In Search of an Understandable Consensus Algorithm

(Extended Version)

Diego Ongaro and John Ousterhout

Stanford University

博客地址：http://blog.csdn.net/hfty290/article/details/42742105 欢迎留言讨论。

摘要

Raft是一个管理日志复制的一个一致性算法。他产生一个结果等同于Paxos算法，与Paxos有着同样的效率，但是与Paxos有不同的结构；这使得Raft相对更容易理解也提供更好工业实现的基础。为了提高可理解性，Raft对一致性的关键要素进行分解，诸如Leader选择，日志分发，安全性，通过更强的一致性{a stronger degree of coherency}来减少必须被接受的状态个数。一组学生学习结果表明Raft相较于Paxos更容易学习。Raft还包括改变机器成员的新算法，其使用重叠多数{overlapping majorites}来保证安全性。

1、Introduction

一致性算法使得多台机器协同工作犹如一个强化的组合，在其中某些成员异常时依然能够工作。正因如此，他们在构建可信赖的大规模关键系统时扮演着关键角色。Paxos

算法统治着一致性算法数十年：多数实现都是基于此或改进版本，并且Paxos已经变成教导一致性的基础工具。

不幸的是，尽管有很多人试图让其变成更容易接近，但是Paxos还是有一些难懂。而且他的结构需要进行复杂的改变才能适应现实系统。因此，系统构建者与学生都在与Paxos{struggling}进行艰难搏斗。

我们在与Paxos进行斗争{struggling}之后，开始去寻找一种新的容易构建与教学的一致性算法。该方法的不寻常之处在于，其主要目标是可理解性：是否可以为现实系统定义一种一致性算法，并且需要比Paxos容易得多。而且，我们希望该算法便利于开发者的直觉，这对于系统构建者是必要的。算法是如何工作的，与为何能够工作都是重要的。

这项工作的结果是一种称为Raft的一致性算法。在设计Raft时我们应用了一些特定的技术来提高可理解性，包括分解（Raft分成Leader election、log replication、safety）与缩小状态空间（相对于Paxos，Raft减少未决的等级与服务器之间可能导致不一致的方式）。包括来自两个学校43个学生的学习显示Raft比Paxos显著容易理解：在学习这两种算法之后，其中33个学生回答Raft比Paxos好。

Raft与现有的一致性算法有很多一致的地方，但是他有如下的创新特性：

1）强健的Leader：Raft使用一个比其他一致性算法强健的Leader关系。例如，日志项只能从Leader流向其他服务器。这使得日志分发的管理变得简单，并且使得Raft变成容易理解。

2）Leader election：Raft使用随机的定时器来选择Leaders。这只是在很多一致性算法原本就需要的心跳基础上增加了少量的机制，而使得解决冲突变成简单与快速。

3）成员关系变更：Raft变更集群中的服务器使用一种新的Joint consensus方法，也就是使用配置重叠部分的多数方法。这使得集群在经历配置变更能够继续正常处理。

我们相信Raft在教育目的与基础实现上都优于Paxos或其他一致性算法。相对更简单与容易理解；被描述成完全足够应付现实系统。他有多个开源实现并且在多个公司中得到应用；他的安全性已经得到证实；并且他的效率可以与其他算法相比拟。

论文的接下来部分将讨论，复制状态机问题 Replicated state machine problem，讨论Paxos的优缺点，描述理解性的通用方法，Raft一致性算法，评估Raft，相关问题讨论。

2、Replicated state machines

一致性算法通常出现在复制状态机的上下文(context of replacated state machines)。在这种方法中，一组服务器的状态机计算出同样的状态，并且即使有些服务器下线也能够继续工作。复制状态机在分布式系统中被用于解决各种可容忍的错误问题。例如，大型系统中有一个单独的leader，例如GFS，HDFS，RAMCloud，典型地使用一个独立的复制状态机来管理Leader election与保存配置信息，以备Leader宕机后信息能够保持。Chubby与ZooKeeper都是使用复制状态机的例子。

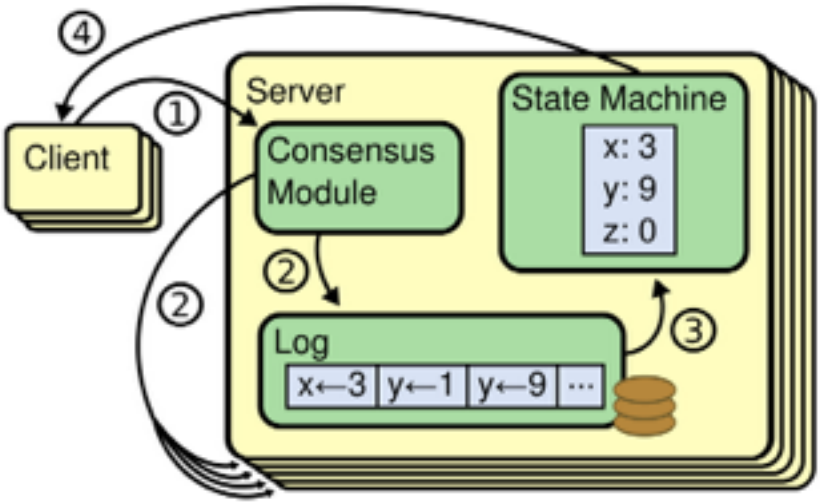


图1、复制状态机结构。一致性算法管理着，来自客户端包含状态机命令的日志复制。状态机执行来自日志中的同样顺序的命令，因此产生了同样的输出。

{每个服务器上由三个模块组成，一致性算法，日志，状态机；其中作为Leader的一致性算法，负责将日志赋值给非Leader的服务器；客户端只能将请求发送给Leader，由Leader将命令复制给其他服务器。}

复制状态机的典型实现是复制日志，如图1.每个服务器在日志中保存一系列命令，其他的状态机按照同样的顺序执行。每个日志以同样的顺序保存同样的命令，因此每个状态机执行同样的命令序列。因为状态机是确定的，每个都计算出同样的状态与同样的顺序输出。

保持复制日志的一致性是一致性算法的工作。服务器上的一致性模块，接收来自客户端的命令，并且添加到他们的日志中。他与其他服务器的一致性模块通讯，以确保每个日志最终以同样的顺序保存同样的请求，即使有些服务器失败。一旦命令被适当地复制，每台服务器的状态机以日志的顺序执行，将输出返回给客户端。结果，服务器（多台）就像来自单个，高度一致的状态机。

现实系统中的一致性算法，具有如下典型的属性：

1）在非拜占庭条件下（non-Byzantine conditions）确保安全性（永远不会返回一个错误的结果），包括网络延迟，分割，与丢包，重复与失序。

2）只要集群中的多数机器工作并且能够互相通信，也能与客户端进行通讯，那么服务将是可使用的（available）。于是，包括五台服务器的集群，能够容忍任何两台服务器的失败。假设服务器的失败是由停机引起，那么当服务器恢复后，能够从持久化存储设备（磁盘）中恢复状态来重新加入到一个集群之中。

3）不依赖时间来确保日志的一致性：错误的时钟与极端的消息延迟，最坏情况下，会导致可使用性问题。

4）通常，只要集群中的多数服务器回复了远程处理请求该命令就能够完成；少量性能差的机器并不影响整体的系统性能。

3、Paxos的问题是什么

最近十年，Leslie Lamport的Paxos协议几乎变成一致性的同义词：该协议是最经常在课上被讲授的，并且绝大多数的一致性实现也是以其为起点。Paxos最早定义一个协议使得达到单个决定变成可能，例如一个单独的复制日志项。此处引用叫做single-decree Paxos（单个命令子集）。Paxos然后合并该协议的多个实例，来为一系列决定提供便利，例如日志（multi-Paxos）。Paxos确保安全性与存活性，并且支持集群成员之间的变更。他的正确性已经得到证明，并且效率也被接受。

不幸的是，Paxos有两个显著的缺点。第一个是Paxos特别难以理解。完整的解释是出名的晦涩；少数人通过巨大的努力，才最终弄懂他。因此，试图以简单地方式来解释Paxos的努力已经有很多次。这些解释集中于single-decree subset，然而这依然是一个挑战。在一个NSDI 2012的非正式调查上，发现只有少数人能够适应Paxos，即使包括一些老练的研究员。我们自己也与Paxos斗争，在阅读了其他Paxos简化描述与设计自己的协议之前都不能完全理解协议，这个过程持续了接近一年。

我们假定，Paxos的晦涩来自于作为其基础的single-decree subset。Single-decree Paxos是愚钝的也是精巧的（dense and subtle）：它分成两个阶段stages，他们之间没有简单的直觉联系，并且无法单独理解。正因如此，使得难以发展关于为何single-decree协议能够工作的直觉。multi-Paxos规则的组成，添加了而外的复杂性与精巧。我们相信在多个决定上达成一致能够被分解成其他更直接与显然的方式。

Paxos的第二个问题是，不能作为构建实际应用的好基础。一个原因是，在multi-Paxos上没有广泛的共识。Lamport的描述几乎都是single-decree Paxos；他描述multi-Paxos的可能方法，但是缺少很多细节。已经有几次试图重试和优化Paxos，但是这些都与Lamport的草图有差别。Chubby系统实现了Paxos类似算法，但是其实现的多数没有被发布。

而且，Paxos架构对于构建应用系统来说是贫瘠的，这是single-decree分解的另一个结果。例如，对于选择一个日志条目集合，将他们合并到一个顺序的日志中就没有多少益处，这仅仅是添加了复杂度。将新条目按照强制顺序添加的方式，来设计一个日志系统（a system around a log）将更简单与高效。另一个问题是，Paxos使用一个端到端均衡的方法作为其核心（尽管他最终建议使用一个弱的leadership作为性能优化）。这给人印象，在

