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.

摘要
Raft是一种用于管理复制日志的共识算法。它产生的结果等同于（多）Paxos，而且与Paxos一样高效，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。为了提高可理解性，Raft分离了共识的关键要素，如领导者选举、日志复制和安全，而且它执行了更强的一致性，以减少必须考虑的状态数量。一项用户研究的结果表明，Raft比Paxos更容易让学生学习。Raft还包括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 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.

共识算法允许一组机器作为一个连贯的团体工作，可以承受一些成员的故障。正因为如此，它们在构建可靠的大规模软件系统中发挥了关键作用。Paxos[15, 16]在过去十年中主导了关于共识算法的讨论：大多数共识的实现都是基于Paxos或受其影响，而Paxos已经成为教学生了解共识的主要工具。

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.

不幸的是，尽管有许多尝试使它更容易实践，Paxos还是相当难理解。此外，它的架构需要复杂的改动来支持实际的系统。因此，系统建设者和学生都在为Paxos而头疼。

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.

在与Paxos纠结许久之后，我们开始寻找一种新的共识算法，为系统建设和教育提供一个更好的基础。我们的方法是不寻常的，因为我们的主要目标是可理解性：我们能否为实用系统定义一个共识算法，并以一种明显比Paxos更容易学习的方式来描述它？此外，我们希望该算法能够促进直觉的发展，这对系统建设者来说是至关重要的。重要的是，不仅要让算法起作用，而且要让它的作用显而易见。

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 in consistent 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.

这项工作的结果是一种叫做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、日志复制和安全分开）和减少状态空间（相对于Paxos，Raft减少了非确定性的程度以及服务器之间的一致方式）。对两所大学的43名学生进行的用户研究表明，Raft明显比Paxos更容易理解：在学习了两种算法后，这些学生中有33人能够更好地回答关于Raft的问题，而不是关于Paxos的问题。

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

Raft在许多方面与现有的共识算法（最值得注意的是Oki和Liskov的Viewstamped Replication[29, 22]）相似，但它有几个新的特点。

- **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.

强领导力。与其他共识算法相比，Raft使用了一种更强的领导形式。例如，日志条目只从领导者流向其他服务器。这简化了复制日志的管理，使Raft更容易理解。

- **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.

领袖选举。Raft使用随机的计时器来选举领导人。这在任何共识算法已经需要的心跳上只增加了少量的机制，同时简单而迅速地解决了冲突。

- **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.

成员变化。Raft改变集群中的服务器集的机制使用了一种新的联合共识方法，其中两个不同配

置的多数在过渡期间重叠。这使得集群在配置变化期间能够继续正常运行。

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.

我们认为Raft比Paxos和其他共识算法更胜一筹，无论是出于教育目的还是作为实施的基础。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；它有几个开源的实现，并被几个公司使用；它的安全属性已经被正式规定和证明；它的效率与其他算法相当。

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).

本文的其余部分介绍了复制的状态机问题（第2节），讨论了Paxos的优点和缺点（第3节），描述了我们对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 Replicated state machines

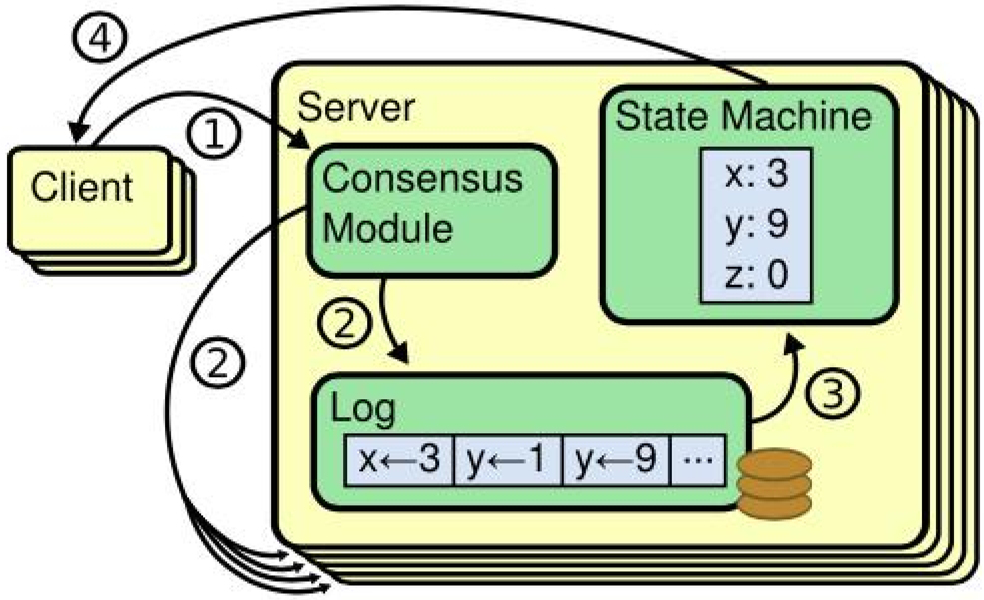


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

图1：复制的状态机架构。共识算法管理着一个包含来自客户的状态机命令的复制日志。状态机处理来自日志的相同的命令序列，因此它们产生相同的输出。

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].

共识算法通常出现在复制状态机的背景下[37]。在这种方法中，服务器集合上的状态机计算相同状态的相同副本，即使一些服务器停机，也能继续运行。复制的状态机被用来解决分布式系统中的各种容错问题。例如，拥有单一集群领导者的大规模系统，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的选举，并存储必须在领导者崩溃后生存的配置信息。复制状态机的例子包括Chubby [2] 和 ZooKeeper [11]。

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.

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。每个日志都包含相同顺序的命令，所以每个状态机处理相同的命令序列。由于状态机是确定性的，每个状态机都计算出相同的状态和相同的输出序列。

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.

保持复制的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，并将它们添加到其日志中。它与其他服务器上的共识模块进行通信，以确保每条日志最终都包含相同顺序的请求，即使一些服务器失败了。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，并将输出结果返回给客户。因此，这些服务器似乎形成了一个单一的、高度可靠的状态机。

Consensus algorithms for practical systems typically have the following properties:
实用系统的共识算法通常具有以下特性：

• They ensure *safety* (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering.

在所有非拜占庭的条件下，包括网络延迟、分区以及数据包丢失、重复和重新排序，它们都能确保安全（永远不会返回错误的结果）。

- 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.

只要任何大多数的服务器都在运行，并且能够相互之间以及与客户进行通信，它们就能完全发挥作用（可用）。因此，一个典型的由五个服务器组成的集群可以容忍任何两个服务器的故障。通过停止服务器假装失效；它们后来可以从稳定存储的状态中恢复并重新加入集群。

- 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.

它们不依赖时间来确保日志的一致性：故障时钟和极端的消息延迟在最坏的情况下会导致可用性问题。

- 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.

在普通情况下，只要集群中的大多数个体对单轮RPC作出反应，就可以完成一个命令；少数慢的服务器不需要影响整个系统的性能。

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.

在过去的十年里，Leslie Lamport的Paxos协议[15]几乎成了共识的代名词：它是课程中最常教授的协议，大多数共识的实现都以它为起点。Paxos首先定义了一个能够在单一决策上达成协议的协议，例如单一复制的日志条目。我们把这个子集称为单决策的Paxos。然后，Paxos结合这个协议的多个实例，以促进一系列的决定，如日志（多Paxos）。Paxos既保证了安全性，又保证了有效性，而且它支持集群成员的变化。它的正确性已被证明，而且在正常情况下是有效的。

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 alter native protocol, a process that took almost a year.

不幸的是，Paxos有两个显著的缺点。第一个缺点是，Paxos特别难理解。完整的解释[15]是出了名的不透明；很少有人能成功地理解它，而且是在付出巨大努力之后。因此，已经有一些尝试用更简

单的术语来解释Paxos [16, 20, 21]。这些解释集中在单决策子集上，然而它们仍然很难。在对NSDI 2012的与会者进行的非正式调查中，我们发现很少有人对Paxos感到满意，即使在经验丰富的研究人员中。我们自己也在为Paxos头疼；在阅读了几个简化的解释和设计了我们自己修改的朴素协议之后，我们才能够理解完整的协议，这个过程花了几乎一年时间。

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 single decree protocol works. The composition rules for multi Paxos 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.

我们假设，Paxos的不透明性来自于它选择单决策子集作为基础。单决策Paxos是密集而微妙的：它分为两个阶段，没有简单的直观解释，不能独立理解。正因为如此，很难发展出关于单决策协议为何有效的直觉。多Paxos的组成规则大大增加了复杂性和微妙性。我们认为，就多个决定达成共识的整体问题（即一个日志而不是一个条目）可以用其他更直接和明显的方式进行分解。

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 agreed-upon 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.

Paxos的第二个问题是，它没有为建立实际的实现提供一个良好的基础。原因之一是没有广泛认同的多Paxos的算法。Lamport的描述主要是关于单决策Paxos的；他勾画了多Paxos的可能方法，但许多细节都没有。已经有一些尝试来充实和优化Paxos，如[26]、[39]和[13]，但这些尝试相互之间以及与Lamport的草图不同。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，它们的细节还没有被公布。

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.

