In一种可理解的共识算法的搜索(扩展版)

Diego Ongaro和John Ousterhout

斯坦福大学

摘要

Raft是一种用于管理复制日志的共识算法。它产生的结果相当于(多)Paxos，其效率与Paxos一样，但其结构与Paxos不同;这使得Raft比Paxos更容易理解，也为构建实用系统提供了更好的基础。为了增强可理解性，Raft分离了共识的关键要素，如leader选举、日志复制和安全性，并强制执行更强的一致性程度，以减少必须考虑的状态数量。用户研究的结果表明，对学生来说，Raft比Paxos更容易学习。Raft还包含了一个改变集群成员的新机制，它使用重叠多数来保证安全。

1 简介

共识算法允许一组机器作为一个连贯的群体工作，可以在其中一些成员的故障中幸存下来。正因为如此，它们在构建可靠的大规模软件系统方面发挥着关键作用。Paxos[15,16]在过去十年中主导了共识-共识算法的讨论:大多数共识的实现都是基于Paxos或受其影响，Paxos已成为教授学生共识的主要工具。

不幸的是，Paxos很难理解，尽管有许多尝试使它更容易接近。此外，它的架构需要进行复杂的修改才能支持实际的系统。因此，系统构建者和学生都在与Paxos作斗争。

在与Paxos斗争之后，我们开始寻找一种新的共识算法，可以为系统构建和教育提供更好的基础。我们的方法是不同寻常的，因为我们的主要目标是可*理解性:*我们能否为实际系统定义一个共识算法，并以一种比Paxos更容易学习的方式描述它?此外，我们希望该算法能够促进对系统构建者至关重要的直觉的发展。重要的不仅是要让算法工作，而且要让它能够明显地工作。

这项工作的成果是一种被称为Raft的共识算法。在设计Raft时，我们应用了特定的技术来提高可理解性，包括分解(Raft将领导者选举、日志复制和安全性分开)和

This tech report is an extended version of [32]; additional material is noted with a gray bar in the margin. Published May 20, 2014.

状态空间减少(相对于Paxos, Raft减少了不确定性的程度和服务器之间不一致的方式)。一项针对两所大学43名学生的用户研究表明，Raft比Paxos更容易理解:在学习了这两种算法后，其中33名学生能够更好地回答关于Raft的问题，而不是关于Paxos的问题。

Raft在许多方面与现有的共识算法相似(最值得注意的是Oki和Liskov的Viewstamped Replication[29,22])，但它有几个新颖的功能:

•强领导(Strong leader): Raft使用了比其他共识算法更强的领导形式。例如，日志条目只从leader流向其他服务器。这简化了对复制日志的管理，使Raft更容易理解。

•首领选举:Raft使用随机计时器来选举首领。这只在任何共识算法已经需要的心跳基础上增加了少量的机制，同时简单快速地解决冲突。

•成员变更:Raft用于更改集群中服务器集的机制使用了一种新的*联合共识*方法，其中两种不同配置的大多数在转换期间重叠。这允许集群在配置更改期间继续正常运行。

我们相信Raft优于Paxos和其他共识算法，无论是出于教育目的还是作为实现的基础。它比其他算法更简单，更容易理解;它的描述足够完整，可以满足实际系统的需要;它有几个开源实现，被几家公司使用;它的安全属性已经被正式指定和证明;并且其效率可以与其他算法相媲美。

本文的其余部分介绍了复制状态机问题(第2节)，讨论了Paxos的优缺点(第3节)，描述了我们实现可理解性的一般方法(第4节)，介绍了Raft共识算法(第5-8节)，评估了Raft(第9节)，并讨论了相关工作(第10节)。

2 复制状态机

共识算法通常出现在*复制状态机*[37]的上下文中。在这种方法中，一组服务器上的状态机计算相同状态的相同副本，即使某些服务器宕机，状态机也可以继续运行。复制状态机有

1

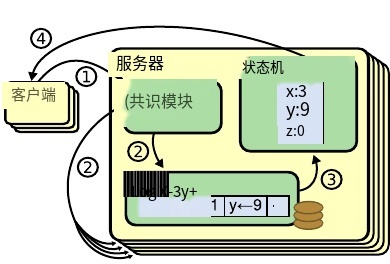


图1:复制状态机架构。consensus -sensus算法管理一个包含来自客户端的状态机命令的复制日志。状态机处理来自日志的相同命令序列，因此它们产生相同的输出。

用于解决分布式系统中的各种容错问题。例如，具有单个集群领导者的大型系统，如GFS[8]、HDFS[38]和RAMCloud[33]，通常使用单独的复制状态机来管理领导者选举和存储必须在领导者崩溃时存活的配置信息。复制状态机的例子包括Chubby[2]和ZooKeeper[11]。

复制的状态机通常使用复制的日志来实现，如图1所示。每台服务器存储一个包含一系列命令的日志，其状态机按顺序执行这些命令。每个日志以相同的顺序包含相同的命令，因此每个状态机处理相同的命令序列。由于状态机是确定性的，所以每个状态机都计算相同的状态和相同的输出序列。

保持复制日志的一致性是共识算法的工作。服务器上的共识模块接收来自客户端的命令，并将其添加到其日志中。它与其他服务器上的共识模块通信，以确保每个日志最终以相同的顺序包含相同的请求，即使某些服务器失败。一旦命令被正确复制，每个服务器的状态机就会按照日志顺序处理它们，并将输出返回给客户端。因此，这些服务器似乎形成了一个单一的、高可靠性的状态机。

用于实际系统的共识算法通常具有以下属性:

•它们确保在所有非拜占庭条件下的*安全性*(永远不会返回不正确的结果)，包括网络延迟、分区、丢包、重复和重新排序。

•只要任何大多数服务器都是可操作的，并且可以相互通信和与客户端通信，它们就是功能齐全的(可用的)。因此，一个典型的五台服务器集群可以容忍任意两台服务器的故障。假定服务器通过停止而失效;它们可能稍后从稳定存储上的状态恢复并重新加入集群。

•它们不依赖于定时来确保日志

的一致性:错误的时钟和极端消息

在最坏的情况下，延迟会导致可用性问题。•在通常情况下，只要集群的大多数响应了一轮远程过程调用，命令就可以完成;少数慢速服务器不需要影响整体系统性能。

3 Paxos有什么问题?

在过去的十年中，Leslie Lamport的Paxos协议[15]几乎已经成为共识的代名词:它是课程中最常教授的协议，并且大多数共识的实现都将其作为起点。Paxos首先定义了一个能够就单个决策达成一致的协议，比如单个复制的日志条目。我们将这个子集称为*单命令Paxos。*然后，Paxos结合该协议的多个实例来促进一系列决策，例如日志(多Paxos)。Paxos确保了安全性和活动性，并且支持更改集群成员。它的正确性已经被证明，在正常情况下是高效的。

不幸的是，Paxos有两个明显的缺点。第一个缺点是Paxos非常难以理解。[15]的完整解释是出了名的不透明;很少有人能成功理解它，只有付出巨大的努力。因此，已经有几次尝试用更简单的术语来解释Paxos[16,20,21]。这些解释侧重于单一法令子集，但它们仍然具有挑战性。在对2012年NSDI与会者的非正式调查中，我们发现很少有人对Paxos感到满意，即使是经验丰富的研究人员。我们自己也在与Paxos作斗争;直到阅读了几份简化的解释，并设计了自己的替代协议，我们才能够理解完整的协议，这个过程花了将近一年的时间。

我们假设Paxos的不透明性源于它选择单一法令子集作为其基础。单法令Paxos密集而微妙:它分为两个阶段，没有简单的直观解释，无法独立理解。正因为如此，很难对单法令协议的工作原理做出直观的解释。多paxos的组合规则增加了额外的复杂性和微妙性。我们认为，在多个决策上达成共识的整体问题(即，一个日志而不是单个条目)可以用其他更直接、更明显的方式来分解。

Paxos的第二个问题是，它没有为构建实际实现提供良好的基础。其中一个原因是，对于多paxos，还没有得到广泛认可的算法。兰波特的描述主要是关于单一法令的Paxos;他概述了多paxos的可能方法，但缺少许多细节。已经有几次尝试充实和优化Paxos，例如[26]、[39]和[13]，但是这些尝试是不同的

2

从彼此和兰波特的草图中。像Chubby[4]这样的系统已经实现了类似paxos的算法，但在大多数情况下，它们的细节还没有公布。

此外，Paxos架构对于构建实际系统来说是一个糟糕的架构;这是单法令分解的另一个后果。例如，独立地选择一组日志条目，然后将它们融合成一个顺序日志，几乎没有什么好处;这只会增加复杂性。围绕日志设计系统更简单、更高效，新条目以受限的顺序依次挂起。另一个问题是，Paxos在其核心使用了对称的点对点方法(尽管它最终提出了一种弱形式的领导作为性能优化)。在一个只会做出一个决策的简化世界里，这是有道理的，但很少有实际的系统使用这种方法。如果必须做出一系列决策，那么先选出一个领导者，然后让领导者来协调这些决策，会更简单、更快捷。