一个简化的世界里只有一个决定会被创建，但是在现实系统中很少使用这种方法。假如有一些列的决定会被创建，首先选择一个leader将是简单与快速的，然后使用leader来协调这些决定。

因此，应用系统与Paxos很少相似。每个实现都从Paxos开始，发现实现他的困难，然后构造一种显著区别的架构。这是耗时并且易错的，并且Paxos的难理解使得问题更加严重。Paxos的表述对于证明理论可能是一个好方法，但是现实应用与Paxos如此不同，以至于该证明没有太多价值。下面来自Chubby的实现者们的评论很典型：

> Paxos算法与现实系统需求上存在显著的鸿沟…最终的系统将基于未证明的协议。

因为这些原因，我们断言，Paxos并没有为系统构建与教育提供一个很好地基础。基于一致性算法在大型软件系统上的重要性，我们决定看看是否可能设计一个相较Paxos有更好特征的替代算法，Raft就是这个经历的结果。

4、可理解性设计

在设计Raft中有几个目标：1）必须完整地为系统构建提供一个完整与现实的基础，因此它显著地减少开发者的设计工作量；2）保证在各种条件下的安全性与典型操作条件下的可用性；3）保证一般性的操作效率；但是我们最主要的目标，与最困难的调整是，可理解性。4）必须能够让大量的受众确切地明白该算法。因此系统构建者在现实的应用中可以做不可避免的扩展或延伸。

在设计Raft中有大量的要点需要在可能的方法中选择。此时，评估候选者是基于可理解性：解析每个候选方法有多难（例如，状态空间的复杂程度，是否有晦涩的相关前提或暗示？），与读者完全理解该方法及其相关的前提或暗示有多容易。

我们意识到在这些分析上有很高的主观意识；尽管如此，首个可应用的技术是广为所知的问题分解：只要可能，将问题分解成可处理的，能解释的，独立理解相关的小问题。例如，Raft就被分解成Leader election，log replication，safety，与membership changes。

第二个方法是减少需要被识别的状态个数以此来简化状态空间，尽可能地增强系统的健壮性与较少不确定性。具体地，日志不允许有空洞，与Raft限制日志之间可能变成不

一致的方式。尽管多数时候我们尝试消除不确定性，但是有些情况，不确定性确实提高了可理解性。特别地，以随机化的方法讨论未决性(randomized approaches introduce nondeterminism)，但是他们倾向于以流行的方式处理所有可能的选择来减少状态空间。我们使用随机化来简化Raft中Leader的选择算法。

5、Raft一致性算法

Raft是用来管理Section 2描述的复制日志的算法。图2描述了算法精要，图3列举了算法的关键属性；这些图片中的精确讨论将在下面依次展开。

Raft首先通过选择一个清晰的Leader来应用一致性，之后给予Leader完全的权限来管理复制日志。Leader从客户端接收日志项，将其分发到其他服务器，并且告诉服务器，什么时候可以安全地应用日志项到他们的状态机上。有一个Leader简化了日志的管理操作。例如，Leader不需要咨询其他服务器就能够决定新条目写入的日志，并且数据流也是简单地从Leader到其他服务器。一个Leader允许失败或者与其他服务器失联，此时新的Leader将被选取出来。

给定的Leader方法，Raft将一致性问题分解成了如下三个子问题，并在下面依次讨论：

1）Leader election：当现存的Leader失败时，一个新的Leader必须被选举出来。

2）Log replication：Leader必须接收从客户端发来的日志项，分发给机器中的服务器，并且要求他们以自己一致。

3）Safety：Raft安全性的关键属性是在图3中描述的状态机安全性：假如任何服务器已经将特定的日志项应用到它的状态机，那么其他服务器就不能将其他的命令应用同一个日志索引。Section5.4将描述Raft如何保证该特性。在选择机制解决方案涉及到的其他限制在Section5.2中描述。

State	
	Persistent state on all servers： **所有服务器的持久化状态：**

State	
	在回复给RPCs之前需要更新到持久化存储之上
currentTerm 当前纪元	服务器看到的最后一个纪元（首次启动时初始化为零，单调递增）
votedFor 投票选择	当前纪元收到的候选者ID（若没有则为null）
log[]	日志条目；每个条目有关于状态机的命令和从Leader接收到条目时的纪元（数组索引从1开始，不是0）
	Volatile state on all servers： 所有服务器上的不稳定状态：
commitIndex	已提交日志条目的最高索引值（初始值为零，单调递增）
lastAppllied	应用到状态机的日志条目的最高索引值（初始值为零，单调递增）
	Volatile state on leaders： **Leaders**的不稳定状态：
	选举后重新初始化
nextIndex[]	对于每一个服务器，记录着下一个要发送的日志条目索引值（初始值为leader的最后一个日志索引+1）
matchIndex[]	对于每一个服务器，已知被复制到该服务器的最高日志条目索引值（初始值为零，单调递增）

关于commitIndex与lastApplied两个字段区别，请看后面。

Rules for Servers	
All Servers：	
1、	如果commitIndex > lastApplied：则递增lastApplied，应用 log[lastApplied] 到状态机之中（5.3）
2、	如果Rpc请求或回复包括纪元T > currentTerm： 设置currentTerm = T，转换成follower（5.1）
Followers（5.2）	
1、	回复candidates与leaders的RPC请求
2、	如果选举超时时间达到，并且没有收到来自当前**leader**或者要求投票的候选者的**AppendEnties RPC**调用：转换角色为**candidate**

Rules for Servers	
Candidates（5.2	
1、	转换成candidate时，开始一个选举： 递增currentTerm；投票给自己；重置election timer；向所有的服务器发送RequestVote RPC请求
2、	**如果获取服务器中多数投票：转换成Leader**
3、	如果收到从新Leader发送的AppendEnties RPC请求：转换成follower
4、	如果选举超时时间达到：开始一次新的选举
Leaders：	
1、	给每个服务器发送初始空的AppendEntires RPCs（heartbeat）；指定空闲时间之后重复该操作以防止election timeouts（5.2）
2、	如果收到来自客户端的命令：将条目插入到本地日志，在条目应用到状态机后回复给客户端（5.3）
3、	如果last log index >= nextIndex for a follower：发送包含开始于nextIndex的日志条目的AppendEnties RPC 如果成功：为follower更新nextIndex与matchIndex（5.3） 如果失败是由于日志不一致：递减nextIndex然后重试（5.3）
4、	*如果存在一个N满足 N>commitIndex，多数的matchIndex[i] >= N，并且log[N].term == currentTerm：设置commitIndex = N （5.3，5.4）*

AppendEntries RPC	
	在回复给RPCs之前需要更新到持久化存储之上
Arguments：	
term	leader服务器的纪元
leaderId	Follower可以通过该ID为客户端进行重定向
prevLogIndex	新条目的前一个日志条目索引（新条目也就是entries）
prevLogTerm	新条目的前一个日志条目索引的纪元 term of prevLogIndex entry
entries[]	用于保存的日志条目（若为空则为心跳消息，可能为了高效而一次发送多条记录）

AppendEntries RPC	
leaderCommit	leader's commitIndex
	Results：
term	currentTerm，for leader to update itself
success	true：follower包括的日志条目中匹配prevLogIndex，prevLogTerm
	Receiver implementation
1、	如果term < currentTerm 则返回false （5.1）
2、	如果日志不包含一个在preLogIndex位置纪元为prevLogTerm的条目，则返回false（5.3）
3、	如果一个已存在的条目与新条目冲突（同样的索引但是不同的纪元），则删除现存的该条目与其后的所有条目（ all that follow it）（5.3）
4、	将不在log中的新条目添加到日志之中
5、	如果leaderCommit > commitIndex，那么设置 commitIndex = min(leaderCommit, index of last new entry）

RequestVote RPC	
	由候选者发起用于收集选票
Arguments：	
term	候选者的纪元
candidateId	候选者请求投票
lastLogIndex	候选者最后日志条目的索引（5.4）
lastLogTerm	候选者最后日志条目的纪元（5.4）
	Results：
term	currentTerm，for candidate to update itself
voteGranted	true：表示候选者获得选票

RequestVote RPC	
	Receiver implementation
1、	如果term < currentTerm 则返回false （5.1）
2、	如果本地的voteFor为空或者为candidateId，并且候选者的日志至少与接受者的日志一样新，则投给其选票

图2：Raft一致性算法的概述（不包括成员变更与日志压缩）。在上述左边表格内的服务器行为被用于描述一些列独立触发并且重复的规则（State、AppendEntiesRPC）；章节名称如5.2指示特性被讨论的地方。在引文【31】中有更精确的算法描述。

Raft guarantee	
Election Safety	在一个特定的纪元中最多只有一个Leader会被选举出来（5.2）
Leader Append-Only	Leader不会在他的日志中覆盖或删除条目，他只执行添加新的条目（5.3）
Log Matching	如果两个日志包含了同样index和term的条目，那么在该index之前的所有条目都是相同的。then the logs are identical in all entries up through the given index.（5.3）
Leader Completeness	如果在一个特定的term上提交了一个日志条目，那么该条目将显示在编号较大的纪元的Leader的日志里。then that entry will be present in the logs of the leaders for all higher-numbered terms。（5.4）
State Machine Safety	如果一个服务器在一个给定的index下应用一个日志条目到他的状态机上，没有其他服务器会在相同index上应用不同的日志条目。

