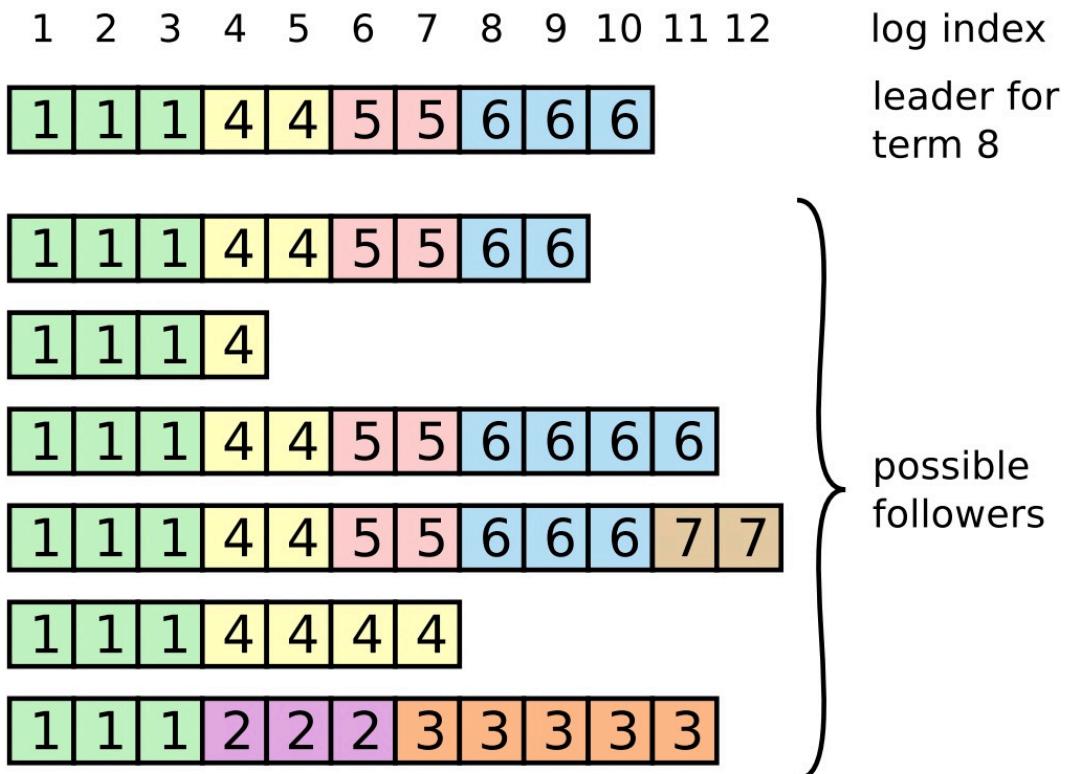


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example,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

Raft 共识算法的性质 - 相关证明：

Leader Append-Only (Leader 只追加) : Leader 绝不会覆写或删除其所记录的日志，只会追加日志

证明：对于不一致的 Follower 日志，Raft 会强制要求 Follower 与 Leader 的日志保持一致。

1. Leader 会为每个 Follower 维持一个 `nextIndex` 变量，代表 Leader 即将通过 AppendEntries RPC 调用发往该 Follower 的日志的 index 值
 2. 在刚刚被选举为一个 Leader 时，Leader 会将每个 Follower 的 `nextIndex` 置为其所保存的最新日志记录的 index 之后
 3. 当有 Follower 的日志与 Leader 不一致时，Leader 的 AppendEntries RPC 调用会失败，Leader 便对该 Follower 的 `nextIndex` 值减 1 并重试，直到 AppendEntries 成功
 4. Follower 接收到合法的 AppendEntries 后，便会移除其在该位置上及以后存储的日志记录，并追加上新的日志记录
 5. 如此，在 AppendEntries 调用成功后，Follower 便会在该 Term 接下来的时间里与 Leader 保持一致
-

Security

Leader 选举限制

在集群运行的过程中，某个 Follower 可能会失效，而 Leader 继续在集群中提交日志记录；当这个 Follower 恢复后，有可能会被选举为 Leader，而它实际上缺少了一些已经提交的日志记录。

