《Raft》读书报告

前言

论文：Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C]//2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). 2014: 305-319.

本读书报告主要是对Raft论文中关键内容的摘取翻译，并附加自己的理解。

论文解读

Abstract

Raft是一种用于日志复制共识算法，与Paxos作用相同、效率相同，但它的结构与Paxos不同，也比Paxos更容易理解、更容易实现。

Raft分离了共识的核心部分，比如：领导人选举、日志复制、安全性，同时增强了一致性程度以减少需要考虑的状态数量。

> In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered.

Raft包括了一种新机制，通过重叠大多数保证集群成员变动的安全性。

> Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

在共识算法领域，Paxos的统治地位可见一斑：

> Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or inflfluenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

但，Paxos晦涩难懂且难以实现。

为了让Raft算法更易于理解，作者使用了分解与减少状态数的方法。

Raft在很多方面与现有共识算法很相似（比如Oki和Liskov的视图戳复制Viewstamped Replication），但也有许多新颖的功能：

- Strong leader
- Leader election：Raft使用随机定时器来选举leader。
- Membership changes：使用joint consensus的新方法处理集群中服务器的变动。

第2节介绍复制状态机的问题（replicated state machine problem）；第3节讨论Paxos的优缺点；第4节描述实现易理解性的大体方法；第5-8节讲解Raft算法；第9节测评Raft算法；第10节讨论最近的相关工作。

2 Replicated state machines

在分布式系统中，常用复制状态机(replicated state machine)解决容错问题。比如，在只有一个leader的大型系统中，通常会使用独立的复制状态机，去管理leader的选举与配置信息的存储（防止leader崩溃）。

复制状态机通过日志实现，每个服务器上的日志保存着一系列命令，服务器上的状态机(state machine)会顺序执行这些命令。每个日志中的命令序列都是相同的，因此每个状态机都执行相同顺序的命令，并得到相同的状态和输出。

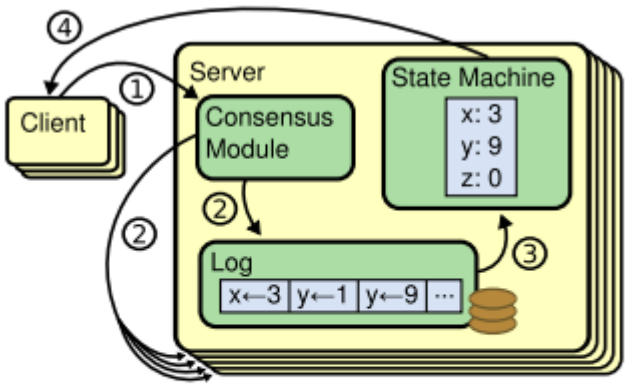


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

复制状态机中，共识算法是为了保证日志的一致性。当服务器上的共识模块(consensus module)收到客户的命令后，会将命令添加到它的日志中，并与其它服务器上的共识模块通信，以保证每个日志中命令序列都是相同的。一旦日志被正确地复制，每台服务器上的状态机就会执行日志中的命令，并将结果返回给客户端。

在实际系统中，共识算法通常有以下特性：

- 在非拜占庭情况下，保证安全性（不返回错误的结果）。
- 在大多数节点可用的情况下，保证可用性。比如5个节点的集群可容忍2个节点的失效。并且，当节点失效时，它们可以恢复并重新加入集群。
- 不依赖时间来保证日志的一致性（？）。
- 只要大多数节点返回响应，命令就算完成，少数较慢的节点不会影响系统性能。

3 What's wrong with Paxos?

过去十年，Paxos就是共识的代名词。

Paxos首先定义了能就单个决策达成共识的协议，被称为单决策Paxos（single-decree Paxos）。然后Paxos组合该协议的多个实例以实现一系列决策，被称为multi-Paxos。

Paxos有两个显著的缺点：

1. 难以理解。论文假定Paxos的复杂性源于它以单决策子集为基础，而单决策Paxos是复杂的。它分为两个阶段，没有简单直观的解释，也无法单独地去理解。因此，这导致multi-Paxos更难以理解。作者相信有更直接显著的方法，就多项决策达成共识。
2. 难以具体实现。一个原因是Lamport（Paxos作者）仅构想了实现multi-Paxos的可能方法，但许多细节有所欠缺。此外，Paxos的架构很难应用到实际系统中。

