

毕 业 设 计

外 文 文 献 译 文 及 原 文



**学 生：**  徐 洋

**学 号：**  201412040126

**院 （系）：** 电气与信息工程学院

**专 业：** 计算机科学与技术

**指导教师：** 齐 勇

2018 **年** 6**月** 19**日**

寻找一种易于理解的一致性算法

摘要：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比Paxos更容易理解并且更易于构建实际的系统。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，日志复制和安全性，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，它使用重叠大多数来保证安全性。

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，日志复制和安全性）和减少状态（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，但是Raft有几个新的特性：

强领导者：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：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：Raft为了调整集群中成员关系使用了新的联合一致性的方法，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。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

一致性算法是在[复制状态机](https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf)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等。

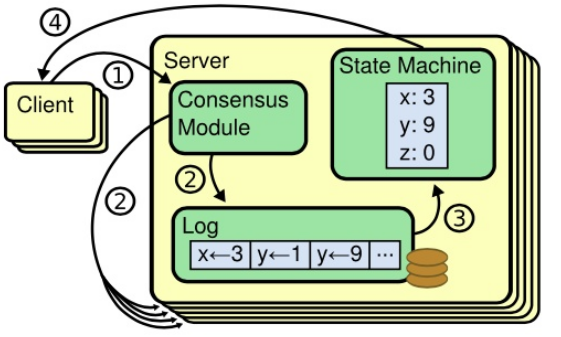


图-1：复制状态机的架构。一致性算法管理来自客户端状态命令的复制日志。状态机处理的日志中的命令的顺序都是一致的，因此会得到相同的执行结果。

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。服务器停止工作了我们就认为它失败了，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。我们把这个子集叫做单一决策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。Paxos确保安全性和活跃性，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，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；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。

**In Search of an Understandable Consensus Algorithm  
(Extended Version)**

Diego Ongaro and John Ousterhout  
Stanford University

**Abstract:** Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

**1 Introduction**

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus. Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture requires complex changes to support practical systems. As a result, both system builders and students struggle with Paxos.

After struggling with Paxos ourselves, we set out to find a new consensus algorithm that could provide a better foundation for system building and education. Our approach was unusual in that our primary goal was understandability: could we define a consensus algorithm for practical systems and describe it in a way that is significantly easier to learn than Paxos? Furthermore, we wanted the algorithm to facilitate the development of intuitions that are essential for system builders. It was important not just for the algorithm to work, but for it to be obvious why it works.

The result of this work is a consensus algorithm called Raft. In designing Raft we applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety) and state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most We believe that Raft notably, Oki and Liskov’s Viewstamped Replication [29, 22]), but it has several novel features: ? Strong leader: Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand. ? Leader election: Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly. ? Membership changes: Raft’s mechanism for changing the set of servers in the cluster uses a new joint consensus approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. It is simpler and more understandable than other algorithms; it is described completely enough to meet the needs of a practical system; it has several open-source implementations and is used by several companies; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

The remainder of the paper introduces the replicated state machine problem (Section 2), discusses the strengths and weaknesses of Paxos (Section 3), describes our general approach to understandability (Section 4), presents the Raft consensus algorithm (Sections 5–8), evaluates Raft (Section 9), and discusses related work (Section 10).

**2 Replicated state machines**

Consensus algorithms typically arise in the context of replicated state machines [37]. In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines are used to solve a variety of fault tolerance problems in distributed systems. For example, large-scale systems that have a single cluster leader, such as GFS [8], HDFS [38], and RAMCloud [33], typically use a separate replicated state machine to manage leader election and store configuration information that must survive leader crashes. Examples of replicated state machines include Chubby [2] and ZooKeeper [11].