此外，Paxos架构对于构建实用系统来说是一个糟糕的架构；这是单决策分解的另一个后果。例如，独立地选择一个日志条目集合，然后将它们拼接成一个连续的日志，这没有什么好处，只会增加复杂性。围绕日志设计一个系统才更简单、有效，新的条目是以受限的顺序依次追加的。另一个问题是，Paxos的核心是使用对称的P2P方法（尽管它最终建议采用弱的领导形式作为性能优化）。这在一个只做一个决定的简化世界中是有意义的，但很少有实际的系统使用这种方法。如果必须做出一系列的决定，首先选举一个领导者，然后让领导者协调这些决定，这样做更简单、快捷。

As a result, practical systems bear little resemblance to Paxos. Each implementation begins with Paxos, covers the difficulties in implementing it, and then develops a significantly different architecture. This is time consuming 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:

因此，实际的系统与Paxos几乎没有相似之处。每个实现都是从Paxos开始的，涵盖了实现它的困难，然后开发出一个明显不同的架构。这样做既费时又容易出错，而理解Paxos的困难又加剧了这个问题。Paxos的公式化对于证明其定理的正确性来说可能是一个很好的表述，但是真正的实现与Paxos差别很大，所以证明的价值不大。以下是来自Chubby实现者的典型评论：

> 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].
>
> 在Paxos算法的描述和真实世界系统的需求之间存在着巨大的差距...最终的系统基于一个未经证实的协议[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.

由于这些问题，我们得出结论，Paxos既不能为系统建设也不能为教育提供一个良好的基础。考虑到共识在大规模软件系统中的重要性，我们决定看看我们是否能设计出一种比Paxos具有更好特性的替代性共识算法。Raft就是这个实验的结果。

4 Designing for understandability

We had several goals in designing Raft: it must provide a complete and practical foundation for system building, so that it significantly reduces the amount of design work required of developers; it must be safe under all conditions and available under typical operating conditions; and it must be efficient for common operations. But our most important goal—and most difficult challenge—was *understandability*. It must be possible for a large audience to understand the algorithm comfortably. In addition, it must be possible to develop intuitions about the algorithm, so that system builders can make the extensions that are inevitable in real-world implementations.

我们在设计Raft时有几个目标：它必须为系统建设提供一个完整而实用的基础，以便大大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可用；它必须对普通操作高效。但我们最重要的目标和最困难的挑战是可理解性。必须让广大受众能够舒适地理解该算法。此外，还必须有可能发展关于该算法的直觉，以便系统构建者能够进行扩展，而这在现实世界的实施中是不可避免的。

There were numerous points in the design of Raft where we had to choose among alternative approaches. In these situations we evaluated the alternatives based on understandability: how hard is it to explain each alternative (for example, how complex is its state space, and does it have subtle implications?), and how easy will it be for a reader to completely understand the approach and its implications?

在Raft的设计中，有许多地方我们必须在备选方法中做出选择。在这些情况下，我们根据可理解性来评估替代方案：解释每个替代方案有多难（例如，它的状态空间有多复杂，是否有微妙的影响）

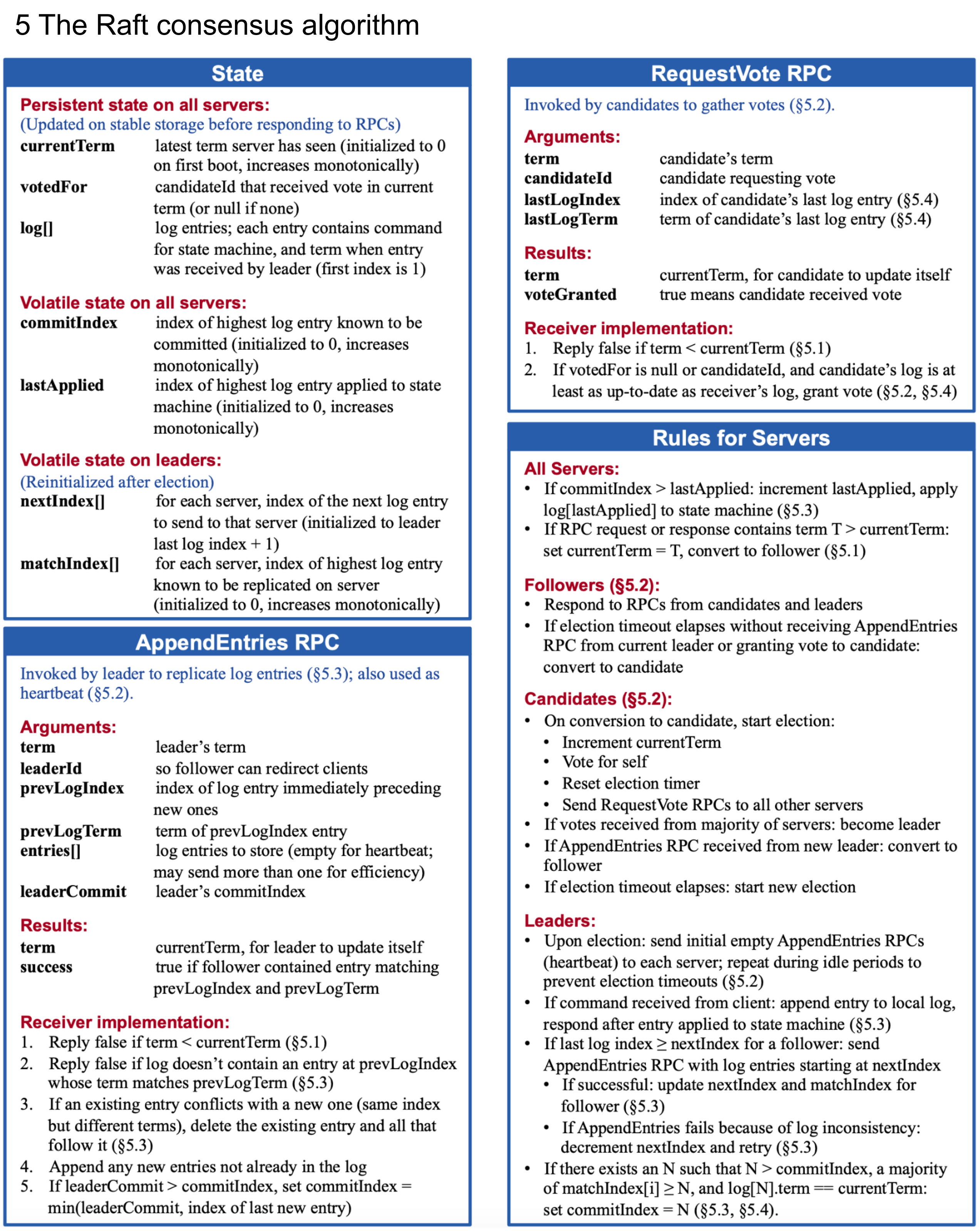
，以及读者完全理解该方法及其影响有多容易？

We recognize that there is a high degree of subjectivity in such analysis; nonetheless, we used two techniques that are generally applicable. The first technique is the well-known approach of problem decomposition: wherever possible, we divided problems into separate pieces that could be solved, explained, and understood relatively independently. For example, in Raft we separated leader election, log replication, safety, and membership changes.

我们认识到，这种分析有很大程度的主观性；然而，我们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：在可能的情况下，我们把问题分成独立的部分，可以相对独立地解决、解释和理解。例如，在Raft中，我们将领袖选举、日志复制、安全和成员变化分开。

Our second approach was to simplify the state space by reducing the number of states to consider, making the system more coherent and eliminating nondeterminism where possible. Specifically, logs are not allowed to have holes, and Raft limits the ways in which logs can become inconsistent with each other. Although in most cases we tried to eliminate nondeterminism, there are some situations where nondeterminism actually improves understandability. In particular, randomized approaches introduce nondeterminism, but they tend to reduce the state space by handling all possible choices in a similar fashion (“choose any; it doesn’t matter”). We used randomization to simplify the Raft leader election algorithm.

我们的第二个方法是通过减少需要考虑的状态数量来简化状态空间，使系统更加连贯，并尽可能消除非确定性。具体来说，不允许日志有漏洞，而且Raft限制了日志相互之间不一致的方式。尽管在大多数情况下，我们试图消除非确定性，但在某些情况下，非确定性实际上提高了可理解性。特别是，随机化的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来减少状态空间（"choose any; it doesn’t matter 选择任何状态；这并不重要"）。我们使用随机化来简化Raft领袖选举算法。



5 The Raft consensus algorithm

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:
1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:
1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):
- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

State 状态

Persistent state on all servers: 所有服务器上的持久性状态	
(Updated on stable storage before responding to RPCs) (在响应RPC之前，在稳定的存储上更新)	
currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically) 服务器看到的最新任期（首次启动时初始化为0，单调增加）
votedFor	candidateId that received vote in current term (or null if none) 当前任期内获得选票的候选人ID（如果没有则为空）
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1) 日志条目;每个条目包含状态机的命令，以及领导者收到条目的任期（第一个索引是1）

Volatile state on all servers: 所有服务器上的易失性状态	
commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically) 已知被提交的最高日志条目的索引(初始化为0，单调增加）
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically) 已应用于状态机的最高日志条目的索引（初始化为0，单调增加）

Volatile state on leaders: 领导者（服务器）上的易失性状态	
(Reinitialized after election) (选举后重新初始化)	
nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1) 对于每个服务器，要发送给该服务器的下一个日志条目的索引（初始化为领导者的最后一个日志索引+1）
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically) 对于每个服务器，已知在服务器上复制的最高日志条目的索引（初始化为0，单调增加）

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).
由领导者调用以复制日志条目（§5.3）；也作为心跳（§5.2）使用