许多系统实现都从Paxos开始，但发现实现上存在困难，然后开发成另一种不同的架构。

为了解决这些，论文设计了Raft。

4 Designing for understandability

设计Raft时的一些目标：

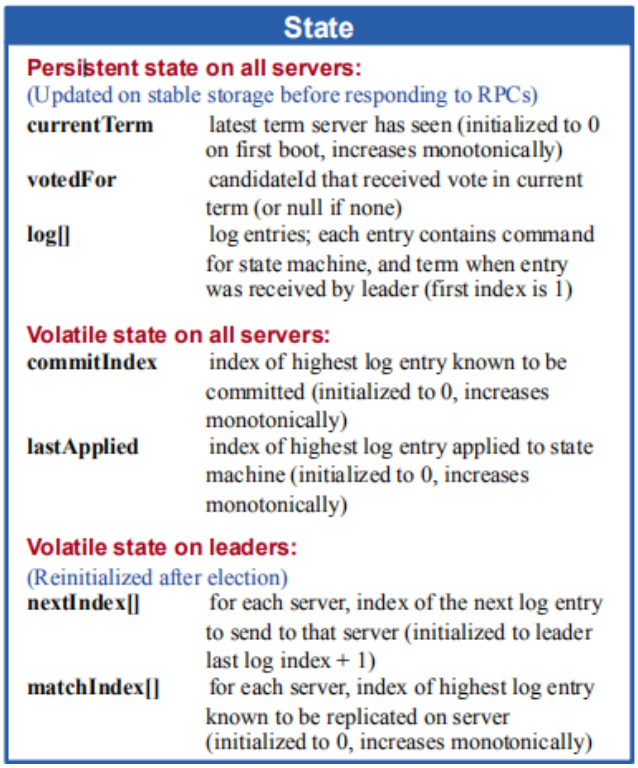
- 必须易于实现。
- 必须在所有情况下保证安全性，在常见情况下保证可用性。
- 对于普通操作必须高效。
- 易于理解（最大的目标与挑战）。

使用两种方法来实现易于理解：

1. 问题分解
2. 减少状态数

5 The Raft consensus algorithm

Raft是一种用于管理第2节中可复制日志（replicated log）的共识算法。图2是算法的压缩总结，以供参考。图3列举了算法的关键属性。



State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

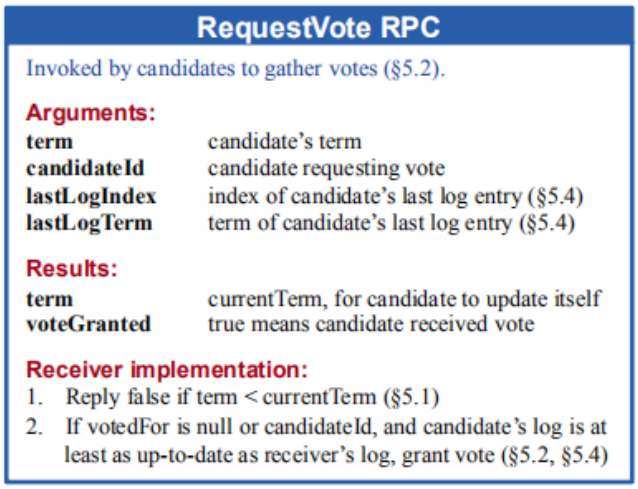
currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

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)



RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

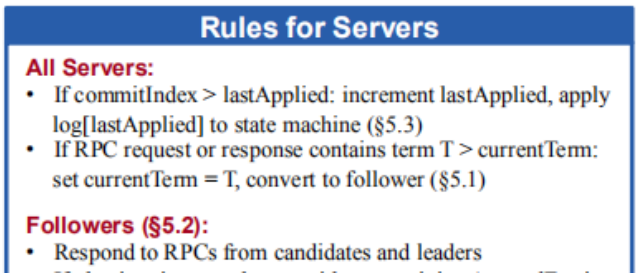
term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

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)



