Raft 一致性算法论文译文

本篇博客为著名的 RAFT 一致性算法论文的中文翻译，论文名为《In search of an Understandable Consensus Algorithm (Extended Version)》(寻找一种易于理解的一致性算法)。

Raft 是一种用来管理日志复制的一致性算法。它和 Paxos 的性能和功能是一样的，但是它和 Paxos 的结构不一样；这使得 Raft 更容易理解并且更易于建立实际的系统。为了提高理解性，Raft 将一致性算法分为了几个部分，例如领导选取（leader selection），日志复制（log replication）和安全性（safety），同时它使用了更强的一致性来减少了必须需要考虑的状态。从用户学习的结果来看，Raft 比 Paxos 更容易学会。Raft 还包括了一种新的机制来使得动态改变集群成员，它使用重叠大多数（overlapping majorities）来保证安全。

1. 引言

一致性算法允许一组机器像一个整体一样工作，即使其中的一些机器出了错误也能正常工作。正因为此，他们扮演着建立大规模可靠的软件系统的关键角色。在过去的十年中 Paxos 一直都主导着有关一致性算法的讨论：大多数一致性算法的实现都基于它或者受它影响，并且 Paxos 也成为了教学生关于一致性知识的主要工具。

不幸的是，尽管在降低它的复杂性方面做了许多努力，Paxos 依旧很难理解。并且，Paxos 需要经过复杂的修改才能应用于实际中。这些导致了系统构构建者和学生都十分头疼。

在被 Paxos 折磨之后，我们开始寻找一种在系统构建和教学上更好的新的一致性算法。我们的首要目标是让它易于理解：我们能不能定义一种面向实际系统的一致性算法并且比 Paxos 更容易学习呢？并且，我们希望这种算法能凭直觉就能明白，这对于一个系统构建者来说是十分必要的。对于一个算法，不仅仅是让它工作起来很重要，知道它是如何工作的更重要。

我们工作的结果是一种新的一致性算法，叫做 Raft。在设计 Raft 的过程中我们应用了许多专门的技巧来提升理解性，包括算法分解（分为领导选

取（leader selection），日志复制（log replication）和安全性（safety））和减少状态（state space reduction）（相对于 Paxos，Raft 减少了非确定性的程度和服务器互相不一致的方式）。在两所学校的 43 个学生的研究中发现，Raft 比 Paxos 要更容易理解：在学习了两种算法之后，其中的 33 个学生回答 Raft 的问题要比回答 Paxos 的问题要好。

Raft 算法和现在一些已经有的算法在一些地方很相似（主要是 Oki 和 Liskov 的 Viewstamped Replication。但是 Raft 有几个新的特性：

- 强领导者（Strong Leader）：Raft 使用一种比其他算法更强的领导形式。例如，日志条目只从领导者发送向其他服务器。这样就简化了对日志复制的管理，使得 Raft 更易于理解。
- 领导选取（Leader Selection）：Raft 使用随机定时器来选取领导者。这种方式仅仅是在所有算法都需要实现的心跳机制上增加了一点变化，它使得在解决冲突时更简单和快速。
- 成员变化（Membership Change）：Raft 为了调整集群中成员关系使用了新的联合一致性（joint consensus）的方法，这种方法中大多数不同配置的机器在转换关系的时候会交迭（overlap）。这使得在配置改变的时候，集群能够继续操作。

我们认为，Raft 在教学方面和实际实现方面比 Paxos 和其他算法更出众。它比其他算法更简单、更容易理解；它能满足一个实际系统的需求；它拥有许多开源的实现并且被许多公司所使用；它的安全特性已经被证明；并且它的效率和其他算法相比也具有竞争力。

这篇论文剩下的部分会讲如下内容：复制状态机（replicated state machine）问题（第 2 节），讨论 Paxos 的优缺点（第 3 节），讨论我们用的为了达到提升理解性的方法（第 4 节），陈述 Raft 一致性算法（第 5~8 节），评价 Raft 算法（第 9 节），对相关工作的讨论（第 10 节）。

2. 复制状态机（Replicated State Machine）

一致性算法是在复制状态机的背景下提出来的。在这个方法中，在一组服务器的状态机产生同样的状态的副本因此即使有一些服务器崩溃了这组服务器也还能继续执行。复制状态机在分布式系统中被用于解决许多有关容错的问题。例如，GFS，HDFS 还有 RAMCloud 这些大规模的系统都是用

一个单独的集群领导者，使用一个单独的复制状态机来进行领导选取和存储配置信息来应对领导者的崩溃。使用复制状态机的例子有 Chubby 和 ZooKeeper。

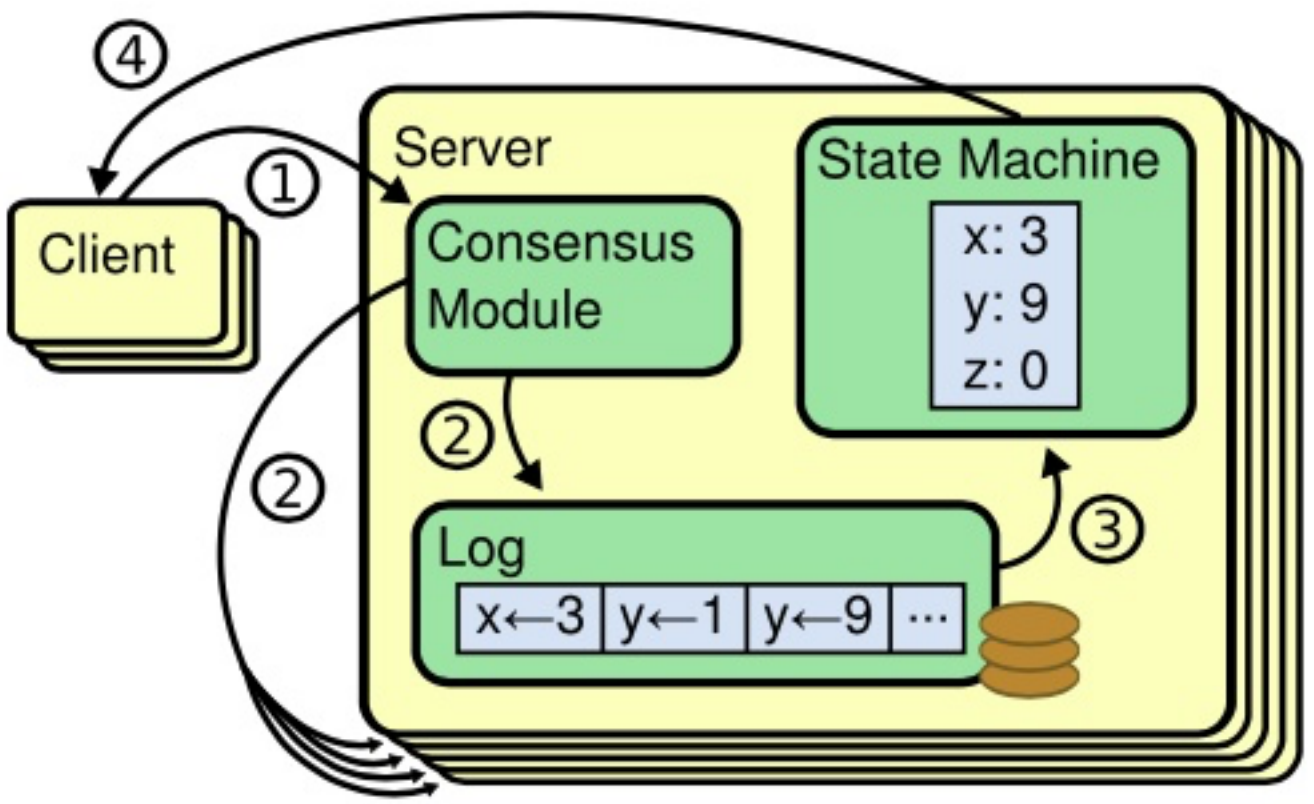

图 -1：复制状态机的架构。一致性算法管理来自客户端状态命令的复制日志。状态机处理的日志中的命令的顺序都是一致的，因此会得到相同的执行结果。

如图 -1 所示，复制状态机是通过复制日志来实现的。每一台服务器保存着一份日志，日志中包含一系列的命令，状态机会按顺序执行这些命令。因为每一台计算机的状态机都是确定的，所以每个状态机的状态都是相同的，执行的命令是相同的，最后的执行结果也就是一样的了。

如何保证复制日志一致就是一致性算法的工作了。在一台服务器上，一致性模块接受客户端的命令并且把命令加入到它的日志中。它和其他服务器上的一致性模块进行通信来确保每一个日志最终包含相同序列的请求，即使有一些服务器宕机了。一旦这些命令被正确的复制了，每一个服务器的状态机都会按同样的顺序去执行它们，然后将结果返回给客户端。最终，这些服务器看起来就像一台可靠的状态机。

应用于实际系统的一致性算法一般有以下特性：

- 确保安全性（从来不会返回一个错误的结果），即使在所有的非拜占庭（Non-Byzantine）情况下，包括网络延迟、分区、丢包、冗余和

乱序的情况下。

- 高可用性，只要集群中的大部分机器都能运行，可以互相通信并且可以和客户端通信，这个集群就可用。因此，一般来说，一个拥有 5 台机器的集群可以容忍其中的 2 台的失败（fail）。服务器停止工作了我们就认为它失败（fail）了，没准一会当它们拥有稳定的存储时就能从中恢复过来，重新加入到集群中。

- 不依赖时序保证一致性，时钟错误和极端情况下的消息延迟在最坏的情况下才会引起可用性问题。

- 通常情况下，一条命令能够尽可能快的在大多数节点对一轮远程调用作出相应时完成，一少部分慢的机器不会影响系统的整体性能。

3. Paxos 算法的不足

在过去的 10 年中，Leslie Lamport 的 Paxos 算法几乎已经成为了一致性算法的代名词：它是授课中最常见的算法，同时也是许多一致性算法实现的起点。Paxos 首先定义了一个能够达成单一决策一致的协议，例如一个单一复制日志条目（single replicated log entry）。我们把这个子集叫做单一决策 Paxos（single-decree Paxos）。之后 Paxos 通过组合多个这种协议来完成一系列的决策，例如一个日志（multi-Paxos）。Paxos 确保安全性和活跃性（liveness），并且它支持集群成员的变更。它的正确性已经被证明，通常情况下也很高效。

不幸的是，Paxos 有两个致命的缺点。第一个是 Paxos 太难以理解。它的完整的解释晦涩难懂；很少有人能完全理解，只有少数人成功的读懂了它。并且大家做了许多努力来用一些简单的术语来描述它。尽管这些解释都关注于单一决策子集问题，但仍具有挑战性。在 NSDI 2012 会议上的一次非正式调查显示，我们发现大家对 Paxos 都感到不满意，其中甚至包括一些有经验的研究员。我们自己也曾深陷其中，我们在读过几篇简化它的文章并且设计了我们自己的算法之后才完全理解了 Paxos，而整个过程花费了将近一年的时间。

我们假定 Paxos 的晦涩来源于它将单决策子集作为它的基础。单决策（Single-decree）Paxos 是晦涩且微妙的：它被划分为两个没有简单直观

解释的阶段，并且难以独立理解。正因为如此，它不能很直观的让我们知道为什么单一决策协议能够工作。为多决策 Paxos 设计的规则又添加了额外的复杂性和精巧性。我们相信多决策问题能够分解为其它更直观的方式。

Paxos 的第二个缺点是它难以在实际环境中实现。其中一个原因是，对于多决策 Paxos （multi-Paxos），大家还没有一个一致同意的算法。Lamport 的描述大部分都是有关于单决策 Paxos （single-decree Paxos）；他仅仅描述了实现多决策的可能的方法，缺少许多细节。有许多实现 Paxos 和优化 Paxos 的尝试，但是他们都和 Lamport 的描述有些出入。例如，Chubby 实现的是一个类似 Paxos 的算法，但是在许多情况下的细节没有公开。

另外，Paxos 的结构也是不容易在一个实际系统中进行实现的，这是单决策问题分解带来的又一个问题。例如，从许多日志条目中选出条目然后把它们融合到一个序列化的日志中并没有带来什么好处，它仅仅增加了复杂性。围绕着日志来设计一个系统是更简单、更高效的：新日志按照严格的顺序添加到日志中去。另一个问题是，Paxos 使用对等的点对点的实现作为它的核心（尽管它最终提出了一种弱领导者的形式来优化性能）。这种方法在只有一个决策被制定的情况下才显得有效，但是很少有现实中的系统使用它。如果要做许多的决策，选择一个领导人，由领带人来协调是更简单有效的方法。

因此，在实际的系统应用中和 Paxos 算法都相差很大。所有开始于 Paxos 的实现都会遇到很多问题，然后由此衍生出了许多与 Paxos 有很大不同的架构。这是既费时又容易出错的，并且理解 Paxos 的难度又非常大。Paxos 算法在它正确性的理论证明上是很好的，但是在实现上的价值就远远不足了。来自 Chubby 的实现的一条评论就能够说明：

> Paxos 算法的描述与实际实现之间存在巨大的鸿沟...最终的系统往往建立在一个没有被证明的算法之上。

正因为存在这些问题，我们认为 Paxos 不仅对于系统的构建者来说不友好，同时也不利于教学。鉴于一致性算法对于大规模软件系统的重要性，我们决定试着来设计一种另外的比 Paxos 更好的一致性算法。Raft 就是这样的一个算法。

4. 易于理解的设计

设计 Raft 的目标有如下几个：

- 它必须提供一个完整的、实际的基础来进行系统构建，为的是减少开发者的工作；
- 它必须在所有情况下都能保证安全可用；
- 它对于常规操作必须高效；
- 最重要的目标是：**易于理解**，它必须使得大多数人能够很容易的理解；
- 另外，它必须能让开发者有一个直观的认识，这样才能使系统构建者们去对它进行扩展。

在设计 Raft 的过程中，我们不得不在许多种方法中做出选择。当面临这种情况时，我们通常会权衡可理解性：每种方法的可理解性是如何的？（例如，它的状态空间有多复杂？它是不是有很细微的含义？）它的可读性如何？读者能不能轻易地理解这个方法和它的含义？

我们意识到对这种可理解性的分析具有高度的主观性；尽管如此，我们使用了两种适用的方式。第一种是众所周知的问题分解：我们尽可能将问题分解成为若干个可解决的、可被理解的小问题。例如，在 Raft 中，我们把问题分解成为了**领导选取（leader election）**、**日志复制（log replication）**、**安全（safety）**和**成员变化（membership changes）**。

我们采用的第二个方法是通过减少需要考虑的状态的数量将状态空间简化，这能够使得整个系统更加一致并且尽可能消除不确定性。特别地，日志之间不允许出现空洞，并且 Raft 限制了限制了日志不一致的可能性。尽管在大多数情况下，我们都都在试图消除不确定性，但是有时候有些情况下，不确定性使得算法更易理解。尤其是，随机化方法使得不确定性增加，但是它减少了状态空间。我们使用随机化来简化了 Raft 中的领导选取算法。

5. Raft 一致性算法

Raft 是一种用来管理第 2 章中提到的复制日志的算法。表 -2 为了方便参考是一个算法的总结版本，表 -3 列举了算法中的关键性质；表格中的这些元素将会在这一章剩下的部分中分别进行讨论。