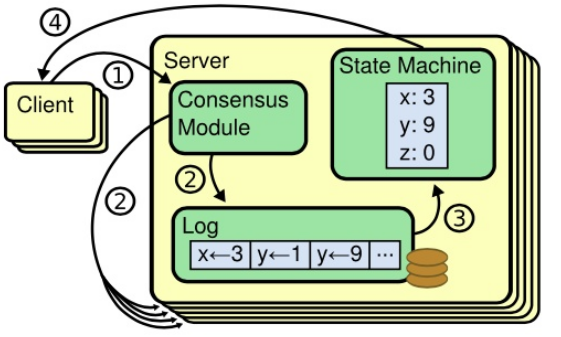


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

Replicated state machines are typically implemented using a replicated log, as shown in Figure 1. Each server stores a log containing a series of commands, which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs.

Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server’s state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

Consensus algorithms for practical systems typically have the following properties: ? They ensure safety (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering. ? They are fully functional (available) as long as any majority of the servers are operational and can communicate with each other and with clients. Thus, a typical cluster of five servers can tolerate the failure of any two servers. Servers are assumed to fail by stopping; they may later recover from state on stable storage and rejoin the cluster. ? They do not depend on timing to ensure the consistency of the logs: faulty clocks and extreme message delays can, at worst, cause availability problems. ? In the common case, a command can complete as soon as a majority of the cluster has responded to a single round of remote procedure calls; a minority of slow servers need not impact overall system performance.

**3 What’s wrong with Paxos?**

Over the last ten years, Leslie Lamport’s Paxos protocol [15] has become almost synonymous with consensus: it is the protocol most commonly taught in courses, and most implementations of consensus use it as a starting point. Paxos first defines a protocol capable of reaching agreement on a single decision, such as a single replicated log entry. We refer to this subset as single-decree Paxos. Paxos then combines multiple instances of this protocol to facilitate a series of decisions such as a log (multi-Paxos). Paxos ensures both safety and liveness, and it supports changes in cluster membership. Its correctness has been proven, and it is efficient in the normal case.

Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [15] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [16, 20, 21]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year.

We hypothesize that Paxos’ opaqueness derives from its choice of the single-decree subset as its foundation. Single-decree Paxos is dense and subtle: it is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. Because of this, it is difficult to develop intuitions about why the singledecree protocol works. The composition rules for multiPaxos add significant additional complexity and subtlety. We believe that the overall problem of reaching consensus on multiple decisions (i.e., a log instead of a single entry) can be decomposed in other ways that are more direct and obvious.

The second problem with Paxos is that it does not provide a good foundation for building practical implementations. One reason is that there is no widely agreedupon algorithm for multi-Paxos. Lamport’s descriptions are mostly about single-decree Paxos; he sketched possible approaches to multi-Paxos, but many details are missing. There have been several attempts to flesh out and optimize Paxos, such as [26], [39], and [13], but these differ from each other and from Lamport’s sketches. Systems such as Chubby [4] have implemented Paxos-like algorithms, but in most cases their details have not been published.

Furthermore, the Paxos architecture is a poor one for building practical systems; this is another consequence of the single-decree decomposition. For example, there is little benefit to choosing a collection of log entries independently and then melding them into a sequential log; this just adds complexity. It is simpler and more efficient to design a system around a log, where new entries are appended sequentially in a constrained order. Another problem is that Paxos uses a symmetric peer-to-peer approach at its core (though it eventually suggests a weak form of leadership as a performance optimization). This makes sense in a simplified world where only one decision will be made, but few practical systems use this approach. If a series of decisions must be made, it is simpler and faster to first elect a leader, then have the leader coordinate the decisions.

As a result, practical systems bear little resemblance to Paxos. Each implementation begins with Paxos, discovers the difficulties in implementing it, and then develops a significantly different architecture. This is timeconsuming and error-prone, and the difficulties of understanding Paxos exacerbate the problem. Paxos’ formulation may be a good one for proving theorems about its correctness, but real implementations are so different from Paxos that the proofs have little value. The following comment from the Chubby implementers is typical: There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. . . . the final system will be based on an unproven protocol [4]. Because of these problems, we concluded that Paxos does not provide a good foundation either for system building or for education. Given the importance of consensus in large-scale software systems, we decided to see if we could design an alternative consensus algorithm with better properties than Paxos. Raft is the result of that experiment.