**In Search of an Understandable Consensus Algorithm**

**摘要**

Raft是管理副本日志的共识算法。它提供了和Paxos相同的功能，并且与Paxos一样高效，但是它的结构与Paxos有一些不同；这使得Raft更容易理解也为构建实际系统提供了更好的基础。为了增强可理解性，Raft将共识的关键分为几块，分别是leader选举，log复制和安全性，并且实现了更强壮的一致性程度以减少需要考虑的情况。研究结果证明对于学生来说Raft比Paxos更容易学习。Raft同时包括一个新的机制去修改成员集群，使用重叠多数（Overlapping Majorities）以保证安全性。

**1 Introduction**

共识算法允许机器集群像一个团结的能够容忍部分成员失败的集体一样工作。正因为如此，他们在构建可靠的大规模软件系统中扮演了关键角色。Paxos[15,16]在过去的十年间主导了共识算法的讨论：很多的共识算法实现都基于Paxos或者受其影响，而且Paxos变成了教导学生共识算法的主要方式。

不幸的是，尽管有很多使Paxos更容易被理解的尝试 ，它还是有一点难以理解。此外，它的架构需要复杂的修改以支持实际的系统。因为如此，系统构建者和学生都不喜欢Paxos。

在与Paxos抗争之后，我们着手去探寻新的可以提供更好的系统构建和教育基础的共识算法。我们的方法不是一般的方法，因为我们的主要目标是可理解性：我们可以对实际系统定义一个共识算法并且用明显比Paxos更简单学习去描述它吗？此外，我们希望算法可以促进直观的对于系统构建者的必要的发展。对于算法如何工作，为什么这么做都是十分重要的。

该项工作的结果是一个叫做Raft的共识算法。在设计Raft过程中我们应用了一些特殊的技术以改变可理解性，包括分解（Raft分隔了leader选举，log复制和安全性）和状态空间减少（相对Paxos，Raft减少了非确定性等级和servers可以和其他不一致的方法）一个针对两所大学43个学生的研究显示Raft相对Paxos明显更好理解：同时学习了两种算法后，其中的33个学生对Raft的问题回答的比Paxos更好。

Raft在很多地方和一些共识算法相似（特别是，Oki和Liskov的Viewstamped Replication[29,22]），但是有一些新特性：

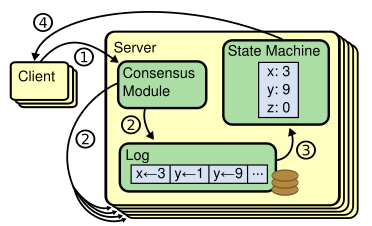
* Strong leader：Raft相对其他共识算法使用了更强的leadership构成。举例来说，log实体仅仅从leader到其他servers。这样简化了复制log的管理并且使得Raft更容易理解。
* Leader election：Raft使用随机定时器选举leader。该方式仅增加了一小部分其他算法早就有的心跳机制，可以使解决冲突更简单和更快。
* Membership changes：Raft对于更改server集群中集合的机制使用新共同共识（joint consensus）方式，这会使得在过渡时有两个不同配置的集合重叠。这样允许集群在配置修改时继续正常运作。

我们相信Raft比Paxos和其他共识算法优秀，不管是教育目的还是作为具体实现的基础。它比其他算法更简单，更易于理解；它的描述足够完整以满足实际系统的需求；它有很多的开源实现并且被很多公司使用；它的安全性属性被正式的指定和证明；它的效率也可以与其他算法相比。

论文的剩余部分介绍了复制状态机问题（第2节），讨论了Paxos的优劣势（第3节），描述了我们如何提高可理解性（第4节），现在的Raft共识算法（第5-8节），评价了Raft（第9节），并且讨论了相关工作（第10节）。

**2 Replicated state machines**

共识算法通常出现在复制状态机场景中。在这个方式中，server集合中的状态机计算相同的状态拷贝并且能在部分server宕机情况下继续执行。复制状态机是用来解决分布式系统中各式各样的容错问题的。举例来说，有一个单独的集群leader的大规模系统，就像GFS[8]，HDFS[38]，和RAMCloud[33]，一般使用一个隔离的复制状态机去管理leader选举和存储Leader崩溃后依然存在的配置信息。复制状态机的例子包括Chubby[2]和ZooKeeper[11]。