状态：

在所有服务器上持久存在的：（在响应远程过程调用 RPC 之前稳定存储的）

名称	描述
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的要执行命令和从领导人处收到时的任期号

在所有服务器上不稳定存在的:

名称	描述
commitIndex	已知的被提交的最大日志条目的索引值（从 0 开始递增）
lastApplied	被状态机执行的最大日志条目的索引值（从 0 开始递增）

在领导人服务器上不稳定存在的:（在选举之后初始化的）

名称	描述
nextIndex[]	对于每一个服务器，记录需要发给它的下一个日志条目的索引（初始化为领导人上一条日志的索引值 +1）
matchIndex[]	对于每一个服务器，记录已经复制到该服务器的日志的最高索引值（从 0 开始递增）

表 -2-i

附加日志远程过程调用 （AppendEntries RPC）

由领导人来调用复制日志（5.3 节）；也会用作 heartbeat

参数	描述
term	领导人的任期号
leaderId	领导人的 id，为了其他服务器能重定向到客户端

prevLogIndex	最新日志之前的日志的索引值
prevLogTerm	最新日志之前的日志的领导人任期号
entries[]	将要存储的日志条目（表示 heartbeat 时为空，有时会为了效率发送超过一条）
leaderCommit	领导人提交的日志条目索引值

返回值	描述
term	当前的任期号，用于领导人更新自己的任期号
success	如果其它服务器包含能够匹配上 prevLogIndex 和 prevLogTerm 的日志时为真

接受者需要实现：

1. 如果 `term < currentTerm`返回 false（5.1 节）
2. 如果在`prevLogIndex`处的日志的任期号与`prevLogTerm`不匹配时，返回 false（5.3 节）
3. 如果一条已经存在的日志与新的冲突（index 相同但是任期号 term 不同），则删除已经存在的日志和它之后所有的日志（5.3 节）
4. 添加任何在已有的日志中不存在的条目
5. 如果`leaderCommit > commitIndex`，将`commitIndex`设置为`leaderCommit`和最新日志条目索引号中较小的一个

表 -2-ii

投票请求 RPC（RequestVote RPC）

由候选人发起收集选票（5.2 节）

参数	描述
term	候选人的任期号
candidateId	请求投票的候选人 id
lastLogIndex	候选人最新日志条目的索引值
lastLogTerm	候选人最新日志条目对应的任期号

返回值	描述
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为 true

接受者需要实现：

1. 如果`term < currentTerm`返回 false（5.1 节）
2. 如果`votedFor`为空或者与`candidateId`相同，并且候选人的日志和自己的日志一样新，则给该候选人投票（5.2 节 和 5.4 节）

表 -2-iii

服务器需要遵守的规则：

所有服务器：

- 如果`commitIndex > lastApplied`，`lastApplied`自增，将`log[lastApplied]`应用到状态机（5.3 节）
- 如果 RPC 的请求或者响应中包含一个 term T 大于 `currentTerm`，则`currentTerm`赋值为 T，并切换状态为追随者（Follower）（5.1 节）

追随者（followers）：5.2 节

- 响应来自候选人和领导人的 RPC
- 如果在超过选取领导人时间之前没有收到来自当前领导人的`AppendEntries RPC`或者没有收到候选人的投票请求，则自己转换状态为候选人

候选人：5.2 节

- 转变为选举人之后开始选举：
  - `currentTerm`自增
  - 给自己投票
  - 重置选举计时器
  - 向其他服务器发送`RequestVote RPC`
- 如果收到了来自大多数服务器的投票：成为领导人
- 如果收到了来自新领导人的`AppendEntries RPC (heartbeat)`：转换状态为追随者
- 如果选举超时：开始新一轮的选举

领导人：

- 一旦成为领导人：向其他所有服务器发送空的AppendEntries RPC（heartbeat）；在空闲时间重复发送以防止选举超时（5.2 节）
- 如果收到来自客户端的请求：向本地日志增加条目，在该条目应用到状态机后响应客户端（5.3 节）
- 对于一个追随者来说，如果上一次收到的日志索引大于将要收到的日志索引（nextIndex）：通过AppendEntries RPC将 nextIndex 之后的所有日志条目发送出去
  - 如果发送成功：将该追随者的 nextIndex和matchIndex更新
  - 如果由于日志不一致导致AppendEntries RPC失败：nextIndex递减并且重新发送（5.3 节）
- 如果存在一个满足N > commitIndex和matchIndex[i] >= N并且log[N].term == currentTerm的 N，则将commitIndex赋值为 N

表 -2-iv

表 -2：Raft 一致性算法的总结（不包括成员变化 membership changes 和日志压缩 log compaction）

性质	描述
选举安全原则（Election Safety）	一个任期（term）内最多允许有一个领导人被选上（5.2 节）
领导人只增加原则（Leader Append-Only）	领导人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（Log Matching）	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头到这个索引位置之间的条目完全相同（5.3 节）
领导人完全原则（Leader Completeness）	如果一个日志条目在一个给定任期内被提交，那么这个条目一定会出现在所有任期号更大的领导人中
状态机安全原则（State Machine Safety）	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（5.4.3 节）

表 -3：Raft 算法保证这些特性任何时刻都成立

Raft 通过首先选出一个领导人来实现一致性，然后给予领导人完全管理复

制日志（replicated log）的责任。领导人接收来自客户端的日志条目，并把它们复制到其他的服务器上，领带人还要告诉服务器们什么时候将日志条目应用到它们的状态机是安全的。通过选出领导人能够简化复制日志的管理工作。例如，领导人能够决定将新的日志条目放到哪，而并不需要和其他的服务器商议，数据流被简化成从领导人流向其他服务器。如果领导人宕机或者和其他服务器失去连接，就可以选取下一个领导人。

通过选出领导人，Raft 将一致性问题分解成为三个相对独立的子问题：

- **领导人选取（Leader election）：** 在一个领导人宕机之后必须要选取一个新的领导人（5.2 节）
- **日志复制（Log replication）：** 领导人必须从客户端接收日志然后复制到集群中的其他服务器，并且强制要求其他服务器的日志保持和自己相同
- **安全性（Safety）：** Raft 的关键的安全特性是 表 -3 中提到的状态机安全原则（State Machine Safety）：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。5.4 节阐述了 Raft 是如何保证这条原则的，解决方案涉及到一个对于选举机制另外的限制，这一部分会在 5.2 节 中说明。

在说明了一致性算法之后，本章会讨论有关可用性（availability）的问题和系统中时序（timing）的问题。

5.1 Raft 基础

一个 Raft 集群包括若干服务器；对于一个典型的 5 服务器集群，该集群能够容忍 2 台机器不能正常工作，而整个系统保持正常。在任意的时间，每一个服务器一定会处于以下三种状态中的一个：*领导人*、*候选人*、*追随者*。在正常情况下，只有一个服务器是领导人，剩下的服务器是追随者。追随者们是被动的：他们不会发送任何请求，只是响应来自领导人和候选人的请求。领导人来处理所有来自客户端的请求（如果一个客户端与追随者进行通信，追随者会将信息发送给领导人）。候选人是用来选取一个新的领导人的，这一部分会在 5.2 节 进行阐释。图 -4 阐述了这些状态，和它们之间的转换；它们的转换会在下边进行讨论。

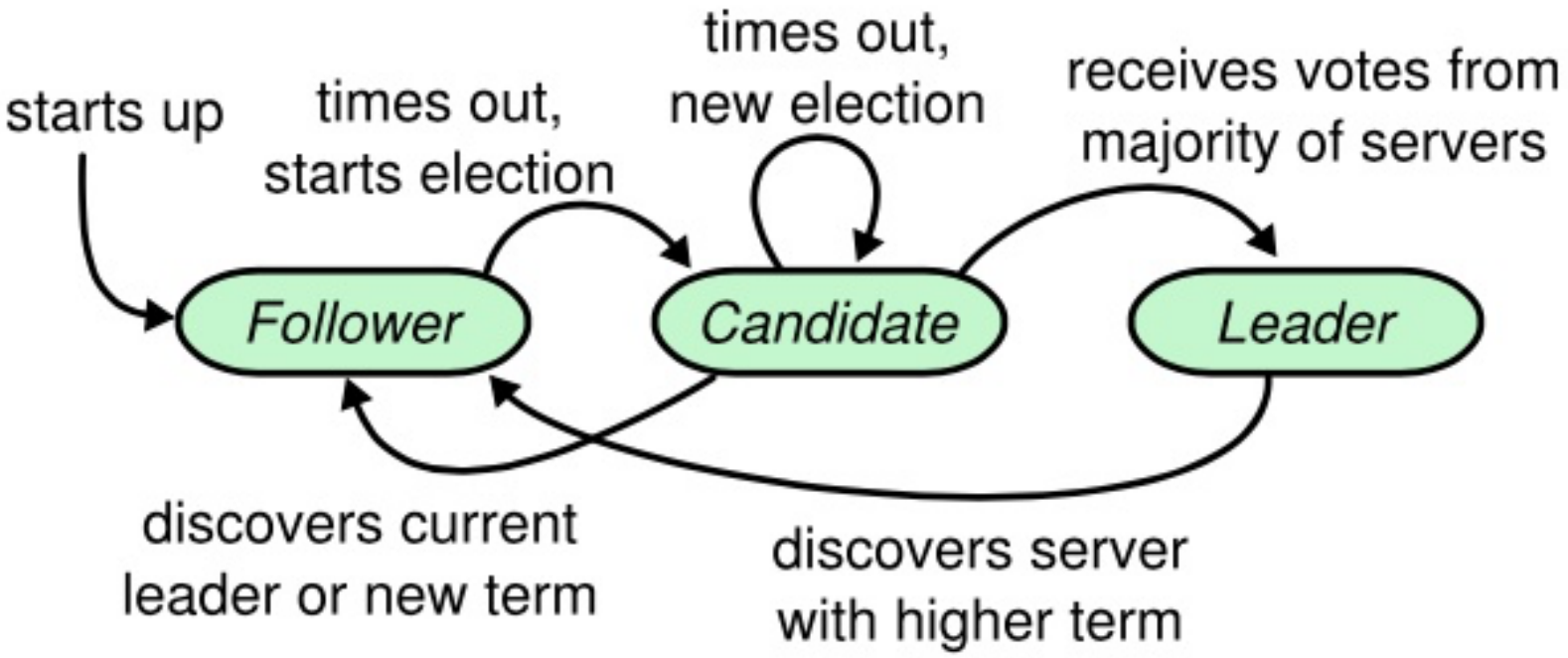

图 -4：服务器的状态。追随者只响应其他服务器的请求。如果追随者没有收到任何消息，它会成为一个候选人并且开始一次选举。收到大多数服务器投票的候选人会成为新的领导人。领导人在它们宕机之前会一直保持领导人的状态。

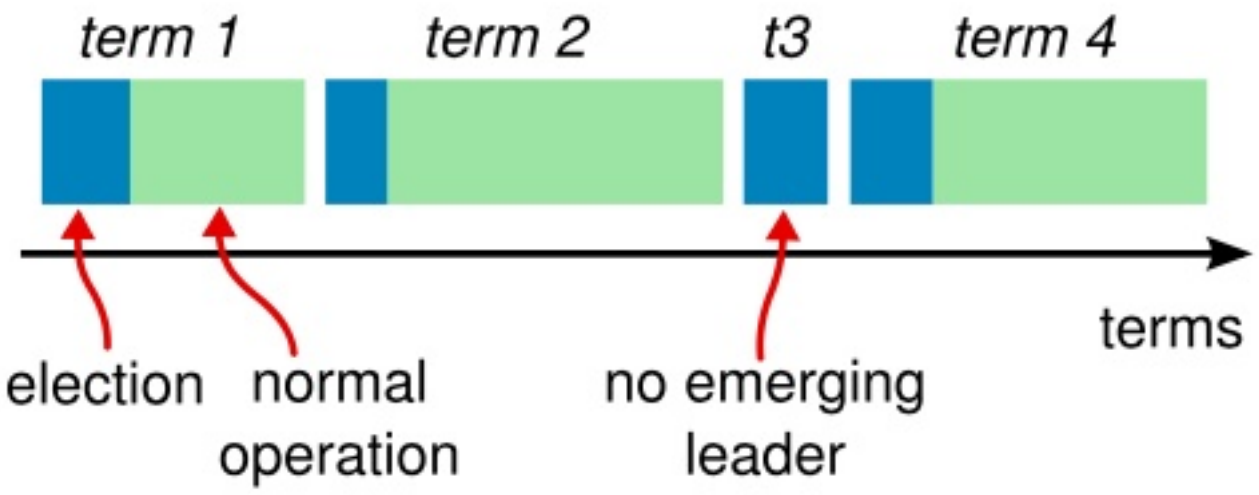

图 -5：时间被分为一个个的任期（term），每一个任期的开始都是领导人选举。在成功选举之后，一个领导人会在任期内管理整个集群。如果选举失败，该任期就会因为没有领带人而结束。这个转变会在不同的时间的不同服务器上观察到。

如 图 -5 所示，Raft 算法将时间划分成为任意不同长度的任期（term）。任期用连续的数字进行表示。每一个任期的开始都是一次选举（election），就像 5.2 节 所描述的那样，一个或多个候选人会试图成为领导人。如果一个候选人赢得了选举，它就会在该任期的剩余时间担任领导人。在某些情况下，选票会被瓜分，有可能没有选出领导人，那么，将会开始另一个任期，并且立刻开始下一次选举。Raft 算法保证在给定的一

个任期最少要有一个领导人。

不同的服务器可能会在任期内观察到多次不同的状态转换，在某些情况下，一台服务器可能看不到一次选举或者一个完整的任期。任期在 Raft 中充当逻辑时钟的角色，并且它们允许服务器检测过期的信息，比如过时的领导人。每一台服务器都存储着一个当前任期的数字，这个数字会单调的增加。当服务器之间进行通信时，会互相交换当前任期号；如果一台服务器的当前任期号比其它服务器的小，则更新为较大的任期号。如果一个候选人或者领导人意识到它的任期号过时了，它会立刻转换为追随者状态。如果一台服务器收到的请求的任期号是过时的，那么它会拒绝此次请求。

Raft 中的服务器通过远程过程调用（RPC）来通信，基本的 Raft 一致性算法仅需要 2 种 RPC。RequestVote RPC 是候选人在选举过程中触发的（5.2 节），AppendEntries RPC 是领导人触发的，为的是复制日志条目和提供一种心跳（heartbeat）机制（5.3 节）。第 7 章加入了第三种 RPC 来在各个服务器之间传输快照（snapshot）。如果服务器没有及时收到 RPC 的响应，它们会重试，并且它们能够并行的发出 RPC 来获得最好的性能。

5.2 领导人选取

Raft 使用一种心跳机制（heartbeat）来触发领导人的选取。当服务器启动时，它们会初始化为追随者。一太服务器会一直保持追随者的状态只要它们能够收到来自领导人或者候选人的有效 RPC。领导人会向所有追随者周期性发送心跳（heartbeat，不带有任何日志条目的 AppendEntries RPC）来保证它们的领导人地位。如果一个追随者在一个周期内没有收到心跳信息，就叫做选举超时（election timeout），然后它就会假定没有可用的领导人并且开始一次选举来选出一个新的领导人。

