Raft 类型与 Multi-Paxos 相似，其有几个新特性：

- Strong leader：在 Raft 中，日志条目只能从 leader 流向其他服务器。
- Leader 选举：Raft 使用随机计时器进行 leader 选举，通过在心跳机制上增加少量机制即可。
- 集群成员变更：Raft 使用了一种新的联合一致性的方法，其他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

复制状态机

一致性算法是在复制状态机的背景下产生的。在这种方法中，一组服务器上的状态机执行相同的日志序列，并且即使某些服务器宕机，也可以继续运行

复制状态机使用复制日志实现。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

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
- 只要集群中过半服务器可以运行，并且可以相互通信和与客户同学呢，一致性算法就可用，假如服务器宕机，它们可以稍后从状态恢复并重新加入集群
- 它们不依赖时序来确保日志的一致性：错误的时钟和极端消息延迟可能在最坏的情况下导致可用性问题
- 通常，只要集群中过半服务器已经响应了单轮 RPC，命令就可以完成，其他慢的服务器不会影响整个系统性能

为可理解性而设计

Raft 算法使用了两种通用的技术来解决这个问题，一个主要是被分成一些模块来处理，比如 leader 选举、日志复制、安全性和成员变更几个部分。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。特别是，所有的日志是不允许有空隙的，并且 Raft 限制了使日志之间不一致的方式

Raft 一致性算法

下面是 Raft 中的节点属性：

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).

Raft 的关键特性（不包括集群成员变更和日志压缩）：

- 选举安全：在给定的任期最多只能有一个 leader
- 只有 leader 能添加条目：leader 永远不会覆盖或者删除自己的日志，它只会增加条目
- 日志匹配原则：如果两个日志在相同的索引位置上的日志条目的任期号相同，那么我们就认为这个日志从开始到这个索引位置之间的条目完全相同
- leader completeness：**如果一个日志条目在一个给定的任期内被提交，那么这个条目一定会出现在所有任期号更大的 leader 中**
- 状态机安全原则：**如果一个服务器已经将给定索引位置的日志条目应用到状态机中，则所有其他服务器不会在该索引位置应用不同的条目**

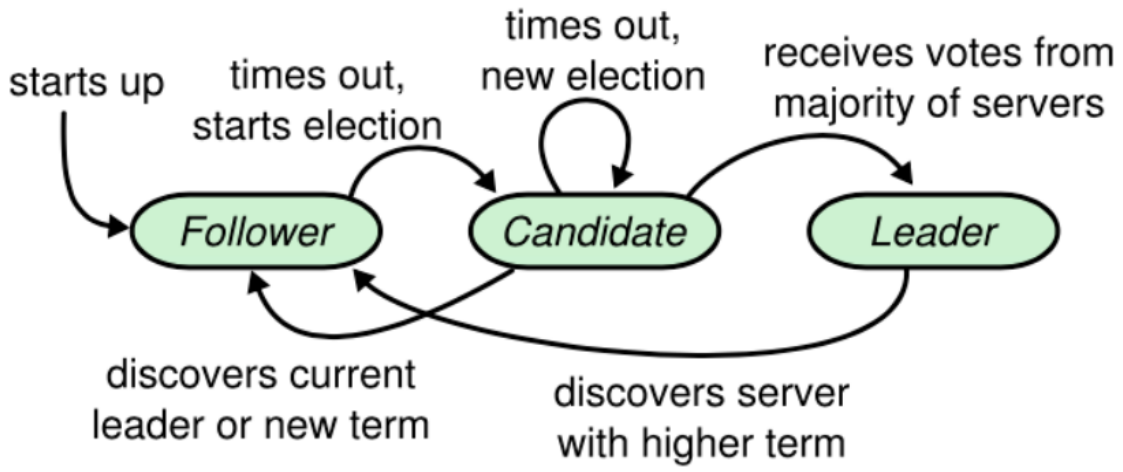
Raft 会选举出一个 leader，leader 从客户端接收日志条目，并把日志复制到其他服务器上，并且在保证安全性的同时通知其他服务器将日志条目应用到它们的状态机中。日志都是从 leader 流向其他服务器。leader 可能宕机，也可能和其他服务器断开连接，此时会选举出一个新的 leader