图1. 复制状态机架构。该共识算法管理一个包括了从client发出的状态机指令的日志副本。该状态机处理相同的从日志中获取的指令队列，所以他们产生相同的输出。

复制状态机一般使用复制日志实现，就如图1所示。每一个server存储一个包含了一系列相同顺序指令的日志，所以每一个状态机处理相同的指令序列。因此状态机是确定性的，每一次计算相同状态和相同输出序列。

保持日志副本一致是共识算法的职责。Server的共识模块收到client发送的指令并将它们写入日志中。它与其他server的共识模块通信以确认所有日志最终包含相同顺序和内容的请求，即使有些server失败了。一旦指令被正确复制了，每一个server的状态机根据日志的顺序处理它们，并且输出被回复给client。结果是，server似乎来自单独的高可靠的状态机。

实际系统的共识算法一般有以下属性：

* 在所有的无拜占庭情况下它们确保安全性（从不返回错误的结果），包括网络延迟，分隔，和包丢失，重复和重排序。
* 它们是完全实用（可用）的只要任意server的多数是工作的并且可以和其他server或client通信。
* 它们不依赖于定时去确认日志的一致性：不完善的时钟和极端的消息延迟，最坏造成可用性问题。
* 在常见场景下，一个指令可以在集群的多数响应一轮RPC调用后立即完成；小部分慢的server不会影响整体系统性能。

**3 What's wrong with Paxos?**

过去的十年间，Leslie Lamport的Paxos协议[15]成为了共识的代名词：它是课程上最广泛教导的，同时大部分共识算法实现使用它作为初始点。Paxos首次定义了协议的单个决定的达成协议的能力，比如一个单独的日志副本实体。我们将该子集称为single-decree Paxos。Paxos然后组合了协议多个实例以促进一系列决定比如一个日志（multi-Paxos）。Paxos同时确保了安全性和活跃性，而且它支持集群成员的改变。它的正确性也被证明了，在正常情况下效率也很不错。

不幸的是，Paxos有两个显著缺点。第一个缺点是Paxos异常难以理解。它的完整解释[15]是公认的不透明；少部分人花费大量功夫成功理解了它。结果是，这里有一些尝试用简单术语来解释Paxos的文章[16,20,21]。这些解释着眼于single-decree子集，这样也是极有挑战的。在NSDI 2012的参加者的非正式调查中，我们发现即使其中有老练的研究员，只有少部分人适应Paxos。我们与Paxos抗争；我们在直到读了一些简化的解释并设计我们自己的替代协议前都不能去理解完整的协议，这个进程花了大约一年。

我们推测Paxos的不透明来源于它选择single-decree子集作为基础。Single-decree Paxos是繁杂与细微的：它分为两个不能简单直观解释和独立理解的阶段。正因为如此，发展关于协议工作原理的思考是很难的。Multi-Paxos的组合规则添加了大量额外的复杂性和细节。我们相信多个决定（比如，一个日志代替一个单独实体）达到共识的问题可以用其他更直接明显方法分解。

第二个问题是没有给构建实用的实现提供一个良好的基础。一个理由是multi-Paxos没有被广泛接受。Lamport的描述更多是关于single-decree Paxos的；他只简单描述了可能实现multi-Paxos的方法，但是缺少了大量细节。有大量的尝试去充实优化Paxos，比如[26]，[39]和[13]，但是这些各自不同也和Lamport的描述不同。像Chubby这样的系统实现了类似Paxos的算法，但是很多场景下他们的处理细节都未公开。

此外，Paxos的架构对于构建实际系统而言太差了；这是拆分single-decree导致的另一个后果。举例来说，选择一个独立的日志实体集合然后将它们合并成一个有序的日志，没有任何益处；这仅仅增加了复杂度。设计一个围绕日志的系统更简单也更有效，新实体以受限的顺序有序的附加。另一个问题是Paxos使用了对称点对点方法作为核心（虽然它最后建议弱结构的领导来作为性能优化）。这在一个简化的只做一个决定的世界是有意义的，但是很少有实际系统会用这个方式。如果一系列决定必须确认，在首次选举leader时会更简单更快，然后有了leader协调决定。