为了开始选举，一个追随者会自增它的当前任期并且转换状态为候选人。然后，它会给自己投票并且给集群中的其他服务器发送 RequestVote RPC。一个候选人会一直处于该状态，直到下列三种情形之一发生：

- 它赢得了选举；
- 另一台服务器赢得了选举；

- 一段时间后没有任何一台服务器赢得了选举

这些情形会在下面的章节中分别讨论。

一个候选人如果在一个任期内收到了来自集群中大多数服务器的投票就会赢得选举。在一个任期内，一台服务器最多能给一个候选人投票，按照先到先服务原则（first-come-first-served）（注意：在 5.4 节 针对投票添加了一个额外的限制）。大多数原则使得在一个任期内最多有一个候选人能赢得选举（表 -3 中提到的选举安全原则）。一旦有一个候选人赢得了选举，它就会成为领导人。然后它会像其他服务器发送心跳信息来建立自己的领导地位并且组织新的选举。

当一个候选人等待别人的选票时，它有可能会收到来自其他服务器发来的声明其为领导人的 AppendEntries RPC。如果这个领导人的任期（包含在它的 RPC 中）比当前候选人的当前任期要大，则候选人认为该领导人合法，并且转换自己的状态为追随者。如果在这个 RPC 中的任期小于候选人的当前任期，则候选人会拒绝此次 RPC， 继续保持候选人状态。

第三种情形是一个候选人既没有赢得选举也没有输掉选举：如果许多追随者在同一时刻都成为了候选人，选票会被分散，可能没有候选人能获得大多数的选票。当这种情形发生时，每一个候选人都会超时，并且通过自增任期号和发起另一轮 RequestVote RPC 来开始新的选举。然而，如果没有其它手段来分配选票的话，这种情形可能会无限的重复下去。

Raft 使用随机的选举超时时间来确保第三种情形很少发生，并且能够快速解决。为了防止在一开始是选票就被瓜分，选举超时时间是在一个固定的间隔内随机选出来的（例如，150~300ms）。这种机制使得在大多数情况下只有一个服务器会率先超时，它会在其它服务器超时之前赢得选举并且向其它服务器发送心跳信息。同样的机制被用于选票一开始被瓜分的情况下。每一个候选人在开始一次选举的时候会重置一个随机的选举超时时间，在超时进行下一次选举之前一直等待。这能够减小在新的选举中一开始选票就被瓜分的可能性。9.3 节 展示了这种方法能够快速的选出一个领导人。

选举是一个理解性引导我们设计替代算法的一个例子。最开始时，我们计划使用一种排名系统：给每一个候选人分配一个唯一的排名，用于在竞争

的候选人之中选择领导人。如果一个候选人发现了另一个比它排名高的候选人，那么它会回到追随者的状态，这样排名高的候选人会很容易地赢得选举。但是我们发现这种方法在可用性方面有一点问题（一个低排名的服务器在高排名的服务器宕机后，需要等待超时才能再次成为候选人，但是如果它这么做的太快，它能重置选举领带人的过程）。我们对这个算法做了多次调整，但是每次调整后都会出现一些新的问题。最终我们认为随机重试的方法是更明确并且更易于理解的。

5.3 日志复制

一旦选出了领导人，它就开始接收客户端的请求。每一个客户端请求都包含一条需要被复制状态机（replicated state machine）执行的命令。领导人把这条命令作为新的日志条目加入到它的日志中去，然后并行的向其他服务器发起 AppendEntries RPC，要求其它服务器复制这个条目。当这个条目被安全的复制之后（下面的部分会详细阐述），领导人会将这个条目应用到它的状态机中并且会向客户端返回执行结果。如果追随者崩溃了或者运行缓慢或者是网络丢包了，领导人会无限的重试 AppendEntries RPC（甚至在它向客户端响应之后）知道所有的追随者最终存储了所有的日志条目。

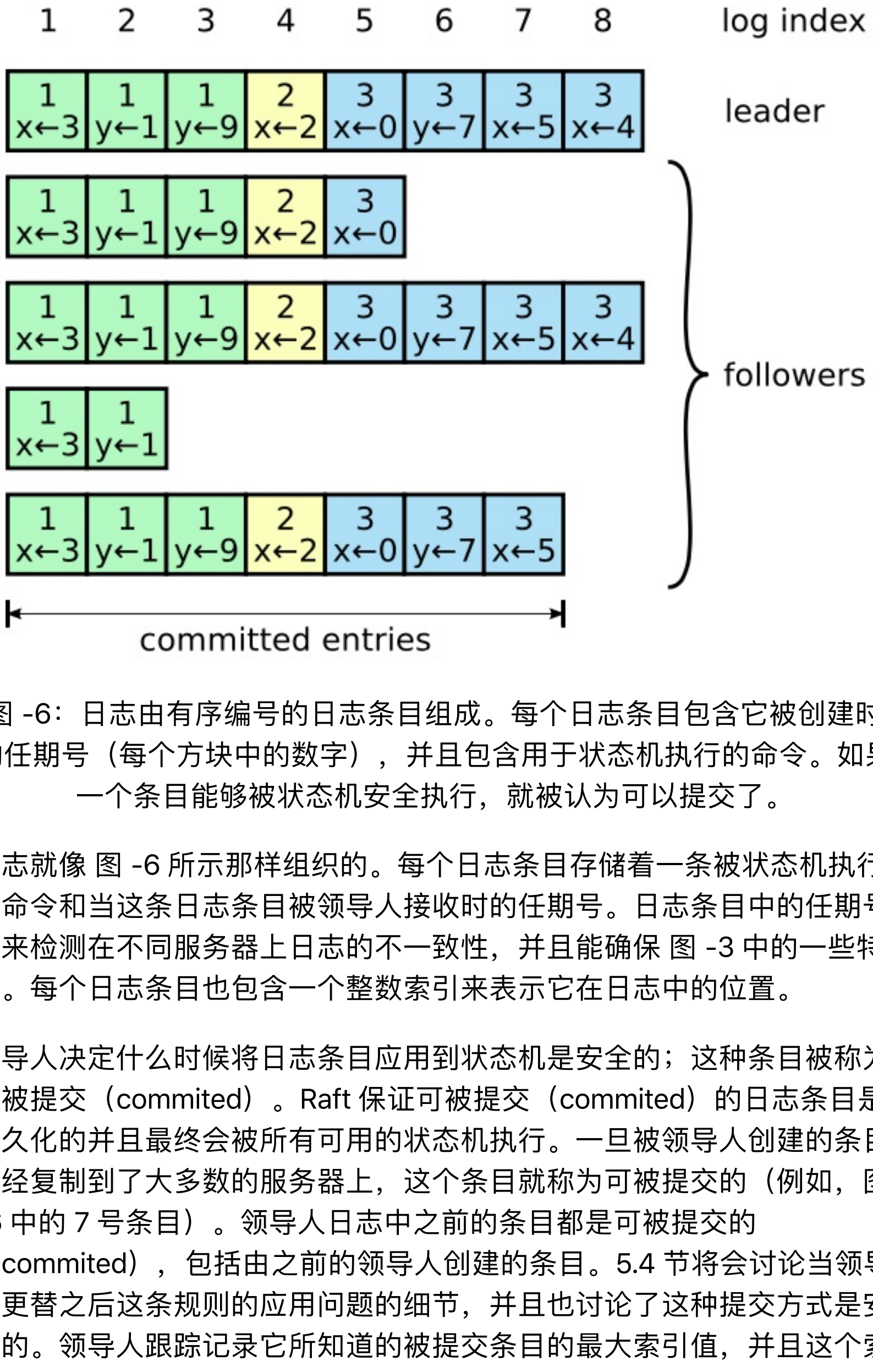

图 -6：日志由有序编号的日志条目组成。每个日志条目包含它被创建时的任期号（每个方块中的数字），并且包含用于状态机执行的命令。如果一个条目能够被状态机安全执行，就被认为可以提交了。

日志就像 图 -6 所示那样组织的。每个日志条目存储着一条被状态机执行的命令和当这条日志条目被领导人接收时的任期号。日志条目中的任期号用来检测在不同服务器上日志的不一致性，并且能确保 图 -3 中的一些特性。每个日志条目也包含一个整数索引来表示它在日志中的位置。

领导人决定什么时候将日志条目应用到状态机是安全的；这种条目被称为可被提交（commited）。Raft 保证可被提交（commited）的日志条目是持久化的并且最终会被所有可用的状态机执行。一旦被领导人创建的条目已经复制到了大多数的服务器上，这个条目就称为可被提交的（例如，图 -6 中的 7 号条目）。领导人日志中之前的条目都是可被提交的（commited），包括由之前的领导人创建的条目。5.4 节将会讨论当领导人更替之后这条规则的应用问题的细节，并且也讨论了这种提交方式是安全的。领导人跟踪记录它所知道的被提交条目的最大索引值，并且这个索

引值会包含在之后的 AppendEntries RPC 中（包括心跳 heartbeat 中），为的是让其他服务器都知道这条条目已经提交。一旦一个追随者知道了一个日志条目已经被提交，它会将该条目应用至本地的状态机（按照日志顺序）。

我们设计了 Raft 日志机制来保证不同服务器上日志的一致性。这样做不仅简化了系统的行为使得它更可预测，并且也是保证安全性不可或缺的一部分。Raft 保证以下特性，并且也保证了 表 -3 中的日志匹配原则（Log Matching Property）：

- 如果在不同日志中的两个条目有着相同的索引和任期号，则它们所存储的命令是相同的。
- 如果在不同日志中的两个条目有着相同的索引和任期号，则它们之间的所有条目都是完全一样的。

第一条特性源于领导人在一个任期里在给定的一个日志索引位置最多创建一条日志条目，同时该条目在日志中的位置也从来不会改变。第二条特性源于 AppendEntries 的一个简单的一致性检查。当发送一个 AppendEntries RPC 时，领导人会把新日志条目紧接着之前的条目的索引位置和任期号都包含在里面。如果追随者没有在它的日志中找到相同索引和任期号的日志，它就会拒绝新的日志条目。这个一致性检查就像一个归纳步骤：一开始空的日志的状态一定是满足日志匹配原则的，一致性检查保证了当日志添加时的日志匹配原则。因此，只要 AppendEntries 返回成功的时候，领导人就知道追随者们的日志和它的是一致的了。

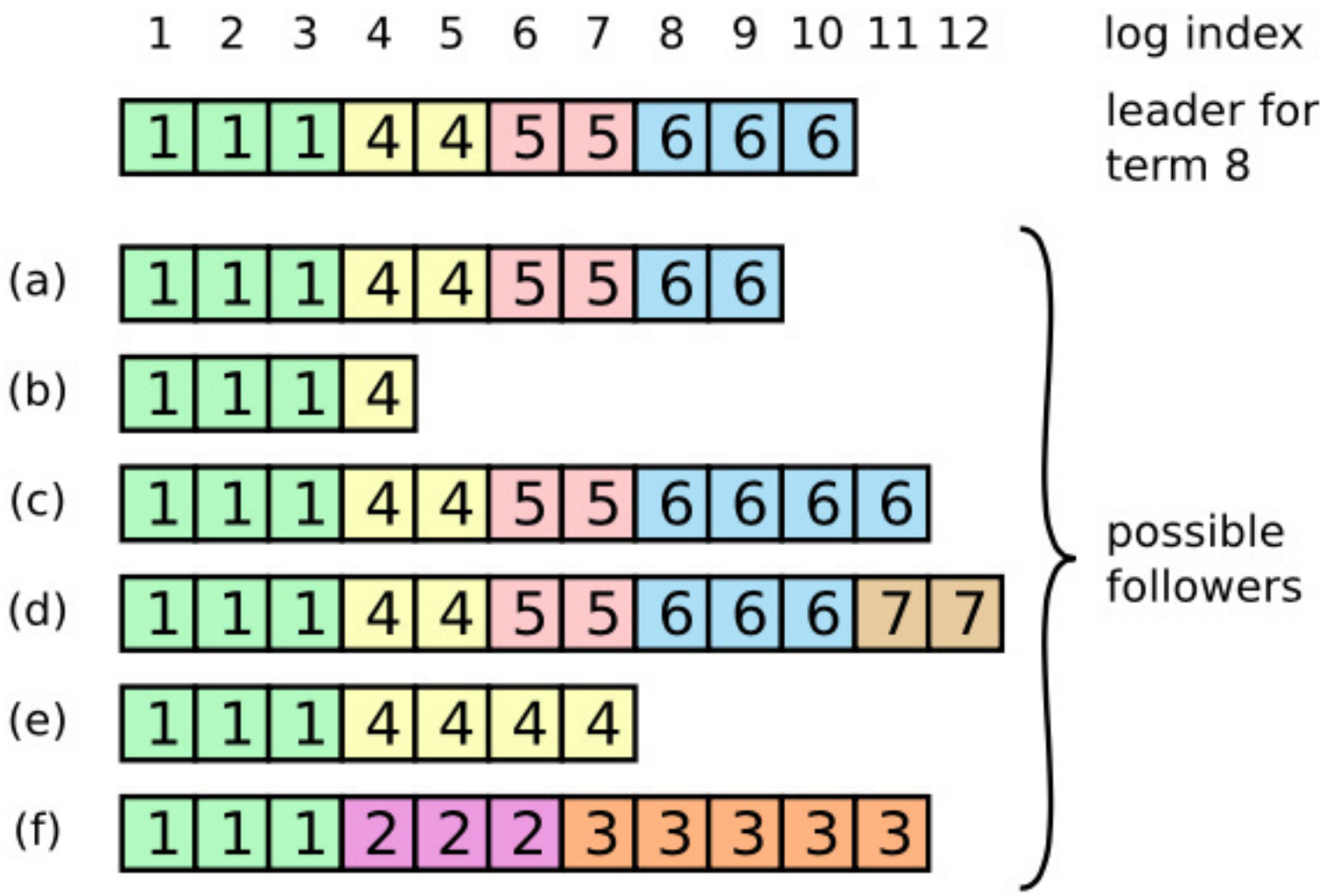

图 -7：当最上边的领导人掌权之后，追随者日志可能有以下情况（a~f）。一个格子表示一个日志条目；格子中的数字是它的任期。一个追随者可能会丢失一些条目（a, b）；可能多出来一些未提交的条目（c, d）；或者两种情况都有（e, f）。例如，场景 f 在如下情况下就会发生：如果一台服务器在任期 2 时是领导人并且往它的日志中添加了一些条目，然后在将它们提交之前就宕机了，之后它很快重启了，成为了任期 3 的领导人，又往它的日志中添加了一些条目，然后在任期 2 和任期 3 中的条目提交之前它又宕机了并且几个任期内都一直处于宕机状态。

在一般情况下，领导人和追随者们的日志保持一致，因此 AppendEntries 一致性检查通常不会失败。然而，领导人的崩溃会导致日志不一致（旧的领导人可能没有完全复制完日志中的所有条目）。这些不一致会导致一系列领导人和追随者崩溃。图 -7 阐述了一些追随者可能和新的领导人日志不同的情况。一个追随者可能会丢失掉领导人上的一些条目，也有可能包含一些领导人没有的条目，也有可能两者都会发生。丢失的或者多出来的条目可能会持续多个任期。