因此，实际系统与Paxos几乎没有相似之处。每种实现都从Paxos开始，发现实现它的困难，然后开发一个截然不同的体系结构。这既耗时又容易出错，而且理解Paxos的困难加剧了这个问题。Paxos的公式对于证明其正确性的定理可能是一个很好的公式，但是实际实现与Paxos如此不同，以至于证明几乎没有价值。Chubby实现者的以下评论很典型:

Paxos算法的描述与实际系统. . . .的需求之间存在很大的差距最终的系统将基于一个未经验证的协议[4]。

由于这些问题，我们得出结论，Paxos不能为系统构建或教育提供良好的基础。考虑到共识在大型软件系统中的重要性，我们决定看看是否可以设计一种比Paxos性能更好的替代共识算法。Raft就是这个实验的结果。

4 为可理解性而设计

我们在设计Raft时有几个目标:它必须为系统构建提供一个完整而实用的基础，这样才能显著减少开发人员所需的设计工作量;它必须在所有条件下都是安全的，并且在典型的操作条件下都是可用的;并且对于常见的操作必须是高效的。但我们最重要的目标——也是最困难的挑战——*是可理解性。*必须有可能让大量观众轻松地理解算法。此外，必须有可能开发关于算法的直觉，以便系统构建者可以进行在现实世界实现中不可避免的扩展。

在Raft的设计过程中，有很多地方需要我们在各种备选方法中进行选择。在这些情况下，我们根据可理解性来评估备选方案:解释每个备选方案有多难(例如，它的状态空间有多复杂，它是否有微妙的含义?)，读者完全理解这种方法及其含义有多容易?

我们认识到，在这样的分析中存在着高度的主观性;尽管如此，我们还是使用了两种普遍适用的技术。第一种技术是众所周知的问题分解方法:只要有可能，我们就把问题分成可以相对独立地解决、解释和理解的独立部分。例如，在Raft中，我们将领导者选举、日志复制、安全性和成员变更分开。

我们的第二种方法是通过减少要考虑的状态数量来简化状态空间，使系统更加连贯，并在可能的情况下消除不确定性。具体来说，日志是不允许有洞的，Raft限制了日志可能变得彼此不一致的方式。尽管在大多数情况下，我们都试图消除非确定性，但在某些情况下，非确定性实际上提高了可理解性。特别是，随机化方法引入了非确定性，但它们倾向于通过以类似的方式处理所有可能的选择(“选择任何;无所谓(It doesn 't matter))。我们使用随机化来简化Raft领袖选举算法。

5 Raft共识算法

Raft是一种算法，用于管理第2节中描述的形式的复制日志。图2以浓缩形式总结了该算法，供参考，图3列出了该算法的关键属性;这些图的元素将在本节的其余部分逐条讨论。

Raft通过首先选举一个杰出的*领导者来实现共识*，然后让领导者完全负责管理复制的日志。leader接受来自客户端的日志条目，在其他服务器上复制它们，并告诉服务器何时可以安全地将日志条目应用到他们的状态机。拥有一个leader简化了对复制日志的管理。例如，leader可以在不咨询其他服务器的情况下决定在日志中放置新条目的位置，数据以简单的方式从leader流向其他服务器。leader可能会出现故障或与其他服务器断开连接，在这种情况下，新的leader会被选举出来。

考虑到领导者方法，Raft将共识问题分解为三个相对独立的子问题，这些子问题将在以下小节中讨论:

•领导者选举:当现有领导者失败时，必须选择新的领导者(第5.2节)。

•日志复制:leader必须接受日志条目

3

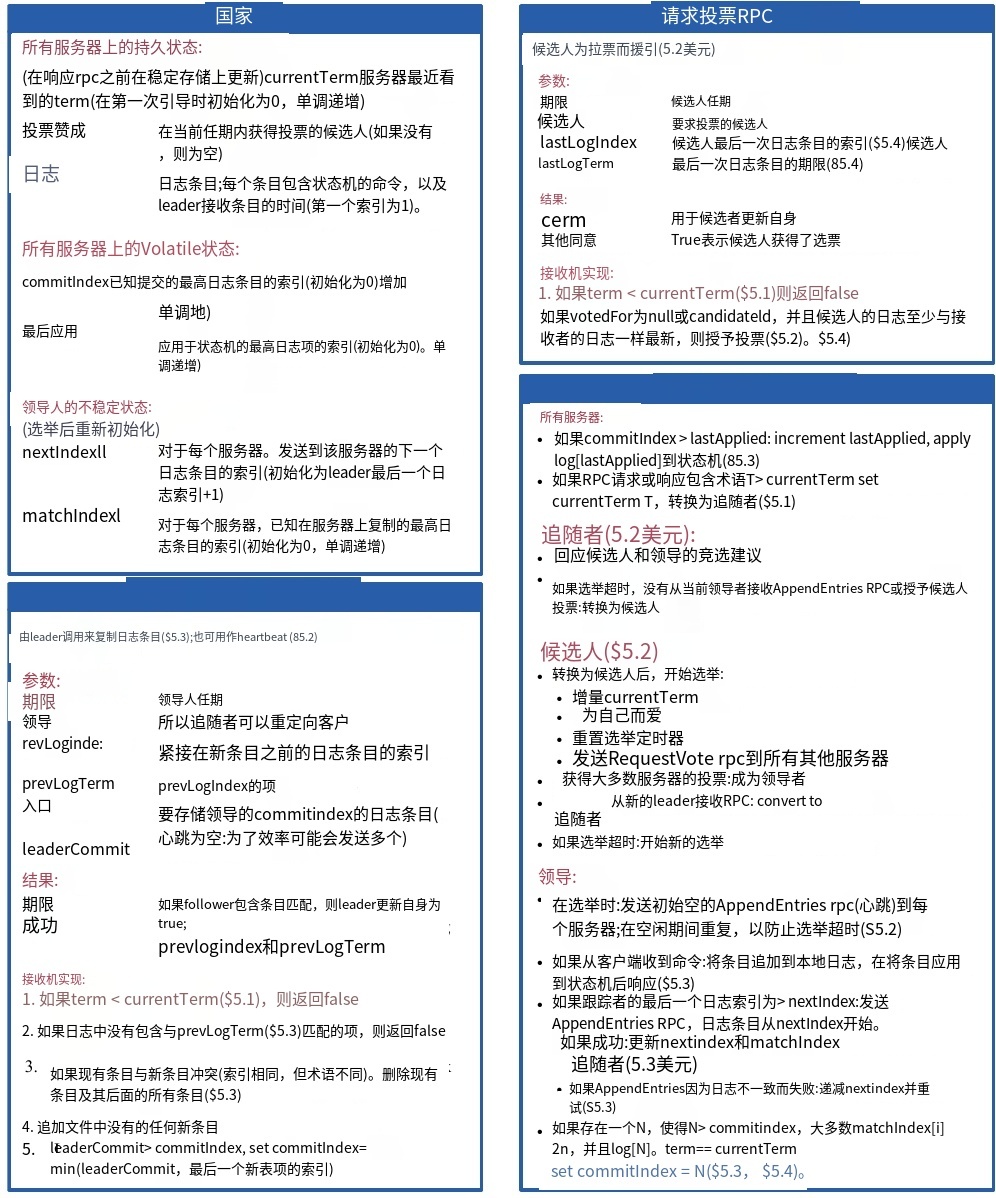


图2:Raft共识算法的浓缩总结(不包括成员变更和日志压缩)。左上角框中的服务器行为被描述为一组独立且反复触发的规则。章节编号，如§5.2表示在哪里讨论了特定的功能。正式规范[31]更精确地描述了该算法。

4

选举安全:一届任期内最多只能选出一位领导人。§5.2

Leader追加- only: Leader永远不会覆盖或删除其日志中的条目;它只会追加新的条目。§5.3

日志匹配:如果两个日志包含具有相同索引和term的条目，则日志在给定索引之前的所有条目中都是相同的。§5.3

Leader完整性:如果一个日志条目在给定的期限内提交，那么该条目将出现在所有更高编号的Leader的日志中。§5.4

状态机安全:如果一个服务器在给定索引上应用了一个日志条目到它的状态机，那么其他服务器将不会为相同的索引应用不同的日志条目。§5.4.3

图3:Raft保证这些属性在任何时候都是正确的。截面号表示讨论每个属性的位置。

并在集群中复制它们，迫使其他日志与自己的日志一致(第5.3节)。

•安全:Raft的关键安全属性是图3中的状态机安全属性:如果任何服务器已经将特定的日志条目应用到其状态机，那么其他服务器就不能对相同的日志索引应用不同的命令。5.4节描述了Raft是如何确保这个属性的;解决方案涉及对第5.2节中描述的选举机制的额外限制。