- 其他的基于 Leader 架构的共识算法都会**保证 Leader 最终会持有所有已提交的日志记录。**
- 一些算法（如 Viewstamped Replication）允许节点在不持有所有已提交日志记录的情况下被选举为 Leader，并通过其他机制将缺失的日志记录发送至新 Leader。而这种机制实际上会为算法引入额外的复杂度。
- 为了简化算法，Raft 限制了日志记录只会从 Leader 流向 Follower，
- 同时 Raft 中，Leader 绝不会覆写它所保存的日志。

在这样的前提下，要提供相同的保证，Raft 就需要限制哪些 Candidate 可以成为 Leader。

- Candidate 为了成为 Leader 需要获得集群内大多数节点的选票，而一个日志记录被提交同样要求它已经被备份到集群内的大多数节点上，那么如果一个 Candidate 能够成为 Leader，投票给它的节点中必然存在节点保存有所有已提交的日志记录。
- Candidate 在发送 RequestVote RPC 调用进行拉票时，它还会附带上自己的日志中最后一条记录的 index 值和 Term ID 值：
 - 其他节点在接收到后会与自己的日志进行比较，如果发现对方的日志落后于自己的日志（首先由 Term ID 决定大小，在 Term ID 相同时由 index 决定大小），就会拒绝这次 RPC 调用。

如此一来，Raft 就能确保被选举为 Leader 的节点必然包含所有已经提交的日志。

来自旧 Term 的日志记录

Leader 在备份当前 Term 的日志记录时，在成功备份至集群大多数节点上后 Leader 即可认为该日志记录已提交。

但如果 Leader 在日志记录备份至大多数节点之前就崩溃了，后续的 Leader 会尝试继续备份该日志。

然而，此时的 Leader 即使在将该日志备份至大多数节点上后都无法立刻得出该日志已提交的结论。

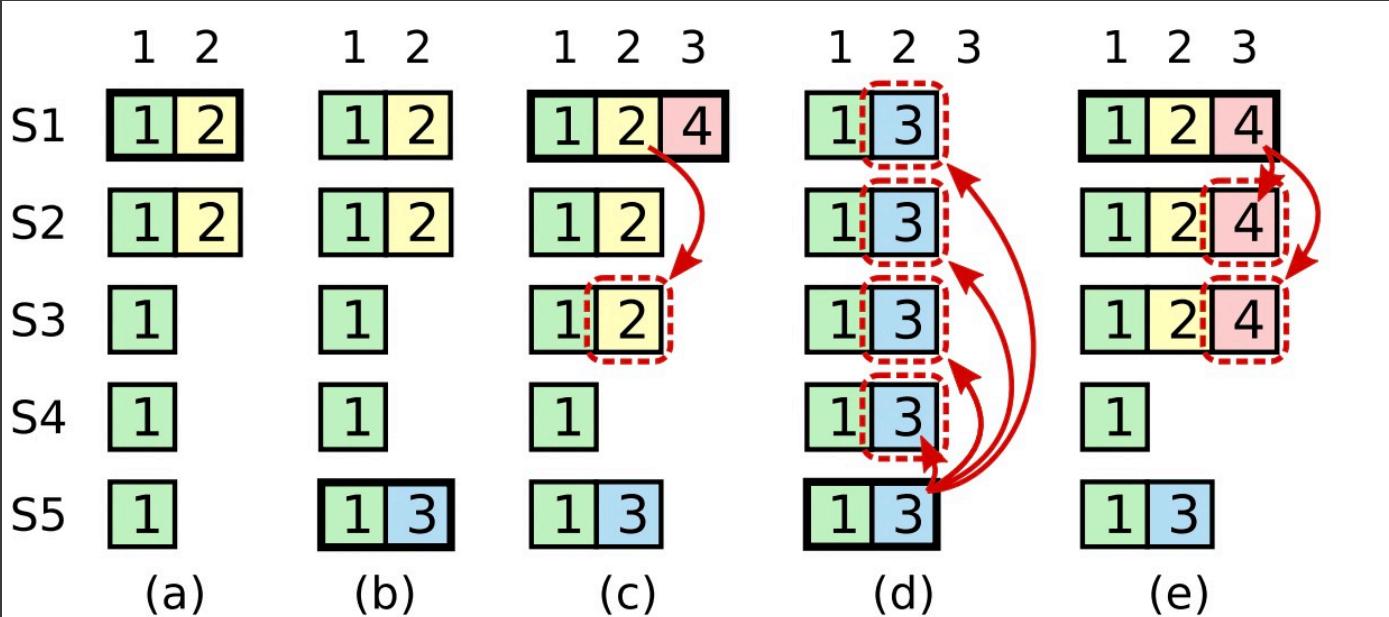


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at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