图3：Raft保证在任何时候这些属性都是正确的。章节编号是具体讨论的位置。

5.1 Raft Basics

一个Raft机器包括多台服务器，典型的是5台，使得系统可以容忍两台服务器的失败。在任何时刻每台服务器处于下面三种状态中：Leader、Follower、Candidate。在通常的操作中，将确切地有一个Leader而其他的服务器都是Follower。Follower是被动的：他们不会发出任何请求，只是响应来自Leader和Candidate的请求。Leader处理所有客户端的请求（如果一个客户端向Follower请求，Follower将重定向到Leader）。第三个状态，

Candidate，用于选择一个新的Leader将在5.2节描述。图4为状态变迁图；在后面将讨论变迁过程。

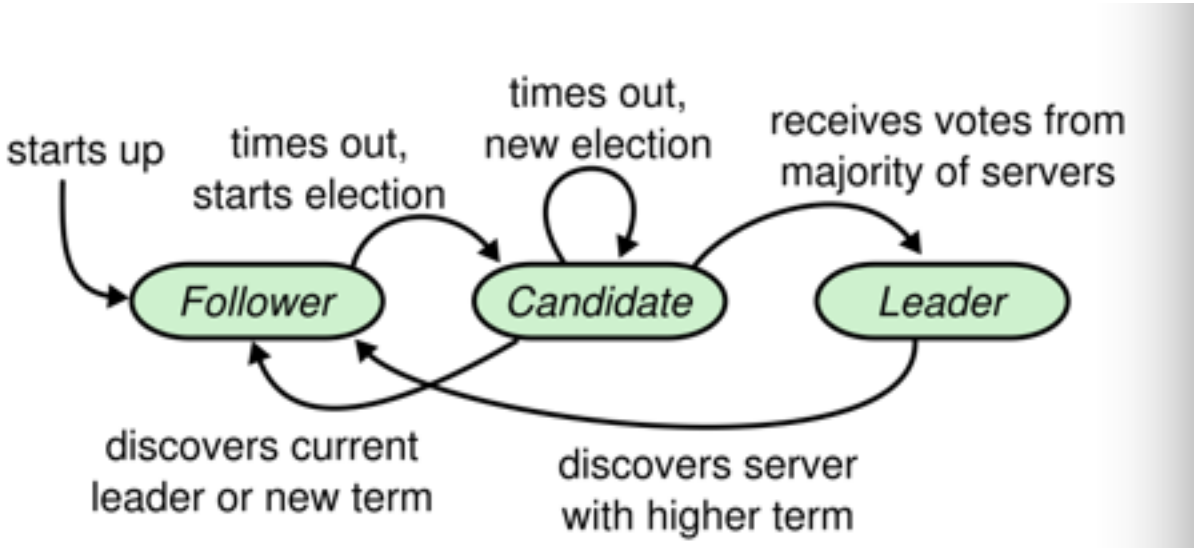


图4：服务器状态图。Follower只响应来自其他服务器的请求（不包括客户端）。如果一个Follower没有收到通信信息（一段时间），则其变成一个Candidate并初始化一个选举。一个Candidate若收到集群中的多数投票就变成了新的Leader。典型地Leader将一直服务直到失败（宕机）。

Raft将时间以任意长度分割成纪元（Terms），显示在图5. 纪元是一个连续的数字。每个纪元开始于一个选举（Election），也就是一个或多个Candidate试图变成Leader的过程，在5.2节描述。如果一个Candidate赢得了选举，那么他在这个纪元剩下的时间内作为Leader向外提供服务。在某些情况下，选举结果产生分歧（票数相当时）。这种情况下该纪元将在没有Leader下结束。一个新的纪元（一个新的选举）将在短时间内发生。Raft保证在给定的纪元里，最多只有一个Leader（可以没有Leader）。

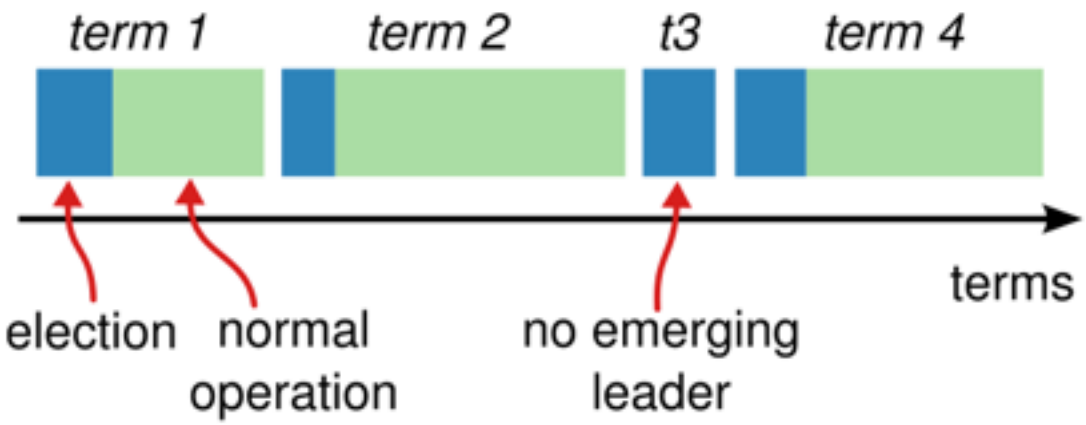


图5：时间被分割成纪元（Terms），每个纪元开始于一次选举。在选举成功后，唯一的Leader管理该集群直到该纪元的结束。有些选举失败（没有得到多数一致），那么这个纪元在没有选举出一个Leader之后就结束。不同服务器上观察到的纪元的转变可能在不同时间完成的。

不同服务器在观察到的纪元的转变上可能是在不同时间，并且在某些情况下，一个服务器可能不会观察到一个选举或整个纪元。纪元在Raft中扮演着一种逻辑时钟的角色【14】，他们使服务器能够检测出过时的Leader信息。每个服务器保存一个当前纪元数值（current term），随着时间的流逝该值是单调递增的。服务器之间通讯时将会交换当前纪元值；如果一个服务器的当前纪元小于另外一个服务器，那么他将更新当前纪元到更大的值。如果一个Candidate或者Leader发现自己的纪元已经过时，会立即转换角色为Follower状态。如果一个服务器收到一个过时纪元的请求，将会拒绝回答。

Raft服务器之间通讯采用远程过程调用（RPCs），并且基础的一致性算法只要求两种类型的RPCs。RequestVote RPCs被Candidate用于在选举时发起（5.2节），而AppendEnties RPCs则被Leader用于复制日志条目与一种类型的心跳（heartbeat）（5.3节）。第7节添加第三种RPC用于在服务器之间传输快照。服务器在一个合理的时间内没有收到回复时会重试该RPCs请求，并且采用并发的方式发送RPCs以提高性能。

5.2 Leader election

Raft使用一种心跳机制来触发Leader election。当服务器启动时，他们作为Follower。一个服务器将保持作为Follower状态，直到他们收到来自Leader或者Candidate的RPCs。Leaders周期性的发送心跳（没有携带日志条目的AppendEnties RPCs）给所有的Follower来保持他们的授权。如果一个Follower在一个election timeout时期内都没有收到信息，那么假设没有可用的Leader，并且开始一个选举来选择一个新的Leader。

为了开始一个选举，Follower递增他的当前纪元并且转换成Candidate状态。接着投票给自己，并行地发送RequestVote RPCs给集群中的其他服务器。Candidate保持这种状态

直到下面三种情况发生：1）他自己赢得了选举；2）另外一个服务器确立他为Leader；3）一个周期时间过去但是没有任何人赢得选举。这些结果将在后面的段落中讨论。

一个Candidate赢得选举表示：在相同的纪元里，他收到集群中多数服务器的投票。每个服务器在给定的纪元里将会投票给最多一个Candidate，采用先到先得的方式。多数原则保证在特定的纪元里只有一个Candidate会赢得选举（选举的安全属性在图3描述）。一个Candidate一旦赢得选举，他变成Leader。接着就会发送心跳消息给所有其他服务器来确立他的权威与防止一个新的选举发生。

在等待投票过程中，一个Candidate可能收到来自其他服务器宣称为Leader的AppendEnties RPCs请求。如果该Leader的纪元（包含在RPC里面）至少与该Candidate的当前纪元一样大，那么Candidate承认该Leader为合法的并且转变自身为Follower状态。如果在RPC中的纪元小于Candidate的当前纪元，那么Candidate将拒绝该RPC并且继续保持在Candidate状态。

第三种结果表示Candidate在一个选举中既没有赢也没有失败：如果有其他Follower也在同时变成Candidate，投票可能被分裂（split votes）（各得到几张）以至于没有Candidate能够得到多数票。当这种情况发生时，每个Candidate将会超时并且开启一个新的选举，使用递增的纪元并发起另外一轮的RequestVote RPCs。然而，没有外部干涉分裂的投票可能无限继续下去。

Raft使用随机化的选举超时时间来减少分裂投票发生并且能够快速解决。1）为了防止分裂投票在首次时发生，选举超时时间是在一个固定时间间隔内随机产生的（例如，150-300ms）。这就分散了服务器以至于在多数情况下只有一个服务器超时（在某个时刻只有一个服务器超时，不会同时多个超时）。他会在其他服务器超时前赢得选举并发送心跳。2）同样的机制也用于处理分裂投票。每个Candidate在开始选举时随机化他的选举超时时间，并且在开始下一次选举之前要等待该超时时间消逝。这样就减少了在下一个新的选举上出现分裂投票的发生。9.3节显示了这种方式能够快速选举一个Leader。