Arguments:	
term	leader's term 领导任期
leaderId	so follower can redirect clients 使追随者可以为客户端重定向
prevLogIndex	index of log entry immediately preceding new ones 紧接在新日志之前的日志条目的索引
prevLogTerm	term of prevLogIndex entry 紧邻新日志条目之前的日志条目的任期
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency) 需要被保存的日志条目(做心跳使用时，内容为空；为了提高效率可一次性发送多个)
leaderCommit	leader's commitIndex 领导者的已知已提交的最高的日志条目的索引

Results:	
term	currentTerm, for leader to update itself 当前任期，领导者会更新自己的任期
success	true if follower contained entry matching prevLogIndex and prevLogTerm 如果跟随者所含有的条目和prevLogIndex以及prevLogTerm匹配，则为真

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
1. 如果 term < currentTerm，则返回 false (§5.1)

2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
2. 如果日志在prevLogIndex处不包含term与prevLogTerm匹配的条目，则返回false（§5.3）。

3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3) 3. 如果一个现有的条目与一个新的条目相冲突（相同的索引但不同的任期），删除现有的条目和后面所有的条目(§5.3)
4. Append any new entries not already in the log 4. 添加日志中任何尚未出现的新条目
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry) 5. 如果leaderCommit > commitIndex，设置commitIndex = min(leaderCommit, 最后一个新条目的索引)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).
候选人为收集选票而调用（§5.2）。

Arguments:	
term	candidate's term 候选人的任期号
candidateId	candidate requesting vote 请求选票的候选人的 Id
lastLogIndex	index of candidate's last log entry (§5.4) 候选人的最后日志条目的索引值
lastLogTerm	term of candidate's last log entry (§5.4) 候选人最后日志条目的任期号

Results:	
term	currentTerm, for candidate to update itself 当前任期号，候选人会更新自己的任期号
voteGranted	true means candidate received vote true 表示候选人获得了选票

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
1. 如果 term < currentTerm，则返回 false (§5.1)

2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)
2. 如果 votedFor 是 null 或 candidateId，并且候选人的日志至少与接收人的日志一样新，则投票（§5.2, §5.4）。

Rules for Servers

All Servers:

• If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
如果commitIndex>lastApplied：增加lastApplied，将log[lastApplied]应用于状态机（§5.3）

• If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)
如果RPC请求或响应包含任期 T > currentTerm： 设置currentTerm = T， 转换为follower（§5.1）。

Followers (§5.2):

• Respond to RPCs from candidates and leaders
对候选人和领导人的RPC作出回应

• If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate
如果选举超时， 没有收到现任Leader的AppendEntries RPC， 也没有给Candidate投票： 转换为Candidate

Candidates (§5.2):

• On conversion to candidate, start election: 在转换为候选人时，开始选举
- • Increment currentTerm 递增 currentTerm
- • Vote for self 给自己投票
- • Reset election timer 重置选举定时器
- • Send RequestVote RPCs to all other servers 向所有其他服务器发送RequestVote RPCs

• If votes received from majority of servers: become leader
 如果获得大多数服务器的投票：成为领导者

• If AppendEntries RPC received from new leader: convert to follower
如果收到来自新领导的AppendEntries RPC：转换为跟随者

• If election timeout elapses: start new election
如果选举超时：开始新的选举

Leaders:

• Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
关于选举：向每个服务器发送初始的空AppendEntries RPC（心跳）；在空闲期间重复，以防止
举超时（§5.2）

• If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
如果收到来自客户端的命令：将条目追加到本地日志，在条目应用于状态机后做出响应（§5.3

• If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
 如果一个跟随者的最后一个日志索引≥nextIndex：发送AppendEntries RPC包含从nextIndex开始的日志条目

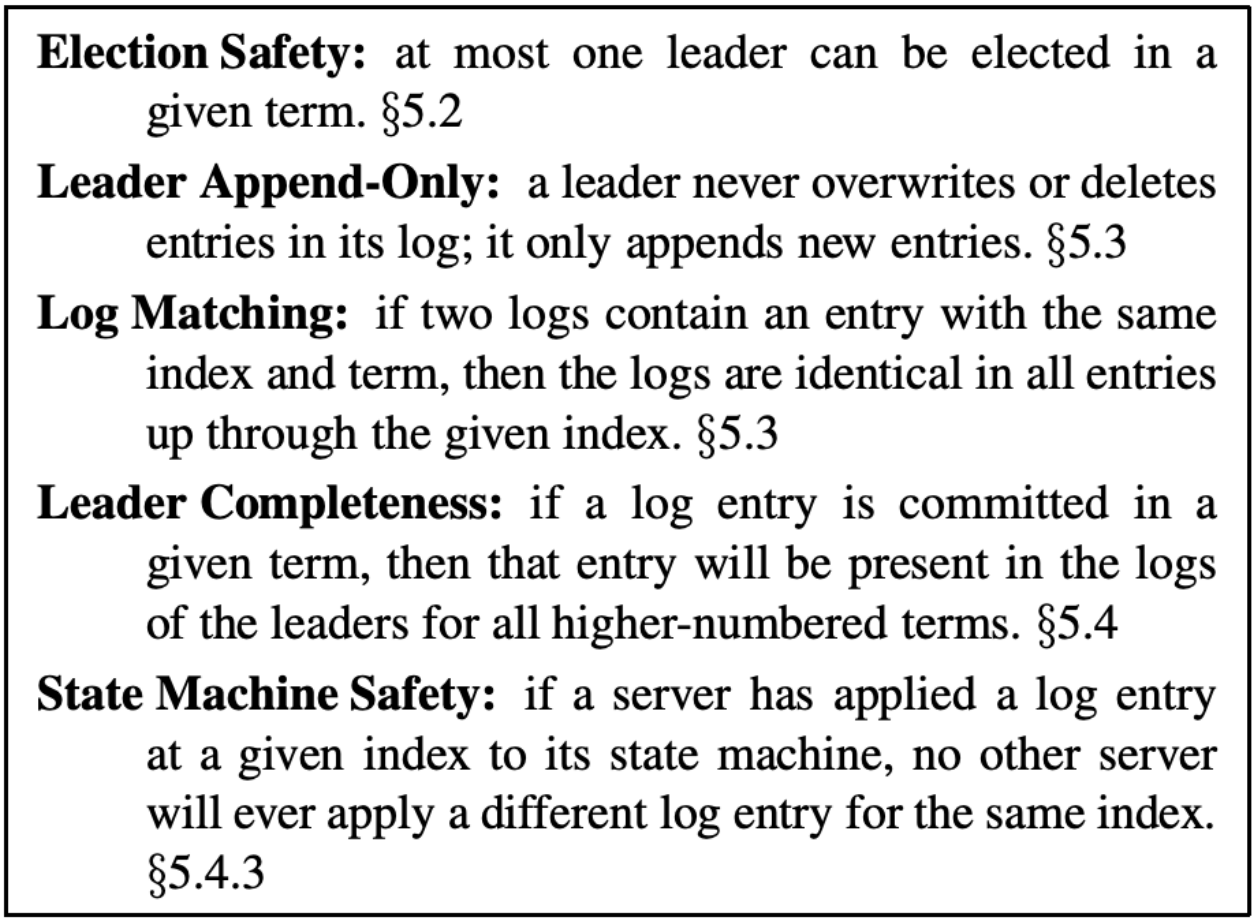
• If successful: update nextIndex and matchIndex for follower (§5.3)
如果成功：为跟随者更新nextIndex和matchIndex（§5.3）

• If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
 如果AppendEntries因为日志不一致而失败：递减NextIndex并重试（§5.3）

• If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).
如果存在一个N，使得N>commitIndex，大多数的matchIndex[i]≥N，并且log[N].term == currentTerm：设置commitIndex = N（§5.3，§5.4）

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.

图2：Raft共识算法的浓缩摘要（不包括成员变化和日志压实）。左上角方框中的服务器行为被描述为一组独立和重复触发的规则。诸如§5.2的章节编号表示讨论特定功能的地方。一个正式的规范[31]更精确地描述了该算法。



Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Election Safety: at most one leader can be elected in a given term. §5.2 选举安全：在一个特定的任期内最多可以选出一名领导人。§5.2
Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3 Leader Append-Only: 领导者从不覆盖或删除其日志中的条目；它只附加新条目。§5.3
Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3 日志匹配: 如果两个日志包含一个具有相同索引和任期的条目, 那么这两个日志的所有条目到给定索引都是相同的。§5.3
Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4 领导者的完整性：如果一个日志条目在某一任期中被承诺，那么该条目将出现在所有更高编号任期的领导者的日志中。§5.4
State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3 状态机安全：如果一个服务器在其状态机上应用了一个给定索引的日志条目，那么其他服务器将永远不会为同一索引应用不同的日志条目。§5.4.3

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

图3：Raft保证这些属性中的每一个在任何时候都是真的。节号表示每个属性的讨论位置。

Raft is an algorithm for managing a replicated log of the form described in Section 2. Figure 2 summarizes the algorithm in condensed form for reference, and Figure 3 lists key properties of the algorithm; the elements of these figures are discussed piecewise over the rest of this section.