在 Raft 算法中，领导人通过强制追随者们复制它的日志来处理日志的不一致。这就意味着，在追随者上的冲突日志会被领导者的日志覆盖。5.4 节会说明当添加了一个额外的限制之后这是安全的。

为了使得追随者的日志同自己的一致，领导人需要找到追随者同它的日志一致的地方，然后删除追随者在该位置之后的条目，然后将自己在该位置之后的条目发送给追随者。这些操作都在 AppendEntries RPC 进行一致性检查时完成。领导人给每一个追随者维护了一个`nextIndex`，它表示领导人将要发送给该追随者的下一条日志条目的索引。当一个领导人开始掌权时，它会将`nextIndex`初始化为它的最新的日志条目索引数 +1（图 -7 中的 11）。如果一个追随者的日志和领导者的不一致，AppendEntries 一致性检查会在下一次 AppendEntries RPC 时返回失败。在失败之后，领导人会将`nextIndex`递减然后重试 AppendEntries RPC。最终`nextIndex`会达到一个领导人和追随者日志一致的地方。这时，AppendEntries 会返回成功，追随者中冲突的日志条目都被移除了，并且添加所缺少的上了领导人的日志条目。一旦 AppendEntries 返回成功，追随者和领导人的日志就一致了，这样的状态会保持到该任期结束。

如果需要的话，算法还可以进行优化来减少 AppendEntries RPC 失败的次数。例如，当拒绝了一个 AppendEntries 请求，追随者可以记录下冲突日志条目的任期号和自己存储那个任期的最早的索引。通过这些信息，领导人能够直接递减`nextIndex`跨过那个任期内所有的冲突条目；这样的话，一个冲突的任期需要一次 AppendEntries RPC，而不是每一个冲突条目需要一次 AppendEntries RPC。在实践中，我们怀疑这种优化是否是必要的，因为 AppendEntries 一致性检查很少失败并且也不太可能出现大量的日志条目不一致的情况。

通过这种机制，一个领导人在掌权时不需要采取另外特殊的方式来恢复日志的一致性。它只需要使用一些常规的操作，通过响应 AppendEntries 一致性检查的失败能使得日志自动的趋于一致。一个领导人从来不会覆盖或者删除自己的日志（表 -3 中的领导人只增加原则）。

这个日志复制机制展示了在第 2 章中阐述的所希望的一致性特性：Raft 能够接受，复制并且应用新的日志条目只要大部分的服务器是正常的。在通常情况下，一条新的日志条目可以在一轮 RPC 内完成在集群的大多数服务器上的复制；并且一个速度很慢的追随者并不会影响整体的性能。

5.4 安全性

之前的章节中讨论了 Raft 算法是如何进行领导选取和复制日志的。然而，到目前为止这个机制还不能保证每一个状态机能按照相同的顺序执行同样的指令。例如，当领导人提交了若干日志条目的同时一个追随者可能宕机了，之后它又被选为了领导人然后用新的日志条目覆盖掉了旧的那些，最后，不同的状态机可能执行不同的命令序列。

这一节通过在领带人选取部分加入了一个限制来完善了 Raft 算法。这个限制能够保证对于固定的任期，任何的领导人都拥有之前任期提交的全部日志条目（表 -3 中的领导人完全原则）。有了这一限制，日志提交的规则就更清晰了。最后，我们提出了对于领导人完全原则的简单证明并且展示了它是如何修正复制状态机的行为的。

5.4.1 选举限制

在所有的以领导人为基础的一致性算法中，领导人最终必须要存储全部已经提交的日志条目。在一些一致性算法中，例如：Viewstamped Replication，即使一开始没有包含全部已提交的条目也可以被选为领导人。这些算法都有一些另外的机制来保证找到丢失的条目并将它们传输给新的领导人，这个过程要么在选举过程中完成，要么在选举之后立即开始。不幸的是，这种方式大大增加了复杂性。Raft 使用了一种更简单的方式来保证在新的领导人开始选举的时候在之前任期的所有已提交的日志条目都会出现在上边，而不需要将这些条目传送给领导人。这就意味着日志条目只有一个流向：从领导人流向追随者。领导人永远不会覆盖已经存在的日志条目。

Raft 使用投票的方式来阻止没有包含全部日志条目的服务器赢得选举。一个候选人为了赢得选举必须要和集群中的大多数进行通信，这就意味着每一条已经提交的日志条目最少在其中一台服务器上出现。如果候选人的日志至少和大多数服务器上的日志一样新（up-to-date，这个概念会在下边有详细介绍），那么它一定包含有全部的已经提交的日志条目。RequestVote RPC 实现了这个限制：这个 RPC（远程过程调用）包括候选人的日志信息，如果它自己的日志比候选人的日志要新，那么它会拒绝候选人的投票请求。

Raft 通过比较日志中最后一个条目的索引和任期号来决定两个日志哪一个更新。如果两个日志的任期号不同，任期号大的更新；如果任期号相同，

更长的日志更新。

5.4.2 提交之前任期的日志条目

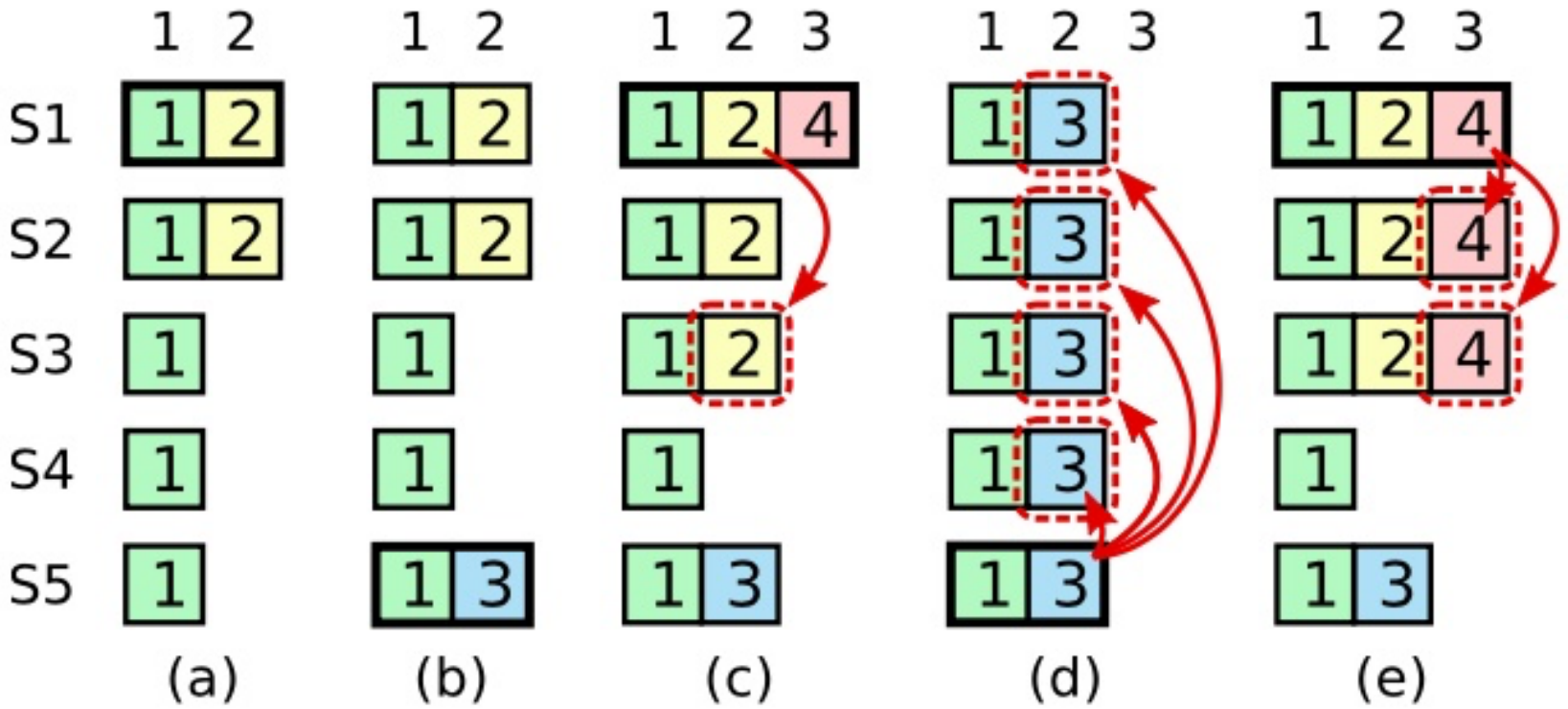

图 -8：如图的时间序列说明了为什么领导人不能通过之前任期的日志条目判断它的提交状态。（a）中的 S1 是领导人并且部分复制了索引 2 上的日志条目。（b）中 S1 崩溃了；S5 通过 S3，S4 和自己的选票赢得了选举，并且在索引 2 上接收了另一条日志条目。（c）中 S5 崩溃了，S1 重启了，通过 S2，S3 和自己的选票赢得了选举，并且继续索引 2 处的复制，这时任期 2 的日志条目已经在大部分服务器上完成了复制，但是还并没有提交。如果在（d）时刻 S1 崩溃了，S5 会通过 S2，S3，S4 的选票成为领导人，然后用它自己在任期 3 的日志条目覆盖掉其他服务器的日志条目。然而，如果在崩溃之前，S1 在它的当前任期在大多数服务器上复制了一条日志条目，就像在（e）中那样，那么这条条目就会被提交（S5 就不会赢得选举）。在这时，之前的日志条目就会正常被提交。

正如 5.3 节 中描述的那样，只要一个日志条目被存在了在多数的服务器上，领导人就知道当前任期就可以提交该条目了。如果领导人在提交之前就崩溃了，之后的领导人会试着继续完成对日志的复制。然而，领导人并不能断定存储在大多数服务器上的日志条目一定在之前的任期中被提交了。图 -8 说明了一种情况，一条存储在了大多数服务器上的日志条目仍然被新上任的领导人覆盖了。

为了消除 图 -8 中描述的问题，Raft 从来不会通过计算复制的数目来提交

之前人气的日志条目。只有领导人当前任期的日志条目才能通过计算数目来进行提交。一旦当前任期的日志条目以这种方式被提交，那么由于日志匹配原则（Log Matching Property），之前的日志条目也都会被间接的提交。在某些情况下，领导人可以安全的知道一个老的日志条目是否已经被提交（例如，通过观察该条目是否存储到所有服务器上），但是 Raft 为了简化问题使用了一种更加保守的方法。

因为当领导人从之前任期复制日志条目时日志条目保留了它们最开始的任期号，所以这使得 Raft 在提交规则中增加了额外的复杂性。在其他的一致性算法中，如果一个新的领导人要从之前的任期中复制日志条目，它必须要使用当前的新任期号。Raft 的方法使得判断日志更加容易，因为它们全程都保持着同样的任期号。另外，和其它的一致性算法相比，Raft 算法中的新领导人会发送更少的之前任期的日志条目（其他算法必须要发送冗余的日志条目并且在它们被提交之前来重新排序）。

5.4.3 安全性论证

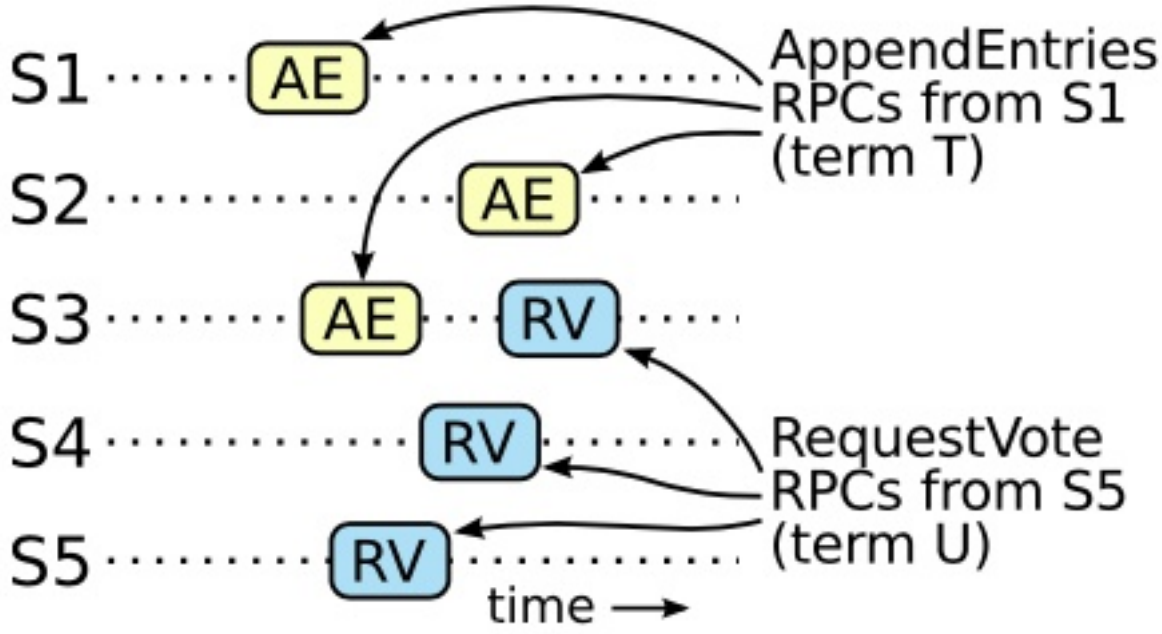

图 -9：如果 S1（任期 T 的领导人）在它的任期提交了一条日志条目，并且 S5 在之后的任期 U 成为了领导人，那么最少会有一台服务器（S3）接收了这条日志条目并且会给 S5 投票。

给出了完整的 Raft 算法，现在我们能够更精确的论证领导人完全原则（Leader Completeness）（这基于 9.2 节 提出的安全性证明）。我们假定领导人完全原则是不成立的，然后推导出矛盾。假定任期 T 的领导人 $leader_T$在它的任期提交了一个日志条目，但是这条日志条目并没有存储在之后的任期中的领导人上。我们设大于 T 的最小的任期 U 的领导人

($leader_U$） 没有存储这条日志条目。

1. 在 $leader_U$ 选举时一定没有那条被提交的日志条目（领导人从来不会删除或者覆盖日志条目）。

2. $leader_T$ 复制了这个条目到集群的大多数的服务器上。因此，只是有一台服务器（投票者）即接收了来自 $leader_T$ 的日志条目并且给 $leader_U$ 投票，就像 图 -9 中所示那样。这个投票者是产生矛盾的关键。

3. 投票者必须在给 $leader_U$ 投票之前接收来自 $leader_T$ 的日志条目；否则它会拒绝来自 $leader_T$ 的 AppendEntries 请求（它的当前任期会比 T 要大）。

4. 投票者会在它给 $leader_U$ 投票时存储那个条目，因为任何中间的领导人都保有该条目（基于假设），领导人从来不会移除这个条目，并且追随者也只会在和领导人冲突时才会移除日志条目。

5. 投票者给 $leader_U$ 投票了，所以 $leader_U$ 的日志必须和投票者的一样新。这就导致了一个矛盾。

6. 首先，如果投票者和 $leader_U$ 最后一条日志条目的任期号相同，那么 $leader_U$ 的日志一定和投票者的一样长，因此它的日志包含全部投票者的日志条目。这是矛盾的，因为在假设中投票者和 $leader_U$ 包含的已提交条目是不同的。