选举是一个”可理解性”如何影响在可选的设计中选择的例子之一。最初，我们计划使用等级系统（ranking system）：每个Candidate被赋予一个唯一的等级（rank）。如果一个Candidate发现另外一个Candidate有更高的等级，那么他将返回Follower状态，使得更

高等级的Candidate能够更容易的赢得下一场选举。我们发现这种方法会在可用性上创建一个晦涩的问题（假如一个高优先级的服务器失败（宕机），一个低等级的服务器可能需要超时并且再次变成一个Candidate，但是如果他这么做太快，他会重置掉一个正在进行的选举）。我们多次调整该算法，但是每次调整都会出现新的问题。我的的最后结论是随机化的重试方法是更显而易见与可理解的。

5.3 Log replication

一旦Leader被选举出来，他就开始为客户端服务。客户端的每次请求都包括要被复制状态机（replicated state machines）执行的命令。Leader将该命令作为一个新的条目插入到他的日志文件，然后以并发方式向其他服务器发起AppendEntries RPCs来复制该条目。当该条目被安全地复制后（后面描述），Leader应用该条目到其状态机并且返回该执行结果给客户端。如果Follower失败或者运行缓慢，或者网络包丢失，Leader将无限次重试该AppendEnties RPCs（即使该Leader已经回复消息给客户端）直到所有的Follower最终都保存了该条目。

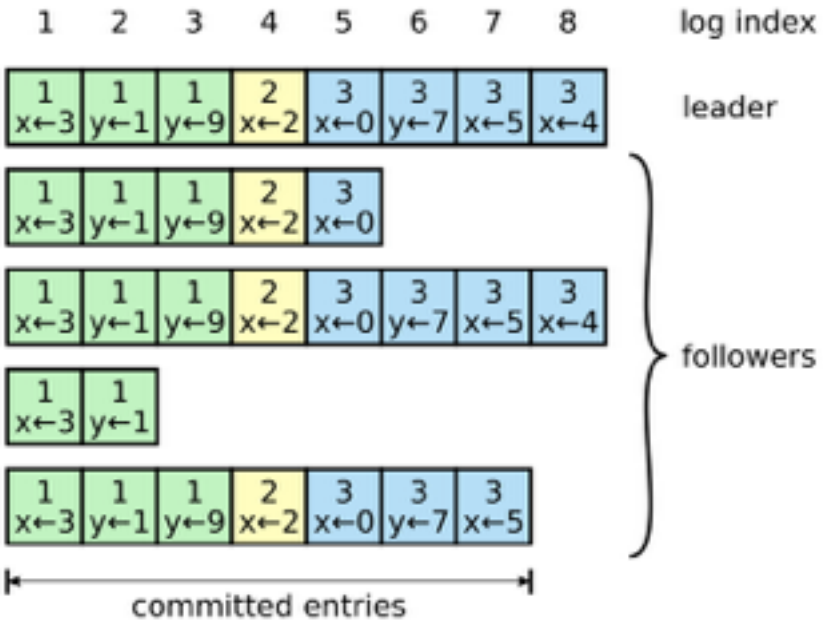


图6：日志由条目组成，按顺序编号。每个条目包括创建时的纪元（每个格子中的号码）与状态机命令。当条目被安全的应用到状态机就认为该条目提交（commited）。

Logs被组织成图6的样子。每条日志条目保存着一个状态机命令与从Leader接收到该条目时的纪元。日志条目中的纪元被用于检测日志间的不一致，来确保图3中的一些属性。每个日志条目还有一个数值型的索引用来标识其在日志中的位置。

Leader会判断什么时候将条目应用到状态机是安全的；这样的条目叫做已提交。Raft保证已提交的条目是持久的并且最终会被所有可用的状态机（其他服务器的状态机）所执行。创建条目的Leader在完成将条目分发给多数的服务器后立即提交（例如图6中的条目7）。这使得在Leader日志中之前的条目也得到提交，包括由之前的Leader创建的条目（对应Leader转换）。在5.4节将讨论变更Leader后应用该规则的晦涩内容，（when applying this rule after leader changes），并且显示了这种提交的定义是安全的。Leader保持他提交的最大index，该index会被用于后面的AppendEntries RPCs（包括心跳），使得其他服务器最终会知道Leader提交的位置。当一个Follower发现一个日志条目在Leader中已经提交，就会按顺序将条目应用到自己的状态机上。

注：客户端的一次日志请求操作触发：1）Leader将该请求记录到自己的日志之中；2）Leader将请求的日志以并发的形式，发送AppendEntries RCPs给所有的服务器；3）Leader等待获取多数服务器的成功回应之后（如果总共5台，那么只要收到另外两台回应），将该请求的命令应用到状态机（也就是提交），更新自己的commitIndex和lastApplied值；4）Leader在与Follower的下一个AppendEntries RPCs通讯中，就会使用更新后的commitIndex，Follower使用该值更新自己的commitIndex；5）Follower发现自己的commitIndex > lastApplied 则将日志commitIndex的条目应用到自己的状态机（这里就是Follower提交条目的时机）。

关于commitIndex与lastApplied两个字段区别：对于Leader来说，将日志条目分发出去之后，收到多数的回复就将该命令应用到状态机（同时更新commitIndex与lastApplied），因此是同时更新这两个值的；而对于Follower来说，则是通过Leader发送来的AppendEntries RPCs请求，发现其中的leaderCommit>commitIndex时，更新commitIndex=leaderCommit，

接着判断如果commitIndex>lastApplied则再将该commitIndex的日志条目应用到状态机，因此是两个步骤完成更新的。

我们设计的Raft日志机制是为了在多服务器之间保持高级别的一致。不但是为了简化系统的行为与使之更加可预测的，而且是确保安全性的重要组件。Raft保持了如下特性，一起组成了图3中的日志匹配属性（Log Matching Property）：

1）如果在不同日志中的两个条目有同样的index与纪元，那么他们存储着同样的命令。

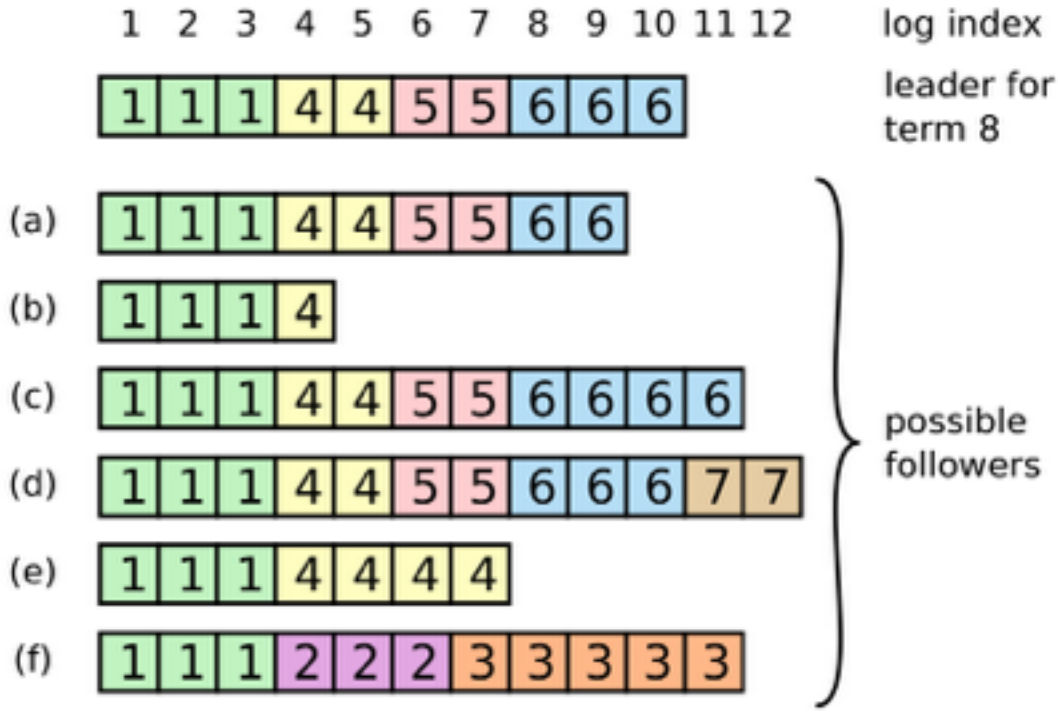
2）如果在不同日志中的两个条目有同样的index与纪元，那么这些日志所有前面的条目是相同的（preceding entires）。

注：不同的日志，应该是表示不同服务器的日志

前一个特性来自，一个Leader在给定的log index与纪元上最多只会创建一个条目，并且日志条目不会改变其在日志文件中的位置。第二个属性是由AppendEntries执行的一个简单地一致性检查保证的。当发送一个AppendEntries RPC，Leader在紧接着新条目的后面包含着前一个条目的index和纪元（prevLogIndex，prevLogTerm两个字段）。如果Follower在他的日志中找不到该index和纪元的条目，那么他拒绝新条目。该一致性检查就像是一个递归步骤：初始时，空的日志状态符合日志匹配属性，并且一致性检查维护日志扩展中的该属性。因此，无论何时AppendEntries返回成功，Leader知道Follower的log与自己的log在该新条目与之前是相同的。