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：

- **leader 选举：当前的 leader 宕机时，一个新的 leader 必须被选举出来**
- 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 中安全性的关键是状态机安全原则，即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应用一条不同的指令

Raft 基础

一个 Raft 集群包含若干服务器节点；通常是 5 个，每个节点都处于这三个状态之一：leader、follower 或 candidate。正常情况下集群中只有一个 leader，其余全是 follower，follower 不会发送任何请求，其只是简单响应来自 leader 和 candidate 的请求。leader 处理所有的客户端请求（如果客户端和 follower 通信，则 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，下面是这三个状态的转换图：



Raft 把时间分割成一些任期，当 follower 计时器过期时，其会转变为 candidate，自己会投自己一票，然后向集群中发送 requestVote RPC，若收到过半节点的选票，则其成为 leader。为了防止不停地瓜分选票的情况出现，计时器每次的过期时长都是随机的。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，一个新的任期会很快重新开始

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。如果一个节点收到包含过期的任期号的请求，它会直接拒绝这个请求

Leader 选举

Raft 使用心跳机制来触发 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 收到有效的 RPC 就一直保持 follower 状态。Leader 周期性地向所有 follower 发送心跳来维持自己的地位

要开始一次选举，follower 先增加自己的任期号并且转换到 candidate 状态。然后投票给自己并且并行地向集群中发送 RequestVote RPC，Candidate 会一直保持当前状态直到以下三件事情之一发生：

- 它自己赢得了选举成为 leader

- 其他的服务器节点成为 leader
- 一段时间之后没有任何 leader 产生

Candidate 在收到同个任期的 Candidate 的 RPC 时，其会忽略，因为 Candidate 首先把票投给自己

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，若 candidate 收到了 leader 服务器节点发来的 RPC 消息，如果这个 leader 的任期号不小于 candidate 的任期号，则 candidate 就会承认该 leader 的合法地位并变成 follower，如果任期号比自己的小，那么 candidate 就会拒绝这次的 RPC 并且继续保持 candidate 状态

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 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，leader 会应用该条目到它的状态机中，然后把执行的结果返回给客户端。

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

Raft 保证所有**已提交的日志条目**都是持久化的并且最终会被可用的状态机执行。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。**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**（其他 leader 可能只是把这个日志加到 follower 的条目中，但是并没有执行，但可以证明的是，新的 leader 一定会包含这些日志，所以可以将之前的日志进行提交）。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

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，同时该日志条目也从来不会改变。这点保证了第一条特性。第二个特性是由 AppendEntries RPC 执行一个简单的一致性检测保证的，在发送 AppendEntries RPC 时，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 找不到包含相同索引位置的任期号和条目，他就会拒绝该新的日志条目，然后通过返回一些信息给 leader，leader 收到信息后，在第二次 RPC 中加入新的索引位置和任期号，直到匹配为止，若匹配，则将 leader 在该位置后的所有日志条目复制到 follower 的日志条目中。即通过强制 follower 复制它的日志来解决不一致的问题。

具体的，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。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。

这个方法可以进行一个优化，就是不是每次只往后退一个位置，而是每次退一个任期，即 follower 包含冲突条目的任期号和自己存储那个任期的第一个 Index，leader 可以跳过那个任期内所有冲突的日志条目来减小 nextIndex。在实践中，我们认为这种优化是没有必要的，因为失败不经常发送而且也不可能有很多不一致的日志条目

安全性

目前为止我们还不能保证每一个状态机会按照相同的顺序执行相同的指令。这里通过对 leader 选举添加一个限制来完善 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