- 在时间点 (a) 时, S1 是 Leader, 并把 $(\text{TermID}=2, \text{index}=2)$ 的日志记录备份到了 S2 上。
- 到了时间点 (b) 时, S1 崩溃, S5 收到 S3、S4、S5 的选票, 被选为 Leader, 并从客户端处接收到日志记录 $(\text{TermID}=3, \text{index}=2)$ 。
- 在时间点 (c) 时, S5 崩溃, S1 重启, 被选举为 Leader, 并继续将先前没有备份的日志记录 $(\text{TermID}=2, \text{index}=2)$ 备份到其他节点上。即便此时 S1 顺利把该日志记录备份到集

群大多数节点上，它仍然不能认为该日志记录已被安全提交。考虑此时 S1 崩溃，S5 将可以收到来自 S2、S3、S4、S5 的选票，成为 Leader（其最后一个日志记录的 Term ID 是 3，大于 2）

- 进入情形 (d)：此时 S5 会继续把日志记录 (TermID=3, index=2) 备份到其他节点上，覆盖掉原本已经备份至大多数节点的日志记录 (TermID=2, index=2)。
- 然而，如果在时间点 (c) S1 成为 Leader 后，同样将当前 Term 的最新日志记录 (TermID=4, index=3) 备份出去并提交，就会进入情形 (e)，此时 S5 便无法再被选举为 Leader。
- 因此，解决该问题的关键在于在备份旧 Term 的日志时也要把当前 Term 最新的日志一并分发出去。

由此，Raft 只会在备份当前 Term 的日志记录时才会通过计数的方式来判断该日志记录是否已被提交；一旦该日志记录完成提交，根据前面提及的 Log Matching 性质，Leader 就能得出之前的日志记录也已被提交。由此，我们便实现了前面提及的 Leader Completeness 性质。文中 5.4.3 节有完整的证明过程，感兴趣的读者可自行查阅。

证得前面 4 条性质后，最后一条 State Machine Safety 性质也可证得：当节点将日志记录应用于其上层状态机时，该日志记录及其之前的所有日志记录必然已经提交。某些节点执行命令的进度可能落后，我们考虑所有节点目前已执行日志记录的 index 值的最小值：Log Completeness 性质保证了未来的所有 Leader 都会持有该日志记录，因此在之后的 Term 中其他节点应用位于该 index 处的日志记录时，该日志记录保存的必然是相同的命令。由此，上层状态机只要按照 Raft 日志记录的 index 值顺序执行命令即可安全完成状态备份。

时序要求

为了提供合理的可用性，集群仍需满足一定的时序要求，具体如下：

broadcastTime << electionTimeout << MTBF

broadcastTime	一台服务器并行的向集群中其他节点发送RPC并且收到它们响应的平均时间
electionTimeout	选举的超时时间
MTBF	是指单个服务器发生故障的时间间隔的平均数

- broadcastTime 应该比 electionTime 小一个数量级，目的是让 leader 能够持续发送心跳来阻止 follower 们开始选举；根据已有的随机化超时前提，这个不等式也将瓜分选票的可能性降低到很小
- electionTimeout 也要比 MTBF 小一个几个数量级，目的是能使系统稳定运行，当 leader 崩溃时，整个集群大约会在 electionTimeout 的时间内不可用

上述的不等式要求：

- broadcastTime 应该比 electionTime 小一个数量级，目的是让 leader 能够持续发送心跳来阻止 follower 们开始选举；根据已有的随机化超时前提，这个不等式也将瓜分选票的可能性降低到很小
- electionTimeout 也要比 MTBF 小一个几个数量级，目的是能使系统稳定运行，当 leader 崩溃时，整个集群大约会在 electionTimeout 的时间内不可用