在介绍了共识算法之后，本节将讨论可用性问题和定时在系统中的作用。

**5.1 Raft basics**

一个Raft集群包含几个服务器;5是一个典型的数字，它允许系统容忍两次故障。在任何给定时间，每台服务器都处于三种状态之一:*领导者、*追随者或*候选人。*在正常运行中，只有一个领导者，所有其他服务器都是追随者。追随者是被动的:他们自己不发出任何请求，只是对领导者和候选人的请求做出回应。领导者处理所有客户的请求(如果客户联系了追随者，追随者会将其重定向给领导者)。第三个状态是candidate，用于选举新的leader，详见5.2节。图4显示了状态及其过渡;下面将讨论这些转换。

Raft将时间划分为任意长度的*项*，如图5所示。术语用连续整数编号。每个任期以*选举开始*，其中一个或多个候选人试图成为第5.2节所述的领导人。如果一名候选人赢得选举，那么他将在剩下的任期内担任领导人。在某些情况下，选举会导致票数不一致。在这种情况下，任期结束时将没有领导人;新任期(重新选举)

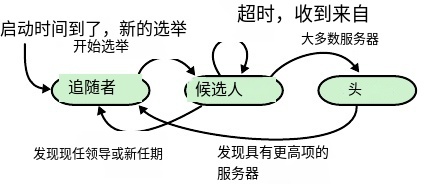


图4:服务器状态。关注者只响应来自其他服务器的请求。如果一个追随者没有收到任何通信，它就成为候选人并发起选举。从整个集群中获得多数选票的候选人成为新的领导者。领导者通常会运作到失败为止。



图5:时间分为任期，每一任期以选举开始。选举成功后，由单个leader管理集群，直到任期结束。有些选举失败，在这种情况下，任期结束而没有选出领导者。在不同的服务器上，可以在不同的时间观察到任期之间的过渡。

很快就会开始。Raft保证在给定的任期内最多有一个leader。

不同的服务器可能会在不同的时间观察任期之间的过渡，在某些情况下，服务器可能不会观察选举甚至整个任期。术语在Raft中就像一个逻辑时钟[14]，它们允许服务器检测过时的信息，比如过时的领导者。每个服务器存储一个*当前*的*术语*号，随着时间单调增加。每当服务器通信时，都会交换当前术语;如果一台服务器的当前项小于另一台服务器的，则将其当前项更新为较大的值。如果候选人或领导者发现其术语过期，则立即恢复到follow -lower状态。如果服务器接收到一个过期的术语号的请求，它会拒绝这个请求。

Raft服务器使用远程过程调用(rpc)进行通信，基本的一致性算法只需要两种类型的rpc。RequestVote rpc是由候选人在选举期间发起的(章节5.2)，而追加条目rpc是由领导者发起的，用于复制日志条目并提供心跳形式(章节5.3)。第5.4节添加了第三个RPC，用于在服务器之间传输快照。如果服务器没有及时收到响应，则重试rpc，并且它们并行地发出rpc以获得最佳性能。

**5.2 Leader election**

Raft使用心跳机制触发领袖选举。当服务器启动时，它们以追随者的身份开始。只要接收到有效消息，服务器就保持追随者状态

5

来自领导人或候选人的rpc。领导者定期向所有追随者发送心跳(不携带日志条目的追加条目rpc)，以维护他们的权威。如果一个follower在一段叫做*选举超时的时间内没有收到任何通信*，那么它就会假设没有可见的leader，并开始选举一个新的leader。

为了开始选举，追随者增加其当前任期并过渡到候选人状态。然后，它为自己投票，并并行地向集群中的每个其他服务器发出RequestVote rpc。候选人继续处于这种状态，直到发生以下三种情况之一:(A)它赢得了选举，(b)另一个服务器确立了自己的领导地位，或者(c)一段时间过去了，没有赢家。这些结果将在下面的段落中单独讨论。

如果候选人在同一任期内获得整个集群中大多数服务器的投票，则该候选人赢得选举。在给定的任期内，每个服务器将以先到先得的方式投票给最多一个候选人(注意:第5.4节增加了对投票的额外限制)。多数决规则确保最多有一名候选人可以赢得特定任期的选举(图3中的选举安全属性)。一旦候选人赢得选举，它就成为领导者。然后，它向所有其他服务器发送心跳消息，以建立其权威并防止新的选举。

在等待投票时，候选人可能会从另一个声称是领导者的服务器收到追加条目 RPC。如果领导者的任期(包含在其RPC中)至少与候选人的当前任期一样长，则候选人将承认领导者是合法的，并返回到追随者状态。如果RPC中的期限小于候选者的当前期限，则候选者拒绝RPC并继续处于候选者状态。

第三种可能的结果是候选人既没有赢得也没有输掉选举:如果许多追随者同时成为候选人，选票可以被分割，这样没有候选人获得多数。当这种情况发生时，每个候选人将超时并通过增加其任期并启动另一轮请求-投票rpc来开始新的选举。然而，如果没有额外的措施，分裂投票可能会无限期地重复。

Raft使用随机选举超时来确保分裂投票的情况很少发生，并且能够迅速得到解决。为了从一开始就防止分裂投票，选举超时从固定的间隔(例如150-300ms)中随机选择。这样分散了服务器，所以在大多数情况下只有一个服务器会超时;它赢得了选举，并在任何其他服务器超时之前发送心跳。同样的机制也用于处理分裂投票。每个候选人在选举开始时重新启动其随机选举超时，并等待该超时结束后再开始下一次选举;这降低了在新的选举中再次出现分裂投票的可能性。第9.3节表明，这种方法可以快速选举出领导人。

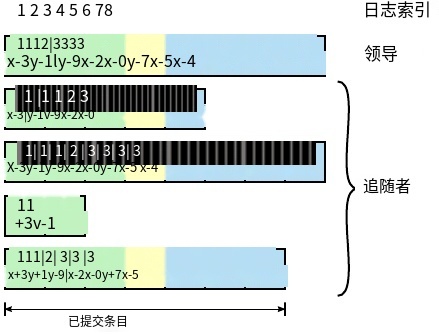


图6:日志由条目组成，按顺序编号。每个条目包含创建它的术语(每个框中的数字)和状态机的命令。如果*将一个条目安全地应用于状态机，则认为该条目已提交*。

选举是一个例子，说明了可理解性如何指导我们在设计方案之间进行选择。最初我们计划使用排名系统:每个候选人被分配一个唯一的排名，用于在竞争候选人之间进行选择。如果一个候选人发现了另一个排名更高的候选人，它就会回到追随者状态，这样排名更高的候选人就能更容易地赢得下一次选举。我们发现，这种方法在可用性方面产生了微妙的问题(如果排名较高的服务器失败，排名较低的服务器可能需要超时并再次成为候选人，但如果它这样做得太快，它可以重置选举领导者的进度)。我们对算法进行了几次调整，但每次调整后都会出现新的边缘情况。最终我们得出结论，随机重试的方法更明显，也更容易理解。

**5.3 Log replication**

一旦选出领导者，它就开始服务客户端请求。每个客户端请求都包含一个要被复制的状态机执行的命令。leader将命令作为一个新条目追加到它的日志中，然后将AppendEntries rpc并行地分发给其他每个服务器以复制该条目。当条目被安全复制(如下所述)时，leader将该条目应用于其状态机并将执行结果返回给客户端。如果follower崩溃或运行缓慢，或者如果网络数据包丢失，leader会无限期地重试追加条目rpc(即使在它对客户端做出响应之后)，直到所有follower最终存储所有日志条目。

日志的组织方式如图6所示。每个日志记录存储一个状态机命令以及leader接收条目时的术语号。日志条目中的术语号用于检测日志之间的不一致性，并确保图3中的一些属性。每个日志条目也有一个整数索引标识-

6

确定其在日志中的位置。

领导者决定何时对状态机应用日志记录是安全的;这样的条目被称为*提交。*Raft保证提交的条目是持久的，最终会被所有可用的状态机执行。一旦创建条目的leader在大多数服务器上复制了该条目(例如，图6中的条目7)，日志条目就会被提交。这也会提交leader日志中所有之前的条目，包括之前的leader创建的条目。第5.4节讨论了在leader更改后应用此规则时的一些微妙之处，并且还表明此承诺的定义是安全的。leader跟踪它知道要提交的最高索引，并将该索引包含在未来的AppendEntries rpc中(包括心跳)，以便其他服务器最终发现。一旦追随者得知一个日志条目被提交，它将该条目应用到其本地状态机(按日志顺序)。

我们设计了Raft日志机制，以保持不同服务器上的日志之间的高度一致性。这不仅简化了系统的行为，使其更具可预测性，而且是确保安全的重要组成部分。Raft维护以下属性，它们共同构成图3中的日志匹配属性:

•如果不同日志中的两个条目具有相同的索引和term，则它们存储相同的命令。

•如果不同日志中的两个条目具有相同的索引和期限，则上述所有条目的日志都是相同的。

第一个属性源于这样一个事实:leader在给定的期限内最多创建一个具有给定日志索引的条目，并且日志条目永远不会改变它们在日志中的位置。第二个属性由AppendEntries执行的简单一致性检查保证。当发送AppendEntries RPC时，leader在其日志中包含条目的索引和期限，该条目立即位于新条目之前。如果follower在其日志中没有找到具有相同索引和期限的条目，则拒绝新条目。一致性检查作为一个归纳步骤:日志的初始空状态满足日志匹配属性，无论何时扩展日志，一致性检查都保持日志匹配属性。因此，每当AppendEntries成功返回时，leader就知道follower的日志与它自己的日志相同。

正常运行时，leader和follower的日志保持一致，因此AppendEntries一致性检查不会失败。然而，leader崩溃可能会导致日志不一致(旧leader可能没有完全复制其日志中的所有条目)。这些不一致可能会在一系列的领导者和追随者崩溃中加剧。图7说明了追随者的日志可能与新领导者的日志不同的方式。追随者可能

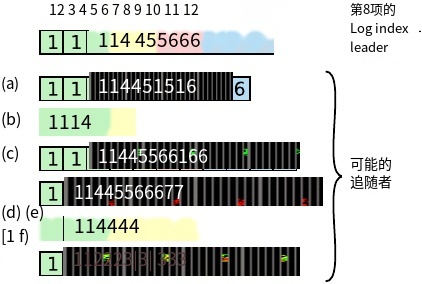


图7:当高层领导掌权时，追随者日志中有可能出现任何一种情况(a-f)。每个方框代表一个日志条目;方框里的数字是它的期限。追随者可能缺少条目(A - b)，可能有额外的未提交条目(c-d)，或者两者都有(e-f)。例如，场景(f)可能发生，如果该服务器是第2项的领导者，在其日志中添加了几个条目，然后在提交任何条目之前崩溃;它迅速重启，成为term 3的leader，并在其日志中添加了更多的条目;在term 2或term 3的任何条目提交之前，服务器再次崩溃，并且保持了几个term的停机状态。

如果缺少leader上存在的条目，它可能有leader上不存在的额外条目，或者两者都有。日志中缺失的和多余的条目可能跨越多个项。

在Raft中，领导者通过强迫追随者的日志复制自己的日志来处理不一致。这意味着追随者日志中冲突的条目将被领导者日志中的条目覆盖。第5.4节将说明，如果再加上一个限制，这是安全的。

为了使追随者的日志与自己的日志保持一致，领导者必须找到两个日志一致的最新日志条目，删除追随者日志中在该点之后的任何条目，并将领导者在该点之后的所有条目发送给追随者。所有这些操作都是对AppendEntries rpc执行的一致性检查的响应。leader为每个follower维护一个*nextIndex*，这是leader将发送给该follower的下一个日志条目的索引。当leader首次掌权时，它将所有nextIndex值初始化为日志中最后一个值之后的索引(图7中的11)。如果follower的日志与leader的日志不一致，AppendEntries一致性检查将在下一个AppendEntries RPC中失败。拒绝后，leader减少nextIndex并重试AppendEntries RPC。最终，nextIndex将达到领导者和追随者日志匹配的点。当这种情况发生时，AppendEntries将成功，它将删除follower日志中的任何冲突条目，并从leader日志(如果有的话)中追加条目。一旦AppendEntries成功，跟随者的日志与领导者的日志一致，并且在学期的剩余时间内保持这种状态。

如果需要，可以对协议进行优化，以减少被拒绝的AppendEntries rpc的数量。例如，当拒绝一个AppendEntries请求时，追随者

7

可以包括冲突项的项和它为该项存储的第一个索引。有了这些信息，leader就可以对nextIndex进行减量以绕过该项中所有冲突的表项;每个有冲突条目的术语需要一个AppendEntries RPC，而不是每个条目一个RPC。在实践中，我们怀疑这种优化是否必要，因为失败很少发生，并且不太可能有许多不一致的条目。

有了这个机制，leader在断电的时候不需要采取任何特殊的动作来恢复日志的一致性。它只是开始正常运行，当追加条目一致性检查失败时，日志会自动收敛。leader永远不会覆盖或删除自己日志中的条目(图3中的leader追加- only属性)。

这种日志复制机制展示了第2节中描述的理想的共识属性:只要大多数服务器正常运行，Raft就可以ac-cept，复制和应用新的日志条目;在正常情况下，一个新条目可以通过一轮rpc复制到集群的大多数;而单个缓慢的跟随者不会影响性能。

**5.4 Safety**

前面的章节描述了Raft如何选择leader和复制日志条目。然而，到目前为止描述的机制还不足以确保每个状态机以相同的顺序执行完全相同的命令。例如，当leader提交多个日志条目时，follower可能不可用，那么它可以被选为leader并用新的条目覆盖这些条目;因此，不同的状态机可能会执行不同的命令序列。

本节通过添加服务器可能被选为leader的限制来完成Raft算法。该限制确保任何给定任期的leader包含在前一任期中提交的所有条目(图3中的leader完整性属性)。考虑到选举限制，我们将使提交规则更加精确。最后，我们给出了leader完整性属性的证明草图，并展示了它如何导致复制状态机的正确行为。

5.4.1 选举限制

在任何基于领导者的共识算法中，领导者最终必须存储所有已提交的日志条目。在一些共识算法中，例如Viewstamped replication -cation[22]，即使leader最初没有包含所有提交的日志项，也可以被选举出来。这些算法包含额外的机制来识别缺失的条目并将其传输给新的领导者，无论是在选举过程中还是之后不久。不幸的是，这导致了相当多的额外机制和复杂性。Raft使用了一种更简单的方法，它保证所有来自先前的提交条目

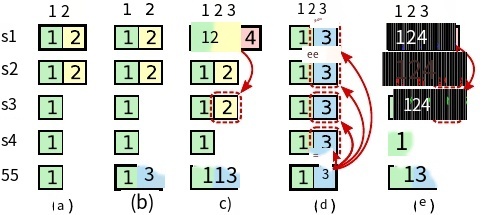


图8:一个时间序列，显示了为什么领导者不能使用旧术语的日志条目来确定承诺。在(a)中，S1是leader，部分复制了索引2处的日志条目。在(b) S1崩溃;S5通过S3、S4和自身的投票被选为第3期的leader，并在log index 2处接受不同的条目。在(c)中，S5崩溃;S1重启，被选为leader，并继续复制。此时，term 2的日志条目已经在大多数服务器上进行了复制，但没有提交。如果S1像(d)中那样崩溃，S5可以被选为leader(来自S2、S3和S4的投票)，并用它自己来自term 3的条目覆盖该条目。然而，如果S1在崩溃前在大多数服务器上复制了当前任期的条目，如(e)，则该条目被提交(S5无法赢得选举)。此时，日志中所有之前的条目也被提交。

从每一个新领导人当选的那一刻起，术语就存在于其上，而无需将这些条目转移到领导人。这意味着日志条目只在一个方向上流动，从领导者到追随者，领导者永远不会覆盖其日志中的现有条目。

Raft使用投票过程来防止候选人赢得选举，除非其日志包含所有已提交的条目。候选人必须联系集群的大多数才能当选，这意味着每个提交的条目必须至少存在于其中一个服务器中。如果候选人的日志至少与该多数服务器中的任何其他日志一样最新(其中“最新”的定义在下面)，那么它将保存所有已提交的条目。RequestVote RPC实现了这个限制:RPC包含关于候选人日志的信息，如果投票人自己的日志比候选人的日志更新，投票人就会拒绝投票。

Raft通过比较日志中最后一个条目的索引和期限来确定两个日志中哪一个是最新的。如果日志的最后条目具有不同的术语，则具有较晚术语的日志更具有最新性。如果日志以相同的期限结束，那么哪个日志越长，哪个日志就越最新。

5.4.2 提交前一项的条目

如5.3节所述，leader知道当前term的条目一旦存储在大多数服务器上，就会被提交。如果leader在提交条目之前崩溃，未来的leader将尝试完成复制该条目。然而，leader不能立即得出结论，认为上一个学期的条目一旦存储在大多数服务器上就被提交了。图-

8

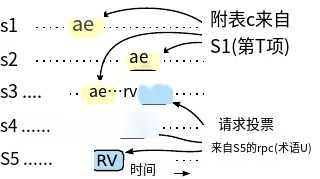


图9:如果S1(任期T的领导者)从其任期提交了一个新的日志条目，并且S5被选为下一个任期U的领导者，那么必须至少有一个服务器(S3)接受该日志条目并投票给S5。

图8说明了一种情况，旧的日志条目存储在大多数服务器上，但仍然可以被未来的leader覆盖。

为了消除图8中出现的问题，Raft从不通过计数副本来提交以前条目的日志条目。只有leader当前任期的日志条目才会通过计数副本提交;一旦以这种方式提交了当前项中的一个条目，那么由于日志匹配属性，所有先前的条目都将间接提交。在某些情况下，leader可以安全地得出一个较旧的日志条目被提交的结论(例如，如果该条目存储在每个服务器上)，但是Raft为了简单起见采取了更保守的方法。