Raft是一种用于管理第2节所述形式的复制日志的算法。图2概括了该算法的浓缩形式以供参考，图3列出了该算法的关键属性；这些数字元素将在本节的其余部分逐一讨论。

Raft implements consensus by first electing a distinguished *leader*, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. Having a leader simplifies the management of the replicated log. For example, the leader can decide where to place new entries in the log without consulting other servers, and data flows in a simple fashion from the leader to other servers. A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

Raft通过首先选举一个不同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。领导接受来自客户端的日志条目，将其复制到其他服务器上，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。有一个领导者可以简化对复制日志的管理。例如，领导者可以决定在日志中放置新条目的位置，而不需要咨询其他服务器，并且数据以一种简单的方式从领导者流向其他服务器。一个领导者可能会失败或与其他服务器断开连接，在这种情况下，会选出一个新的领导者。

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems, which are discussed in the subsections that follow:
考虑到领导方法，Raft将共识问题分解为三个相对独立的子问题，这将在后面的小节中讨论：

- **Leader election**: a new leader must be chosen when an existing leader fails (Section 5.2).
领袖选举：当现有领袖失败时，必须选择一个新的领袖（第5.2节）。

- **Log replication**: the leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own (Section 5.3).
日志复制：领导者必须接受来自客户端的日志条目，并在整个集群中进行复制，追使其他日志与自己的日志一致（第5.3节）。

- **Safety**: the key safety property for Raft is the State Machine Safety Property in Figure 3: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index. Section 5.4 describes how Raft ensures this property; the solution involves an additional restriction on the election mechanism described in Section 5.2.
安全性：Raft的关键安全属性是图3中的状态机安全属性：如果任何服务器对其状态机应用了一个特定的日志条目，那么其他服务器不得对同一日志索引应用不同的命令。第5.4节描述了Raft如何确保这一属性；该解决方案涉及对第5.2节中描述的选举机制的额外限制。

After presenting the consensus algorithm, this section discusses the issue of availability and the role of timing in the system.
在介绍了共识算法之后，本节讨论了系统中的可用性问题和计时的作用。

5.1 Raft basics

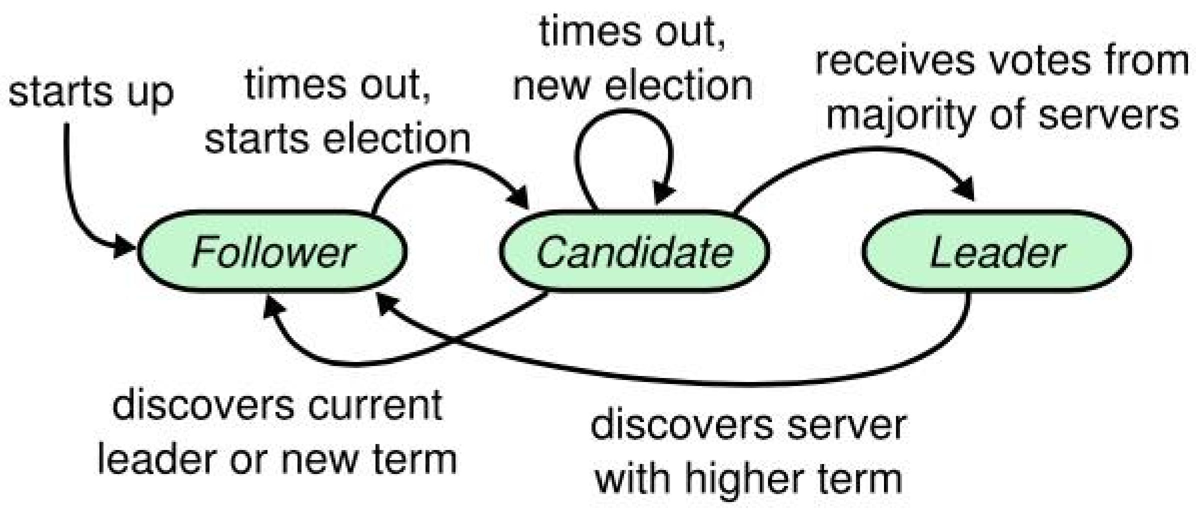

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

图4：服务器状态。Follower只对来自其他服务器的请求做出回应。如果一个Follower没有收到任何通信，它就会成为一个Candidate并发起选举。获得整个集群中大多数人投票的Candidate成为新的Leader。Leader通常会一直运行，直到他们失效。

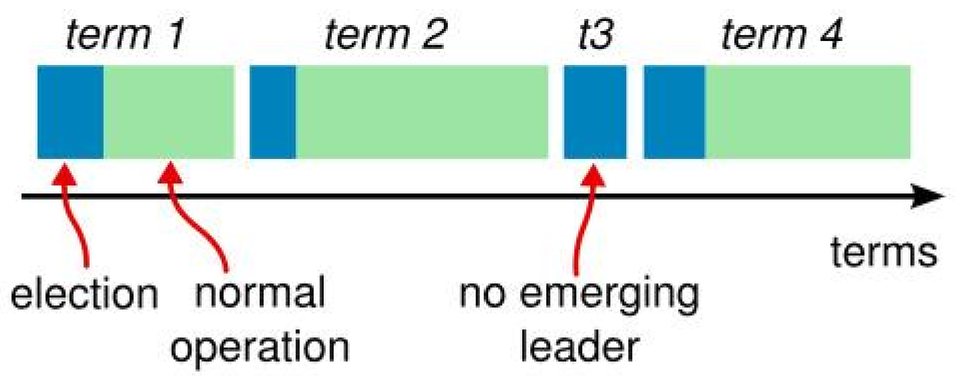


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.
图5：时间被划分为几个任期，每个任期以选举开始。选举成功后，由一个领导者管理集群，直到任期结束。有些选举失败，在这种情况下，任期结束时没有选择领导者。在不同的服务器上，可以在不同的时间观察到任期之间的转换。

A Raft cluster contains several servers; five is a typical number, which allows the system to tolerate two failures. At any given time each server is in one of three states: *leader*, *follower*, or *candidate*. In normal operation there is exactly one leader and all of the other servers are followers. Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). The third state, candidate, is used to elect a new leader as described in Section 5.2. Figure 4 shows the states and their transitions; the transitions are discussed below.

一个Raft集群包含几个服务器；五个是典型的数字，这使得系统可以容忍两个故障服务器。在任何时候，每个服务器都处于三种状态之一：领导者、追随者或候选者。在正常操作中，有且仅有一个领导者，其他所有的服务器都是追随者。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端的请求（如果客户端联系追随者，追随者将其重定向到领导者）。第三种状态，候选人，被用来选举一个新的领导者，如第5.2节所述。图4显示了这些状态和它们的转换；下面将讨论这些转换。

Raft divides time into *terms* of arbitrary length, as shown in Figure 5. Terms are numbered with consecutive integers. Each term begins with an *election*, in which one or more candidates attempt to become leader as described in Section 5.2. If a candidate wins the election, then it serves as leader for the rest of the term. In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election) will begin shortly. Raft ensures that there is at most one leader in a given term.

如图5所示，Raft将时间划分为任意长度的任期。任期用连续的整数来编号。每个任期以选举开始, 其中一个或多个候选人试图成为领导者，如第5.2节所述。如果一个候选人在选举中获胜，那么他将在剩下的任期内担任领导者。在某些情况下，选举的结果是分裂票。在这种情况下，任期结束时将没有领导者；新的任期（新的选举）将很快开始。Raft确保在一个给定的任期内最多有一个领导者。

Different servers may observe the transitions between terms at different times, and in some situations a server may not observe an election or even entire terms. Terms act as a logical clock [14] in Raft, and they allow servers to detect obsolete information such as stale leaders. Each server stores a *current term* number, which increases monotonically over time. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other's, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state. If a server receives a request with a stale term number, it rejects the request.

不同的服务器可能会在不同的时间观察任期之间的转换，在某些情况下，一个服务器可能不会观察一个选举甚至整个任期。任期在Raft中充当了逻辑时钟[14]，它们允许服务器检测过时的信息，如过时的领导者。每个服务器都存储一个当前的任期编号，该编号随时间单调地增加。每当服务器进行通信时，就会交换当前任期；如果一个服务器的当前任期比另一个服务器的小，那么它就会将其当前任期更新为较大的值。如果一个候选者或领导者发现它的任期已经过时，它将立即恢复到跟随者状态。如果一个服务器收到的请求是一个过时的任期编号，它将拒绝该请求。

Raft servers communicate using remote procedure calls (RPCs), and the basic consensus algorithm requires only two types of RPCs. RequestVote RPCs are initiated by candidates during elections (Section 5.2), and Append Entries RPCs are initiated by leaders to replicate log entries and to provide a form of heartbeat (Section 5.3). Section 7 adds a third RPC for transferring snapshots between servers. Servers retry RPCs if they do not receive a response in a timely manner, and they issue RPCs in parallel for best performance.

Raft服务器使用远程过程调用（RPCs）进行通信，基本的共识算法只需要两种类型的RPCs。RequestVote RPCs由候选人在选举期间发起（第5.2节），Append Entries RPCs由领导者发起，用于复制日志条目并提供一种心跳形式(第5.3节）。第7节增加了第三个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，它们会重试RPC，并且为了获得最佳性能，它们会并行地发出RPC。

5.2 Leader election

Raft uses a heartbeat mechanism to trigger leader election. When servers start up, they begin as followers. A server remains in follower state as long as it receives valid RPCs from a leader or candidate. Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the *election timeout*, then it assumes there is no viable leader and begins an election to choose a new leader.

Raft使用心跳机制来触发领导者选举。当服务器启动时，它们开始是追随者。只要服务器收到来自领导者或候选人的有效RPC，它就一直处于跟随者状态。领导者定期向所有追随者发送心跳（AppendEntries RPCs，不携带任何日志条目），以保持他们的权威。如果追随者在一段被称为选举超时的时间内没有收到任何通信，那么它就认为没有可行的领导者，并开始选举以选择一个新的领导者。

To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster. A candidate continues in this state until one of three things happens: (a) it wins the election, (b) another server establishes itself as leader, or (c) a period of time goes by with no winner. These outcomes are discussed separately in the paragraphs below.