在这个不等式中，broadcastTime 及 MTBF 由集群架构所决定，electionTimeout 则可由运维人员自行配置。

Candidate 与 Follower 崩溃

目前来讲我们讨论都是 Leader 失效的问题。

对于 Candidate 和 Follower 而言，它们分别是 RequestVote 和 AppendEntries RPC 调用的接收方：

- 当 Candidate 或 Follower 崩溃后，RPC 调用会失败；Raft RPC 失败时会不断重试 RPC，直至 RPC 成功；
 - 除外，RPC 调用也有可能已经生效，但接收方在响应前就已失效，为此 Raft 保证 RPC 的幂等性，在节点重启后收到重复的 RPC 调用也不会有所影响。
-

集群成员变更

系统总是可能需要做出变更，例如移除一些节点或增加一些节点。

当然，集群可以被全部关闭后，调整配置文件，再全部重启，这样也能完成集群配置变更，但这样会导致系统出现一段时间的不可用。而 Raft 则引入了额外的机制来允许集群在运行中变

更自己的成员配置。

在进行配置变更时，直接从旧配置切换至新配置是不可行的，源于不同的节点不可能原子地完成配置切换，而这之间可能会有一些时间间隙使得集群存在两个不同的“大多数”。

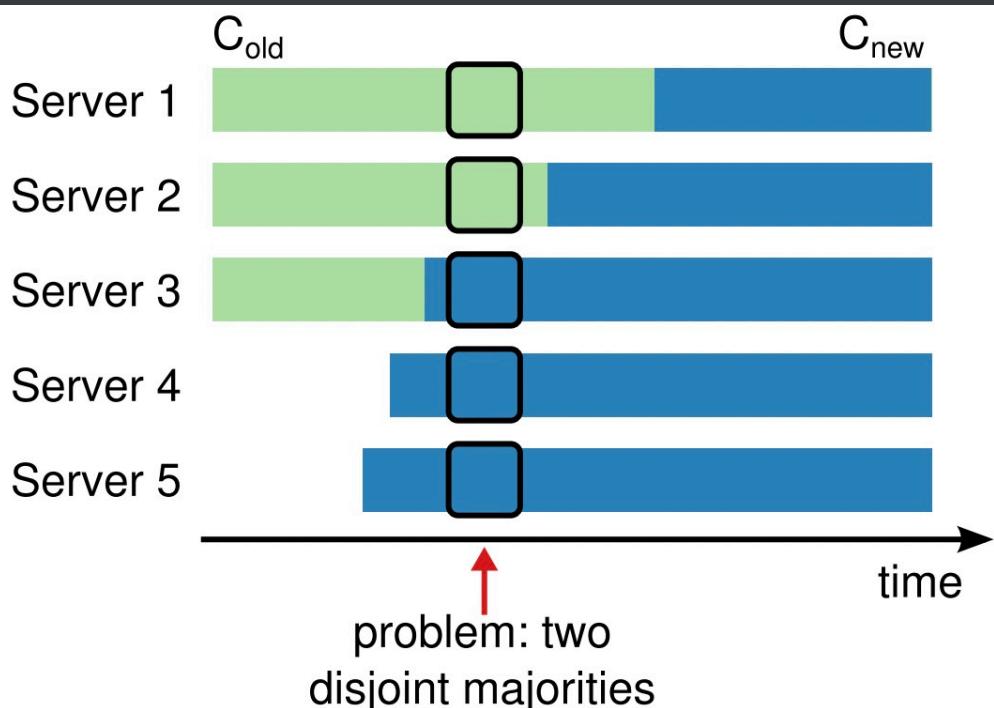


Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

如上图所示，集群逐渐地从旧配置切换至新配置，那么在箭头标记的位置就出现了两个不同的“大多数”：

- S1、S2 构成 C_{old} 的大多数，S3、S4、S5 构成 C_{new} 的大多数。
- 在这一时间间隔内，处于两个配置的节点可能会选出各自的 Leader，引入 Split-Brain 问题。
- 问题的关键在于，在这段时间间隔中， C_{old} 和 C_{new} 都能够独立地做出决定。

为了解决这个问题，Raft 采用二阶段的方式来完成配置切换：在 C_{old} 与 C_{new} 之间，引入一个被称为 Joint Consensus 的特殊配置 $C_{old,new}$ 作为迁移状态。该配置有如下性质：