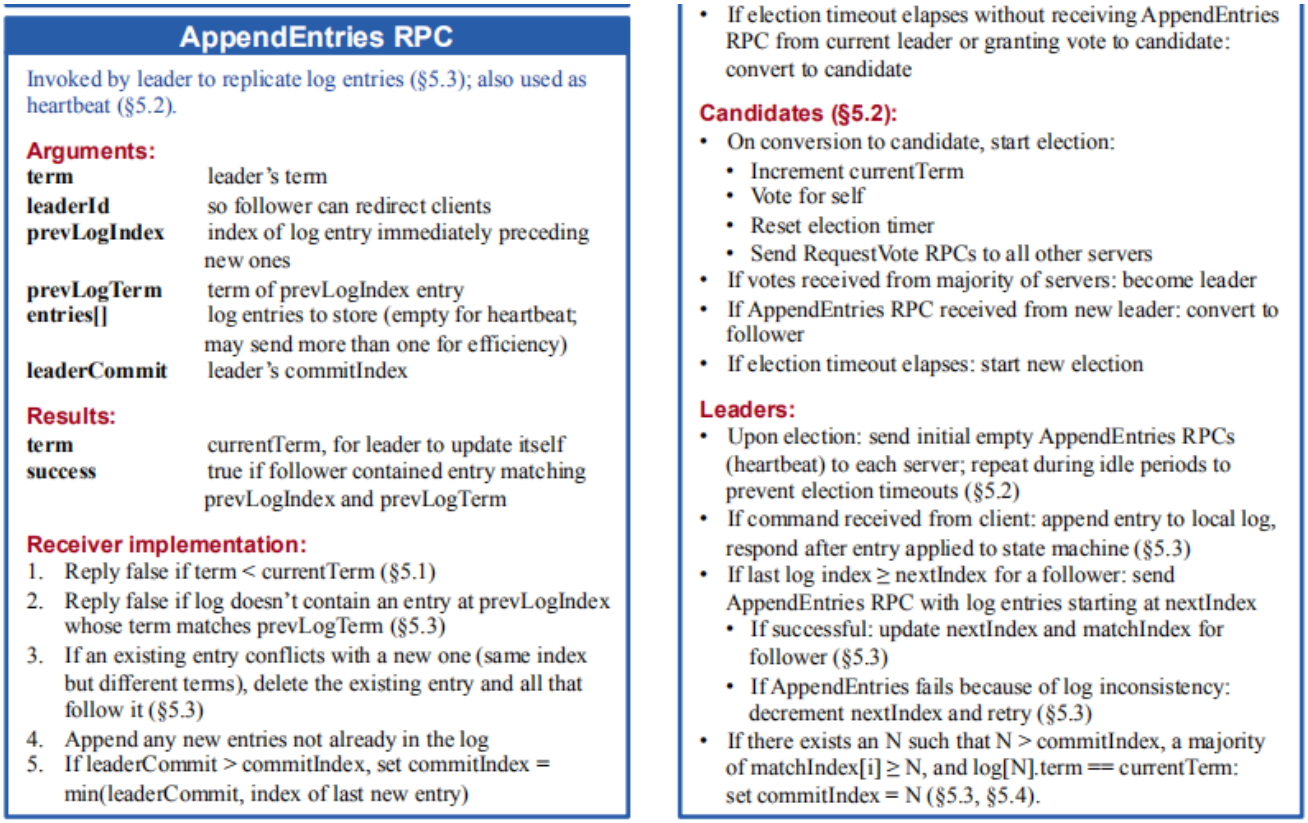
Rules for Servers

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



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)

- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

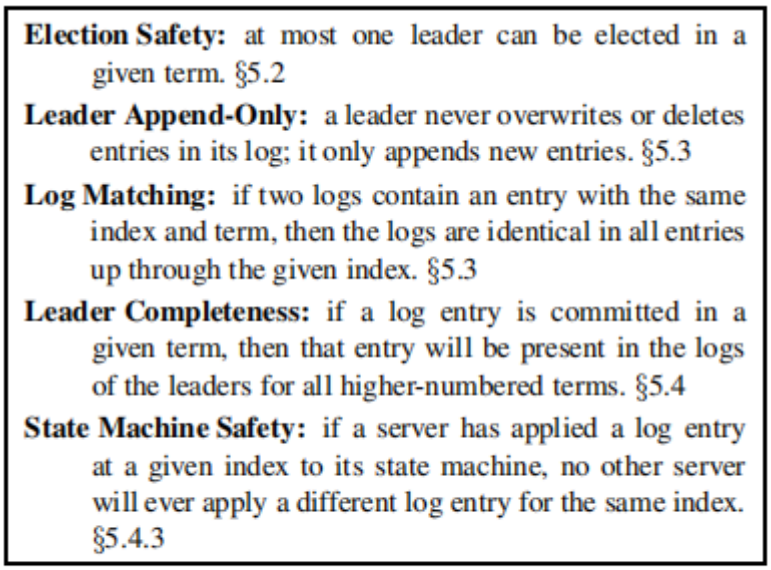
Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

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



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

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

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

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

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

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

Raft首先会选举一个独立的leader，让leader全权负责日志复制。leader从客户端获取命令（日志记录 log entries），然后将其复制给其它服务器，并告诉其它服务器什么时候执行才是安全的。当leader失效时，会选举出新的leader。

通过leader机制，Raft将共识问题分解为三个子问题：

- Leader election（5.2节）
- Log replication（5.3节）

- Safety：在日志中的同一位置，不同服务器上的状态机应执行相同命令（5.4节）

在介绍了共识算法之后，本节将讨论可用性和定时(timing)在系统中的作用。

5.1 Raft basics

Raft集群包含多个节点，通常有5个节点就能容忍2个节点出错。

任意时刻，集群中节点只有三种状态：leader、follower、candidate。在正常情况下，有一个leader，其它都是follower。follower节点不发出请求，只响应leader或candidate的请求。leader节点处理所有客户端请求，如果follower收到客户端，会转交给leader。candidate状态则用于leader选举。各状态间的相互转换如下图：

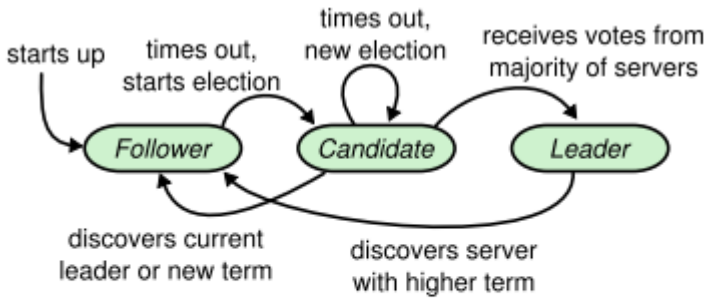


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

如下图，Raft将时间分为不同长度的任期(terms)，terms的序号是连续的。每个term都开始于一次选举election，选举中一或多个candidate竞争一个leader，如果某个candidate赢得选举，将在当前term的剩余时间内担任leader。

在某些情况下，可能会出现相等的票数（比如A和B都得到3票），那么当前term就会以无leader的情况(**no leader**)结束，一个新的term很快就会开始。Raft会保证一个term最多只有一个leader。