为了开始选举，追随者增加它的当前任期并过渡到候选状态。然后，它为自己投票，并向集群中的每个其他服务器发出RequestVote RPCs。候选者继续处于这种状态，直到发生三种情况之一。(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)一段时间内没有赢家。这些结果将在下面的段落中分别讨论。

A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis (note: Section 5.4 adds an additional restriction on votes). The majority rule ensures that at most one candidate can win the election for a particular term (the Election Safety Property in Figure 3). Once a candidate wins an election, it becomes leader. It then sends heartbeat messages to all of the other servers to establish its authority and prevent new elections.

如果一个候选人在同一任期内获得了整个集群中大多数服务器的投票，那么它就赢得了选举。每台服务器在给定的任期内最多为一名候选人投票，以先来后到为原则（注：第5.4节对投票增加了一个额外的限制）。少数服从多数的原则保证了最多只有一名候选人能够在某一任期内赢得选举（图3中的选举安全属性）。一旦一个候选人在选举中获胜，它就成为领导者。然后，它向所有其他服务器发送心跳信息，以建立其权威并防止新的选举。

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader’s term (included in its RPC) is at least as large as the candidate’s current term, then the candidate recognizes the leader as legitimate and returns to follower state. If the term in the RPC is smaller than the candidate’s current term, then the candidate rejects the RPC and continues in candidate state.

在等待投票的过程中，候选人可能会收到另一个服务器的AppendEntries RPC，声称自己是领导者。如果领导者的任期（包括在其RPC中）至少与候选人的当前任期一样大，那么候选人就会承认领导者是合法的，并返回到跟随者状态。如果RPC中的任期比候选者当前的任期小，那么候选者拒绝RPC，继续处于候选状态。

The third possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election by incrementing its term and initiating another round of Request Vote RPCs. However, without extra measures split votes could repeat indefinitely.

第三个可能的结果是，一个候选人既没有赢得选举，也没有失去选举：如果许多追随者同时成为候选人，选票可能被分割，因此没有候选人获得多数。当这种情况发生时，每个候选人都会超时，并通过增加其任期和启动新一轮的请求投票RPC来开始新的选举。然而，如果没有额外的措施，分裂的投票可能会无限期地重复。

Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150–300ms). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out. The same mechanism is used to handle split votes. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election. Section 9.3 shows that this approach elects a leader rapidly.

Raft使用随机的选举超时，以确保分裂投票很少发生，并能迅速解决。为了从一开始就防止分裂投票，选举超时是从一个固定的时间间隔中随机选择的（例如，150-300ms）。这就分散了服务器，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信号。同样的机制被用来处理分裂投票。每个候选人在选举开始时重新启动其随机的选举超时，并等待超时过后再开始下一次选举；这减少了在新的选举中再次出现分裂票的可能性。第9.3节显示，这种方法可以迅速选出一个领导者。

Elections are an example of how understandability guided our choice between design alternatives. Initially we planned to use a ranking system: each candidate was assigned a unique rank, which was used to select between competing candidates. If a candidate discovered another candidate with higher rank, it would return to follower state so that the higher ranking candidate could more easily win the next election. We found that this approach created subtle issues around availability (a lower-ranked server might need to time out and become a candidate again if a higher-ranked server fails, but if it does so too soon, it can reset progress towards electing a leader). We made adjustments to the algorithm several times, but after each adjustment new corner cases appeared. Eventually we concluded that the randomized retry approach is more obvious and understandable.

选举是一个例子，说明可理解性是如何指导我们在设计方案之间进行选择的。最初我们计划使用一个排名系统：每个候选人被分配一个独特的排名，用来在竞争的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，它就会回到追随者的状态，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一些微妙的问题（如果一个排名较高的服务器失败了，一个排名较低的服务器可能需要超时并再次成为候选人，但如果它过早地这样做，它可能会重置选举领导者的进展）。我们对算法进行了多次调整，但每次调整后都会出现新的边界案例。最终我们得出结论，随机重试的方法更加明显和容易理解。

5.3 Log replication

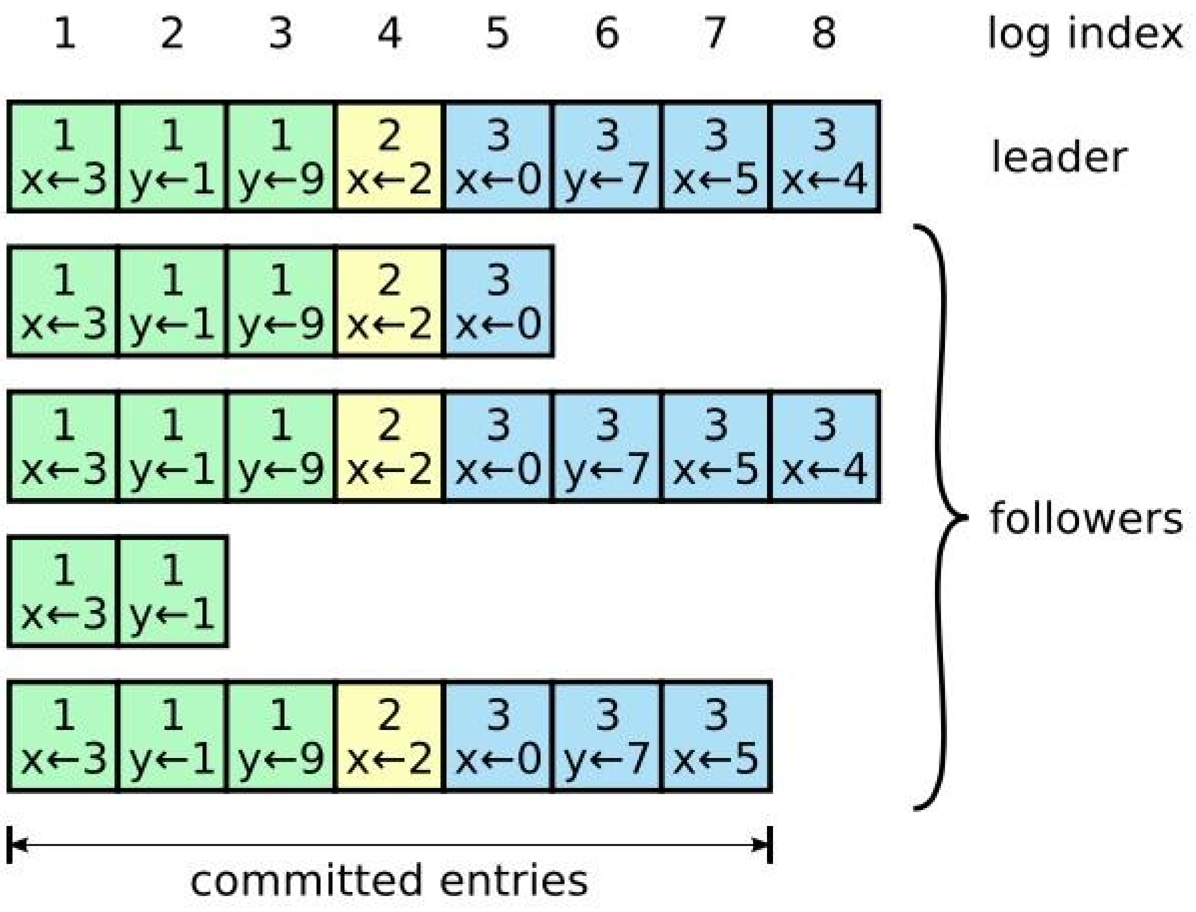

1 2 3 4 5 6 7 8
log index
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
leader
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
1 x←3 | 1 y←1
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5
followers
committed entries

Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.
图6：日志是由条目组成的，这些条目按顺序编号。每个条目都包含创建它的任期（每个方框中的数字）和状态机的命令。如果一个条目可以安全地应用于状态机，那么该条目就被认为是承诺的。

Once a leader has been elected, it begins servicing client requests. Each client request contains a command to be executed by the replicated state machines. The leader appends the command to its log as a new entry, then it uses AppendEntries RPCs in parallel to each of the other servers to replicate the entry. When the entry has been safely replicated (as described below), the leader applies the entry to its state machine and returns the result of that execution to the client. If followers crash or run slowly, or if network packets are lost, the leader retries Append Entries RPCs indefinitely (even after it has responded to the client) until all followers eventually store all log entries.

一旦一个领导者被选出，它就开始为客户端请求提供服务。每个客户端请求都包含一个要由复制的状态机执行的命令。领导者将命令作为一个新的条目附加到它的日志中，然后它使用AppendEntries RPCs平行于其他每个服务器来复制该条目。当条目被安全复制后（如下所述），领导者将条目应用于其状态机，并将执行结果返回给客户端。如果跟随者崩溃或运行缓慢，或者网络数据包丢失，领导者会无限期地重试Append Entries RPCs（甚至在它响应了客户端之后），直到所有跟随者最终存储所有的日志条目。

Logs are organized as shown in Figure 6. Each log entry stores a state machine command along with the term number when the entry was received by the leader. The term numbers in log entries are used to detect inconsistencies between logs and to ensure some of the properties in Figure 3. Each log entry also has an integer index identifying its position in the log.