- 日志记录会被备份到 C_{old} 及 C_{new} 的节点上
- 两份配置中的任意机器都能成为 Leader
- 选举或提交日志记录要求得到来自 C_{old} 和 C_{new} 的两个不同的“大多数”的同意

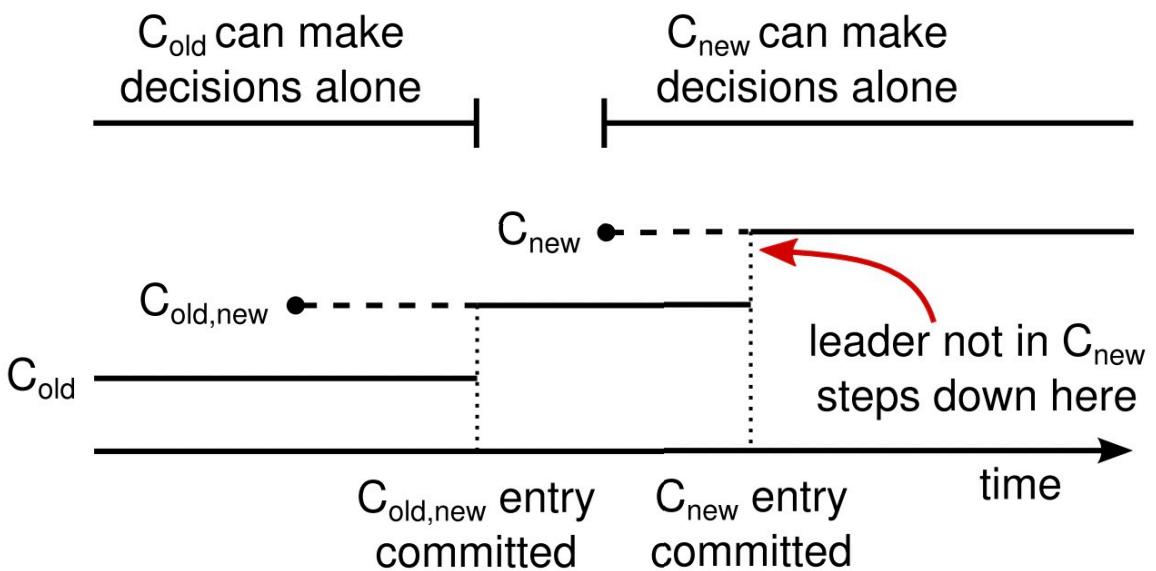


Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a 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.

上图显示了配置切换的时序，其中可以看到不存在 C_{old} 和 C_{new} 都能独立作出决定的时间段。

1. 首先对新节点进行CaughtUp追数据
2. 全部新节点完成CaughtUp之后，向新老集合发送Cold+new命令
 1. 对于这种特殊的配置切换日志，节点在接收到时就会立刻切换配置，不会等待日志提

交,

3. 如果新节点集合多数和老节点集合多数都应答了Cold+new, 就向新老节点集合发送Cnew命令
4. 如果新节点集合多数应答了Cnew, 完成节点切换

集群成员变更带来的其他的问题:

1 首先, 配置变更可能会引入新的节点, 这些节点不包含之前的日志记录, 完成日志备份可能会需要较长的时间, 而这段时间可能导致集群无法提交日志, 引入一段时间的服务不可用。

- 为此, Raft 在节点变更配置之前还引入了一个额外的阶段: 此时节点会以不投票成员的形式加入集群, 开始备份日志, Leader 在计算“大多数”时也不会考虑它们; 等到它们完成备份后, 它们就能回到正常状态, 完成配置切换。

2 此外, 集群的 Leader 有可能不属于 Cnew。在这种情况下, Leader 在完成 Cnew 的配置变更日志提交后才能变更自己的配置并关闭。也就是说, 在 Leader 提交 Cnew 的日志时, 那段时间里它会需要管理一个不包含自己的集群:

- 它会把日志记录备份出去, 但不会把自己算入“大多数”。
- 直到 Cnew 日志完成提交, Cnew 才能够独立做出决定, 才能够在原 Leader 降级后在 Cnew 集群中选出新的 Leader;
- 在那之前, 来自 Cold 的节点有可能被选为 Leader。