结果是，实际系统使用类似Paxos的算法。每一个实现都始于Paxos，探索实现过程中的难点，然后开发一个明显不同的架构。这是非常耗时且容易出错的，而且理解Paxos的难度也会增加。Paxos的公式对于提供它的正确性理论是很好的，但是实际实现是不同于Paxos以至于证明的价值很少。下面Chubby实现者的评论是典型：

Paxos算法和实际系统需求有很明显的间隙，最终的系统将会基于未被证明的协议[4]。

因为这些原因，我们总结得出Paxos对于构建实际系统和教育都不适合作为基础。基于大规模软件系统中一致性的重要性，我们决定看看我们是否能够设计一个比Paxos有更好属性的替代共识算法。Raft正是实验的结果。

**4 Designing for understandability**

我们在设计Raft时有一些目标：它必须给系统构建提供一个完整的实用的基础，所以它明显减少了开发者的设计工作量；它必须在各种条件下是安全的，在一般操作环境下是可用的；对于常见操作必须是高效的。但是我们最重要的目标——也是最难的挑战——可理解性。它必须能够让大量的听众对算法感到舒适。此外，它必须可以发展关于算法的思考，所以系统构建者可以使用在真实实现中必须的扩展。

在Raft的设计中有很多点是需要我们在替代方案中选择的。这样的环境下我们基于可理解性评估了替代方案：解释每一个替代方案有多难（举例来说，它的状态空间多复杂，是否有细微的含义？），对于读者来说要完全理解它的方法和含义会多简单？

我们承认在这样的分析中有很高的主观性；尽管如此，我们使用两种普遍适用的技术。第一种技术是知名的问题分解方法：我们尽可能的把问题分解成可以解决的，解释的，相对独立理解的。举例来说，在Raft中我们分离了leader选举，log复制，安全性和成员变更。

我们的第二个方法是通过减少需要考虑的状态，使得系统更连贯和尽可能消除非确定性来简化状态空间。特别是日志不允许有缺口，并且Raft限制了日志会变得和其他不一致的方式。虽然在大多数情况下我们尝试消除非确定性，然而有些情况下非确定性可以提升可理解性。尤其是，随机方法引入非确定性，但是他们倾向于通过在相似风格（“任意选择；没关系的”）时处理所有的可能选项来减少状态空间。我们使用随机去简化Raft的leader选举算法。