7. 除此之外， $leader_U$ 的最后一条日志的任期号一定比投票者的大。另外，它也比 T 要大，因为投票者的最后一条日志条目的任期号最小也要是 T（它包含了所有任期 T 提交的日志条目）。创建 $leader_U$ 最后一条日志条目的上一任领导人必须包含已经提交的日志条目（基于假设）。那么，根据日志匹配原则（Log Matching），$leader_U$ 也一定包含那条提交的日志条目，这也是矛盾的。

8. 这时就完成了矛盾推导。因此，所有比任期 T 大的领导人一定包含所有在任期 T 提交的日志条目。

9. 日志匹配原则（Log Matching）保证了未来的领导人也会包含被间接

提交的日志条目，就像 图 -8 中（d）时刻索引为 2 的条目。

通过给出了 领导人完全原则（Leader Completeness），我们能够证明 表 -3 中的状态机安全原则（State Machine Safety），状态机安全原则（State Machine Safety）讲的是如果一台服务器将给定索引上的日志条目应用到了它自己的状态机上，其它服务器的同一索引位置不可能应用的是其它条目。在一个服务器应用一条日志条目到它自己的状态机中时，它的日志必须和领导人的日志在该条目和之前的条目上相同，并且已经被提交。现在我们来考虑在任何一个服务器应用一个指定索引位置的日志的最小任期；日志完全特性（Log Completeness Property）保证拥有更高任期号的领导人会存储相同的日志条目，所以之后的任期里应用某个索引位置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。

最后，Raft 算法需要服务器按照日志中索引位置顺序应用日志条目。和状态机安全特性结合起来看，这就意味着所有的服务器会应用相同的日志序列集到自己的状态机中，并且是按照相同的顺序。

5.5 追随者和候选人崩溃

截止到目前，我们只讨论了领导人崩溃的问题。追随者和候选人崩溃的问题解决起来要比领导人崩溃要简单得多，这两者崩溃的处理方式是一样的。如果一个追随者或者候选人崩溃了，那么之后的发送给它的 RequestVote RPC 和 AppendEntries RPC 会失败。Raft 通过无限的重试来处理这些失败；如果崩溃的服务器重启了，RPC 就会成功完成。如果一个服务器在收到了 RPC 之后但是在响应之前崩溃了，那么它会在重启之后再次收到同一个 RPC。因为 Raft 中的 RPC 都是幂等的，因此不会有什么问题。例如，如果一个追随者收到了一个已经包含在它的日志中的 AppendEntries 请求，它会忽视这个新的请求。

5.6 时序和可用性

我们对于 Raft 的要求之一就是安全性不依赖于时序（timing）：系统不能仅仅因为一些事件发生的比预想的快一些或慢一些就产生错误。然而，可用性（系统可以及时响应客户端的特性）不可避免的要依赖时序。例如，如果消息交换在服务器崩溃时花费更多的时间，候选人不会等待太长的时间来赢得选举；没有一个稳定的领导人，Raft 将无法工作。

领导人选取是 Raft 中对时序要求最关键的地方。Raft 会选出并且保持一个稳定的领导人只有系统满足下列时序要求（timing requirement）：

broadcastTime << electionTimeout << MTBF

在这个不等式中，broadcastTime指的是一台服务器并行的向集群中的其他服务器发送 RPC 并且收到它们的响应的平均时间；electionTimeout指的就是在 5.2 节 描述的选举超时时间；MTBF指的是单个服务器发生故障的间隔时间的平均数。broadcastTime应该比electionTimeout小一个数量级，为的是使领导人能够持续发送心跳信息（heartbeat）来阻止追随者们开始选举；根据已经给出的随机化选举超时时间方法，这个不等式也使得瓜分选票的情况变成不可能。electionTimeout也要比MTBF小几个数量级，为的是使得系统稳定运行。当领导人崩溃时，整个大约会在electionTimeout的时间内不可用；我们希望这种情况仅占全部时间的很小的一部分。

broadcastTime和MTBF是由系统决定的性质，但是electionTimeout是我们必须做出选择的。Raft 的 RPC 需要接收方将信息持久化的保存到稳定存储中去，所以广播时间大约是 0.5 毫秒到 20 毫秒，这取决于存储的技术。因此，electionTimeout一般在 10ms 到 500ms 之间。大多数的服务器的MTBF都在几个月甚至更长，很容易满足这个时序需求。

6. 集群成员变化

截止到目前，我们都假定集群的配置（加入到一致性算法的服务器集合）是固定的。在实际中，我们会经常更改配置，例如，替换掉那些崩溃的机器或者更改复制级别。虽然通过关闭整个集群，升级配置文件，然后重启整个集群也可以解决这个问题，但是这回导致在更改配置的过程中，整个集群不可用。另外，如果存在需要手工操作，那么就会有操作失误的风险。为了避免这些问题，我们决定采用自动改变配置并且把这部分加入到了 Raft 一致性算法中。

为了让配置修改机制能够安全，那么在转换的过程中在任何时间点两个领导人不能再同一个任期被同时选为领导人。不幸的是，服务器集群从旧的配置直接升级到新的配置的任何方法都是不安全的，一次性自动的转换所

有服务器是不可能的，所以集群可以在转换的过程中划分成两个单独的组（如 图 -10 所示）。

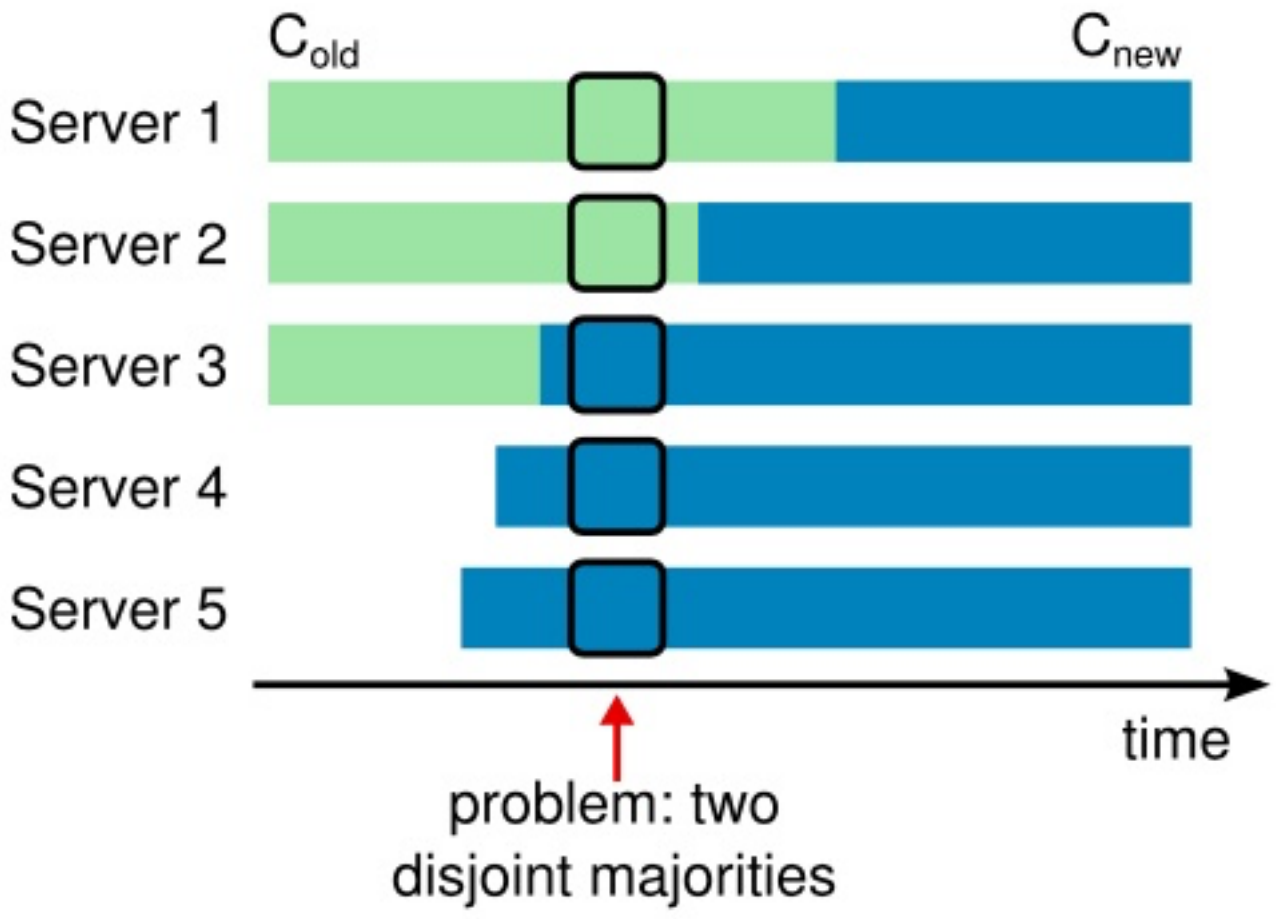

图 -10：从一个配置切换到另一个配置是不安全的因为不同的服务器会在不同的时间点进行切换。在这个例子中，集群数量从三台转换成五台。不幸的是，在一个时间点有两个服务器能被选举成为领导人，一个是在使用旧的配置的机器中（C_{old}）选出的领导人，另一个领导人是通过新的配置（C_{new}）选出来的。

为了保证安全性，集群配置的调整必须使用两阶段（two-phase）方法。有许多种实现两阶段方法的实现。例如，一些系统在第一个阶段先把旧的配置设为无效使得它无法处理客户端请求，然后在第二阶段启用新的配置。在 Raft 中，集群先切换到一个过渡配置，我们称其为共同一致（joint consensus）；一旦共同一致被提交了，然后系统再切换到新的配置。共同一致是旧的配置和新的配置的组合：

- 日志条目被复制给集群中新、老配置的所有服务器。
- 新、老配置的服务器都能成为领导人。
- 需要分别在两种配置上获得大多数的支持才能达成一致（针对选举和提交）

共同一致允许独立的服务器在不影响安全性的前提下，在不同的时间进行配置转换过程。此外，共同一致可以让集群在配置转换的过程中依然能够响应服务器请求。

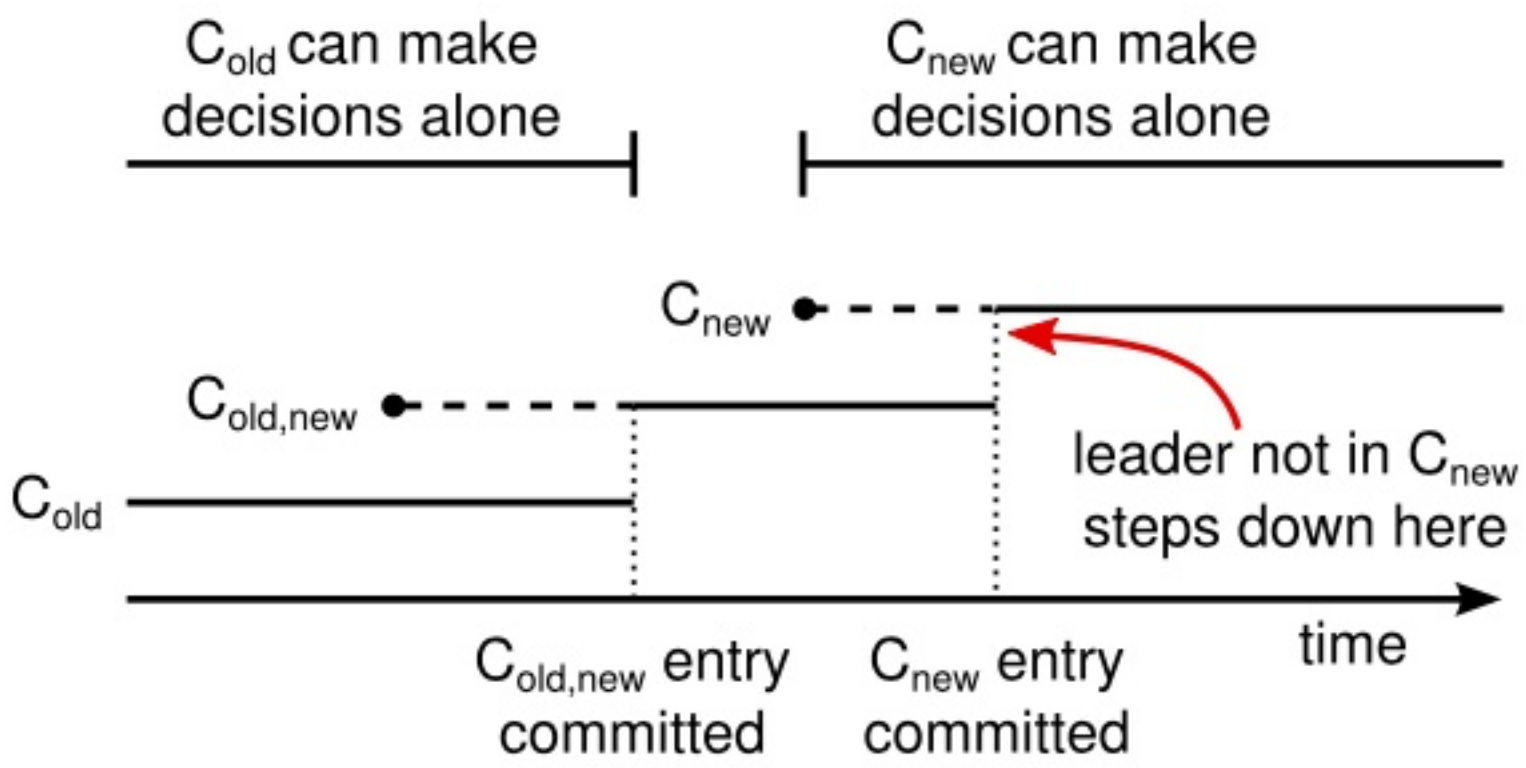

图 -11：集群配置变更的时间线。虚线表示的是已经被创建但是还没提交的配置条目，实线表示的是最新提交的配置条目。领导人首先在它的日志中创建 $C_{old,new}$配置条目并且将它提交到 $C_{old,new}$（使用旧配置的大部分服务器和使用新配置的大部分服务器）。然后创建它创建 C_{new}配置条目并且将它提交到使用新配置的大部分机器上。这样就不存在 C_{old}和 C_{new}能够分别同时做出决定的时刻。

集群配置在复制日志中用特殊的日志条目来存储和通信；图 -11 展示了配置变更的过程。当一个领导人接收到一个改变配置 C_{old} 为 C_{new} 的请求，它会为了共同一致以前面描述的日志条目和副本的形式将配置存储起来（图中的 $C_{old,new}$）。一旦一个服务器将新的配置日志条目增加到它的日志中，它就会用这个配置来做出未来所有的决定（服务器总是使用最新的配置，无论它是否已经被提交）。这意味着领导人要使用 $C_{old,new}$ 的规则来决定日志条目 $C_{old,new}$ 什么时候需要被提交。如果领导人崩溃了，被选出来的新领导人可能是使用 C_{old} 配置也可能是 $C_{old,new}$ 配置，这取决于赢得选举的候选人是否已经接收到了 $C_{old,new}$ 配置。在任何情况下， C_{new} 配置在这一时期都不会单方面的做出决定。