日志的组织方式如图6所示。每个日志条目都存储了一个状态机命令，以及领导者收到该条目时的任期编号。日志条目中的任期编号被用来检测日志之间的不一致，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，用于识别它在日志中的位置。

The leader decides when it is safe to apply a log entry to the state machines; such an entry is called *committed*. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines. A log entry is committed once the leader that created the entry has replicated it on a majority of the servers (e.g., entry 7 in Figure 6). This also commits all preceding entries in the leader's log, including entries created by previous leaders. Section 5.4 discusses some subtleties when applying this rule after leader changes, and it also shows that this definition of commitment is safe. The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out. Once a follower learns that a log entry is committed, it applies the entry to its local state machine (in log order).

领导决定何时将日志条目应用于状态机是安全的；这样的条目被称为承诺。Raft保证所提交的条目是持久的，最终会被所有可用的状态机执行。一旦创建该条目的领导者将其复制到大多数服务器上，该日志条目就会被提交（例如，图6中的条目7）。这也会提交领导者日志中所有之前的条目，包括之前领导者创建的条目。第5.4节讨论了在领导者变更后应用这一规则时的一些微妙之处，它还表明这种承诺的定义是安全的。领导者会跟踪它所知道的已承诺的最高索引，并且它在未来的AppendEntries RPC（包括心跳）中包括该索引，以便其他服务器最终获知。一旦跟随者得知一个日志条目被提交，它就会将该条目应用到它的本地状态机（按日志顺序）。

We designed the Raft log mechanism to maintain a high level of coherency between the logs on different servers. Not only does this simplify the system's behavior and make it more predictable, but it is an important component of ensuring safety. Raft maintains the following properties, which together constitute the Log Matching Property in Figure 3:

我们设计的Raft日志机制在不同服务器上的日志之间保持高度的一致性。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，它们共同构成了图3中的日志匹配属性：

• If two entries in different logs have the same index and term, then they store the same command.
如果不同日志中的两个条目具有相同的索引和任期，那么它们存储的是同一个命令。

• If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.
如果不同日志中的两个条目具有相同的索引和任期，那么日志中的所有前面的条目都是相同的。

The first property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log. The second property is guaranteed by a simple consistency check performed by AppendEntries. When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries. If the follower does not find an entry in its log with the same index and term, then it refuses the new entries. The consistency check acts as an induction step: the initial empty state of the logs satisfies the Log Matching Property, and the consistency check preserves the Log Matching Property whenever logs are extended. As a result, whenever AppendEntries returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries.

第一个属性来自于这样一个事实，即一个领导者在一个给定的任期中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来保证。当发送AppendEntries RPC时，领导者包括其日志中紧接新条目之前的条目的索引和任期。如果跟随者在其日志中没有找到具有相同索引和任期的条目，那么它将拒绝新条目。一致性检查作为一个归纳步骤：日志的初始空状态满足了日志匹配属性，并且每当日志被扩展时，一致性检查都会保留日志匹配属性。因此，每当AppendEntries成功返回时，领导者知道跟随者的日志与自己的日志在新条目之前是相同的。

During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails. However, leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all of the entries in its log). These inconsistencies can compound over a series of leader and follower crashes. Figure 7 illustrates the ways in which followers' logs may differ from that of a new leader. A follower may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both. Missing and extraneous entries in a log may span multiple terms.

在正常运行期间，领导者和追随者的日志保持一致，所以AppendEntries一致性检查从未失败。然而，领导者崩溃会使日志不一致（老领导者可能没有完全复制其日志中的所有条目）。这些不一致会在一系列领导者和追随者的崩溃中加剧。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。跟随者可能会丢失领导者的条目，可能会有领导者没有的额外条目，或者两者都有。日志中的缺失和不相干的条目可能跨越多个任期。

In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log. Section 5.4 will show that this is safe when coupled with one more restriction.

在Raft中，领导者通过强迫追随者的日志重复自己的日志来处理不一致的情况。这意味着追随者日志中的冲突条目将被领导者日志中的条目覆盖。第5.4节将表明，如果再加上一个限制，这就是安全的。

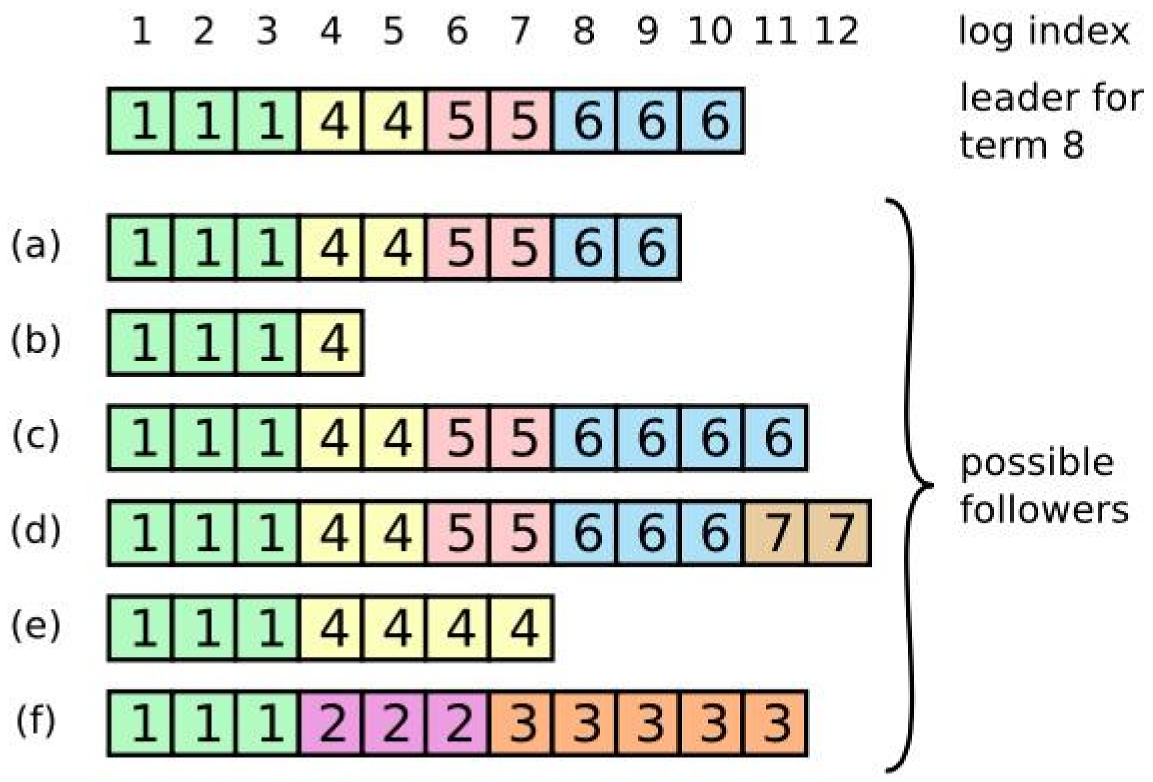


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

图7：当最上方的领导者掌权时，在追随者的日志中可能会出现（a-f）的任何一种情况。每个盒子代表一个日志条目；盒子里的数字是它的任期。一个追随者可能缺少条目（a-b），可能有额外的未承诺的条目（c-d），或者两者都有（e-f）。例如，场景f会发生在如果该服务器是第2期的领导者，在其日志中增加了几个条目，然后在提交任何条目之前就崩溃了；它很快重新启动，成为第3期的领导者，并在其日志中增加了几个条目；在第2期或第3期的任何条目被提交之前，该服务器再次崩溃，并持续了几个任期。

To bring a follower’s log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower’s log after that point, and send the follower all of the leader’s entries after that point. All of these actions happen in response to the consistency check performed by AppendEntries RPCs. The leader maintains a *nextIndex* for each follower, which is the index of the next log entry the leader will send to that follower. When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log (11 in Figure 7). If a follower’s log is inconsistent with the leader’s, the AppendEntries consistency check will fail in the next AppendEntries RPC. After a rejection, the leader decrements nextIndex and retries the AppendEntries RPC. Eventually nextIndex will reach a point where the leader and follower logs match. When this happens, AppendEntries will succeed, which removes any conflicting entries in the follower’s log and appends entries from the leader’s log (if any). Once AppendEntries succeeds, the follower’s log is consistent with the leader’s, and it will remain that way for the rest of the term.