Raft在提交规则中引入了这种额外的复杂性，因为当leader复制以前的条目时，日志条目会保留其原始的条目编号。在其他共识算法中，如果一个新的leader从先前的“term”中重新复制条目，它必须用新的“term number”这样做。Raft的方法使得对日志条目的推理变得更容易，因为它们随着时间的推移和跨日志保持相同的术语数。此外，Raft中的新领导者发送的以前术语的日志条目比其他算法要少(其他算法必须发送冗余的日志条目来重新编号，然后才能提交)。

**5.4.3 Safety argument**

给定完整的Raft算法，我们现在可以更精确地论证前导完备性属性成立(该论证基于安全性证明;参见第9.2节)。我们假设前导完备性不成立，然后证明一个矛盾。假设项T(leaderT)的leader从它的term中提交了一个日志条目，但是该日志条目没有被某个未来term的leader存储。考虑最小的项U > T，它的leader(leaderU)没有存储该条目。

1. 承诺的条目必须在其选举时从leaderU’s日志中缺席(领导者永远不会删除或覆盖条目)。

2. leaderT复制了大多数集群上的条目，leaderU收到了大多数集群的投票。因此，至少有一个服务器(“投票人”)同时接受了来自leaderT的条目并投票

leaderU，如图9所示。选民是达成矛盾的关键。

3. 投票人必须*在投票给*leaderU;*之前*接受来自leaderT的承诺条目，否则它将拒绝来自leaderT的AppendEntries请求(其当前期限将高于T)。

4. 投票人在投票给leaderU时仍然存储了该条目，因为每个介入的领导者都包含该条目(假设)，领导者永远不会删除条目，而追随者只在与领导者冲突时删除条目。

5. 投票人把选票投给了leaderU，所以leaderU’s的日志必须和投票人的日志一样是最新的。这就导致了两个矛盾之一。

6. 首先，如果选民和leaderU共享相同的最后一个日志项，那么leaderU’s的日志必须至少和选民的日志一样长，所以它的日志包含选民日志中的每一个条目。这是一个矛盾，因为投票者包含了提交的条目，而leaderU被假定不包含。

7. 否则，leaderU’s上一个对数项一定大于选民的对数项。而且，它比T大，因为投票人的最后一个日志项至少是T(它包含了来自项T的承诺条目)，创建leaderU’s最后一个日志项的早期领导者必须在其日志中包含承诺条目(通过假设)。然后，根据日志匹配属性，leaderU’s日志也必须包含提交的条目，这是一个矛盾。

8. 这就完成了矛盾。因此，所有大于T的项的前导必须包含在T项中提交的所有来自T项的条目。

9. 日志匹配属性保证未来的leader也将包含间接提交的条目，如图8(d)中的索引2。

给定Leader完整性属性，我们可以证明图3中的状态机安全属性，该属性指出，如果一个服务器在其状态机的给定索引上应用了一个日志条目，那么其他服务器将不会为相同的索引应用不同的日志条目。当服务器向其状态机应用日志条目时，它的日志必须与该条目之前的leader日志相同，并且该条目必须被提交。现在考虑任何服务器应用给定日志索引的最低期限;日志完整性属性保证所有较高项的前导项将存储相同的日志条目，因此在较低项中应用索引的服务器将应用相同的值。因此，状态机安全属性保持不变。

最后，Raft要求服务器按照log - index顺序应用条目。结合状态机安全属性，这意味着所有服务器将以相同的顺序向其状态机应用完全相同的日志条目集。

9

5.5 Follower和candidate崩溃

在此之前，我们一直关注领导者的失败。folo -lower和candidate崩溃比leader崩溃处理起来要简单得多，它们的处理方式都是一样的。如果一个follower或candidate崩溃，那么未来发送给它的RequestVote和AppendEntries rpc将失败。Raft通过无限重试来处理这些失败;如果崩溃的服务器重新启动，那么RPC将成功完成。如果服务器在完成RPC后但在响应之前崩溃，那么它将在重新启动后再次接收相同的RPC。Raft rpc是幂等的，所以这不会造成伤害。例如，如果follower接收到一个AppendEntries请求，该请求包含其日志中已经存在的日志条目，那么它将忽略新请求中的这些条目。

5.6 时间和可用性

我们对Raft的一个要求是，安全性不能依赖于时间:系统不能仅仅因为某些事件发生得比预期的快或慢而产生不正确的结果。然而，可用性(系统及时响应客户端的能力)必须不可避免地依赖于时间。例如，如果mes-sage交换的时间比服务器崩溃之间的典型时间更长，候选人就不会熬夜到足以赢得选举;如果没有一个稳定的领导者，Raft就无法取得进展。

领袖选举是Raft中时机最关键的一个方面。只要系统满足以下时间*要求，Raft就能选出并维持一个稳定的领袖:*

*广播时间*≪选举时间timeout≪MTBF

在这个*不等式中，广播时间*是服务器向集群中的每个服务器并行发送rpc并接收它们的响应所需的平均时间;*electionTime-out*是5.2节中描述的选举超时;*MTBF*是单个服务器的平均故障间隔时间。广播时间应该比选举超时时间少一个数量级，这样领导者才能可靠地发送心跳消息，以阻止追随者开始选举;考虑到选举超时使用的随机化方法，这种不平等也使得分裂投票不太可能发生。选举超时应该比MTBF小几个数量级，这样系统才能稳步前进。当leader崩溃时，系统将在大致的选举超时时间内不可用;我们希望这只代表整体时间的一小部分。

广播时间和MTBF是底层系统的属性，而选举超时是我们必须选择的。Raft的rpc通常要求接收方将信息持久化到稳定的存储中，因此广播时间可能在0.5ms到20ms之间，具体取决于存储技术。因此，选举超时时间很可能在10ms到500ms之间。典型

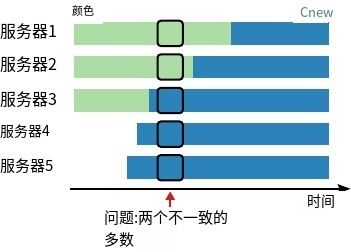


图10:直接从一种配置切换到另一种配置是不安全的，因为不同的服务器会在不同的时间切换。在这个例子中，集群从3台服务器增长到5台。不幸的是，在一个时间点上，两个不同的领导者可以在同一任期内当选，一个是大多数旧配置(Cold)，另一个是大多数新配置(Cnew)。

服务器mtbf是几个月或更长时间，这很容易满足时间要求。

6 集群成员变更

到目前为止，我们已经假设*集群配置(*参与共识算法的服务器集)是固定的。在实践中，偶尔需要更改配置，例如在服务器出现故障时更换服务器或更改复制程度。虽然这可以通过使整个集群脱机，更新配置文件，然后重新启动集群来完成，但这将使集群在切换期间不可用。此外，如果有任何手动步骤，它们可能会导致操作人员出错。为了避免这些is- issues，我们决定将配置更改自动化，并将其纳入Raft共识算法中。

为了保证配置更改机制的安全性，在过渡期间必须不存在两个领导人可能在同一任期内当选的情况。不幸的是，任何让服务器直接从旧配置切换到新配置的方法都是不安全的。一次自动切换所有服务器是不可能的，因此集群在转换期间可能会分裂成两个独立的多数(参见图10)。

为了确保安全，配置更改必须使用两阶段的方法。实现这两个阶段的方法有很多种。例如，一些系统(例如[22])使用第一阶段禁用旧配置，使其无法处理客户端请求;然后第二阶段启用新配置。在Raft中，集群首先切换到我们称之为*联合共识的过渡配置;*一旦联合共识被提交，系统就会过渡到新的配置。联合共识结合了旧配置和新配置:

•日志条目被复制到两种配置中的所有服务器。

10

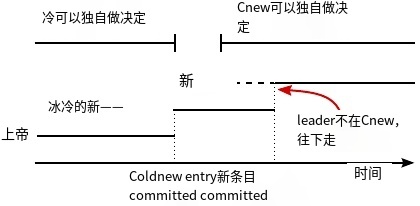


图11:配置更改的时间轴。虚线显示已创建但未提交的配置项，实线显示最新提交的配置项。leader首先在其日志中创建*Cold,new*配置条目，并将其提交给*Cold,new* (majority of *Cold*和majority of *Cnew)。*然后，它创建*Cnew*条目并将其提交给大多数*Cnew。*在任何时间点上，柯尔德和*Cnew*都不可能同时独立做出决定。