3 最后, 那些从集群中被移除出去的节点可能会在配置切换完成后干扰新集群的运行。这些节点不会再接收到 Leader 的心跳, 于是它们就会超时并发起选举。这时它们会发起 RequestVote RPC 调用, 其中包含新的 Term ID, 而这可能导致新的 Leader 自动降级为 Follower, 导致服务不可用。最终新集群会选出一个新的 Leader, 但被移除的节点依旧不会接收到心跳信息, 它们会再次超时, 再次发起选举, 如此循环往复。

- 为解决此问题, 节点在其“确信” Leader 仍存活时会拒绝 RequestVote RPC 调用: 如果距离节点上一次接收到 Leader 心跳信息过去的时间小于 Election Timeout 的最小值, 那么节点便会“确信” Leader 仍然存活。考虑前面提到的时序要求, 这确实能够在大多数情况下避免该问题。

日志压缩

随着 Raft 集群的不断运行，节点上的日志体积会不断增大，这会逐渐占用节点的磁盘资源，此外过长的日志也会延长节点重放日志的耗时，引入服务可用性问题。为此，集群需要对过往的日志进行压缩。

快照是进行日志压缩最简便的方案。

- 在进行快照时，状态机当前的完整状态会被写入到持久存储中，而后就能够安全地把直到该时间点以前的日志记录移除了。
- 完成快照后，Raft 也会在快照中记录其所覆盖到的最新日志记录的 index 和 Term ID，以便后面的日志记录能够被继续追加。
- 为了兼容前面提到的集群成员配置变更，快照同样需要记录下当前的集群成员配置。

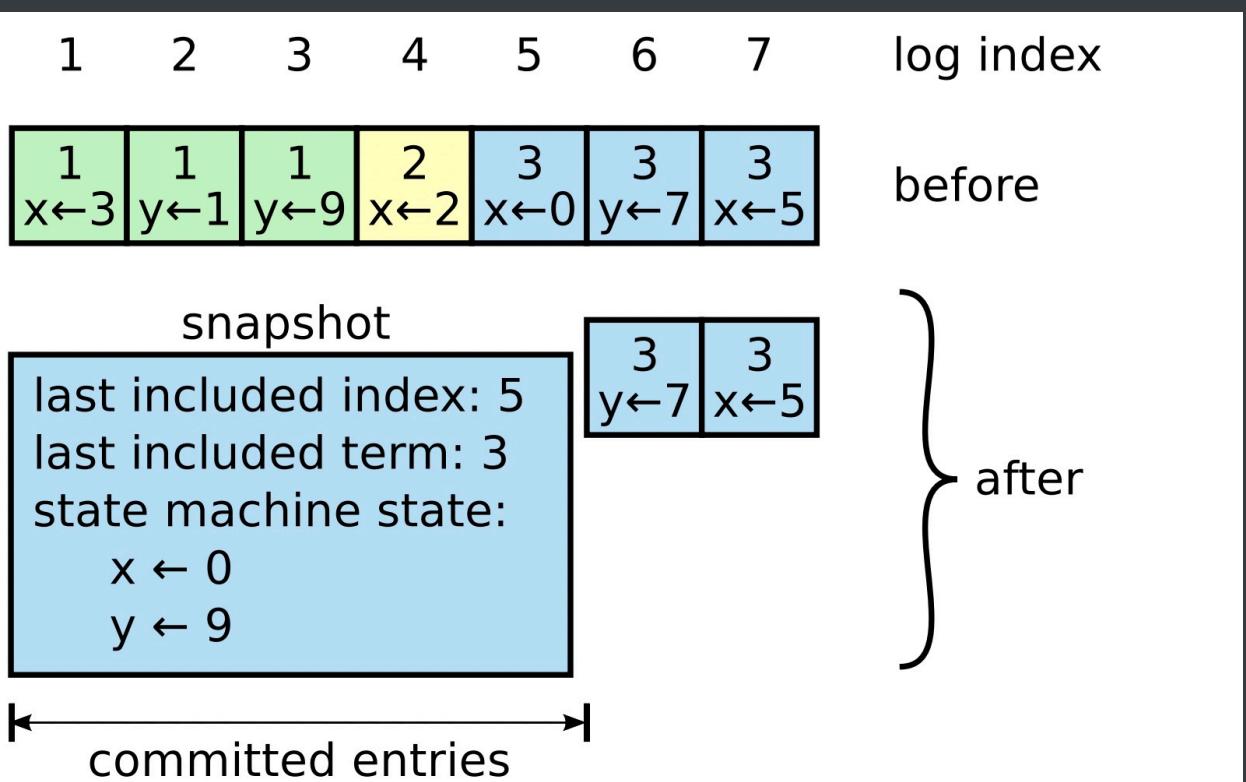


Figure 12: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables x and y in this example). The snapshot's last included index and term serve to position the snapshot in the log preceding entry 6.