在通常操作过程中，Leader的日志与Follower保持一致，因此AppendEntries一致性检查总是不会失败。然而，Leader奔溃能够导致日志的不一致（旧的Leader可能没有将其日志中的所有条目复制出去）。这些不一致能够在一系列的Leader与Follower崩溃中增加。图7描述了Follower的日志与新的Leader不一致的方式。一个Follower可能没有当前已经在新

Leader上的条目，也可能有一些不在新Leader上的条目。缺失与增多的（Missing and



extraneous）条目在一个日志中可能跨越多个纪元。

图7：当在顶端的Leader获取权力时，在Follower中可能出现a-f的场景。在格子中间的数值表示纪元。一个Follower可能缺少条目（a-b），也可能有而外的未提交的条目（c-d），或者两者都出现（e-f）。例如，场景（f）可能发生在该服务器在纪元2时是Leader。添加多个条目到他的日志，然后在提交他们之前崩溃了。然后马上重启，在纪元3时又变成Leader，并且添加其他一些条目到他的日志，在纪元2到纪元3之间的条目被提交之前，该服务器再次崩溃，并且持续了多个纪元。

注：Leader在接收到客户端的请求后，会将条目先写入到自己的日志之中，然后并发地使用AppendEntries RPCs发送给其他服务器，当收到多数服务器的成功返回后，将条目应用到状态机，然后才返回给客户端。因此，如果Leader未来得及将日志复制出去，就宕机，那么客户端必定能够检测到，该请求失败了。(Leader收到客户端的请求后，处理成功了，在返回给客户端前网络断开，因此客户端最终得不到通知，当下次网络通之后，客户端重试该请求，Raft要求命令是幂等的)

Raft中，Leader以强制Follower复制其的日志的方式来处理不一致（就是强制要求Follower必须与Leader一致）。意味着Follower日志中的冲突条目将被来自Leader的日志条目覆盖。在5.4节，将显示在结合更多限制时这样做是安全的。

为了使得Follower的日志与自己保持一致，Leader必须找到该Follower的日志与自己日志相同的最后位置，然后删除掉Follower该位置之后的条目，并且发送Leader该位置之后的条目给Follower。所有这些动作发生在由AppendEntries RPCs执行的一致性检查的响应上。Leader为每个Follower保存一个nextIndex值，表示Leader下次将发送给该Follower的日志条目索引。当一个Leader获得权利时，他将所有的nextIndex初始化成其日志的最后一项（如图7中的11）。如果一个Follower的日志与Leader的不一致，在下一次AppendEntries RPC的AppendEntries一致性检查中将会失败。在一次拒绝之后，Leader会递减该nextIndex并且再次尝试AppendEntries RPC。直到最终nextIndex到达Leader与Follower日志一致的位置。此时，AppendEntries将会成功，会删除Follower日志中冲突的条目，并且插入来自Leader日志的条目。一旦AppendEntries成功，Follower的日志就与Leader一致了，并且会在接下来的纪元里保持。

如果需要，该协议可以优化被AppendEntries RPCs拒绝的次数。例如：当拒绝一个AppendEntries请求时，Follower可以包含冲突条目的纪元与在该纪元下保存的第一个索引。根据这些信息，Leader能够递减nextIndex来跳过该纪元下所有冲突的条目；一个AppendEntries RPC要求冲突条目的每个纪元，而不是一个RPC一个条目。我们怀疑该优化的必要性，因为错误发生并不常见，并且也不会出现很多的不一致条目。

介于这种机制，Leader在获取权利时并不需要使用一些特殊的动作来保存日志一致性。他只是开始一个普通的操作（normal operation），并且日志在相应错误的AppendEntries一致性检查时会自动的汇聚。Leader永远不需要覆盖或者删除他自己的日志（图3 Leader Append-Only 属性）。

日志复制机制显示了在图2中描述的一致性特性的价值：Raft能够接受，复制，与应用新的日志条目，只要多数服务器处于工作状态；通常情况下，一个新的条目能够被一轮的RPCs复制给集群中的多数机器；并且单个慢得Follower不会影响执行性能。

5.4 Safety

上述的章节描述了Raft如何选举Leader与复制日志条目。然而，到目前为止该机制的描述还不能完全有效的保证，每个状态机确切地按照同样的顺序执行同样的命令。例如：一个Follower可能在Leader提交多条日志条目时处于不可用状态，然后他被选举为Leader并且使用新条目覆盖这些条目；结果，不同的状态机可能执行了不同的命令序列。

注：1）Server-A（Leader）提交了条目7-10，Server-B（Follower）处于不可用状态；2）Server-A宕机；3）Server-B恢复，并且变成Leader，此时他最后一个条目是6，并且开始接受客户端的请求；4）Server-B（新的Leader）收到新的条目写入到7-10，并提交。5）Server-A恢复，变成Follower，Server-B复制新的条目7-10覆盖到Server-A的同样位置，此时Server-A与Server-B提交的命令其实就不一样了。

这章添加一个哪些服务器可以被选举为Leader的限制来完善Raft算法。该限制保证了Leader在任何给定的纪元，包含了前一个纪元所有已经被提交的条目（来自图3的Leader完整性特性）。给定该选举限制，我们接着使提交的规则更加精确。最终，我们描述了Leader完整性属性的证明并且显示了他如何使得复制状态机有正确的行为。

5.4.1 Election restriction

在任何Leader-Base一致性算法中，Leader最终必须保存所有已经提交的日志条目，在某些一致性算法中，例如Viewstamped Replication[22]，一个初始时并不包括所有已提交条目的服务器也能够被选举为Leader。要么在选举过程中或选举之后，这些算法包括了附加的机制来识别缺少的条目与传输这些条目到新的Leader上。不幸的是，这导致了相当多的机制与复杂性。Raft使用一个简单地方法，他保证来自前一个纪元的所有已提交的条目出现在通过选举产生的新的Leader上，而不需要传输这些条目到新的Leader。意味着，日志条目只会按照一个方向流动，从Leader到Follower，并且Leader永远不会覆盖他们日志里已存在的条目。

Raft使用选举过程来预防，一个Candidate，只有他的日志包括所有(Leader)已提交的条目(Leader提交并不意味着这个Candidate也提交，只是意味着多数的服务器持有该条目的日志），才可能赢得选举。Candidate必须联系集群中的超过一半的服务器来进行一

个选举，意味着任何一个已提交的条目至少必须出现在这些服务器的一台上（注：Leader只有在多数服务器返回AppendEntries RPC成功时，才会提交该条目条目）。如果Candidate的日志至少与这些服务器的任何一台同样新（up-to-date）（up-to-date将在后面准确定义），那么他就持有了所有那些已提交的条目（then it will hold all the committed entries）。RequestVote RPC实现了这种限制：RPC包括了Candidate的日志信息，并且当投票者的日志比Candidate更新时会拒绝该投票。

注：RequestVote RPC请求包括四个字段: Term、CandidateId、LastLogIndex、LastLogTerm。

Raft决定两个日志那个更新，采用比较日志的最后一个条目index与纪元的方式。先看纪元，纪元更大的更新，若相等，则继续比较条目的index，index更大的更新。

5.4.2 Commiting entries from previous terms

就像5.3节所描述的，一旦条目被保存到多数的服务器上，Leader就知道当前纪元的该条目已经被提交。如果Leader在提交条目之前宕机，后续的Leader将会尝试完成该条目的复制操作。然而，Leader不能绝对地断定，一个前一个纪元的条目被保存到多个服务器上时就一定已经提交。图8展示了一种情况，一个旧的条目即使保存到多数服务器上，依然可能会被之后的Leader给覆盖掉。

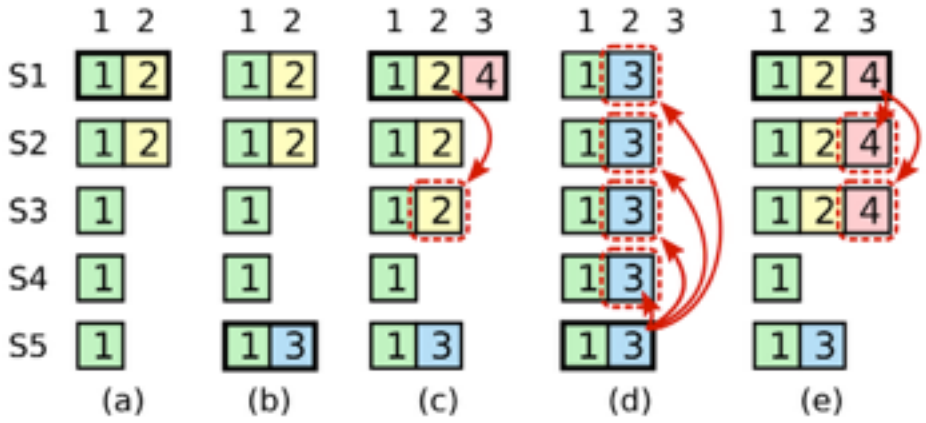


图8：显示了Leader为何不能决定提交旧纪元的日志条目的时间序列。

(a) S1是Leader，并且部分地复制了index-2；

(b) S1宕机，S5得到S3、S4、S5的投票当选为新的Leader（S2不会选择S5，因为S2的日志较S5新），并且在index-2写入到一个新的条目，此时是纪元3（注：之所以是纪元3，是因为在纪元2的选举中，S3、S4、S5至少有一个参与投票，也就是至少有一个知道纪元2，虽然他们没有纪元2的日志）；

(c) S5宕机，S1恢复并被选举为Leader，并且开始继续复制日志（也即是将来自纪元2的index-2复制给了S3），此时纪元2的index-2已经复制给了多数的服务器，但是还没有提交；