•任一配置中的任何服务器都可以担任领导者。

•协议(用于选举和进入承诺)需要从旧配置和新配置中分别获得多数。

联合共识允许单个服务器在不同时间在配置之间转换，而没有共同承诺的安全性。此外，联合共识允许集群在整个配置更改期间继续为客户端请求提供服务。

集群配置使用复制日志中的特殊条目进行存储和通信;图11展示了配置更改过程。当领导者收到将配置从*C*old更改为*C*new的请求时，它将联合共识的配置((*C*old,new在图中)存储为日志条目，并使用前面描述的机制复制该条目。一旦给定的服务器将新的配置条目添加到其日志中，它将在所有未来的决策中使用该配置(服务器总是在其日志中使用最新的配置，而不管该条目是否已提交)。这意味着领导者将使用*C*old,new的规则来确定*C*old,new的日志条目何时被提交。如果leader崩溃，可能会在*Cold或Cold*,new下选择一个新的leader，这取决于获胜的候选人是否收到了*Cold,new。*无论如何，*Cnew*不能在此期间单方面作出决定。

一旦cecold、new被提交，任何一方都不能在没有对方批准的情况下做出决策，并且Leader完全性属性保证了只有拥有*C*old,new日志条目的服务器才能被选为Leader。现在leader可以安全地创建一个描述*C*new的日志条目并将其复制到集群中。同样，此配置将在看到后立即在每个服务器上生效。当在*C*new规则下提交了新配置后，旧配置就无关紧要了，不在新配置中的服务器可以被关闭。如图11所示，不存在*C*old和*C*new都能做出单边决策的时间;这就保证了安全性。

重新配置还有三个问题需要解决。第一个问题是新服务器最初可能不存储任何日志条目。如果它们以这种状态添加到集群中，它们可能需要相当长的一段时间才能赶上，在此期间可能无法提交新的日志条目。为了避免可用性差距，Raft在配置更改之前引入了一个额外的阶段，在这个阶段中，新服务器作为无投票成员加入集群(leader向它们复制日志条目，但它们不被认为是majority)。一旦新服务器赶上了集群的其他服务器，重新配置就可以按照上面描述的方式进行。

第二个问题是集群领导者可能不是新配置的一部分。在这种情况下，leader一旦提交了*C*new日志条目，就会退出(返回到follower状态)。这意味着会有一段时间(虽然是committing*C*new)，当leader正在管理一个不包括它自己的集群时;它复制日志条目，但不把自己算作多数。leader转换发生在*C*new被提交的时候，因为这是新配置可以独立运行的第一个点(总是可以从*C*new)中选择leader)。在这一点之前，可能只有来自*C*old的一个服务器可以被选为leader。

第三个问题是，被移除的服务器(那些不在*C*new)中的服务器)可能会扰乱集群。这些服务器将不会接收到心跳，因此它们将超时并开始新的选举。然后，它们将发送带有新术语号的RequestVote rpc，这将导致当前的领导者恢复到追随者状态。新的领导者最终会被选举出来，但是被移除的服务器会再次超时，这个过程会重复，导致可用性差。

为了防止这个问题，当服务器认为当前的leader存在时，会忽略RequestVote rpc。具体来说，如果服务器在当前leader的最小选举超时时间内收到RequestVote RPC，则它不会更新其任期或授予其投票。这不会影响正常的选举，其中每个服务器在开始选举之前至少等待最小的选举超时。然而，它有助于避免被移除的服务器造成的中断:如果一个领导者能够将心跳发送到其集群，那么它就不会被更大的term - number所推翻。

**7 Log compaction**

Raft的日志在正常运行期间会增长，以纳入更多的客户端请求，但在实际系统中，它无法无限制地增长。随着日志的增长，它会占用更多的空间，并且需要更多的时间来重放。如果没有某种机制来丢弃日志中积累的过时信息，这最终会导致可用性问题。

快照是最简单的压缩方法。在快照中，将整个当前系统状态写入稳定存储上的*快照*，然后将整个日志写入

11

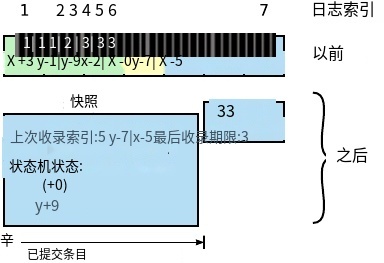


图12:服务器用一个新的快照替换其日志中的已提交条目(索引1到5)，该快照只存储当前状态(本例中的变量*x*和*y*)。快照最后包含的索引和术语用于将快照定位在条目6之前的日志中。

这个点将被丢弃。快照在Chubby和ZooKeeper中使用，本节的其余部分将描述Raft中的快照。

压缩的增量方法，如日志清理[36]和日志结构合并树[30,5]，也是可能的。这些方法一次对一小部分数据进行操作，因此它们随着时间的推移更均匀地分散了压缩的负载。它们首先选择一个累积了许多已删除和覆盖对象的数据区域，然后更紧凑地重写该区域的活对象，并释放该区域。与快照相比，这需要显著的附加机制和复杂性，快照通过始终对整个数据集进行操作来简化问题。虽然日志清理需要对Raft进行修改，但状态机可以使用与快照相同的接口实现LSM树。

图12展示了Raft中快照的基本思想。每个服务器独立地进行快照，只覆盖其日志中提交的条目。大部分工作包括状态机将其当前状态写入快照。Raft还在快照中包含了少量的元数据:*最后*一个*包含*的*索引*是快照替换的日志中最后一个条目的索引(状态机已经应用的最后一个条目)，*最后*一个*包含*的*term*是这个条目的term。保留这些是为了支持对快照之后的第一个日志条目的AppendEntries一致性检查，因为该条目需要一个先前的日志索引和期限。为了启用集群成员变更(第6节)，快照还包括截至上次包含索引的日志中的最新配置。一旦服务器完成写入快照，它可能会删除所有日志条目，直到最后包含的索引，以及任何先前的快照。

虽然服务器通常是独立地拍摄快照，但leader必须偶尔将快照发送给落后的follower。这种情况发生在leader已经丢弃了它需要发送给follower的下一个日志条目的时候。幸运的是，这种情况在正常操作中是不太可能发生的:一个跟随者已经跟上了

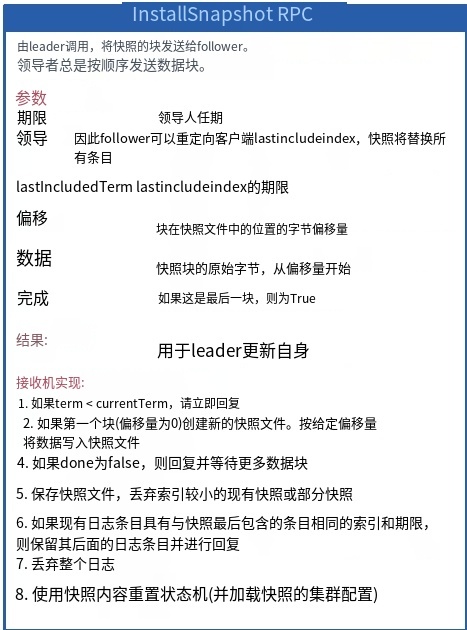


图13:InstallSnapshot RPC的摘要。快照被分成块进行传输;这让follow -lower在每个chunk上都有一个生命的迹象，这样它就可以重置它的选举计时器。

Leader应该已经有这个条目了。但是，异常缓慢的追随者或新服务器加入集群(第6节)则不会。让这样一个追随者保持最新状态的方法是，领导者通过网络向它发送快照。

leader使用一个名为InstallSnapshot的新RPC向远远落后的follower发送快照;参见图13。当跟踪者接收到带有此RPC的快照时，它必须决定如何处理其现有的日志条目。通常，快照将包含接收者日志中尚未包含的新信息。在这种情况下，关注者会丢弃它的整个日志;它全部被快照取代，并且可能有与快照冲突的未提交条目。如果跟随者收到一个描述其日志前缀的快照(由于重传或错误)，则快照所覆盖的日志条目被删除，但快照后面的条目仍然有效，必须保留。

这种快照方法违背了Raft的强leader原则，因为follower可以在leader不知情的情况下进行快照。然而，我们认为这种背离是有道理的。虽然有一个领导者有助于在达成共识时避免相互冲突的决策，但在快照时已经达成了共识，所以没有决策冲突。数据仍然只会从领导者流向傻瓜

12

追随者现在可以重新组织他们的数据。

我们考虑了另一种基于领导者的方法，其中只有领导者会创建快照，然后它会将这个快照发送给每个追随者。然而，这种方法有两个缺点。首先，将快照发送给每个关注者会浪费网络带宽并减慢快照进程。每个关注者已经拥有了生成自己的快照所需的信息，对于服务器来说，从其本地状态生成快照通常比通过网络发送和接收快照要便宜得多。其次，leader的实现会更加复杂。例如，leader需要在向follower复制新的日志条目的同时向其发送快照，这样才不会阻塞新的客户端请求。

还有两个问题会影响快照的性能。首先，服务器必须决定何时进行快照。如果服务器快照太频繁，就会浪费磁盘带宽和能量;如果快照频率过低，则有耗尽存储容量的风险，并且会增加重启时重播日志所需的时间。一个简单的策略是，当日志达到固定大小(以字节为单位)时拍摄快照。如果将这个大小设置为明显大于快照的预期大小，那么用于快照的磁盘带宽开销就会很小。