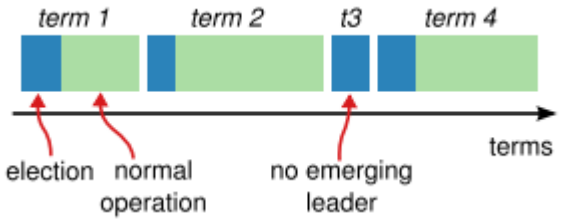


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different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

不同节点可能在不同时间察觉到term的变动，在某些情况下，节点甚至察觉不到某次选举或整个term。

在Raft中，terms扮演了逻辑时钟的角色，可以让节点发现一些过期的信息，比如已失效的leader。

每个节点都保存着一个**当前term序号(current term number)**，并随着时间递增。**当节点进行通信时，会交换当前term的序号，如果节点发现它的term序号较小时，它会将其更新成那个较大的值。**当leader或candidate发现它的term已过期时，它会立即变为follower状态。如果节点收到一个携带已过期term序号的请求，它会拒绝这个请求。

Raft中，节点间使用RPCs通信，最基础的共识算法只需要两种类型的RPCs：

- RequestVote RPCs，由candidate发起，用于选举。
- AppendEntries RPCs，由leader发起，用于复制日志和心跳提醒。

5.2 Leader election

Raft采用一种心跳机制来触发leader选举。当节点刚启动时，它处于follower状态。当它收到来自candidate或leader的RPC请求时，它会保留为follower状态。因此，leader会定期地发送心跳提醒(heartbeats)给所有follower，以维持它的统治。当follower在一段时间内(election timeout)都没有收到消息时，它会假定当前leader已失效，然后开始新的竞选。

当开始竞选，follower会将当前term序号加1，然后转变为candidate状态。它先会投票给自己，然后并行地发送RequestVote RPC请求给其它节点。直到以下三种情况出现，节点才会改变candidate状态：

1. 节点赢得选举。
2. 其它节点成为leader。
3. 一个周期时间过去任未选出leader。

当candidate节点得到集群中大多数节点针对当前同一term的投票，它将赢得选举胜利。在一个term内，节点只能对最多一个candidate进行投票，并遵循先到先得的原则。得到大多数投票，能保证在一个term内，最多只有一个candidate获胜。一旦某个candidate获胜成为leader，它将定期发送心跳heartbeats信息给其它所有节点，以维持它的统治。

当candidate在等待投票过程中，收到刚成为leader节点的AppendEntries RPC请求，且请求的term序号**大于等于**它的term序号时，它将会认可这个leader的合法性，并转变为follower状态。如果请求的term序号小于它的term序号时，它会拒绝请求并保持candidate状态。

当没有candidate胜出，比如大多数follower同时变为candidate，导致没有candidate能获得大多数投票时，每个candidate都会等待超时，然后将当前term序号加1，开始新一轮的竞选。当然，如果没有额外的措施，这种情况可能无限重复。

Raft使用**随机化**的选举等待时间来解决上述**投票分裂(split votes)**问题：每个candidate节点开始竞选时，会从某个区间（比如150-300ms）内随机选择**选举等待时间(election timeout)**。这将导致在大多情况下，只有一个节点率先超时进入下一次选举，并在其它节点超时之前赢得选举。

5.3 Log replication

一旦某个leader被选举出来，它将处理客户端请求。每个客户端请求包含一条可被复制状态机执行的命令。leader会将这条命令追加到它的日志中，然后并行地发送AppendEntries RPCs请求，复制这条命令给其它节点。当这条命令被安全地复制，leader将让它的状态机执行这条命令，并返回结果给客户端。如果followers崩溃或网络堵塞，leader将会重复地发送请求，直到所有follower都保存了这条命令。

日志如下图所示，类似于有序数组，日志中每条记录都包含一条命令和对应的term序号。

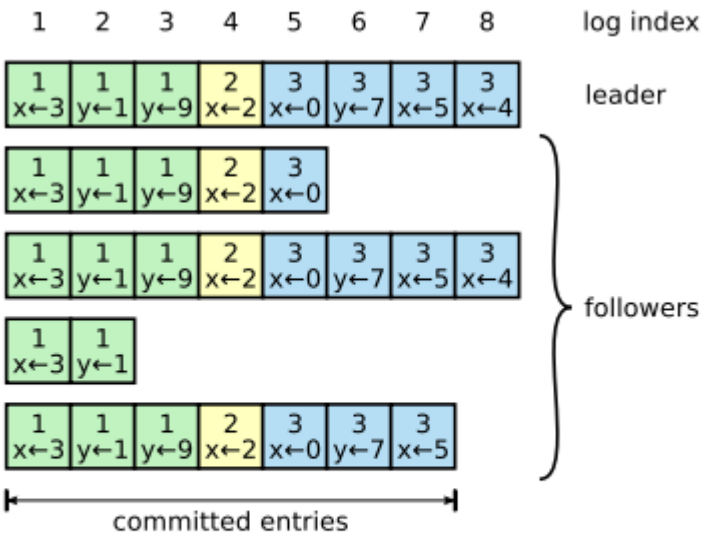


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

如果某条日志记录可以安全地执行，那leader会标记这条记录为**已提交的(committed)。Raft会保证已提交的记录是持久的（不会被删除或覆盖），且最终被所有节点执行。**当记录**被复制到大多数节点**时，它就可以被标记为已提交的，比如上图中的第7条记录。leader会保存日志中**已提交记录的最大索引**，比如上图中最大已提交索引为7。它会在AppendEntries RPCs请求中包含这个值，以便让其它节点知晓。**一旦某个follower节点得知某条记录已提交，它便会执行记录中的命令**（按日志中记录的顺序）。