(d) S1再次宕机，并且S5恢复又被选举为Leader（通过S2、S3、S4投票），然后覆盖Follower中的index-2为来自纪元3的index-2；（注：此时出现了，来自纪元2中的index-2已经复制到三台服务器，还是被覆盖掉）

(e) 然而，如果S1在宕机之前已经将其当前纪元（纪元4）的条目都复制出去，然后该条目被提交（那么S5将不能赢得选举，因为S1、S2、S3的日志是4比S5都新）。此时所有在前的条目都会被很好地提交。

注：之所以关注该问题，是由于当一个新Leader当选时，由于所有成员的日志进度不同，很可能需要继续复制前面纪元的日志条目。因为即使为前面纪元的日志复制到多数服务器并且提交，如步骤C。但是在D中还是可能被覆盖，这就产生了不一致。

为了避免图8所描述的问题，Raft采用计算副本数的方式，使得永远不会提交前前面纪元的日志条目。通过计算副本数，只有Leader的当前纪元被提交；一旦来自当前纪元的条目采用这种方式提交了，那么由于Log Matching Property使得早先的条目会被间接地提交。在某些情况下，Leader可以确定一个旧的日志条目已经被安全提交（比如，该条目保存到所有的服务器上），但是Raft为了简单化，而采用更加保守的方法。

注：此处的意思是，在c阶段，S1成为Leader，此时的纪元是4。S1一样向其他服务器发送日志2，当发送到多数服务器S1，S2，S3时，此时并不提交该日志，而是继续复制日志4，直到日志4到达多数服务器后，提交日志4，由于Log Matching Property属性，会自动提交日志2（包括自己与其他服务器）。如果提交了4之后宕机，S5就不会被选举为新的Leader，如果在提交4之前宕机，那么日志2，日志4还是可能被覆盖，但是由于没有提交，也就没有执行日志中的命令，即使被覆盖也无关系。

当Leader复制前一个纪元的日志条目时，日志条目保持其原来的纪元，使得Raft在提交规则上产生而外的复杂性。在其他一致性算法中，如果新的Leader复制先前纪元的条目，他必须使用新的纪元来做这件事。Raft的方式使得日志条目容易被判断（reason），因为他们全程使用同一个纪元。此外的，Raft中的Leader相比其他算法，发送更少之前纪元的条目（其他算法在能够提交前，必须发送冗余的日志条目来重新计算数值）。

5.4.3 Safety argument

在给定完整的Raft算法后，现在能够更加精确地讨论Leader所持有的完全性（Completeness Property）（这些讨论基于安全性证明，见9.2节）。我们假设Leader完全性没有保持，然后证明此时产生冲突。假设来自Term-T的Leader-T在他的纪元里提交了一个日志条目，但是该日志条目没有存储到Leader的将来某个纪元里。设U为最小的纪元U > T，并且Leader-U没有存储该条目。

1、在Leader-U的选举时刻，该条目必须不在Leader-U的日志中（因为Leader不会删除或覆盖一个条目）。

2、Leader-T复制条目给集群中的多数机器，并且Leader-U收到来自集群中多数服务器的投票。意味着，至少存在一个服务器（投票者），同时接收了来自Leader-T的条目，并且跳票给了Leader-U。如图9所示；该投票者就就产生冲突的关键。

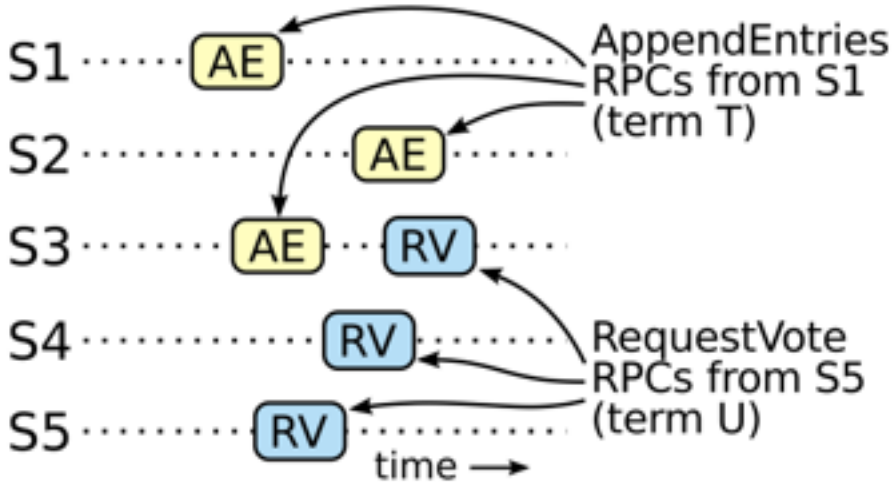


图9：如果S1（纪元T时的Leader）在他的纪元里提交了一个新的条目，S5在之后的纪元U里被选举为Leader，那么必须存在至少一个服务器（S3）既接受了日志条目又投票给了S5.

3、该选民肯定在投票给Leader-U之前，已经接收了来自Leader-T的该已提交的条目；否则他必须拒绝来自Leader-T的AppendEntries请求（因为他的当前纪元比纪元T高）。

4、该选民在投票给Leader-U时，依然存储着该条目，因为任何可能干扰的Leader都包含着该条目（根据假设，U是最小的Leader不包含给条目的纪元），Leader永远不会移除条目，并且Follower只有在与Leader冲突时才移除条目。

5、该选民投票给了Leader-U，因此Leader-U的日志必须至少与该选民一样新。这是导致两者矛盾之一。

6、首先，如果该选民与Leader-U有同样的最后日志纪元，那么Leader-U的日志必须至少与该选民一样长（index大小），因此Leader-U的日志包含了任何在该选民日志里的条目。这是一个矛盾，因此该选民包含了该提交的条目，但是Leader-U的假设是没有该条目。

7、其他情况（该选民与Leader-U有不同的最后日志纪元），Leader-U的最后纪元必须大于该选民的最后纪元。此外，Leader-U的纪元U大于T，因为该选民的最后日志纪元至少为T（它包括了来自T的条目）。创建Leader-U日志最后日志条目的Leader（也就是发生在纪元U选举时，Leader-U的最后一个日志条目创建者）在其日志中肯定包括了该已提交的条目（根据假设U是最早的一个Leader不包括该条目的纪元）。接着，根据Log Matching Property，Leader-U的日志必须也包括该提交的条目，这是一个矛盾。

注：在进行纪元U的选举中，Leader-U的最后一个日志条目的创建者Leader-U-1，根据假设肯定包括了条目T（根据假设U是最早的一个Leader不包括该条目的纪元），另外根据假设该条目T是已提交的，因此就保证了多数的服务器包括了条目T。由于纪元U选举了Leader-U，那么Leader-U的日志只有包括了条目T才有可能得到多数服务器的投票，因此Leader-U的日志必须也包括该提交的条目。

8、这就完成了冲突。因此，所有来自大于纪元T的Leader，必须包括所有在纪元T时提交的条目。

9、Log Matching Property保证了，将来的Leader将包含被间接提交的条目，例如图8-d中的index-2；

给定了Leader Completeness Property（一个已经提交的日志条目，肯定会出现在后面纪元的Leader日志之中），就能够证明图3中的状态机安全属性 State Machine Safety Property，即：如果一个服务器在给定的index上提交了一个日志条目到他的状态机，没有其他服务器会应用一个不同的日志到同样的index。在一个服务器应用一个日志条目到他的状态机时，他的日志与Leader的日志，在该日志条目之前的条目上必须是一致的，而且已经提交。现在考虑所有服务器应用给定日志索引的最小纪元（Now consider the lowest term in which any server applies a given log index），Log Completeness Property保证来自更高纪元的Leader将存储同样的日志条目，因此在后面的纪元里应用该index时会应用同样的内容。因此State Machine Safety Property得到保持。

最后，Raft要求服务器按照日志index来引用条目。集合State Machine Safety Property，这意味着所有的服务器将会确切地应用同样的日志条目以同样的顺序到他们状态机。

5.5 Follower and candidate crashes

到目前为止我们都关注着Leader的异常。Follower与Candidate宕机相对于Leader宕机简单得多，并且他们采用同样的方式处理。如果一个Follower或者Candidate宕机，那么后面发送给他们的RequestVote与AppendEntries RPCs都将失败。Raft使用无限制的重试来处理这种错误；如果服务器恢复了，那么RPC请求将能够成功。如果一个服务器在完成RPC请求的处理，在回复之前宕机（注：请求处理成功了，在发送回复消息时宕机），那么在服务器恢复后，将会收到同样的RPC请求。Raft里面的RPC是幂等的，因此这不会有任何危害。例如：一个Follower收到一个AppendEntries请求，请求中的日志条目已经出现在日志之中，他就会忽略该请求的这个条目（注：一个请求可以发送多个条目，可能其中有部分是已经存在的，那么剩下的还是需要处理）。

5.6 Timing and availablity

Raft的要求之一是，安全性必须不依赖与时间：系统必须不会仅仅因为某些时间比预期更快或更慢发生而产生错误结果。然而，可用性（系统在可以合适的时间内回复给客户端）不可避免地依赖于时间。例如：如果消息交换在服务器宕机与正常时耗时更长，Candidate将不会等待太长时间来赢得选举（Cand will not stay up long enough to win an election）；如果没有一个稳定的Leader，Raft将不能前进