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，删除追随者日志中该点之后的任何条目，并将该点之后领导者的所有条目发送给追随者。所有这些动作都是为了响应AppendEntries RPC所进行的一致性检查而发生的。领导为每个跟随者维护一个nextIndex，它是领导将发送给该跟随者的下一个日志条目的索引。当领导者第一次上台时，它将所有的nextIndex值初始化为其日志中最后一条的索引（图7中的11）。如果跟随者的日志与领导者的日志不一致，在下一个AppendEntries RPC中，AppendEntries一致性检查将失败。在拒绝之后，领导者会递减nextIndex并重试AppendEntries RPC。最终，nextIndex将达到一个领导者和追随者日志匹配的点。当这种情况发生时，AppendEntries就会成功，它将删除跟随者日志中任何冲突的条目，并追加领导者日志中的条目（如果有的话）。一旦AppendEntries成功，追随者的日志就与领导者的日志一致了，并且在剩下的任期里，它将保持这种状态。

If desired, the protocol can be optimized to reduce the number of rejected AppendEntries RPCs. For example, when rejecting an AppendEntries request, the follower can include the term of the conflicting entry and the first index it stores for that term. With this information, the leader can decrement nextIndex to bypass all of the conflicting entries in that term; one AppendEntries RPC will be required for each term with conflicting entries, rather than one RPC per entry. In practice, we doubt this optimization is necessary, since failures happen infrequently and it is unlikely that there will be many inconsistent entries.
如果需要，该协议可以被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，跟随者可以包括冲突条目的任期和它为该任期存储的第一个索引。有了这些信息，领导者可以递减nextIndex以绕过该任期中的所有冲突条目；每个有冲突条目的任期将需要一个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有必要，因为故障不常发生，而且不太可能有很多不一致的条目。

With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the Append Entries consistency check. A leader never overwrites or deletes entries in its own log (the Leader Append-Only Property in Figure 3).
有了这种机制，领导者在掌权时不需要采取任何特别的行动来恢复日志的一致性。它只是开始正常的操作，而日志会自动收敛以应对Append Entries一致性检查的失败。一个领导者从不覆盖或删除自己日志中的条目（图3中的 "Leader Append-Only"属性）。

This log replication mechanism exhibits the desirable consensus properties described in Section 2: Raft can accept, replicate, and apply new log entries as long as a majority of the servers are up; in the normal case a new entry can be replicated with a single round of RPCs to a majority of the cluster; and a single slow follower will not impact performance.
这种日志复制机制表现出第2节中所描述的理想的共识属性：只要大多数服务器是正常的，Raft就可以接受、复制和应用新的日志条目；在正常情况下，一个新的条目可以通过单轮RPC复制到集群的大多数；而且一个缓慢的跟随者不会影响性能。

5.4 Safety

The previous sections described how Raft elects leaders and replicates log entries. However, the mechanisms described so far are not quite sufficient to ensure that each state machine executes exactly the same commands in the same order. For example, a follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences.
前面的章节描述了Raft如何选举领导和复制日志条目。然而，到目前为止所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，当领导者提交几个日志条目时，一个跟随者可能无法使用，然后它可能被选为领导者，并用新的条目覆盖这些条目；结果，不同的状态机可能执行不同的命令序列。

This section completes the Raft algorithm by adding a restriction on which servers may be elected leader. The restriction ensures that the leader for any given term contains all of the entries committed in previous terms (the Leader Completeness Property from Figure 3). Given the election restriction, we then make the rules for commitment more precise. Finally, we present a proof sketch for the Leader Completeness Property and show how it leads to correct behavior of the replicated state machine.

本节对Raft算法进行了完善，增加了对哪些服务器可以被选为领导者的限制条件。该限制确保了任何给定任期的领导者都包含了之前任期中承诺的所有条目（图3中的Leader Completeness属性）。考虑到选举限制，我们会使承诺的规则更加精确。最后，我们提出一个Leader Completeness属性的证明草图，并说明它如何导致副本状态机的正确行为。

5.4.1 Election restriction

In any leader-based consensus algorithm, the leader must eventually store all of the committed log entries. In some consensus algorithms, such as Viewstamped Replication [22], a leader can be elected even if it doesn't initially contain all of the committed entries. These algorithms contain additional mechanisms to identify the missing entries and transmit them to the new leader, either during the election process or shortly afterwards. Unfortunately, this results in considerable additional mechanism and complexity. Raft uses a simpler approach where it guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader. This means that log entries only flow in one direction, from leaders to followers, and leaders never overwrite existing entries in their logs.

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承诺的日志条目。在一些共识算法中，例如Viewstamped Replication[22], 即使最初不包含所有已承诺的条目，也可以选出一个领导者。这些算法包含额外的机制来识别缺失的条目，并在选举过程中或之后不久将它们传送给新的领导者。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，它保证从每一个新的领导者当选的那一刻起，以前的所有承诺条目都存在于其身上，而不需要将这些条目传输给领导者。这意味着日志条目只在一个方向流动，即从领导者到追随者，而且领导者永远不会覆盖他们日志中的现有条目。

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries. A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate's log is at least as up-to-date as any other log in that majority (where "up-to-date" is defined precisely below), then it will hold all the committed entries. The RequestVote RPC implements this restriction: the RPC includes information about the candidate's log, and the voter denies its vote if its own log is more up-to-date than that of the candidate.

Raft使用投票程序来防止候选人赢得选举，除非其日志包含所有承诺的条目。候选人必须与集群中的大多数人联系才能当选，这意味着每个已承诺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和该多数人中的任何其他日志一样是最新的（这里的 "最新 "在下面有精确的定义），那么它将包含所有承诺的条目。RequestVote RPC实现了这一限制：RPC包括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及时，则拒绝投票。

Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

Raft通过比较日志中最后条目的索引和任期来确定两个日志中哪一个是最新的。如果日志的最后条目有不同的任期，那么任期较晚的日志是最新的。如果日志以相同的任期结束，那么哪一个日

志的任期更长，哪一个就更新。

5.4.2 Committing entries from previous terms

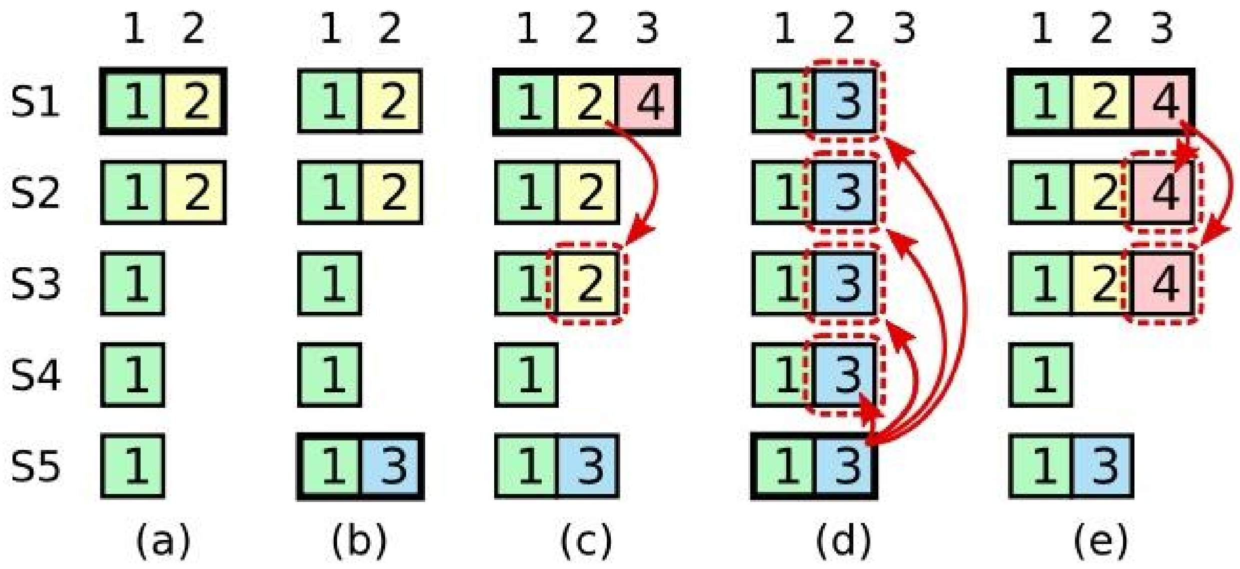


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

图8：一个时间序列显示了为什么领导者不能使用较早的任期的日志条目来确定承诺。在（a）中，S1是领导者，部分复制了索引2的日志条目。在(b)中，S1崩溃了；S5以S3、S4和它自己的投票当选为第三任期的领导者，并接受了日志索引2的不同条目。在（c）中，S5崩溃了；S1重新启动，被选为领导者，并继续复制。在这一点上，第2项的日志条目已经在大多数服务器上复制，但它没有被提交。如果S1像(d)那样崩溃，S5可以被选为领导者（有S2、S3和S4的投票），并用它自己的第3任期的条目覆盖该条目。然而，如果S1在崩溃前在大多数服务器上复制了其当前任期的条目，如(e)，那么这个条目就被承诺了（S5不能赢得选举）。在这一点上，日志中所有前面的条目也被提交。

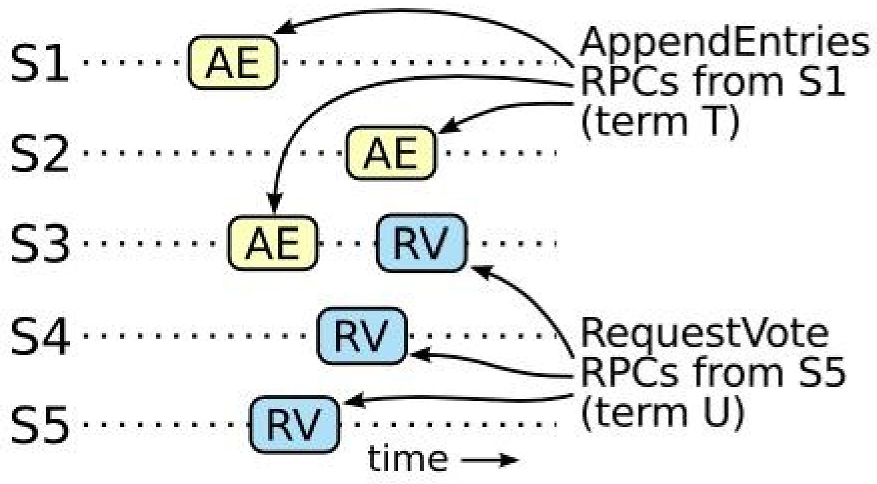

S1
S2
S3
S4
S5
AE
AE
AE
RV
RV
RV
AppendEntries RPCs from S1 (term T)
RequestVote RPCs from S5 (term U)
time

Figure 9: If S1 (leader for term T) commits a new log entry from its term, and S5 is elected leader for a later term U, then there must be at least one server (S3) that accepted the log entry and also voted for S5.
图9：如果S1（任期T的领导者）在其任期内提交了一个新的日志条目，而S5被选为后来任期U的领导者，那么至少有一个服务器（S3）接受了该日志条目，并且也为S5投票。

As described in Section 5.3, a leader knows that an entry from its current term is committed once that entry is stored on a majority of the servers. If a leader crashes before committing an entry, future leaders will attempt to finish replicating the entry. However, a leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers. Figure 8 illustrates a situation where an old log entry is stored on a majority of servers, yet can still be overwritten by a future leader.