Raft会维持如下特性，这些特性构成了图3种的**日志匹配属性(Log Matching Property)**：

- 如果不同日志中的两条记录拥有相同的索引和任期号，那么它们存储了相同的指令。
- 如果不同日志中的两条记录拥有相同的索引和任期号，那么它们之前的所有日志条目也都相同。

一个任期号至多对应一个leader，且leader在任期号内的一个索引处只能创建一条记录，这保证第一条特性。第二条特性由AppendEntries RPCs请求时**一致性检查**保证：当leader发送AppendEntries RPCs请求时，会将新日志记录的**上一条记录的索引和term号（其实就是已提交记录的最大索引和其term号）**包含在内。如果follower在它日志中没有找到**相同的索引和term号**，它就会拒绝请求（保证一致性）。

在某些情况下，旧leader发生崩溃，那么新的leader与follower日志可能不同：新leader中的日志记录可能比某些follower多，也可能比某些follower少。

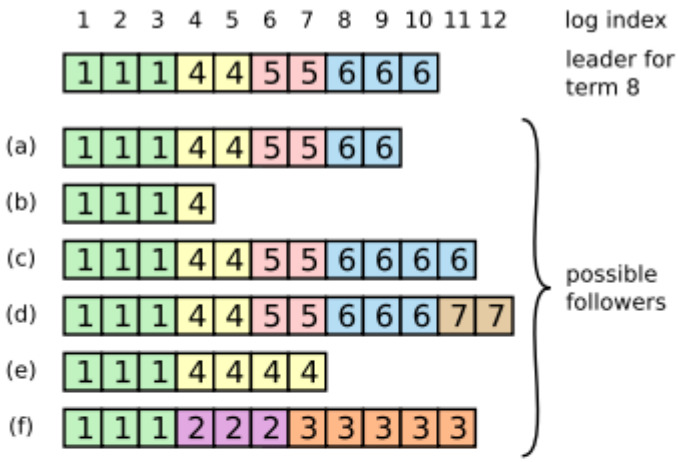


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, 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中，leader通过强制follower的日志保持与自己一致，来解决不一致问题（5.4节将解释，当附加了一个至多个限制时，这是安全的）。

leader为了让follower的日志与自己一致，**会找到两者日志中一致且索引最大的记录，然后删除follower日志中的剩余记录，再将leader日志中剩余记录追加到follower日志中。**

leader为每个follower都维护了一个**下一个索引值(nextIndex)**，用于标识下一个将发送给follower的日志记录。当leader首次启动时，它会初始所有nextIndex为它日志中最后一条记录的下一个索引。如果follower的日志与leader的不一致，AppendEntries RPC请求将会失败。当leader收到拒绝响应时，它会将follower对应的nextIndex减1，然后再重试。最终，nextIndex会达到leader和follower中日志相一致的索引处。此时，AppendEntries RPC请求将成功响应，follower会删除冲突的部分，并将leader发来的记录追加到它的日志中。

这部分算法可以被优化：当follower发现不一致时，返回term序号和term内的第一个索引，leader借此可以跳过这个term。但在实践中，作者发现这种优化没有必要，因为失败不经常发送，且冲突记录并不多。

通过这种机制，既能保证一致性，又能保证少数过慢的机器不会影响系统性能。

5.4 Safety

上述机制还存在一个问题：不能保证每个节点都按相同顺序执行了相同的命令。比如，某个follower处于失效状态时，leader提交了多条日志记录，而后这个follower生效被选为新leader，它将覆盖这些已提交的记录。

本节通过对leader选举进行限制，来完善Raft算法。这种限制能保证：在任意term内，leader都包含了之前各term内已提交的日志记录。

5.4.1 Election restriction

在选举时，Raft规定只有日志中包含**所有已提交记录**的candidate才能赢得竞选，这就意味着所有已提交记录必须至少存在于一个节点上。

如果candidate的日志至少与大多数节点的日志一样**新(up-to-date)**，那么它就拥有全部的已提交记录。

这个投票限制通过RequestVote RPC请求实现：RPC请求会包含candidate的日志信息，**如果投票者的日志比candidate的新，它就会拒绝投票。**

Raft通过比较两个日志中最后一条记录来判断谁更新：**term序号大的更新；term序号相等但日志更长的更新。**

5.4.2 Committing entries from previous terms

由上文可知，leader只有得知大多数节点保存了当前日志记录后，才会提交该记录。如果leader在提交该记录前崩溃了，那么新leader会尝试继续完成该记录的复制。然而，如果前一条记录被复制到大多数节点，leader也无法判断该记录是否已提交。下图展示了在旧leader下被复制到大多数节点的记录，可能被新leader覆盖。