一旦 $C_{old,new}$ 被提交，那么无论是 C_{old} 还是 C_{new}，在没有经过他人批准的情况下都不可能做出决定，并且领导人完全特性（Leader

Completeness Property）保证了只有拥有 $C_{old,new}$ 日志条目的服务器才有可能被选举为领导人。这个时候，领导人创建一条关于 C_{new} 配置的日志条目并复制给集群就是安全的了。另外，每个服务器在收到新的配置的时候就会立即生效。当新的配置在 C_{new} 的规则下被提交，旧的配置就变得无关紧要，同时不使用新的配置的服务器就可以被关闭了。如 图 -11，C_{old} 和 C_{new} 没有任何机会同时做出单方面的决定；这就保证了安全性。

针对重新配置提出了三个问题。第一个问题是一开始的时候新的服务器可能没有任何日志条目。如果它们在这个状态下加入到集群中，那么它们需要一段时间来更新追赶，在这个阶段它们还不能提交新的日志条目。为了避免这种可用性的间隔时间，Raft 在配置更新的时候使用了一种额外的阶段，在这个阶段，新的服务器以没有投票权的身份加入到集群中来（领导人复制日志给他们，但是不把它们考虑到大多数中）。一旦新的服务器追赶上了集群中的其它机器，重新配置可以像上面描述的一样处理。

第二个问题是，集群的领导人可能不是新配置的一员。在这种情况下，领导人就会在提交了 C_{new} 日志之后退位（回到跟随者状态）。这意味着有这样的一段时间，领导人管理着集群，但是不包括自己；它复制日志但是不把它自己看作是大多数之一。当 C_{new} 被提交时，会发生领导人过渡，因为这时是新的配置可以独立工作的最早的时间点（总是能够在 C_{new} 配置下选出新的领导人）。在此之前，可能只能从 C_{old} 中选出领导人。

第三个问题是，移除不在 C_{new} 中的服务器可能会扰乱集群。这些服务器将不会再接收到心跳（heartbeat），所以当选举超时时，它们就会进行新的选举过程。它们会发送带有新的任期号的 RequestVote RPC，这样会导致当前的领导人回退成跟随者状态。新的领导人最终会被选出来，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，导致整体可用性大幅降低。

为了避免这个问题，当服务器确认当前领导人存在时，服务器会忽略 RequestVote RPC。特别的，当服务器在当前最小选举超时时间内收到一个 RequestVote RPC，它不会更新当前的任期号或者投出选票。这不会影响正常的选举，每个服务器在开始一次选举之前，至少等待一个最小选举超时时间。然而，这有利于避免被移除的服务器扰乱：如果领导人能够发送心跳给集群，那么它就不会被更大的任期号废除。

7. 日志压缩

Raft 产生的日志在持续的正常操作中不断增长，但是在实际的系统中，它不会无限的增长下去。随着日志的不断增长，它会占据越来越多的空间并且花费更多的时间重置。如果没有一个机制使得它能够废弃在日志中不断累积的过时的信息就会引起可用性问题。

快照（snapshot）是最简单的压缩方式。在快照中，全部的当前系统状态都被写入到快照中，存储到持久化的存储中，然后在那个时刻之前的全部日志都可以被丢弃。在 Chubby 和 ZooKeeper 中都使用了快照技术，这一章的剩下的部分会介绍 Raft 中使用的快照技术。

增量压缩（incremental approaches）的方法，例如日志清理（log cleaning）或者日志结构合并树（log-structured merge trees），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压力。首先，他们先选择一个已经积累的大量已经被删除或者被覆盖对象的区域，然后重写那个区域还活跃的对象，之后释放那个区域。和简单操作整个数据集合的快照相比，需要增加复杂的机制来实现。状态机可以使用和快照相同的接口来实现 LSM tree，但是日志清除方法就需要修改 Raft 了。

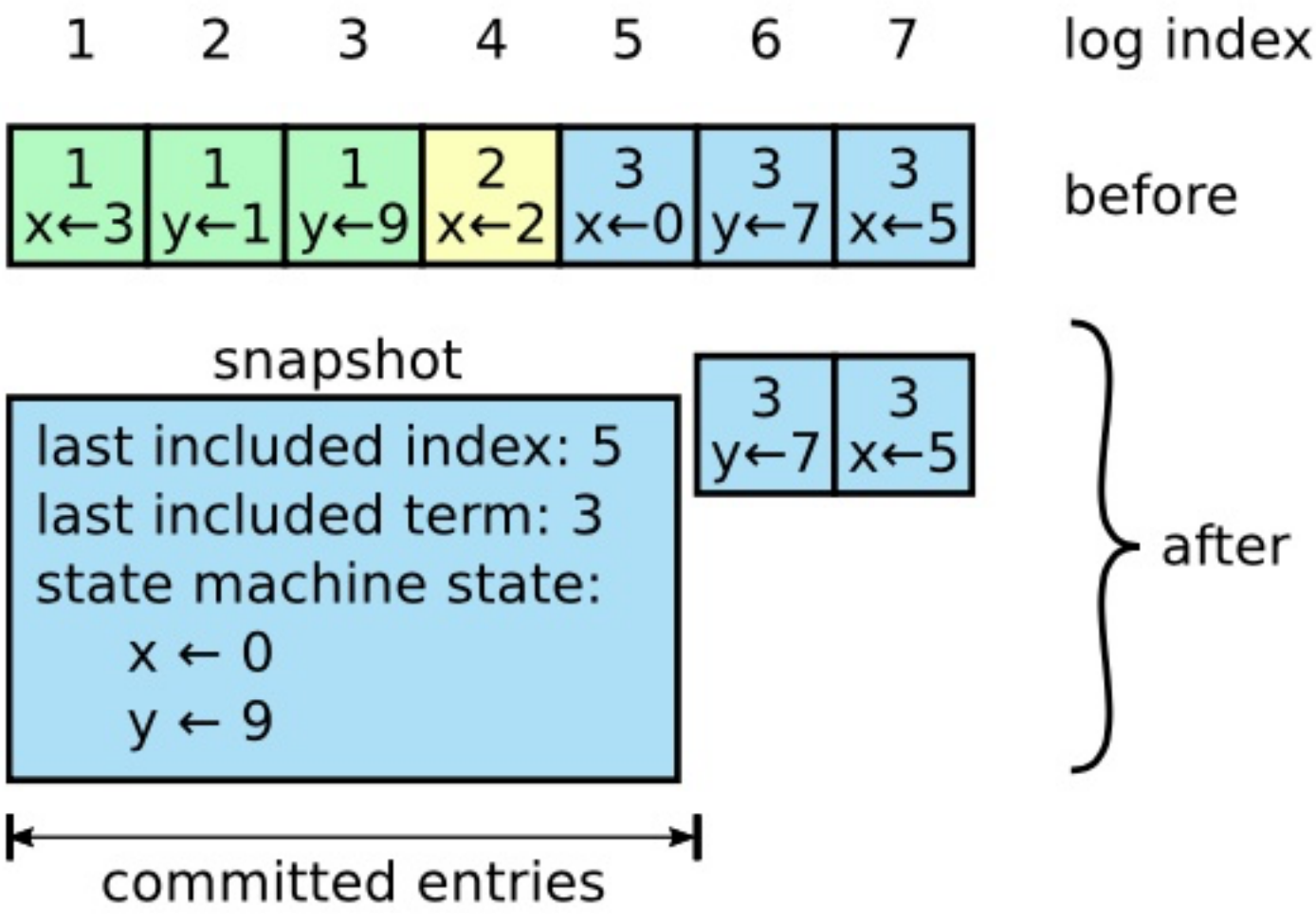

图 -12：一个服务器用新的快照替换了从 1 到 5 的条目，快照值存储了当前的状态。快照中包含了最后的索引位置和任期号。

图 -12 展示了 Raft 中快照的基础思想。每个服务器独立的创建快照，只包括已经被提交的日志。主要的工作包括将状态机的状态写入到快照中。Raft 也将一些少量的元数据包含到快照中：最后被包含的索引（last included index）指的是被快照取代的最后的条目在日志中的索引值（状态机最后应用的日志），最后被包含的任期（last included term）指的是该条目的任期号。保留这些数据是为了支持快照前的第一个条目的附加日志请求时的一致性检查，因为这个条目需要最后的索引值和任期号。为了支持集群成员更新（第 6 章），快照中也将最后的一次配置作为最后一个条目存下来。一旦服务器完成一次快照，他就可以删除最后索引位置之前的所有日志和快照了。

尽管通常服务器都是独立的创建快照，但是领导人必须偶尔的发送快照给一些落后的跟随者。这通常发生在当领导人已经丢弃了下一条需要发送给跟随者的日志条目的时候。幸运的是这种情况不是常规操作：一个与领导人保持同步的跟随者通常都会有这个条目。然而一个运行非常缓慢的跟随者或者新加入集群的服务器（第 6 章）将不会有这个条目。这时让这个跟随者更新到最新的状态的方式就是通过网络把快照发送给它们。

** 安装快照 RPC（InstallSnapshot RPC）**

在领导人发送快照给跟随者时使用调用。领导人总是按顺序发送。

参数	描述
term	领导人的任期
leaderId	为了追随者能重定向到客户端
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	分块在快照中的偏移量
data[]	快照块的原始数据
done	如果是最后一块数据则为真

返回值	描述
term	currentTerm，用于领导人更新自己

接受者需要实现：

1. 如果`term < currentTerm`立刻回复
2. 如果是第一个分块（offset 为 0）则创建新的快照
3. 在指定的偏移量写入数据
4. 如果 `done`为 false，则回复并继续等待之后的数据
5. 保存快照文件，丢弃所有存在的或者部分有着更小索引号的快照
6. 如果现存的日志拥有相同的最后任期号和索引值，则后面的数据继续保留并且回复
7. 丢弃全部日志
8. 能够使用快照来恢复状态机（并且装载快照中的集群配置）

表 -13：InstallSnapshot RPC 的总结。为了便于传输，快照都是被分成分块的；每个分块都给了跟随者生存的信号，所以跟随者可以重置选举超时计时器。

在这种情况下领导人使用一种叫做安装快照（InstallSnapshot）的新的 RPC 来发送快照给太落后的跟随者；见 表 -13。当跟随者通过这种 RPC 接收到快照时，它必须自己决定对于已经存在的日志该如何处理。通常快照会包含没有在接收者日志中存在的信息。在这种情况下，跟随者直接丢弃它所有的日志；这些会被快照所取代，但是可能会和没有提交的日志产生冲突。如果接收到的快照是自己日志的前面部分（由于网络重传或者错误），那么被快照包含的条目将会被全部删除，但是快照之后的条目必须是正确的和并且被保留下来。

这种快照的方式背离了 Raft 的强领导人原则（strong leader principle），因为跟随者可以在不知道领导人情况下创建快照。但是我们认为这种背离是值得的。领导人的存在，是为了解决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，这时不存在冲突了，所以没有领导人也是可以的。数据依然是从领导人传给跟随者，只是跟随者可以重新组织它们的数据了。

我们考虑过一种替代的基于领导人的快照方案，即只有领导人创建快照，然后发送给所有的跟随者。但是这样做有两个缺点。第一，发送快照会浪费网络带宽并且延缓了快照处理的时间。每个跟随者都已经拥有了所有产生快照需要的信息，而且很显然，自己从本地的状态中创建快照比通过网络接收别人发来的要经济。第二，领导人的实现会更加复杂。例如，领导人需要发送快照的同时并行的将新的日志条目发送给跟随者，这样才不会阻塞新的客户端请求。

还有两个问题影响了快照的性能。首先，服务器必须决定什么时候应该创建快照。如果快照创建的过于频繁，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，它就要承受耗尽存储容量的风险，同时也增加了从日志重建的时间。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。如果这个阈值设置的显著大于期望的快照的大小，那么快照对磁盘压力的影响就会很小了。

第二个影响性能的问题就是写入快照需要花费显著的一段时间，并且我们还不希望影响到正常操作。解决方案是通过写时复制（copy-on-write）的技术，这样新的更新就可以被接收而不影响到快照。例如，具有函数式数据结构的状态机天然支持这样的功能。另外，操作系统的写时复制技术的支持（如 Linux 上的 fork）可以被用来创建完整的状态机的内存快照（我们的实现就是这样的）。

8. 客户端交互

这一节将介绍客户端是如何和 Raft 进行交互的，包括客户端是如何发现领导人的和 Raft 是如何支持线性化语义（linearizable semantics）的。这些问题对于所有基于一致性的系统都存在，并且 Raft 的解决方案和其他的也差不多。

Raft 中的客户端将所有请求发送给领导人。当客户端启动的时候，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领导人，那么那个服务器会拒绝客户端的请求并且提供它最近接收到的领导人的信息（附加条目请求包含了领导人的网络地址）。如果领导人已经崩溃了，那么客户端的请求就会超时；客户端之后会再次重试随机挑选服务器的过程。

我们 Raft 的目标是要实现线性化语义（linearizable semantics）（每一次操作立即执行，在它调用和收到回复之间只执行一次）。但是，如上述所说，Raft 是可以多次执行同一条命令的：例如，如果领导人在提交了这条日志之后，但是在响应客户端之前崩溃了，那么客户端会和新的领导人重试这条指令，导致这条命令就被再次执行了。解决方案就是客户端对于每一条指令都赋予一个唯一的序列号。然后，状态机跟踪每条指令最新的序列号和相应的响应。如果接收到一条指令，它的序列号已经被执行了，那么就立即返回结果，而不重新执行指令。

只读（read-only）的操作可以直接处理而不需要记录日志。但是，在不增加任何限制的情况下，这么做可能会冒着返回过期数据 (stale data) 的风险，因为领导人响应客户端请求时可能已经被新的领导人作废了，但是它还不知道。线性化的读操作必须不能返回过期数据，Raft 需要使用两个额外的措施在不使用日志的情况下保证这一点。首先，领导人必须有关于被提交日志的最新信息。领导人完全原则（Leader Completeness Property）保证了领导人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。为了知道这些信息，它需要在它的任期里提交一条日志条目。Raft 中通过领导人在任期开始的时候提交一个空白的没有任何操作的日志条目到日志中去来进行实现。第二，领导人在处理只读的请求之前必须检查自己是否已经被废除了（如果一个更新的领导人被选举出来，它自己的信息就已经过期了）。Raft 中通过让领导人在响应只读请求之前，先和集群中的大多数节点交换一次心跳（heartbeat）信息来处理这个问题。另外，领导人可以依赖心跳机制来实现一种租约的机制，但是这种方法依赖时序来保证安全性（它假设时间误差是有界的）。

9. 实现和评价

我们已经为 RAMCloud 实现了 Raft 算法作为存储配置信息的复制状态机的一部分，并且帮助 RAMCloud 协调故障转移。这个 Raft 实现包含大约 2000 行 C++ 代码，其中不包括测试、注释和空行。这些代码是开源的。同时也有大约 25 个其他独立的第三方的基于这篇论文草稿的开源实现，针对不同的开发场景。同时，很多公司已经部署了基于 Raft 的系统。