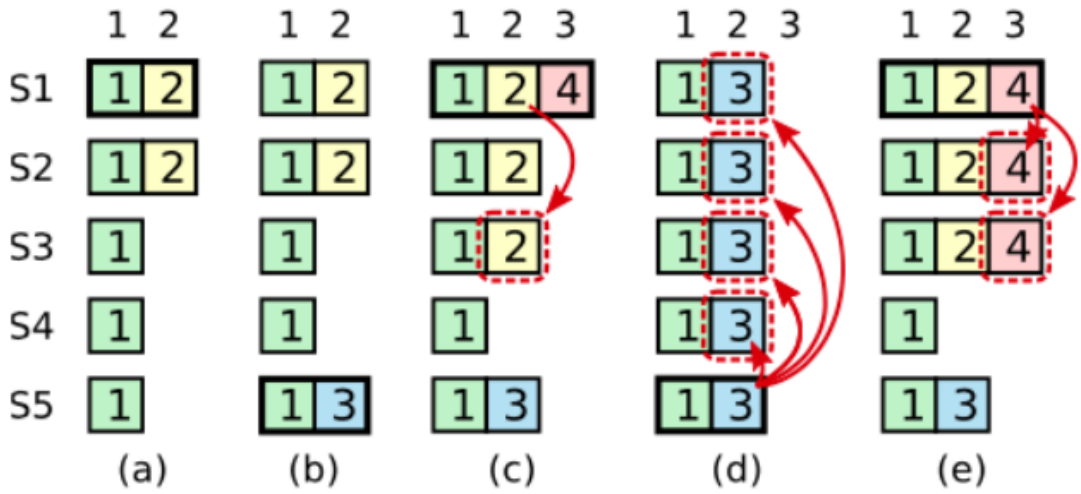
选举限制

如果 candidate 的日志至少和过半的服务器节点一样新，那么他一定包含了所有已经提交的日志条目，RequestVote RPC 执行了这一的限制：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

禁止提交不是当前任期内的日志条目

以下图为例：

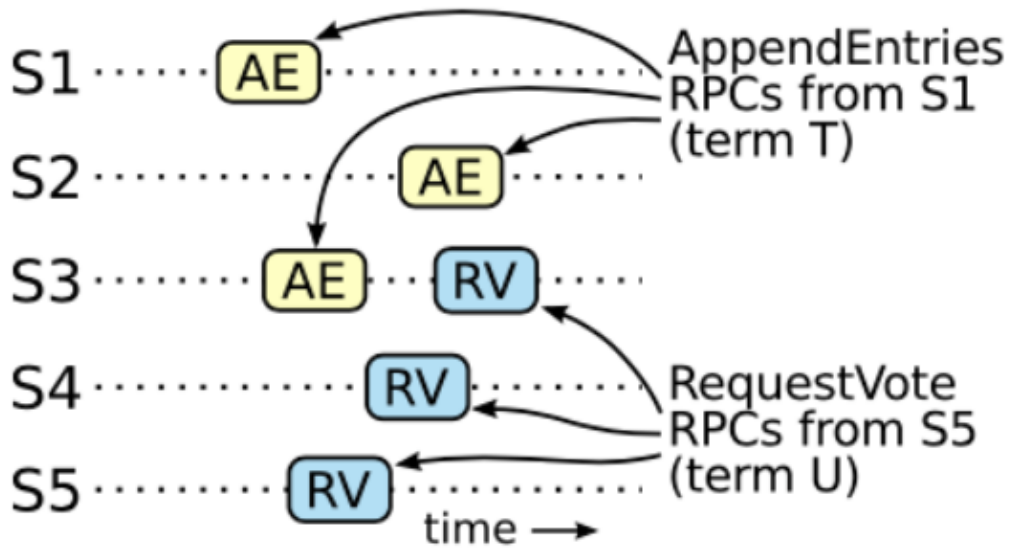


在 c 中 S1 是 leader，在 4 日志进来前，其将 2 日志复制到 s1、s2、s3 中，并提交，然后又进来了一个日志 4，但是 s1 还没来得及将其复制给其他两台机器后就宕机了，此时 s5 在任期5选举为了 leader，因为其日志比较新，然后其会将日志 3 复制到其他所有服务器上，达到过半后又进行提交，此时就会导致 index 2 的日志被提交了两次，这样就会出现问题，因此，我们要求 leader 只能提交当前任期内的日志，前面任期的日志只能随着新任期的日志一起提交，不过以 d 为例，假设其不能提交，那么查找 3 日志的内容则会查不出来，假设一段时间内都没有请求，这样日志3就一直不能提交了。所以这里**引入了 no-op 日志**，即当选举为 leader 后，马上追加一条 no-op 日志，并立即复制到其他节点，no-op 日志提交后，前面任期的日志也就能一起提交了，这样就解决了这个问题