第二个性能问题是，写快照可能会花费相当多的时间，我们不希望这会延迟正常的操作。解决方案是使用写时复制(copy-on-write)技术，这样就可以接受新的更新，而不会影响正在写的快照。例如，用功能数据结构构建的状态机自然就支持这一点。或者，操作系统的写时复制(copy-on-write)支持(例如Linux上的fork)可以用来创建整个状态机的内存快照(我们的实现使用了这种方法)。

8 客户端交互

本节描述客户端如何与Raft交互，包括客户端如何查找集群leader以及Raft如何支持线性化语义[10]。这些问题适用于所有基于共识的系统，Raft的解决方案与其他系统类似。

Raft的客户端会将所有的请求发送给领导者。当客户端第一次启动时，它会连接到一个随机选择的服务器。如果客户端的第一选择不是leader，该服务器将拒绝客户端的请求，并提供最近收到的leader的信息(AppendEntries请求包括leader的网络地址)。如果leader崩溃，客户端请求将超时;然后客户端用随机选择的服务器再次尝试。

我们对Raft的目标是实现线性化语义(每个操作看起来都是瞬间执行的，正好一次，在它的调用和响应之间的某个点上)。然而，正如目前所描述的，Raft可以多次执行命令:例如，如果l

eader在提交日志条目后崩溃，但在响应客户端之前，客户端将用新的leader重试命令，导致它被执行第二次。解决方案是让客户端为每条命令分配唯一的序列号。然后，状态机跟踪为每个客户端处理的最新序列号，以及相关的响应。如果它接收到一个序列号已经执行过的命令，它会立即响应，而不会重新执行请求。

只读操作可以在不向日志中写入任何内容的情况下处理。然而，如果没有额外的措施，这将冒着返回陈旧数据的风险，因为响应请求的leader可能已经被一个它不知道的新leader所取代。Lin-earizable读取一定不能返回过时的数据，Raft需要两个额外的预防措施来保证这一点，而不使用日志。首先，leader必须拥有关于哪些条目被提交的最新信息。Leader完整性属性保证Leader拥有所有提交的条目，但在其任期开始时，它可能不知道哪些是提交的条目。为了找出答案，它需要提交其任期内的一个条目。Raft通过让每个leader在其任期开始时提交一个空白的*无操作*条目到日志中来处理这个问题。其次，leader必须在处理只读请求之前检查它是否已经被删除(如果一个最近的leader被选举出来，它的信息可能是陈旧的)。Raft通过让leader在响应只读请求之前与集群的大多数节点交换心跳消息来处理这个问题。或者，leader可以依靠心跳机制来提供一种形式的租约[9]，但这将依赖于安全的定时(它假设有界时钟倾斜)。

9 实现和评估

我们已经将Raft作为复制状态机的一部分实现，该状态机存储RAMCloud[33]的配置信息，并协助RAMCloud协调器的故障转移。Raft实现包含大约2000行c++代码，不包括测试、注释或空白行。源代码可以在[23]免费获得。基于本文的草稿，Raft还有大约25个独立的第三方开源实现[34]处于不同的开发阶段。此外，各种公司正在部署基于raft的系统[34]。

本节的其余部分使用三个标准来评估Raft:可理解性、正确性和性能。

9.1 可理解性

为了衡量Raft相对于Paxos的可理解性，我们对斯坦福大学高级操作系统课程和加州大学伯克利分校分布式计算课程的高年级本科生和研究生进行了一项实验研究。我们录制了Raft和Paxos的视频讲座，并制作了相应的小测验。Raft讲座除了日志压缩之外，涵盖了本文的内容;Paxos

13

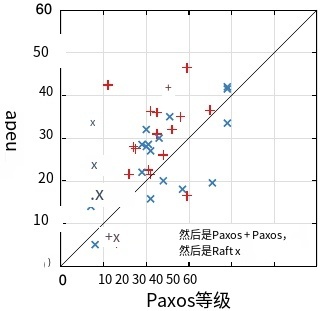


图14:散点图比较43名参与者在Raft和Paxos测试中的表现。对角线(33)上方的点代表在Raft测试中得分较高的参与者。

讲座涵盖了足够的材料来创建一个等效的复制状态机，包括单命令Paxos、多命令Paxos、重新配置和实践中需要的一些优化(例如领导者选举)。这些小测验测试了学生对算法的基本理解，也要求他们对极端情况进行推理。每个学生看一个视频，做相应的小测验，看第二个视频，再做第二个小测验。大约一半的参与者先做了Paxos部分，另一半先做了Raft部分，以解释个人在表现和从第一部分研究中获得的经验上的差异。我们比较了参与者在每次测验中的得分，以确定参与者是否对Raft有更好的理解。

我们试图在Paxos和Raft之间进行尽可能公平的比较。实验在两个方面对Paxos有利:43名参与者中有15人报告说他们之前有过Paxos的一些经验，Paxos视频比Raft视频长14%。如表1所示，我们已经采取措施来减轻潜在的偏见来源。我们所有的资料都可供查阅[28,31]。

平均而言，参与者在Raft测试中的得分比在Paxos测试中的得分高4.9分(在可能的60分中，Raft的平均得分为25.7分，Paxos的平均得分为20.8分);图14显示了他们的个人得分。*配对t检验*表明，在95%的置信度下，Raft分数的真实分布均值至少比Paxos分数的真实分布均值大2.5点。

我们还创建了一个线性回归模型，可以根据三个因素预测新学生的测验分数:他们参加的测验，他们之前的Paxos经验程度，以及

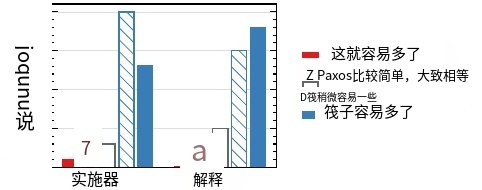


图15:使用5分制量表，参与者被问及(左)他们认为哪种算法更容易在一个功能正常、正确和高效的系统中实现，以及(右)哪种算法更容易向CS研究生解释。

他们学习算法的先后顺序。该模型预测，测验的选择会产生12.5分的差异，有利于Raft。这明显高于观察到的4.9分的差异，因为许多实际学生之前都有Paxos的经验，这对Paxos有很大帮助，而对Raft的帮助略小。奇怪的是，该模型还预测，已经参加过Paxos测试的人在Raft上的得分要低6.3分;虽然我们不知道原因，但这似乎确实具有统计学意义。

我们还在测试结束后对参与者进行了调查，看看他们觉得哪种算法更容易实现或解释;这些结果如图15所示。绝大多数参与者报告说Raft更容易实现和解释(每个问题41人中有33人)。然而，这些自我报告的感受可能不如参与者的测验分数可靠，并且参与者可能因为我们假设Raft更容易理解的知识而产生偏见。

关于Raft用户研究的详细讨论可以在[31]上找到。

**9.2 Correctness**

我们已经开发了第5节中描述的共识机制的正式规范和安全性证明。正式规范[31]使用TLA+规范语言[17]使图2中总结的信息完全精确。它大约有400行长，并作为证明的主体。对于任何实现Raft的人来说，它本身也很有用。我们用TLA证明系统[7]机械地证明了对数完备性。然而，这个证明依赖于没有经过机械检查的不变量(例如，我们没有证明规范的类型安全性)。此外，我们已经写了一个状态机安全属性的非正式证明[31]，它是完整的(它只依赖于规范)和rela-

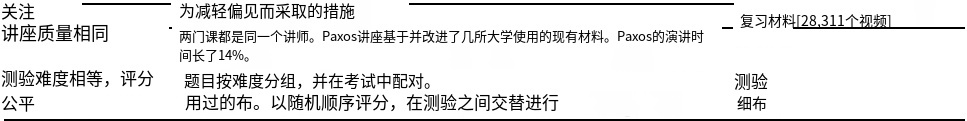


表1:研究中对Paxos可能存在的偏见的关注，采取的应对措施，以及可用的其他材料。

14

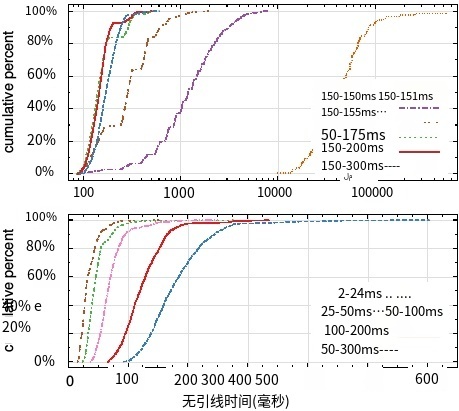


图16:检测和替换崩溃leader的时间。上面的图改变了选举超时的随机性，下面的图缩放了最小选举超时。每条线代表1000次试验(除了“150-150ms”的100次试验)，对应于一个特定的选举超时选择;例如，“150-155ms”表示选举超时随机且均匀地选择在150ms和155ms之间。测量是在一个由5台服务器组成的集群上进行的，广播时间大约为15ms。9台服务器集群的结果是相似的。

非常精确(大约3500字长)。

**9.3 Performance**

Raft的性能与其他共识算法(如Paxos)类似。性能最重要的情况是当一个已建立的领导者复制新的日志条目时。Raft使用最少数量的消息(从leader到半个集群的单次往返)来实现这一点。也有可能进一步提高Raft的性能。例如，它很容易支持批处理和流水线请求，以获得更高的吞吐量和更低的延迟。文献中已经针对其他算法提出了各种优化;其中许多可以应用于Raft，但我们将其留给未来的工作。

我们使用Raft实现来衡量Raft领导者选举算法的性能，并回答了两个问题。第一，选举过程收敛得快吗?第二，领袖崩溃后能达到的最小停机时间是多少?