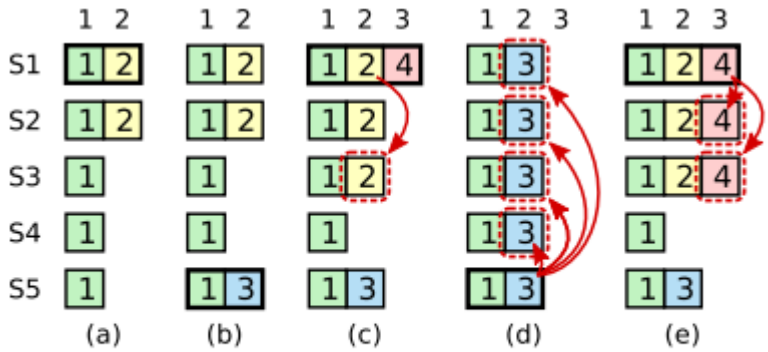


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at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

图8描述了一种情况：(a)中，S1被选为leader(term 2)，复制2索引处的记录到S1、S2。(b)中，S1崩溃，S5被选为leader(term 3)，投票来自S3、S4、S5，它在2索引处获得新记录。(c)中，S5崩溃，S1重新被选为leader(term 4)，为了保证日志一致，它会继续复制索引2处term 2内的记录到其它节点，如果得到超过半数的复制成功响应，它就会提交该记录（也意味着会执行该记录）。但在(d)中，S1崩溃，S5重新选为leader，它也会继续复制索引2处term 3内的记录到其它节点，此时，就会覆盖索引2处term 2的已提交记录（S1执行过的命令被覆盖，违背持久性）。如果，回到(c)中，S1复制索引2处term 2内的记录到其它节点，得到大多数成功响应，但并不标记这个记录为已提交，而是等到自己term内的记录复制到大多数，才标记自己term内的记录已提交，那就会避免这个问题，既能提交旧term内的记录，也能保证一致。

为了解决这个问题，Raft不会通过统计是否复制到大多数节点，来提交之前任期(term)内的日志记录。**只有当前term内的日志记录，才会通过统计是否复制到大多数节点来提交。**因为，一旦当前term内的记录被提交，所有此前的记录都会被提交（参考5.3节）。

5.4.3 Safety argument

此节讨论Raft的安全性，主要通过反证法进行论证。假设任期T内的leaderT在任期内提交了一个日志记录，但是该日志记录没有被存储到未来某个任期U的 leaderU 中，然后反证这种情况不可能发生。

详细内容略。

5.5 Follower and candidate crashes

如果follower或candidate崩溃，发送给它们的RPC请求会无限重试，直到它们重启后成功响应。

5.6 Timing and availability

Raft中，系统的安全性不能依靠定时：不能因为某些事件发生得过慢或过快，就产生错误的结果。但是系统的可用性（规定时间内响应用户）却必须依靠定时，比如：消息传播过慢，导致candidate没有足够时间赢得选举，进而导致选不出稳定的leader。

定时是leader选举的关键，为了选举并维持一个稳定的leader，系统必须满足如下定时需求：

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

广播时间是节点并行地发生RPC请求且收到响应的平均时间；选举超时时间参考5.2节；MTBF是节点故障恢复的平均时间。广播时间应该比选举超时时间小一个量级，这样 leader 才能通过发送心跳消息来阻止 follower 开始进入选举状态。选举超时时间应该比平均故障间隔时间小上几个数量级，这样才能在leader崩溃期间，快速选出新leader保证系统可用。

广播时间和MTBF由具体情况而定，但选举超时时间可自定义。通常，广播时间大约是0.5ms-20ms，因此选举超时时间可定义为10ms-500ms。

6 Cluster membership changes

实际中，某些时刻需要更新系统的配置（系统节点信息），即集群变更，比如某些节点永久失效需要被替换时。一个方法是将所有节点下线，更新完成后再手动重启，但这样会导致系统某段时间不可用。此外，通过手动操作也会产生意料之外的错误。于是，Raft针对集群变更设计自动化机制。

当然，在集群变更过程中，会产生一些问题：由于一次性改变所有节点配置是不可能的，那么在某个时刻，旧配置节点与新配置节点共存，可能出现两个leader。如下图，1、2在旧配置（只有3个节点）中，3、4、5在新配置（有5个节点）中，1获得了2的投票，它就会认为得到大多数投票，成为leader；而5获得3、4的投票，它也会成为leader。这样同一时刻就有两个leader。

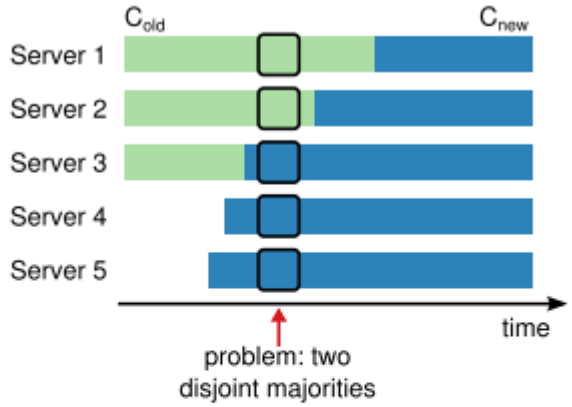


Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one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

为了保证集群变更时的安全性，需使用**两阶段(two-phase)方法**。这种方法有许多实现方式，比如：一些系统，在第一阶段先停止旧配置节点，让其不能接收请求；然后在第二个阶段启动新配置节点。