> 这个问题就是所谓的 “幽灵复现” 问题

安全性论证

下面讨论一下 leader completeness，假设有个最新的 leader，其不包含上一个 leader 提交的一个日志条目，我们看看会有什么问题：



> 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. 首先 U 一定在成为 leader 时就没有该提交的日志条目，因为 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
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（说明在 T 和 U 之间还有其他的 leader）。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（他包含了来自任期 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。（因为不满足该条件的 leader 一定有第一个，我们验证第一个是不可能出现的即可）
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

通过 leader completeness 特性，我们就能证明状态机安全特性。**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；leader completeness 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。

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

Follower 和 candidate 崩溃

这两者的崩溃的处理方式是一样的，并且比 leader 崩溃要简单的多。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不好的影响。**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。

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：

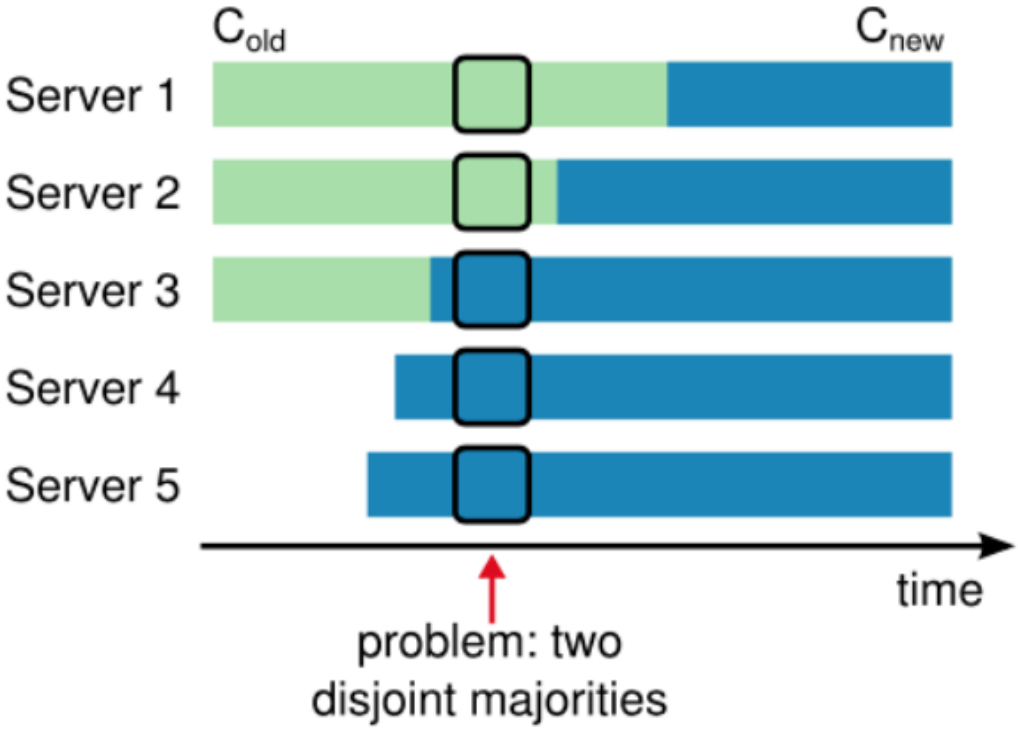
> 广播时间（broadcastTime） << 选举超时时间（electionTimeout） << 平均故障间隔时间（MTBF）

集群成员变更

目前我们假设集群的配置是不变的。但是实践中可能会偶尔改变集群配置，比如替换那些宕机的机器或者改变复制程度。

一种方式是先使整个集群下线，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，此外如果是手工操作，可能会有操作失误的风险。

为了使配置变更机制能够安全，我们需要保证在转换的过程中不能存在某个时间点，使得同一个任期内出现两个 leader。然而从旧配置转换到新的配置的方案都是不安全的，所以在转换期间可能划分为两个独立的大多数：



> 假设集群从 3 台变为 5 台，那么可能会导致有一个任期，有两个 leader 被选出，一个获得旧配置里过半机器的投票，一个获得新配置里的过半机器的投票

为了保证安全性，配置变更需要采用一种两阶段的方法。比如，有些系统在第一阶段停掉旧的配置，这时不能处理客户端请求，第二阶段再启用新的配置。在 Raft 中，集群先切换到一个过渡的配置，我们称为联合一致的状态；一旦联合一致状态被提交了，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：