Leader election是Raft中对时间最为关切的一部分。Raft将能够选举与保持一个稳定的Leader，只要满足如下的时间要求（timing requirement）：

broadcastTime <= electionTimeout <= MTBF

broadcastTime是服务器并行地发送RPCs到集群中的各个服务器，并且收到回复的平均时间；electionTimeout是在5.2节中描述的选举超时时间；MTBF为单个服务器宕机的平均时间。

broadcastTime时间应该适当地小于electionTimeout，使得Leader能够放心地发送心跳消息来阻止Follower开始一个新的选举（注：也就是如果ElectionTimeout设置的过小，小于broadcastTime，那么在Leader的心跳消息到来之前，就超时，那么就会导致不断的选举）；使用随机化的方法来设置选举超时时间，这个不等式同样使得不可能产生分裂投票（this inequality also makes split votes unlikely）。electionTimeout应该适当地小于MTBF，使得系统能够继续前进。当Leader宕机，系统将会在electionTimeout之内变得不可用，我们希望这只是整体上非常小的一个时间段。

broadcastTime与MTBF都是由系统决定，相对的electionTimeout是需要我们自己选择的。Raft的RPCs要求接收方将持久化信息到稳定存储中，因此broadcastTime可能在0.5ms ~ 20ms之间，依赖存储技术不同耳钉。因此electionTimeout可能需要在10ms ~ 500ms之间。典型的MTBF都是数个月或更长，很容易满足时间要求。

6、Cluster membership changes

到目前为止我们假设集群的配置（configuration，就是参加到一致性算法之中的服务器）是固定不变的。现实中，偶尔有必要修改该配置，例如，替换宕机的服务器，修改复制级别（degree of replication）。尽管，可以采用将整个集群下线，更新配置文件，然后重启集群的方式完成，这将会导致在修改的过程中集群处于不可用状态。另外，如果有一些手工操作，可能导致操作失败。为了避免这些问题，我们决定自动适应配置修改并且将他们融入到Raft一致性算法之中。

为了来配置变更机制的安全性，在改变的过程中必须不存在某个时刻出现两个Leader被选举。不幸的是，服务器直接从旧配置却换到新配置的任何方法都是不安全的。不可能所有服务器立刻自动却换，因此集群潜在却换的过程中可能分裂成两个独立的多数（如图10）。

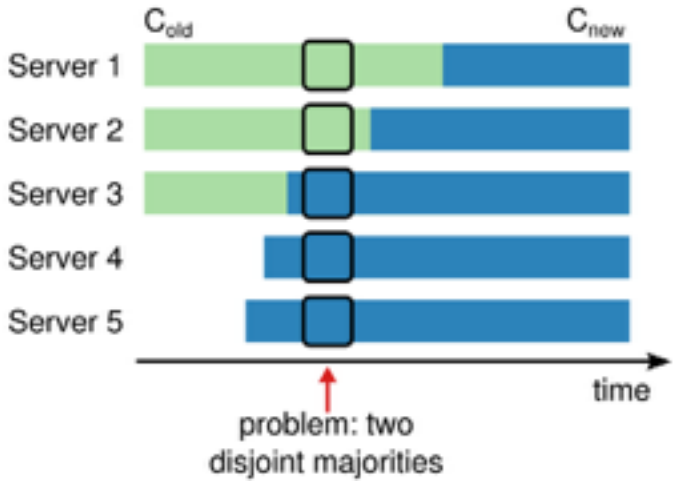


图10：直接从旧配置修改到新配置是不安全的，因为各服务器的却换是在不同时刻进行的。如此处，集群从三台服务器变更到五台服务器（在这个时刻有三台完成却换，另外两台还没有）。不幸的是，在某个点同样的纪元上可能有两个不同的Leader被选举出来，其中一个为在旧的配置上，另外一个为新的配置上。

为了确保安全性，配置变更必须使用两阶段方法（two-phase approach）。实现两阶段有很多方法。例如：有些系统【22】使用第一阶段来使旧配置不可使用，因此他就不能

处理客户端请求；然后第二阶段启动新配置。在Raft中集群首先却换到一种过度配置（transitional configuration）叫做共同一致（joint consensus）；一旦joint consensus已经提交，接着系统却换到新的配置。Joint consensus是旧配置与新配置的结合体。

1）在两个配置中，日志条目被复制到所有的服务器；

2）来自新旧配置中的任何服务器都可能成为Leader；

3）Agreement（关于选举与条目提交）要求在旧配置与新配置上的独立多数同意。

Joint consensus在不妥协安全的情况下，允许独立的服务器在不同时刻却换配置信息。此外，joint consensus允许集群在却换配置时继续向客户端提供服务。

集群配置使用特殊的条目在复制日志时被存储与通讯(集群配置也是一个日志项，只是内容为配置信息而不是日志，同样该日志项需要通过Leader进行分发与提交和应用)；图11显示了配置变更过程。当Leader接受到一个请求，要求将配置从C-old却换成C-new，他会为Joint consensus保存配置（图中使用C-old-new表示）为一个日志条目并且使用之前描述的机制复制出去。当一个特定的服务器添加新配置到他的日志中，使用该配置到所有将来的决定中（一个服务器总是使用最后的配置到他的日志中，不管该条目是否被提交）。这意味着Leader将使用C-old-new的规则来决定日志条目的提交。如果Leader宕机，新的Leader可能在C-old或者C-old-new上被选举出来，这取决于Candidate是否接收到C-old-new。在任何情况下，C-new在该过程中不会单方面做出决定。

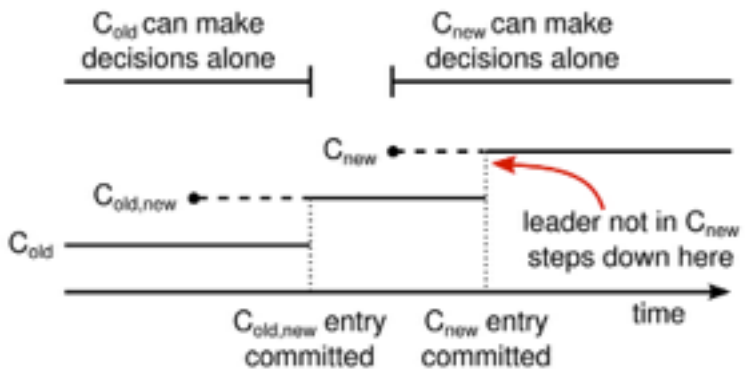


图11：配置变革的时间线。横向的破折线表示配置条目已经被创建但是还没有提交，实线显示最后提交的配置条目。Leader首先在其日志中创建C-old-new配置条目，并提交C-old-new（得到C-old与C-new中多数的允许）。接着，创建C-new条目，当得到C-new多数允许时提交。在这个过程中，没有任何一个点C-old与C-new能够同时独立做出决定。

一旦C-old-new已经被提交，不管C-old还是C-new在没有得到其他人得批准时都不能做出决定，并且Leader Completeness Property确保，只有一个具有C-old-new日志条目的服务器能够被选举为Leader。现在Leader能够创建一个日志条目来描述C-new，并且复制该条目到集群中。再次，每个服务器在见到该条目之后，配置就会生效（每个服务器见到C-new，就是用C-new的配置）。在C-new的规则下，当新配置已经提交，旧配置就无关紧要了，并且不在新配置中的服务器可以关机。就如图11，没有任何一个时刻C-old与C-new同时可以决出决策；这保证了安全性。

在重配置时，还有三个问题需要提出。1）新的服务器可能并没有保存任何日志条目。如果以这种状态添加到集群，则需要一段时间来追赶，在这段时间，他不能提交新的日志。为了避免可用性间隙（gaps），Raft提出一个在配置变更前的附加阶段，新服务器作为没有投票权限的成员，添加到集群中（Leader也复制日志给他们，但是并不把他们作为多数的成员 but they are not considered for majorities）。一旦新服务器已经追上集群，那么开始上面描述的重配置过程。

2）当前的Leader可能不是新配置的成员。此时，Leader会在提交C-new日志条目之后让位（变成Follower状态)。这意味着，将会有一段时间（提交C-new之后），Leader会管理一个不包括自己的集群；他会复制日志到集群，但是并不把自己计算到多数中。当C-new被提交时，将发生Leader的转换，因为这时新配置才能够独立操作（将总是能够在C-new配置下选择一个Leader）。在此之前，他可能是只能从C-old下选举一个Leader。

3）移除不在C-new中的服务器可能扰乱集群。这些服务器将不会再接收到心跳，因此他们会选举超时，并开始一个新的选举。他们将会发送RequestVote RPCs给C-new中的成员，这会导致C-new中当前的Leader变成Follower状态。一个新的Leader最终会被选举出来，但是之前那些服务器会再次超时，然后重复上述操作，导致了低可用性。

为了防止这个问题，服务器在确认当前Leader存在时，会忽略RequestVote RPCs。特别地，如果一个服务器在当前Leader的最小选举超时之内，收到一个RequestVote RPC，他不会更新纪元或者投出选票。这不会影响正常的选举，每个服务器会在开始一个选举之前，至少等待一个最小的选举超时。然而，着避免了来自被移除服务器的干扰。如果一个Leader能够从集群中获取心跳，那么他就不会被更高的纪元废黜。

7、Log compaction

Raft的日志在正常的操作过程中不断增长，来合并更多客户端请求，但是在一个现实系统中，他不能无限制增长。一旦日志太大，它需要更多地空间与更长的时间来replay。在没有采用一定机制来丢弃日志里累积起来的过时的信息，将会导致可用性问题。