在Raft中，集群首先切换到一个中间状态，称为**联合一致(joint consensus)**。一旦joint consensus被提交，系统就可以过渡到新配置。在joint consensus状态中，系统同时包含旧配置和新配置，并遵循以下机制（相当于**旧新配置的并集**）：

- 日志会复制到新旧配置中的所有节点。
- 新旧配置中的节点都可以成为leader。
- 选举和日志提交，**既需要得到旧配置中大多数节点的同意，也需要得到新配置中大多数节点的同意**。这能保证新旧配置leader不会同时出现，比如上图10中，5想要成为leader，就需要同时得到旧配置（1、2、3）下的大多数投票，和新配置（1、2、3、4、5）下的大多数投票。

集群配置作为**特殊的日志记录**进行存储。下图展示了配置变更过程。当leader收到配置变更的请求($C_{old} \rightarrow C_{new}$)时，它会将中间态配置$C_{old,new}$（相当于新旧配置的并集）存储为一个日志记录，然后按照如上机制复制到系统其它节点。（一旦节点添加了新的配置记录到日志中，它将会使用该配置进行未来的全部决策。且节点总是使用日志中最新的配置记录，无论该记录是否提交）。leader会使用$C_{old,new}$配置来判断$C_{old,new}$记录是否已提交，如果leader宕机，一个**新的leader会从**C_{old}**或**$C_{old,new}$**配置的机器中选出**。在任何情况中，C_{new}的配置都不会产生任何决策。

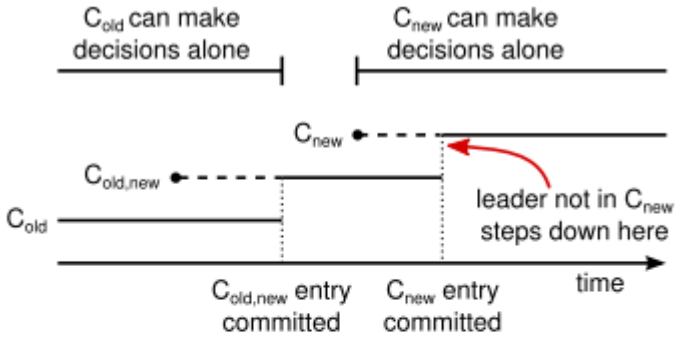


Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{\text{old,new}}$ configuration entry in its log and commits it to $C_{\text{old,new}}$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

一旦$C_{old,new}$配置已提交，不管C_{old}或C_{new}配置的节点都不能单独决策（因为按照中间态的机制，决策既要得到旧配置中大多数节点的同意，也需要得到新配置中大多数节点的同意）。且$C_{old,new}$配置已提交能保证它被复制到大多数节点，即只有$C_{old,new}$配置的节点才能选为leader。此时，leader可以安全地创建C_{new}配置的日志记录，并复制到集群。当C_{new}配置被提交，不在新配置下的节点将被关闭。

关于配置变更还有3个问题：

1. 新节点没有存储任何日志记录。新节点刚加入系统时，需要一段时间赶上进度。为了避免这段时间系统不可用，Raft在配置变更阶段之前引入一个新阶段。在这个阶段中，新节点加入到系统中，但不参与投票，leader依然会将日志复制给它们。直到新节点的日志赶上其它节点，才会进入配置变更阶段。
2. leader 可能不是新配置节点中的一员。在这种情况下，leader 一旦提交了C_{new}配置的日志记录（复制到大多数）就会退位（回到 follower 状态）。然后新配置中的节点将被选为leader，在此之前只有旧配置中的节点才会被选为leader。
3. 被移除的节点可能会扰乱集群。这些节点因为没收到心跳提醒，会超时，然后开始新一轮选举，并发送带有新term号的RequestVote RPC请求给新配置下的节点，这会导致新配置下的leader变为follower状态。为了防止这个问题，**当节点知晓当前leader存在时，它会拒绝RequestVote RPC请求，即使请求的term号更大。**

根据上述集群变更机制，重新考虑图10中的情况。首先，4、5节点先要将日志追上进度，再参与投票。此时，作为leader的3节点收到集群变更的命令，但它只将$C_{old,new}$配置记录发送到4、5节点，就失效了。然后，1节点（它并不知道4、5节点，但获得1、2节点的投票）成为新的leader。此时，5节点也收到3、4、5节点的投票，但它不能成为leader，因为它还要得到旧配置（1、2、3节点）中大多数节点的同意。

7 Log compaction

Raft中，日志会越来越长，这会导致日志占用空间越来越大，重新执行日志花费的时间也会更长。为了防止日志无限增长，Raft需要压缩日志。

快照是最简单的压缩方法。系统会将当前状态以快照方式持久化存储，而该时间点前的日志全部丢弃。

Raft中的快照机制如下图。每个节点独立地生成快照，具体工作是将状态机的当前状态存储到快照中，因此只有已提交的日志记录才会被快照替代（已提交才会被状态机执行）。Raft也会在快照中保存相关元数据：last included index是被快照替代的日志序列中最后一个记录的索引，last included term是被快照替代的日志序列中最后一个记录的term号，它们主要用于支持AppendEntries RPC请求时的一致性检查。为了支持集群成员变更，还会包含被快照