为了衡量leader的选举，我们反复让一个由5台服务器组成的集群的leader崩溃，并计时检测到崩溃并选出新leader所需的时间(见图16)。为了生成最坏的情况，每次试验中的服务器都有不同的日志长度，因此有些候选人没有资格成为领导者。此外，为了鼓励分裂投票，我们的测试脚本在终止其进程之前触发了来自leader的心跳rpc的同步广播(这近似于leader在崩溃之前复制新日志条目的行为

)。leader在其心跳间隔内均匀随机崩溃，这是所有测试最小选举超时的一半。因此，最小可能的停机时间大约是最小选举超时的一半。

图16中最上面的图表显示，在选举超时中进行少量的随机化就足以避免选举中的分裂投票。在没有随机性的情况下，在我们的测试中，由于许多分裂选票，领导人选举持续花费超过10秒的时间。仅仅增加5ms的随机性就有很大帮助，导致停机时间的中位数为287ms。使用更多的随机性可以改善最坏情况的行为:在50ms的随机性下，最坏情况的完成时间(超过1000次试验)为513ms。

图16底部的图表显示，可以通过减少选举超时来减少停机时间。在选举超时为12-24ms的情况下，平均只需要35ms就能选出一个leader(最长的一次试验花费了152ms)。然而，将超时时间降低到超过这个点就违反了Raft的计时要求:leader很难在其他服务器开始新的选举之前广播心跳。这可能会导致不必要的leader变更，降低整个系统的可用性。我们建议使用保守的选举超时，比如150-300ms;这样的超时不太可能造成不必要的领导人更换，仍然会提供良好的可用性。

**10 Related work**

已经有许多与共识算法相关的出版物，其中许多属于以下类别之一:

•Lamport对Paxos[15]的原始描述，并试图更清楚地解释它[16,20,21]。

•Paxos的细化，填补缺失的细节，修改算法，为实现提供更好的基础[26,39,13]。

•实现共识算法的系统，如Chubby [2,4]， ZooKeeper[11,12]和Span-ner[6]。Chubby和Spanner的算法细节尚未公布，尽管它们都声称是基于Paxos的。ZooKeeper的算法已经发表了更详细的内容，但它与Paxos有很大的不同。

•可应用于Paxos的性能优化[18,19,3,25,1,27]。

•Oki和Liskov的Viewstamped Replication (VR)，这是一种与Paxos同时开发的共识替代方法。最初的描述[29]与分布式交易的协议交织在一起，但核心共识协议在最近的更新[22]中已经分离。VR使用了一种基于领导者的方法，这与Raft有许多相似之处。

Raft和Paxos之间最大的区别在于Raft的强大领导力:Raft将领导者选举作为共识协议的重要组成部分，并且它涉及到

15

在leader中尽可能多地赋值功能。这种方法的结果是一个更简单的算法，更容易理解。例如，在Paxos中，领导者选举与基本共识协议是或正交的:它仅作为性能优化，而不是实现共识所必需的。然而，这导致了额外的机制:Paxos既包括用于基本共识的两阶段协议，也包括用于领导者选举的单独机制。相比之下，Raft将领导人选举直接纳入共识算法，并将其作为共识两阶段中的第一阶段。这导致比Paxos中的机制更少。

与Raft一样，VR和ZooKeeper都是基于领导者的，因此与Paxos相比，Raft具有许多优势。然而，Raft的机制比VR或ZooKeeper少，因为它最小化了非领导者的功能。例如，Raft中的日志条目只向一个方向流动:从AppendEntries rpc中的leader向外流动。在VR中，日志条目是双向流动的(领导人可以在选举过程中接收日志条目);这就导致了额外的机制和复杂性。ZooKeeper发布的描述也将日志条目传输到leader或从leader传输日志条目，但其实现显然更像Raft[35]。

Raft的消息类型比我们所知道的任何其他基于共识的日志复制算法都要少。例如，我们计算了VR和ZooKeeper用于基本共识和成员变更的消息类型(不包括日志压缩和客户端交互，因为这些几乎独立于算法)。VR和ZooKeeper各自定义了10种不同的消息类型，而Raft只有4种消息类型(两个RPC请求和它们的响应)。Raft的消息比其他算法更密集一些，但它们总体上更简单。此外，VR和ZooKeeper在领导者更换期间传输整个日志;将需要额外的消息类型来优化这些机制，使其具有实用性。

Raft的强领导方法简化了算法，但它排除了一些性能优化。例如，Egalitarian Paxos (EPaxos)可以在某些条件下通过无引线方法实现更高的性能。EPaxos利用状态机命令中的交换性。任何服务器都可以通过一轮通信提交命令，只要其他被提议的命令与它并发地进行通信。但是，如果同时提出的命令不能相互交换，则EPaxos需要额外的一轮通信。由于任何服务器都可以提交命令，因此EPaxos可以很好地平衡服务器之间的负载，并且能够在WAN设置中实现比Raft更低的延迟。然而，它给Paxos增加了显著的复杂性。

在其他工作中已经提出或实现了几种不同的集群成员变更方法，包括兰波特的原始提案[15]、VR[22]和SMART[24]。我们为Raft选择了联合共识方法，因为它利用了共识-共识协议的其余部分，因此成员变更只需要很少的额外机制。兰波特的基于α的方法对Raft来说不是一个选择，因为它假设没有领导者也能达成共识。与VR和SMART相比，Raft的重新配置算法的优点是，成员变更可以在不限制正常请求处理的情况下发生;相比之下，VR在配置更改期间停止所有正常处理，SMART对未完成请求的数量施加了类似α的限制。Raft的方法也比VR或SMART添加了更少的机制。

11结论

算法的设计通常以正确性、效率和/或简洁性为主要目标。虽然这些都是值得追求的目标，但我们认为可理解性同样重要。在开发人员将算法呈现为实际实现之前，其他目标都无法实现，而实际实现将不可避免地偏离并扩展到已发布的形式。除非开发人员对算法有深刻的理解，并能创建关于它的直觉，否则他们很难在实现中保留其理想的属性。

在本文中，我们讨论了分布式共识问题，其中一个被广泛接受但难以理解的算法Paxos多年来一直挑战着学生和开发人员。我们开发了一种新的算法Raft，我们已经证明它比Paxos更容易理解。我们也相信Raft为系统构建提供了更好的基础。将可理解性作为主要的设计目标改变了我们处理Raft设计的方式;随着设计的进行，我们发现自己反复使用了一些技术，比如分解问题和简化状态空间。这些技术不仅提高了Raft的可理解性，也让我们更容易相信它的正确性。

12致谢

如果没有Ali Ghodsi、David Mazi的同事以及伯克利CS 294-91和斯坦福CS 240的学生的支持，这项用户研究是不可能完成的。斯科特·克莱默(Scott Klemmer)帮助我们设计了用户研究，尼尔森·雷(Nelson Ray)在统计分析方面给了我们建议。用于用户研究的Paxos幻灯片大量借鉴了Lorenzo Alvisi最初创建的幻灯片。特别感谢David Mazi的eres和Ezra Hoch在Raft中发现了微妙的漏洞。许多人对论文和用户研究材料提供了有益的反馈，包括Ed Bugnion, Michael Chan, Hugues Evrard，

16

Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, robert van Renesse, Mendel Rosenblum, Nico-las Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia，24位匿名会议审稿人(有副本)，尤其是我们的牧童Eddie Kohler。沃纳·沃格尔斯(Werner Vogels)在推特上发了一个早期草稿的链接，这给了Raft很大的曝光机会。这项工作得到了Gigascale Systems -tems研究中心和多尺度系统中心的支持，这两个研究中心是由半导体研究公司计划的focus中心研究计划资助的六个研究中心中的两个，由MARCO和DARPA赞助的半导体研究公司计划STARnet提供支持，得到了美国国家科学基金会0963859号拨款的支持，并得到了Facebook、b谷歌、Mel-lanox、NEC、NetApp、SAP和三星的资助。Diego Ongaro由Junglee公司斯坦福大学研究生奖学金资助。

参考文献

[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. M´ery, 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., AND CHERITON, D. Leases: An efficient fault- 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. http://github.com/logcabin/logcabin.

17

[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’N EIL, 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., ANDCHANSLER, 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.

18