- 日志条目被复制到集群中新、旧配置的所有服务器
- 新、旧配置的服务器都可以成为 leader
- **达成一致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

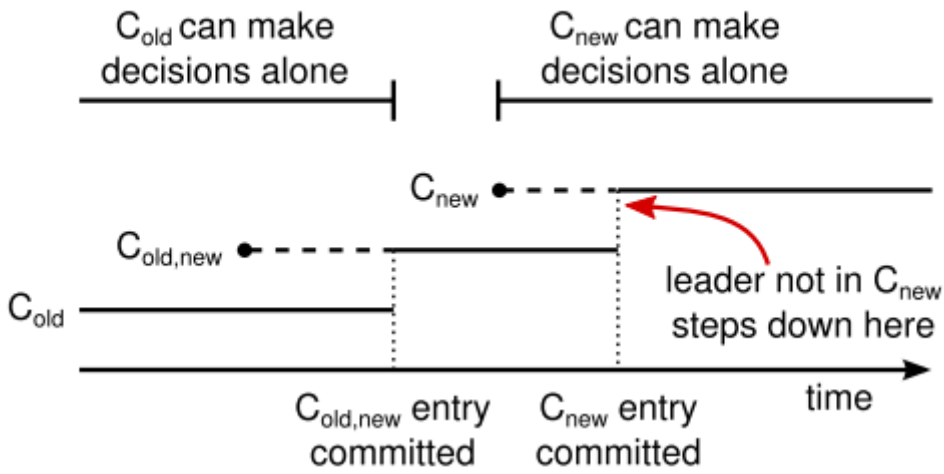


Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a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

集群的配置是用特殊的日志条目来存储的，当一个配置的日志添加到服务器日志条目中时，服务器就会直接用这个新的配置来做出未来的决策（无需该配置日志提交）。当一个 leader 接收到一个改变配置 C_old 到 C_new 的请求，它就将联合一致的配置 C_old, new 存储为一个日志，并按照前面的方式进行复制，leader 会使用 C_old, new 的规则（**分别在新旧配置上获得过半支持**）来决定 C_old, new 的日志条目是什么是否被提交的，如果 leader 崩溃了，新 leader 可能是在 C_old 配置也可能是在 C_new 配置下选出来的，C_new 在这一时期不能做出单方面决定

一旦 C_old, new 被提交，那么整个系统就是需要分别在两种配置上达成过半支持才能成为 leader，leader 完整性特性保证只有拥有 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 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。当新的配置在 C_new 规则下被提交，旧的配置就无关紧要了，同时不使用新配置的服务器就可以关闭了

关于配置变更还有三个问题：

1. 新的服务器一开始可能没有存储任何日志条目，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，这段时间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
2. 集群的 leader 可能不是新配置中的一员，这种情况下，leader 一旦提交了 C_new 日志条目就会退位。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 Leader ）。在此之前，可能只能从 C_old 中选出 Leader