快照是压缩的最简单方法。在快照中，当前系统完整的状态最为一个快照写入到持久化存储中，那么在这个点之前的完整日志可以丢弃。快照在Chubby与ZooKeeper中都有使用，该节的后续将简述Raft中的快照。

采用增量的方法来压缩，例如log cleaning【36】与log-structured merge trees【30，5】，都是可行的。这些操作每次只处理数据的一部分，因此他们可以分摊压缩的负载。他们首先选择数据的一个，已经累积了大量删除与覆盖对象的区域，然后他们覆盖当前活动的对象到这个区域，然后释放该区域。这相对于快照，简单地在整个数据集合上操作，要求显著的附加机制与复杂性。当日志清理需要修改Raft，状态机可以以实现LSM trees同样的接口来实现快照。

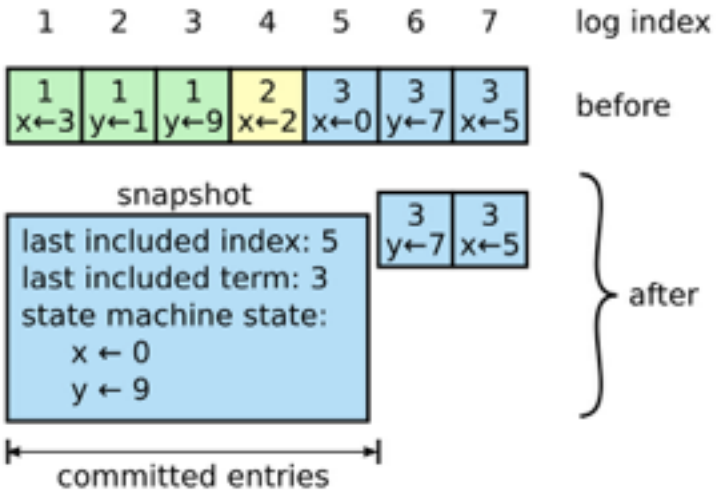


图12：一个服务器使用一个新的快照（仅仅包括当前状态，例子中的x与y）替换日志中已提交的条目（index 1-5）。快照的last included index与term用于指明快照的位置是出于条目6的前面。

图12显示了Raft中快照的基本想法。每个服务器各自创建快照，只保存日志中的已提交条目。主要工作包括将状态机的当前状态写入到快照中。Raft也在快照中加入一些metadata：last included index为被快照取代的最后条目在日志中的索引（应用到状态机的最后一个条目），last inclued term，表示该条目的纪元。保留这些用于支持在快照之后的首个条目的AppendEntries一致性检查，因为该条目需要有前一个条目的log index与纪元。为了能够实现集群关系改变（第六节），快照也将最新的配置作为日志的最后一个条目保存。

InstallSnapshot RPC	
	由Leader调用用于向Follower发送快照chunks。Leader总是按照顺序发送chunks。
Arguments：	
term	Leader当前的纪元
leaderId	**使得Follower能够为客户端请求进行重定向**
lastIncludeIndex	快照将会替换掉该index及其之前的日志条目
lastIncludedTerm	**lastIncludeIndex的纪元**
offset	chunk数据在快照中的字节偏移量
data[]	从偏移量开始的，快照原始数据
done	如果为最后一个chunk，则为true
Arguments：	
term	当前纪元，for leader to update itself
Receiver implementation	
1、	如果参数的纪元小于当前纪元，则立即返回错误
2、	如果*offset=0*，则为首个*chunk*，则创建快照文件
3、	在给定的*offset*上，将数据写入到快照文件之中
4、	回复，并且若*done*为*fasle*则等待更多地数据
5、	保存快照文件，丢弃已存在或者不完整的具有更小*index*的快照

InstallSnapshot RPC	
6、	如果存在与请求中*lastIndex*、*lastTerm*一样的条目，那么保留该条目之后的，并回复
7、	丢弃整个日志
8、	使用快照内容进行重置状态机（并且加载快照中的集群配置）

一旦服务器写快照完成，他可能删除快照所包括最后条目的所有前面条目，以及之前的快照。

图13：InstallSnapshot RPC的概要。快照会分成多个chunks进行传输；每个chunk都给了一个存活的信号，因此Follower就能够重置选举超时时间。

接管服务器通常独自构造快照，Leader偶尔必须发送快照给落后的Follower。这发生在Leader已经删除要发送给Follower的下一个条目。幸运的是，这种状态不会是一个通常的操作：一个与Leader保持同步的Follower将已经有该条目。然而，一个特别慢得Follower或者刚加入集群的服务器将不会有该条目。使这种Follower跟上进度的方法是Leader通过网络发送快照。

Leader使用一种新的RPC，InstallSnapshot来将快照发送给当前落后过多的Follower。如图13.当一个Follower通过RPC收到一个快照，他必须决定如何处理当前存在的条目。通常快照将包括没在接受者日志中的消息。此时，Follower丢弃他的整个日志；这些将都由快照取代，可能存在已提交的条目与快照相冲突。如果接收到的快照为日志的前面部分（由于重复传输或错误），那么被该快照覆盖的条目将被删除，但是快照之后的条目必须是正确的必须被保留。

这种快照方法偏离了Raft中的Leader原则，因为Follower能够在不需要Leader允许时制造快照。然而，我们觉得这种偏离是合适的。Leader是为了帮助避免一致性决策时的冲突，一致性在快照时已经达成，因此没有决定是冲突的。数据依然只从Leader流向Follower，只是Follower现在可以重新组织他们的数据。

我们知道有一种选择是采用Leader-based方法，只有Leader创建快照，然后将快照发送给所有的Follower。然而，这有两个缺点，1）发送快照给Follower将浪费带宽并且使得快照过程变慢。每个Follower都有产生快照的数据信息，并且比Leader创建快照然后通过

网络传输给Follower便宜。2）Leader的实现将更加复杂。例如：Leader需要与复制新条目并行地发送快照给Follower，才能不会阻塞客户端的请求。

影响快照性能还有两个问题。1）服务器必须决定何时进行快照。如果一个服务器快照过于频繁，将会浪费磁盘带宽与能源；如果太慢，会危及到存储容量，同时会增加从日志重重做的时间。一个简单地策略是当日志达到一个固定大小是构造快照。如果这个大小设置成显著大于预期的快照，那么磁盘的带宽将影响很小。

2）记录下快照将会需要一个较长的时间，而且我们并不像这个操作影响正常的操作。方法是使用copy-on-write技术，使得新的更新能够被接受而不影响快照的记录。例如，函数化数据结构的状态机天然支持。另外，操作系统的copy-on-write支持（Linux的fork操作），能够用于创建一个完整状态机的内存快照（我们的实现使用该方法）。

8、Client interaction

这节讲述客户端如何与Raft进行交互，包括客户端如何发现集群的Leader与Raft如何支持线性化的语义【10】。这些问题适用于所有以consensus-based系统，并且Raft的解决方法与其他系统类似。

Raft的客户端发送所有请求到Leader。当一个客户端首次启动，随机连接到一个服务器。如果客户端的首次选择不是Leader，该服务器会拒绝客户端的倾情，并且提供当前所知道最新的Leader（AppendEntries请求包括了Leader得网络地址）。如果Leader宕机，客户端请求将会超时；客户端然后再次随机选择服务器。

Raft的目标是实现线性语义（似乎每个操作执行的瞬间，只有一次，在它的调用和响应之间的某一点）。然而，到目前为止Raft可能重复执行一条命令：例如，如果Leader在提交日志条目之后与回复给客户端之前宕机，客户端将会到一个新的Leader上重试该命令。解决方法是客户端为每个命令赋予一个唯一的序列号。然后，状态机跟踪每个客户端的最新序列号以及相关的响应。如果接收到一个序列号已经被执行的命令，会立即返回而不会重复执行该请求。

只读操作能够在不写任何日志情况下被处理。然而，如果没有附加的条件，这可能有返回过时数据的风险，因为Leader回复该请求时可能并不知道自己已经被一个新的Leader所取代（注：在回复时可能还不知道已经有一个新的Leader，自己已经被废黜）。线性化读取必须不返回过时的数据，Raft在不使用日志时，有两个注意事项来保证这点。1）Leader必须有被提交条目的最新信息。Leader Completeness Property保证了，Leader具有所有已提交的条目，但是在开始他的纪元时，他可能不知道哪些是已提交的条目（but at the start of its term, it may not know which those are）。为了找出，它需要在自己的纪元上提交一个条目。Raft采用，每个Leader在开始时提交一个空操作条目到他的日志中。2）Leader必须在处理一个只读请求前，检查自己是否已经被罢免（如果一个更新的Leader已经当选，那么他的信息可能是过时的）。Raft采用，在回复客户端的只读请求前，向集群中的多数服务器发送心跳的方式（来确定自己的Leader地位）。可选的，Leader可以依赖心跳机制来提供一种租约机制【9】，但是这导致依赖时间来保证安全性（此时需要假定时间误差是有界的）。

9、Implementation and evaluation

我们已经实现了Raft，作为一个为RAMCloud保存配置信息的，复制状态机的一部分，并且与RAMCloud的错误处理相协调（assists in failover of the RAMCloud coordinator）。Raft的实现包括大约2000行C++代码，不包括测试、注释与空行。源代码可以免费获取【23】。还有差不多25中独立的在各种开发状态的Raft第三方开源实现【34】，都是基于本篇文章的草稿。同样，各种公司正在部署Raft-based系统。

该节的余下部分将通过三个部分来评价Raft：可理解性，正确性，性能。

9.1、Understandability

9.2、Correctness

9.3、Performance

10、Related work

11、Conclusion

12、Acknowledgments