替代的日志序列中最后一条配置信息。一旦快照成功保存，节点会删除包括last included index在内的之前所有日志记录。

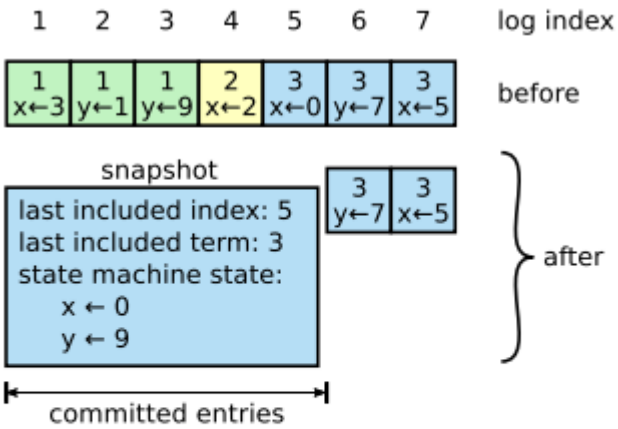
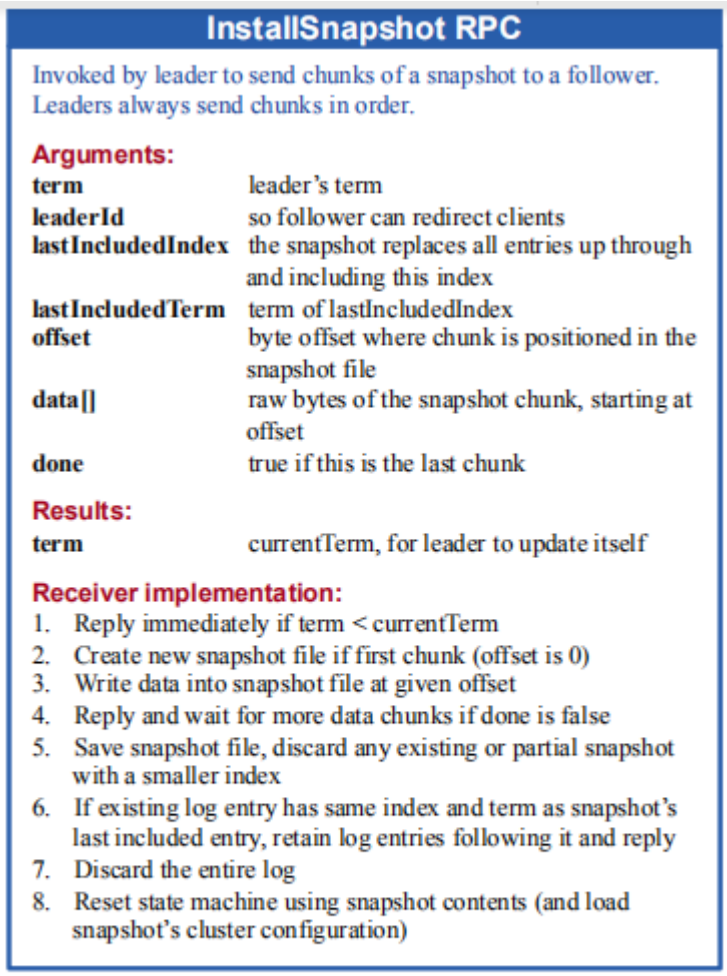


Figure 12: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables *x* and *y* in this example). The snapshot's last included index and term serve to position the snapshot in the log preceding entry 6.

当leader需要发送已被快照替代的日志记录给某些follower时，它应该发送快照给对方，这种情况发生在某些follower较慢或刚加入集群时。

leader使用InstallSnapshot RPC请求（如下图）发送快照给一些落后的follower。当follower收到InstallSnapshot RPC请求时，它会丢弃快照之内的日志记录，只保留不在快照中的剩余记录，若已提交索引小于快照最后索引，还会更新自己的已提交索引，最后将快照应用到状态机中。



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
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
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. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

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.

8 Client interaction

当一个客户端初次启动时，它会随机连接到一个节点。若节点并非leader，它会返回当前leader节点的信息给客户端。如果leader宕机了，客户端会请求超时，然后再尝试连接一个随机节点。

Raft需实现线性语义化(linearizable semantics)，即一条指令只执行一次。实际情况中，一条指令可能执行多次，比如leader执行完某条指令，但在响应客户端时宕机了，客户端会重试连接，并重新发送这条指令，那么这条指令就会被执行两次。解决方法是对每条指令附加一个唯一标识，当状态机收到一条标识重复的指令时，它会直接返回而不执行。

只读操作不需要写入到日志中，但是系统可能会返回过期的数据，比如：响应只读请求的leader可能不知道新leader已经出现了，因为不产生日志则不需要发送日志复制请求。Raft采取两个额外措施来解决这个问题：

1. leader必须知道哪些记录是已提交的。根据特性，leader一定包含所有已提交记录，但新leader一开始并不知道哪些记录是已提交的。对此，**Raft要求每个leader在任期开始时，提交一个空的no-op类型的日志记录。**
2. leader在处理只读请求之前，必须检查自己是否已过时。Raft要求leader在响应只读请求时，必须向集群内过半节点请求并响应一次心跳信息。

9 Implementation and evaluation

略。

10 Related work

略。

11 Conclusion

略。

12 Acknowledgments

略。