3. 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

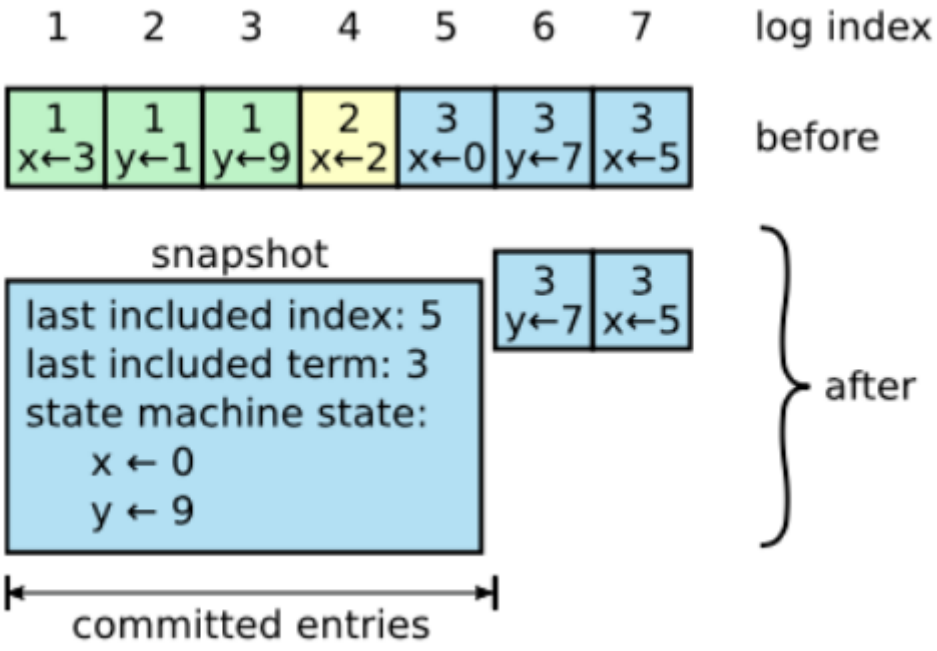
   为了防止这种问题，当服务器认为当前 leader 存在时，服务器会忽略RequestVote RPCs 。特别的，当服务器在最小选举超时时间内收到一个 RequestVote RPC，它不会更新任期号或者投票。这不会影响正常的选举，每个服务器在开始一次选举之前，至少等待最小选举超时时间。这有利于避免被移除的服务器的扰乱：如果 leader 能够发送心跳给集群，那它就不会被更大的任期号影响

日志压缩

Raft 的日志会随着客户端的请求不断增长，而实际中，日志不能无限增长，其会占用越来越多的空间

快照技术是日志压缩最简单的方法。在快照技术中，将整个系统当前的状态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

增量压缩方法，比如日志清理或者日志结构合并数（log-structure 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能力。其选择一个已经有大量被覆盖或删除的对象的数据区域（比如 x = 0，而后面的日志有 x = 1，那 x = 0 就无效了），然后重写该区域还有用的对象，之后就释放该区域。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。



上图展示了 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。主要是讲状态机自己的状态写入快照中。Raft 快照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，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。**并且为了支持上面的集群成员变更，快照中还包括截至 the last included index 的最新配置。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

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 follower，通常是 leader 把需要发送给 follower 的下一条日志条目丢弃了，才需要发送快照，而这种情况一般来说不会发生，一个与 leader 保持同步的 follower 通常都会有该日志条目。然而一个运行缓慢的 follower 或者新加入集群的服务器不会有这个条目，此时让 follower 更快达到最新状态的方式就是让 leader 通过网络把快照发给它

Leader 使用 InstallSnapshot RPC 来发生快照给落后比较多的 follower。当 follower 收到带有这种 RPC 的快照时，follower 需要进行一定的处理。通常该快照会包含接受者日志中没有的信息。在这种情况下，follower 丢弃它所有的日志，并且其整个日志都被快照所取代，并且可能有一些没提交的条目会和快照产生冲突。如果接收到的快照是自己日志的前面部分，那么自己日志中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