这一章会从三个方面来评估 Raft 算法：可理解性、正确性和性能。

9.1 可理解性

为了比较 Paxos 和 Raft 算法的可理解性，我们针对高层次的本科生和研究生，在斯坦福大学的高级操作系统课程和加州大学伯克利分校的分布式计算课程上，进行了一次学习的实验。我们分别拍了针对 Raft 和 Paxos 的视频课程，并准备了相应的小测验。Raft 的视频讲课覆盖了这篇论文除了日志压缩之外的所有内容；Paxos 课程包含了足够的资料来创建一个等价的复制状态机，包括单决策 Paxos，多决策 Paxos，重新配置和一些实际系统需要的性能优化（例如领导人选举）。小测验测试一些对算法的基本理解和解释一些示例。每个学生都是看完第一个视频，回答相应的测试，再看第二个视频，回答相应的测试。大约有一半的学生先进行 Paxos 部分，然后另一半先进行 Raft 部分，这是为了说明两者独立的区别从第一个算法处学来的经验。我们计算参加人员的每一个小测验的得分来看参与者是否对 Raft 的理解更好。

因素	消除偏见的手段	复习材料
相同的讲课质量	使用相同的讲师。Paxos 的讲义是基于之前在几所大学中使用的材料的并且做了改进。Paxos 的讲义要长 14%	视频
相同的测试难度	用难度给问题分组，在测试中成对出现	测验
公平的打分	使用红字标题。随机顺序打分，两个测验交替进行。	红字标题

表 -1：考虑到的可能造成偏见的因素，以及解决方案和对应的复习材料

我们尽可能的使得 Paxos 和 Raft 的比较更加公平。这个实验偏爱 Paxos 表现在两个方面：43 个参加者中有 15 个人在之前有一些 Paxos 的经验，并且 Paxos 的视频要长 14%。如表 -1 总结的那样，我们采取了一些措施来减轻这种潜在的偏见。我们所有的材料都可供审查。

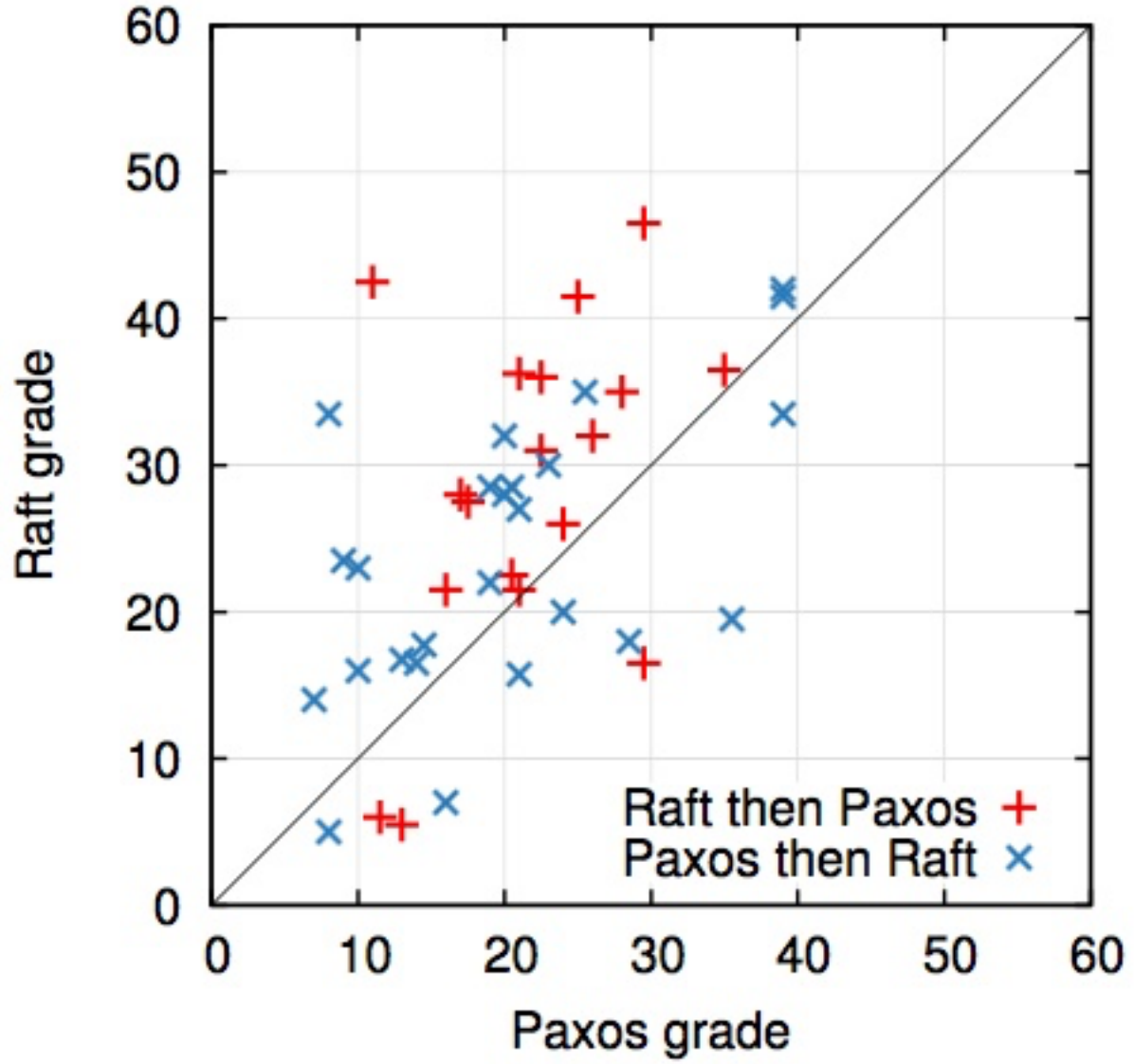

图 -14：表示了 43 个学生在 Paxos 和 Raft 的小测验中的成绩的散点图。在对角线之上的点表示在 Raft 获得了更高分数的学生。

参加者平均在 Raft 的测验中比 Paxos 高 4.9 分（总分 60，那么 Raft 的平均得分是 25.7，而 Paxos 是 20.8）；图 -14 展示了每个参与者的得分。一对 t - 测试表明，拥有 95% 的可信度，真实的 Raft 分数分布至少比 Paxos 高 2.5 分。

我们也建立了一个线性回归模型来预测一个新的学生的测验成绩，基于以下三个因素：他们使用的是哪个小测验，之前对 Paxos 的经验，和学习算法的顺序。模型显示，对小测验的选择会产生 12.5 分的差别在对 Raft 的好感度上。这显著的高于之前的 4.9 分，因为很多学生在之前都已经有了对于 Paxos 的经验，这相当明显的帮助 Paxos，对 Raft 就没什么太大影响了。但是奇怪的是，模型预测对于先进行 Paxos 小测验的人而言，Raft 的小测验得分会比 Paxos 低 6.3 分；我们不知道为什么，但这在统计学上是这样的。

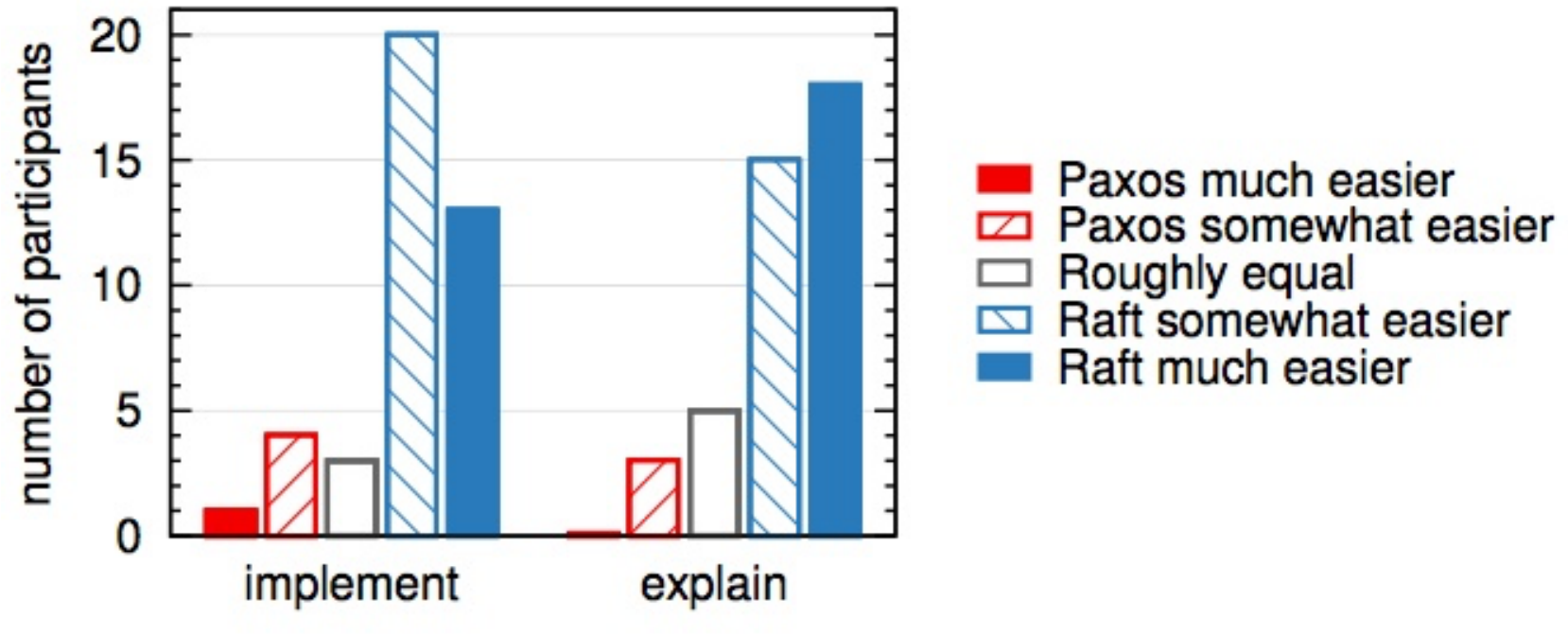

图 -15：通过一个 5 分制的问题，参与者（左边）被问哪个算法他们觉得在一个高效正确的系统里更容易实现，右边被问哪个更容易向学生解释。

我们同时也在测验之后调查了参与者，他们认为哪个算法更加容易实现和解释；这个的结果在图 -15 上。压倒性的结果表明 Raft 算法更加容易实现和解释（41 人中的 33 个）。但是，这种自己报告的结果不如参与者的成绩更加可信，并且参与者可能因为我们的 Raft 更加易于理解的假说而产生偏见。

关于 Raft 用户学习有一个更加详细的讨论，详见 http://ramcloud.stanford.edu/~ongaro/thesis.pdf

9.2 正确性

在第 5 章，我们已经进行了一个正式的说明，和对一致性机制的安全性证明。这个正式说明通过 TLA+ 让 表 -2 中的信息非常清晰。它大约有 400 行并且充当了证明的主题。同时对于任何想实现的人也是十分有用的。我们非常机械的通过 TLA 证明系统证明了日志完全特性（Log Completeness Property）。然而，这个证明依赖的约束前提还没有被机械证明（例如，我们还没有证明这个说明中的类型安全 type safety）。而且，我们已经写了一个非正式的证明关于状态机安全性质是完备的，并且是相当清晰的（大约 3500 个词）。

9.3 性能

Raft 和其他一致性算法例如 Paxos 有着差不多的性能。在性能方面，最重要的关注点是，当领导人被选举成功时，什么时候复制新的日志条目。Raft 通过很少数量的消息包（一轮从领导人到集群大多数机器的消息）就达成了这个目的。同时，进一步提升 Raft 的性能也是可行的。例如，很容易通过支持批量操作和管道操作来提高吞吐量和降低延迟。对于其他一致性算法已经提出过很多性能优化方案；其中有很多也可以应用到 Raft 中来，但是我们暂时把这个问题放到未来的工作中去。

我们使用我们自己的 Raft 实现来衡量 Raft 领导人选举的性能并且回答以下两个问题。首先，领导人选举的过程收敛是否快速？第二，在领导人宕机之后，最小的系统宕机时间是多久？

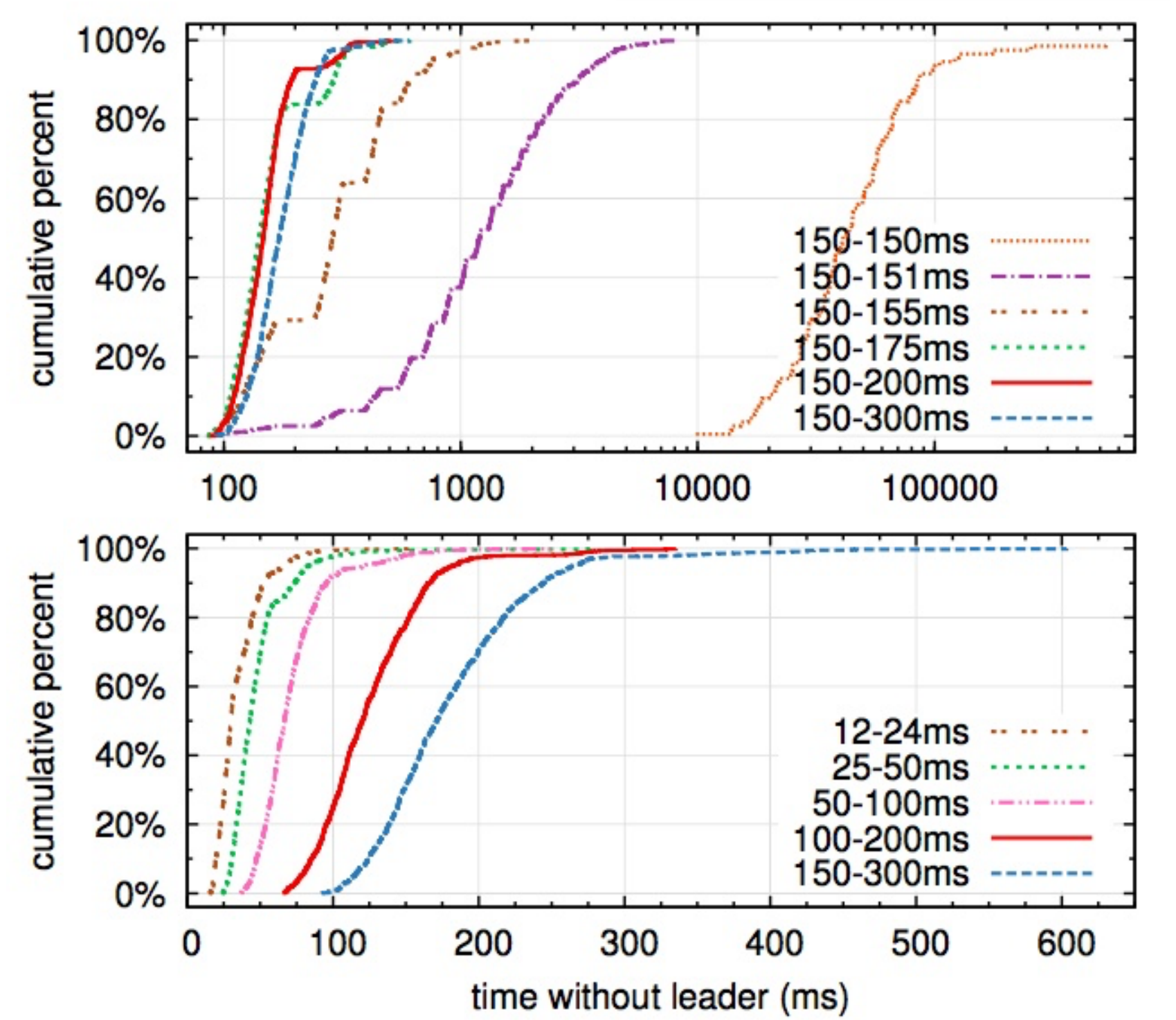

图 -16：发现并替换一个已经崩溃的领导人的时间。上面的图考察了在选举超时时间上的随机化程度，下面的图考察了最小超时时间。每条线代表

了 1000 次实验（除了 150-150 毫秒只试了 100 次），和相应的确定的选举超时时间。例如，150-155 毫秒意思是，选举超时时间从这个区间范围内随机选择并确定下来。这个实验在一个拥有 5 个节点的集群上进行，其广播时延大约是 15 毫秒。对于 9 个节点的集群，结果也差不多。