对于 Raft 来说，每个节点会独立地生成快照。相比起只由 Leader 生成快照，这样的设计避免了 Leader 频繁向其他 Follower 传输快照而占用网络带宽，况且 Follower 也持有着足以独立生成快照的数据。

尽管如此，当某个 Follower 落后太多或是集群加入了新节点时，Leader 仍然会需要将自己持有的快照传输给 Follower。为此，Raft 提供了专门的 InstallSnapshot RPC 接口。

InstallSnapshot RPC: 由 Leader 进行调用，用于将组成快照的文件块发给指定 Follower。Leader 总会按照顺序发送文件块。

参数：

term	leader任期
leaderId	Leader id, 为了能帮助客户端重定向到Leader服务器
lastIncludedIndex	快照中包含的最后日志条目的索引值
lastIncludedTerm	快照中包含的最后日志条目的任期号
offset	当前发送的文件块在整个快照文件中的偏移值
data[]	快照块的原始数据
done	如果是最后一块数据则为真

Follower 在接收到该 RPC 调用后会进行以下操作：

1. 若 `term < currentTerm` , `return`
2. 若 `offset == 0` , 创建快照文件
3. 在快照文件的指定 `offset` 处写入 `data[]`
4. 若 `done == false` , 返回响应，并继续等待其他文件块
5. 移除已有的或正在生成的在该快照之前的快照
6. 如果有一个日志记录的 Term ID 及 index 值与该快照所包含的最后一个日志记录相同，那么便保留该日志记录之后的日志记录，并返回响应
7. 移除被该快照覆盖的所有日志记录
8. 使用该快照的内容重置上层状态机，并载入快照所携带的集群成员配置

客户端交互

最后，我们再来聊聊客户端如何与 Raft 集群进行交互。

首先，客户端需要能够得知目前 Raft 集群的 Leader 是谁。在一开始，客户端会与集群中任意的一个节点进行通信：如果该节点不是 Leader，那么它会把上一次接收到的 AppendEntries RPC 调用中携带的 Leader ID 返回给客户端；如果客户端无法连接至该节点，那么客户端就会再次随机选取一个节点进行重试。

Raft 的目标之一是为上层状态机提供日志记录的 exactly-once 语义，但如果 Leader 在完成日志提交后、向客户端返回响应之前崩溃，客户端就会重试发送该日志记录。为此，客户端需要为自己的每一次通信赋予独有的序列号，而上层状态机则需要为每个客户端记录其上一次通信所携带的序列号以及对应的响应内容，如此一来当收到重复的调用时状态机便可在不执行该命令的情况下返回响应。

对于客户端的只读请求，Raft 集群可以在不对日志进行任何写入的情况下返回响应。然而，这有可能让客户端读取到过时的数据，源于当前与客户端通信的“Leader”可能已经不是集群的实际 Leader，而它自己并不知情。

为了解决此问题，Raft 必须提供两个保证。首先，Leader 持有关于哪个日志记录已经成功提交的最新信息。基于前面提到的 Leader Completeness 性质，节点在成为 Leader 后会立刻添加一个空白的 no-op 日志记录；此外，Leader 还需要知道自己是否已经需要降级，为此 Leader 在处理只读请求前需要先与集群大多数节点完成心跳通信，以确保自己仍是集群的实际 Leader。

结语

至此，本文已对 Raft 论文的内容进行了完整的总结。总体而言，Raft 的论文为 Raft 提供了很详实的介绍，论文各处的 API Specification 也为他人实现 Raft 算法提供了很好的基础。Raft 算法也存在着一些在论文中也没有提及的细节及优化方式，有机会的话我会在新的博文中介绍这部分的内容，敬请期待。