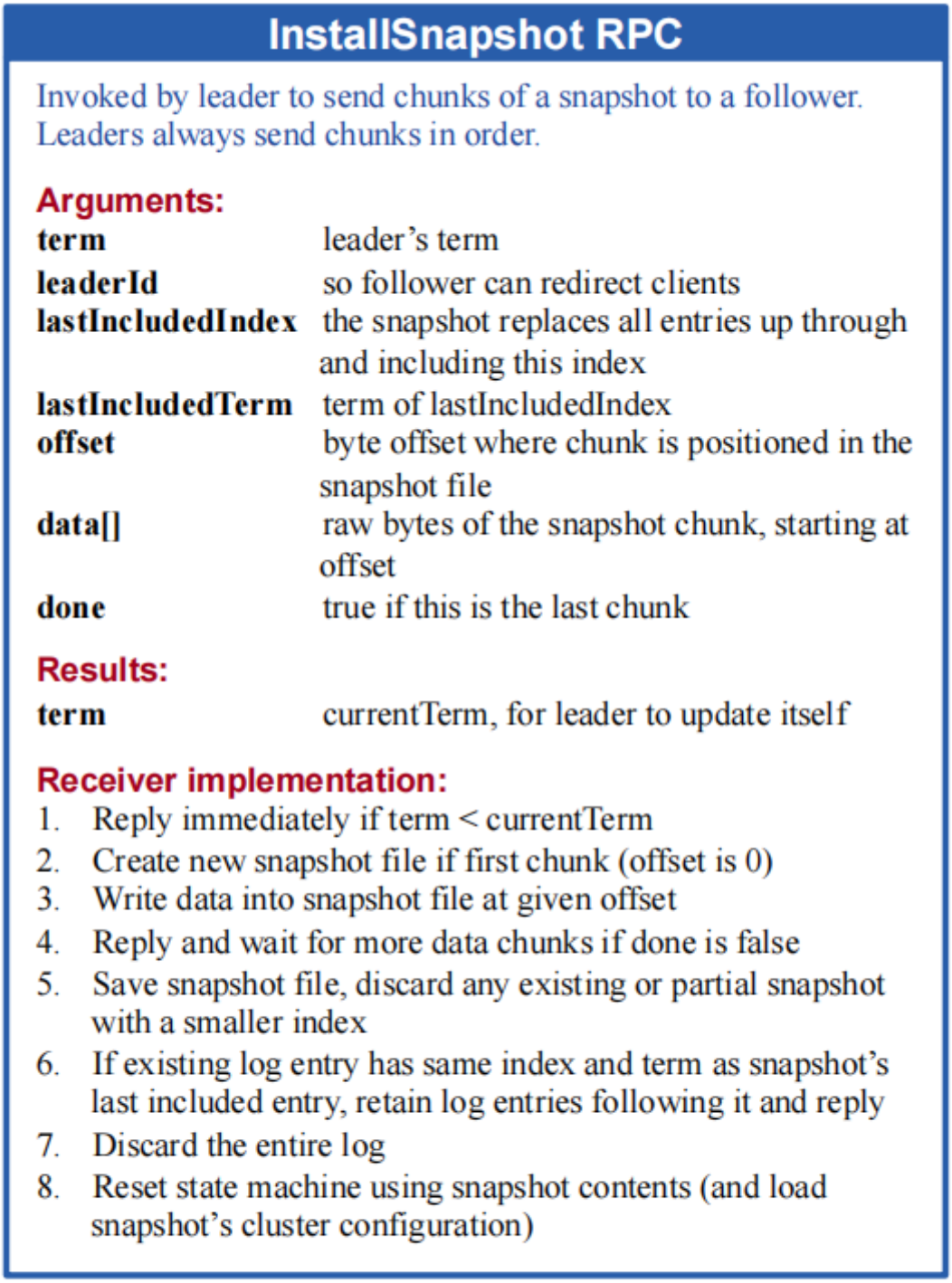


Figure 13: A summary of the InstallSnapshot RPC. Snapshots are split into chunks for transmission; this gives the follower a sign of life with each chunk, so it can reset its election timer.

这种快照方式违反了 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。但是因为 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

我们考虑过基于 leader 的快照方案，在该方案中，只有 leader 创建快照，然后 leader 会发送它的快照给所有的 follower。但是这样做会有两个缺点：第一，发送快照会浪费网络带宽并且延缓了快照过程。本身每个 follower 都已经拥有了创建自己的快照所需要的信息，而且 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快和节约资源。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 replay 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）

客户端交互

这里主要介绍客户端如何与 Raft 进行交互。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该服务器最近接收到的 leader 的信息（AppendEntries 请求包含了 leader 的 ip 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

Raft 的目标是实现线性化语义（每一次操作立即执行，且只执行一次，在调用和回复之间执行）。但是 Raft 可能执行同一条命令多次，比如 leader 在提交日志条目之后，在响应客户端之前崩溃了，那么客户端会和新的 leader 重试这条命令，导致这条命令被执行两次，解决方案就是客户端对每条指令赋予一个唯一的序列号，然后状态机跟踪每个客户端以及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

对于只读操作，我们可以直接处理，而不记录日志，但是，如果不采取其他措施，这样可能会有返回过时数据的风险。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被替代了。**线性化的读操作肯定不会返回过时数据。**Raft 使用了两个额外措施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。这时就通过一个提交一个 no-op 日志条目来处理
- 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）。**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。**另一种方案是，leader 通过心跳机制来实现一种租约形式，但这种方法依赖 timing 来保证安全性