为了衡量领导人选举，我们反复的使一个拥有五个节点的服务器集群的领导人宕机，并计算需要多久才能发现领导人已经宕机并选出一个新的领导人（见图 -16）。为了构建一个最坏的场景，在每一的尝试里，服务器都有不同长度的日志，意味着有些候选人是没有成为领导人的资格的。另外，为了促成选票瓜分的情况，我们的测试脚本在终止领导人之前同步的发送了一次心跳广播（这大约和领导人在崩溃前复制一个新的日志给其他机器很像）。领导人均匀的随机的在心跳间隔里宕机，也就是最小选举超时时间的一半。因此，最小宕机时间大约就是最小选举超时时间的一半。

图 -16 上面的图表表明，只需要在选举超时时间上使用很少的随机化就可以大大避免选票被瓜分的情况。在没有随机化的情况下，在我们的测试里，选举过程由于太多的选票瓜分的情况往往都需要花费超过 10 秒钟。仅仅增加 5 毫秒的随机化时间，就大大的改善了选举过程，现在平均的宕机时间只有 287 毫秒。增加更多的随机化时间可以大大改善最坏情况：通过增加 50 毫秒的随机化时间，最坏的完成情况（1000 次尝试）只要 513 毫秒。

图 -16 中下面的图显示，通过减少选举超时时间可以减少系统的宕机时间。在选举超时时间为 12-24 毫秒的情况下，只需要平均 35 毫秒就可以选举出新的领导人（最长的一次花费了 152 毫秒）。然而，进一步降低选举超时时间的话就会违反 Raft 的时间不等式需求：在选举新领导人之前，领导人就很难发送完心跳包。这会导致没有意义的领导人改变并降低了系统整体的可用性。我们建议使用更为保守的选举超时时间，比如 150-300 毫秒；这样的时间不大可能导致没有意义的领导人改变，而且依然提供不错的可用性。

10. 相关工作

已经有很多关于一致性算法的工作被发表出来，其中很多都可以归到下面的类别中：

- Lamport 关于 Paxos 的原始描述，和尝试描述的更清晰的论文。
- 关于 Paxos 的更详尽的描述，补充遗漏的细节并修改算法，使得可以提供更加容易的实现基础。
- 实现一致性算法的系统，例如 Chubby，ZooKeeper 和 Spanner。对于 Chubby 和 Spanner 的算法并没有公开发表其技术细节，尽管他们都声称是基于 Paxos 的。ZooKeeper 的算法细节已经发表，但是和 Paxos 有着很大的差别。
- Paxos 可以应用的性能优化。
- Oki 和 Liskov 的 Viewstamped Replication（VR），一种和 Paxos 差不多的替代算法。原始的算法描述和分布式传输协议耦合在了一起，但是核心的一致性算法在最近的更新里被分离了出来。VR 使用了一种基于领导人的方法，和 Raft 有很多相似之处。

Raft 和 Paxos 最大的不同之处就在于 Raft 的强领导特性：Raft 使用领导人选举作为一致性协议里必不可少的部分，并且将尽可能多的功能集中到了领导人身上。这样就可以使得算法更加容易理解。例如，在 Paxos 中，领导人选举和基本的一致性协议是正交的：领导人选举仅仅是性能优化的手段，而且不是一致性所必须要求的。但是，这样就增加了多余的机制：Paxos 同时包含了针对基本一致性要求的两阶段提交协议和针对领导人选举的独立的机制。相比较而言，Raft 就直接将领导人选举纳入到一致性算法中，并作为两阶段一致性的第一步。这样就减少了很多机制。

像 Raft 一样，VR 和 ZooKeeper 也是基于领导人的，因此他们也拥有一些 Raft 的优点。但是，Raft 比 VR 和 ZooKeeper 拥有更少的机制因为 Raft 尽可能的减少了非领导人的功能。例如，Raft 中日志条目都遵循着从领导人发送给其他人这一个方向：附加条目 RPC 是向外发送的。在 VR 中，日志条目的流动是双向的（领导人可以在选举过程中接收日志）；这就导致了额外的机制和复杂性。根据 ZooKeeper 公开的资料看，它的日志条目也是双向传输的，但是它的实现更像 Raft。

和上述我们提及的其他基于一致性的日志复制算法中，Raft 的消息类型更少。例如，我们数了一下 VR 和 ZooKeeper 使用的用来基本一致性需要和成员改变的消息数（排除了日志压缩和客户端交互，因为这些都比较独立且和算法关系不大）。VR 和 ZooKeeper 都分别定义了 10 中不同的消息类型，相对的，Raft 只有 4 中消息类型（两种 RPC 请求和对应的响

应）。Raft 的消息都稍微比其他算法的要信息量大，但是都很简单。另外，VR 和 ZooKeeper 都在领导人改变时传输了整个日志；所以为了能够实践中使用，额外的消息类型就很必要了。

Raft 的强领导人模型简化了整个算法，但是同时也排斥了一些性能优化的方法。例如，平等主义 Paxos （EPaxos）在某些没有领导人的情况下可以达到很高的性能。平等主义 Paxos 充分发挥了在状态机指令中的交换性。任何服务器都可以在一轮通信下就提交指令，除非其他指令同时被提出了。然而，如果指令都是并发的被提出，并且互相之间不通信沟通，那么 EPaxos 就需要额外的一轮通信。因为任何服务器都可以提交指令，所以 EPaxos 在服务器之间的负载均衡做的很好，并且很容易在 WAN 网络环境下获得很低的延迟。但是，他在 Paxos 上增加了非常明显的复杂性。

一些集群成员变换的方法已经被提出或者在其他的工作中被实现，包括 Lamport 的原始的讨论，VR 和 SMART。我们选择使用共同一致（joint consensus）的方法因为它对一致性协议的其他部分影响很小，这样我们只需要很少的一些机制就可以实现成员变换。Raft 没有采用 Lamport 的基于 α 的方法是因为它假设在没有领导人的情况下也可以达到一致性。和 VR 和 SMART 相比较，Raft 的重新配置算法可以在不限制正常请求处理的情况下进行；相比较而言，VR 需要停止所有的处理过程，SMART 引入了一个和 α 类似的方法，限制了请求处理的数量。和 VR、SMART 比较而言，Raft 的方法同时需要更少的额外机制来实现。

11. 总结

算法的设计通常会把正确性，效率或者简洁作为主要的目标。尽管这些都是很有意义的目标，但是我们相信，可理解性也是一样的重要。在开发者把算法应用到实际的系统中之前，这些目标没有一个会被实现，这些都会必然的偏离发表时的形式。除非开发人员对这个算法有着很深的理解并且有着直观的感觉，否则将会对他们而言很难在实现的时候保持原有期望的特性。

在这篇论文中，我们尝试解决分布式一致性问题，但是一个广为接受但是十分令人费解的算法 Paxos 已经困扰了无数学生和开发者很多年了。我们创造了一种新的算法 Raft，显而易见的比 Paxos 要容易理解。我们同时也相信，Raft 也可以为实际的实现提供坚实的基础。把可理解性作为设计的

目标改变了我们设计 Raft 的方式；这个过程是我们发现我们最终很少有技术上的重复，例如问题分解和简化状态空间。这些技术不仅提升了 Raft 的可理解性，同时也使我们坚信其正确性。

12. 鸣谢

这项研究必须感谢以下人员的支持：Ali Ghodsi，David Mazie res，和伯克利 CS 294-91 课程、斯坦福 CS 240 课程的学生。Scott Klemmer 帮我们设计了用户调查，Nelson Ray 建议我们进行统计学的分析。在用户调查时使用的关于 Paxos 的幻灯片很大一部分是从 Lorenzo Alvisi 的幻灯片上借鉴过来的。特别的，非常感谢 DavidMazieres 和 Ezra Hoch，他们找到了 Raft 中一些难以发现的漏洞。许多人提供了关于这篇论文十分有用的反馈和用户调查材料，包括 Ed Bugnion，Michael Chan，Hugues Evrard，Daniel Giffin，Arjun Gopalan，Jon Howell，Vimalkumar Jeyakumar，Ankita Kejriwal，Aleksandar Kracun，Amit Levy，Joel Martin，Satoshi Matsushita，Oleg Pesok，David Ramos，Robbert van Renesse，Mendel Rosenblum，Nicolas Schiper，Deian Stefan，Andrew Stone，Ryan Stutsman，David Terei，Stephen Yang，Matei Zaharia 以及 24 位匿名的会议审查人员（可能有重复），并且特别感谢我们的领导人 Eddie Kohler。Werner Vogels 发了一条早期草稿链接的推特，给 Raft 带来了极大的关注。我们的工作由 Gigascale 系统研究中心和 Multiscale 系统研究中心给予支持，这两个研究中心由关注中心研究程序资金支持，一个是半导体研究公司的程序，由 STARnet 支持，一个半导体研究公司的程序由 MARCO 和 DARPA 支持，在国家科学基金会的 0963859 号批准，并且获得了来自 Facebook，Google，Mellanox，NEC，NetApp，SAP 和 Samsung 的支持。Diego Ongaro 由 Junglee 公司，斯坦福的毕业团体支持。

引用

1. BOLOSKY, W. J., BRADSHAW, D., HAAGENS, R. B., KUSTERS, N. P., AND LI, P. Paxos replicated state machines as the basis of a high-performance data store. In Proc. NSDI’11, USENIX Conference on Networked Systems Design and Implementation (2011), USENIX, pp. 141–154.

2. BURROWS, M. The Chubby lock service for loosely- coupled distributed systems. In Proc. OSDI'06, Sympo- sium on Operating Systems Design and Implementation (2006), USENIX, pp. 335–350.
3. CAMARGOS, L. J., SCHMIDT, R. M., AND PEDONE, F. Multicoordinated Paxos. In Proc. PODC'07, ACM Sym- posium on Principles of Distributed Computing (2007), ACM, pp. 316–317.
4. CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In Proc. PODC'07, ACM Symposium on Principles of Distributed Computing (2007), ACM, pp. 398–407.
5. CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In Proc. OSDI'06, USENIX Symposium on Operating Systems Design and Implementation (2006), USENIX, pp. 205–218.
6. CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KAN- THAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In Proc. OSDI'12, USENIX Conference on Operating Systems Design and Implemen- tation (2012), USENIX, pp. 251–264.
7. COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ, S., RICKETTS, D., AND VANZETTO, H. TLA+ proofs. In Proc. FM'12, Symposium on Formal Methods (2012), D. Giannakopoulou and D. Me´ry, Eds., vol. 7436 of Lec- ture Notes in Computer Science, Springer, pp. 147–154.
8. GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In Proc. SOSP'03, ACM Symposium on Operating Systems Principles (2003), ACM, pp. 29–43.
9. GRAY,C.,ANDCHERITON,D.Leases:Anefficientfault- tolerant mechanism for distributed file cache consistency. In Proceedings of the 12th ACM Ssymposium on Operating Systems Principles (1989),

pp. 202–210.
10. HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. ACM Trans- actions on Programming Languages and Systems 12 (July 1990), 463–492.
11. HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In Proc ATC'10, USENIX Annual Technical Con- ference (2010), USENIX, pp. 145–158.
12. JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup sys- tems. In Proc. DSN'11, IEEE/IFIP Int'l Conf. on Depend- able Systems & Networks (2011), IEEE Computer Society, pp. 245–256.
13. KIRSCH, J., AND AMIR, Y. Paxos for system builders. Tech. Rep. CNDS-2008-2, Johns Hopkins University, 2008.
14. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. Commununications of the ACM 21, 7 (July 1978), 558–565.
15. LAMPORT, L. The part-time parliament. ACM Transac- tions on Computer Systems 16, 2 (May 1998), 133–169.
16. LAMPORT, L. Paxos made simple. ACM SIGACT News 32, 4 (Dec. 2001), 18–25.
17. LAMPORT, L. Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison- Wesley, 2002.
18. LAMPORT, L. Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2005.
19. LAMPORT, L. Fast paxos. Distributed Computing 19, 2 (2006), 79–103.
20. LAMPSON, B. W. How to build a highly available system using consensus. In Distributed Algorithms, O. Baboaglu and K. Marzullo, Eds. Springer-Verlag, 1996, pp. 1–17.
21. LAMPSON, B. W. The ABCD's of Paxos. In Proc. PODC'01, ACM Symposium on Principles of Distributed Computing (2001), ACM, pp. 13–13.
22. LISKOV, B., AND COWLING, J. Viewstamped replica- tion revisited.

Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.
23. LogCabin source code. logcabin/logcabin.
24. LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART way to migrate replicated stateful services. In Proc. Eu- roSys'06, ACM SIGOPS/EuroSys European Conference on Computer Systems (2006), ACM, pp. 103–115.
25. MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: building efficient replicated state machines for WANs. In Proc. OSDI'08, USENIX Conference on Operating Systems Design and Implementation (2008), USENIX, pp. 369–384.
26. MAZIE` RES, D. Paxos made practical.http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf , Jan. 2007.
27. MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In Proc. SOSP'13, ACM Symposium on Operating System Principles (2013), ACM.
28. Raft user study. http://ramcloud.stanford.edu/~ongaro/userstudy/.
29. OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In Proc. PODC'88, ACM Symposium on Principles of Distributed Computing (1988), ACM, pp. 8–17.
30. O'NEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (LSM-tree). Acta Informat- ica 33, 4 (1996), 351–385.
31. ONGARO, D. Consensus: Bridging Theory and Practice. PhD thesis, Stanford University, 2014 (work in progress).http://ramcloud.stanford.edu/~ongaro/thesis.pdf
32. ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In Proc ATC'14, USENIX Annual Technical Conference (2014), USENIX.
33. OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIE`RES, D., MI- TRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M.,

STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. Com- munications of the ACM 54 (July 2011), 121–130.
34. Raft consensus algorithm website. http://raftconsensus.github.io.
35. REED, B. Personal communications, May 17, 2013.
36. ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. 10 (February 1992), 26–52.
37. SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Com- puting Surveys 22, 4 (Dec. 1990), 299–319.
38. SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In Proc. MSST'10, Symposium on Mass Storage Sys- tems and Technologies (2010), IEEE Computer Society, pp. 1–10.
39. VAN RENESSE, R. Paxos made moderately complex. Tech. rep., Cornell University, 2012.