**5 The Raft consensus algorithm**

Raft是一个像第2节描述的结构那样管理日志副本的算法。图2以浓缩的结构作为参考总结了算法，图3列出了算法的关键属性；这些图的元素在该节的剩余部分分段讨论。

|  |
| --- |
| State |
| 所有server上的持久化状态（在回复RPCs前更新稳定存储）：  currentTerm：server看到的最新term（首次引导被初始化为0，单调递增）  voteFor：当前term收到的投票的候选人ID（如果没有就为null）  log[]：日志实体；每个实体包含状态机指令，当实体被leader接收时的期（初始索引1）  所有server上的挥发性状态：  commitIndex：已知的提交的日志实体最高索引值（初始为0，单调递增）  lastApplied：状态机应用的日志实体最高索引值（初始为0，单调递增）  Leaders的挥发性状态（选举后重新初始化）：  nextIndex[]：对于每一个server的下一个会发送的日志实体的索引（初始为leader最后的日志索引+1）  matchIndex[]：对于每一个server的复制的已知日志实体的最高索引值（初始为0，单调递增） |
| RequestVote RPC |
| 由候选者调用以收集选票(5.2节)  参数：  term：候选者的term  candidateId：候选者要求投票  lastLogIndex：候选者的最新日志实体索引（5.4节）  lastLogTerm：候选者的最新日志实体term（5.4节）  结果：  term：当前term，给候选者更新自身的term  voteGranted：如果候选者收到则返回true  接收者实现：  1.如果term<currentTerm则回复false（5.1节）  2.如果votedFor是空的或者candidatedId，且候选者的日志最少是和接收者的日志是最新的，则投票（5.2，5.4节） |
| AppendEntries RPC |
| 由leader调用去复制日志实体（5.3节）；同时用作心跳（5.2节）  参数：  term：leader的term  leaderId：跟随者因此可以重定向clients  prevLogIndex：上一个日志实体的索引值  prevLogTerm：prevLogIndex对应实体的term  entries[]：要存储的日志实体（心跳时是空的；为了效率可能发送多个）  leaderCommit：leader的commitIndex  结果：  term：currentTerm，给leader更新自身  success：如果跟随者有匹配prevLogIndex和prevLogTerm的实体则返回true  接收者实现：  1.如果term<currentTerm返回false（5.1节）  2.如果日志在prevLogIndex上没有实体的term为prevLogTerm，返回false（5.3节）  3.如果已有的实体和新的冲突（相同索引但是不同term），删除已有的实体和它之后的实体  4.添加任意不在日志中的实体  5.如果leaderCommit>commitIndex，设置commitIndex=min(leaderCommit, 最新实体的索引) |
| Rules for Servers |
| 所有servers：   * 如果commitIndex>lastApplied；增加lastApplied，应用log[lastApplied]给状态机（5.3节） * 如果RPC请求或回复包含term T>currentTerm；设置currentTerm为T，转换为跟随者（5.1节）   跟随者（5.2节）：   * 回复候选者和leaders的RPC调用 * 如果没有接收到来自当前leader的AppendEntries RPC或给候选者的投票，则选举超时：转换为候选者   候选者（5.2节）：   * 转换为候选者，开始选举：   + 增加currentTerm   + 投给自己   + 重置选举计时器   + 发送RequestVote RPC给所有其他servers * 如果从大多数server中收到投票：变成leader * 如果从一个新leader处收到AppendEntries RPC：转换为跟随者 * 如果选举超时：开始新选举   Leaders：   * 当选：发送初始空AppendEntries RPC（心跳）给所有server；在空闲期间重复发送以避免选举超时（5.2节） * 如果从client处收到指令：添加实体到本地日志，在实体应用到状态及后回应（5.3节） * 如果对于一个跟随者最新的日志index≥nextIndex：发送从nextIndex开始的日志实体的AppendEntries RPC   + 如果成功：更新跟随者的nextIndex和matchIndex   + 如果由于日志不一致失败：降低nextIndex并重试 * 如果存在一个N使得N>commitIndex，多数matchIndex[i]≥N，且log[N].term==currentTerm：设置commitIndex=N（5.3，5.4节） |

图2：一个Raft共识算法的简短总结（排除成员修改和日志压缩）。Server的行为被描述为独立重复触发的规则集合。小节编号表示何处会说明具体特性。正式的规范[31]描述算法更确切。

|  |
| --- |
| Election Safety：最多只有一个Leader会被选举出来（5.2节）  Leader Append-Only：Leader绝不在它的日志中复写或删除实体；只会添加新实体（5.3节）  Log Matching：如果两个日志包含一个相同索引和term的实体，则这两份日志中所有给定索引的实体都是相同的。（5.3节）  Leader Completeness：如果一个日志实体在给定的term中被提交，则对于所有更高序号的term，该实体会存在于leader的日志中。（5.4节）  State Machine Safety：如果一个server给它自己的状态机应用了一个给定索引的日志实体，没有其他server会在相同索引应用不同的日志实体。（5.4.3节） |

图3：Raft一直保证这些属性。小节编号表示何处会说明具体特性。

Raft实现共识的方式是首先选举一个突出的leader，然后给这个leader完整的管理日志副本的职责。Leader从clients处收到日志实体，将它们复制到其他server上，然后告诉server什么时候将日志实体应用到他们的状态机是安全的。Leader简化了日志副本的管理。举例来说，leader可以在没有询问其他server的情况下决定将新实体放在log的什么位置，而且数据流根据简单的风格从leader到其他server。Leader可以失败或者与其他server断连，这时会选举一个新leader。

鉴于leader的方法，Raft将共识问题分解为3个相对独立的子问题，将会在接下去的字节中讨论：

* Leader Selection：当已有的leader失败时必须选一个新leader（5.2节）。
* Log replication：Leader必须接受client发送的日志实体并在急集群中复制它们，要求其他日志以它的为准（5.3节）。
* Safety：Raft关键的安全性属性是图3中的状态机安全属性：如果所有server给它自己的状态机应用特定的日志实体，则没有其他server会在相同日志索引上应用不同的指令。5.4节描述了Raft怎么确保该属性的；解决方案设计一个5.2节描述的选举机制的额外限制。

提出该共识算法后，该节讨论可用性问题和系统中时机的作用。

**5.1 Raft Basics**

一个Raft集群