如第5.3节所述，一旦一个条目被存储在大多数服务器上，领导者就知道其当前任期的条目被提交。如果一个领导者在提交条目之前崩溃了，未来的领导者将试图完成对该条目的复制。然而，领导者不能立即得出结论，一旦前一届的条目被存储在大多数服务器上，它就被提交了。图8说明了这样一种情况：一个旧的日志条目被存储在大多数服务器上，但仍然可以被未来的领导者所覆盖。

To eliminate problems like the one in Figure 8, Raft never commits log entries from previous terms by counting replicas. Only log entries from the leader's current term are committed by counting replicas; once an entry from the current term has been committed in this way, then all prior entries are committed indirectly because of the Log Matching Property. There are some situations where a leader could safely conclude that an older log entry is committed (for example, if that entry is stored on every server), but Raft takes a more conservative approach for simplicity.

为了消除类似图8中的问题，Raft从不通过计算副本数量来提交以前的日志条目。只有领导者当前任期的日志条目是通过计算副本数量来提交的；一旦当前任期的条目以这种方式被提交，那么由于日志匹配特性，所有之前的条目都被间接提交。在某些情况下，领导者可以安全地断定一个较早的日志条目已被提交（例如，如果该条目存储在每个服务器上），但Raft为了简单起见，采取了更保守的方法。

Raft incurs this extra complexity in the commitment rules because log entries retain their original term numbers when a leader replicates entries from previous terms. In other consensus algorithms, if a new leader re-replicates entries from prior "terms," it must do so with its new "term number." Raft's approach makes it easier to reason about log entries, since they maintain the same term number over time and across logs. In addition, new leaders in Raft send fewer log entries from previous terms than in other algorithms (other algorithms must send redundant log entries to renumber them before they can be committed).

Raft在承诺规则中产生了这种额外的复杂性，因为当领导者复制以前任期的条目时，日志条目会保留其原始任期编号。在其他共识算法中，如果一个新的领导者重新复制之前 "任期"中的条目，它必须用新的 "任期号 "来做。Raft的方法使得对日志条目的推理更加容易，因为它们在不同的时间和不同的日志中保持着相同的任期编号。此外，与其他算法相比，Raft的新领导从以前的任期中发送较少的日志条目（其他算法必须发送多余的日志条目，在它们被提交之前对其重新编号）。

5.4.3 Safety argument

Given the complete Raft algorithm, we can now argue more precisely that the Leader Completeness Property holds (this argument is based on the safety proof; see Section 9.2). We assume that the Leader Completeness Property does not hold, then we prove a contradiction. Suppose the leader for term T (leader$_T$) commits a log entry from its term, but that log entry is not stored by the leader of some future term. Consider the smallest term U > T whose leader (leader$_U$) does not store the entry.

基于完整的Raft算法，我们现在可以更精确地论证领导者完备性属性是否成立（这个论证是基于安全证明的，见第9.2节）。我们假设领导者完备性属性不成立，然后我们证明一个矛盾。假设任期T的领导者（leader$_T$）从其任期中提交了一个日志条目，但是这个日志条目没有被未来某个任期的领导者所存储。考虑最小的任期U > T，其领导者（leader$_U$）不存储该条目。

1. The committed entry must have been absent from leader$_U$'s log at the time of its election (leaders never delete or overwrite entries).

承诺的条目在leader$_U$当选时必须它不在的日志中（leader从不删除或改写条目）。

2. leader$_T$ replicated the entry on a majority of the cluster, and leader$_U$ received votes from a majority of the cluster. Thus, at least one server ("the voter") both accepted the entry from leader$_T$ and voted for leader$_U$, as shown in Figure 9. The voter is key to reaching a contradiction.

leader$_T$在集群的大多数服务器上复制了该条目，而领导者U从集群的大多数服务器上获得了投票。因此，至少有一台服务器（"投票者"）既接受了leader$_T$的条目，又投票给leader$_U$，如图9所示。这个投票者是达成矛盾的关键。

3. The voter must have accepted the committed entry from leader$_T$ *before* voting for leader$_U$; otherwise it would have rejected the AppendEntries request from leader$_T$ (its current term would have been higher than T).

投票者在投票给leader$_U$之前，必须接受leader$_T$的承诺条目；否则它就会拒绝leader$_T$的AppendEntries请求（它的当前任期会比T高）。

4. The voter still stored the entry when it voted for leader$_U$, since every intervening leader contained the entry (by assumption), leaders never remove entries, and followers only remove entries if they conflict with the leader.

投票者在投票给leader$_U$时仍然储存了该条目，因为每个介入的领导者都包含该条目（根据假设），领导者从不删除条目，而追随者只有在与领导者冲突时才会删除条目。

5. The voter granted its vote to leader$_U$, so leader$_U$'s log must have been as up-to-date as the voter's. This leads to one of two contradictions.

投票者将自己的选票投给leader$_U$，所以leader$_U$的日志肯定和选民的一样是最新的。这导致了两个矛盾中的一个。

6. First, if the voter and leader$_U$ shared the same last log term, then leader$_U$'s log must have been at least

as long as the voter's, so its log contained every entry in the voter's log. This is a contradiction, since the voter contained the committed entry and leader$_U$ was assumed not to.

首先，如果投票者和leader$_U$共享最后一个日志项，那么leader$_U$的日志至少要和投票者的一样长，所以它的日志包含了投票者日志中的每一个条目。这是一个矛盾，因为投票者包含了已承诺的条目，而leader$_U$被认为不包含。

7. Otherwise, leader$_U$'s last log term must have been larger than the voter's. Moreover, it was larger than T, since the voter's last log term was at least T (it contains the committed entry from term T). The earlier leader that created leader$_U$'s last log entry must have contained the committed entry in its log (by assumption). Then, by the Log Matching Property, leader$_U$'s log must also contain the committed entry, which is a contradiction.

否则，leader$_U$的最后一个日志的任期一定比投票者的大。而且，该任期比T大，因为投票人的最后一个日志任期至少是T(它包含T项的承诺条目）。创建leader$_U$的最后一个日志的早期leader，其日志中一定包含了已承诺的条目（根据假设）。那么，根据日志匹配属性，leader$_U$的日志也必须包含已承诺的条目，这是个矛盾。

8. This completes the contradiction. Thus, the leaders of all terms greater than T must contain all entries from term T that are committed in term T.

这就完成了矛盾。因此，所有大于T的任期的领导必须包含任期T的所有条目，这些条目在任期T中被承诺。

9. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly, such as index 2 in Figure 8(d).

日志匹配属性保证了未来的领导也将包含间接承诺的条目，如图8(d)中的索引2。

Given the Leader Completeness Property, we can prove the State Machine Safety Property from Figure 3, which states that if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. At the time a server applies a log entry to its state machine, its log must be identical to the leader's log up through that entry and the entry must be committed. Now consider the lowest term in which any server applies a given log index; the Log Completeness Property guarantees that the leaders for all higher terms will store that same log entry, so servers that apply the index in later terms will apply the same value. Thus, the State Machine Safety Property holds.

基于领导者完备性属性，我们可以证明图3中的状态机安全属性，即如果一个服务器在其状态机上应用了一个给定索引的日志条目，那么没有其他服务器会在同一索引上应用一个不同的日志条目。当一个服务器在其状态机上应用一个日志条目时，直到该条目的日志必须与领导者的日志相同，并且该条目必须被提交。现在考虑任何服务器应用一个给定的日志索引的最低任期；日志完整性属性保证所有更高任期的领导者将存储相同的日志条目，因此在以后的任期中应用索引的服务器将应用相同的值。因此，状态机安全属性成立。

Finally, Raft requires servers to apply entries in log index order. Combined with the State Machine Safety Property, this means that all servers will apply exactly the same set of log entries to their state machines, in the same order.

最后，Raft要求服务器按照日志索引顺序应用条目。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将相同的日志条目集应用于其状态机。

5.5 Follower and candidate crashes

Until this point we have focused on leader failures. Follower and candidate crashes are much simpler to handle than leader crashes, and they are both handled in the same way. If a follower or candidate crashes, then future RequestVote and AppendEntries RPCs sent to it will fail. Raft handles these failures by retrying indefinitely; if the crashed server restarts, then the RPC will complete successfully. If a server crashes after completing an RPC but before responding, then it will receive the same RPC again after it restarts. Raft RPCs are idempotent, so this causes no harm. For example, if a follower receives an AppendEntries request that includes log entries already present in its log, it ignores those entries in the new request.

在这之前，我们一直专注于领导者的失败。跟随者和候选人的崩溃比领导者的崩溃要简单得多, 而且它们的处理方式都是一样的。如果一个追随者或候选人崩溃了，那么未来发送给它的RequestVote和AppendEntries RPC将会失败。Raft通过无限期地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后但在响应前崩溃，那么它将在重新启动后再次收到相同的RPC。Raft RPC是幂等的，所以这不会造成任何伤害。例如，如果一个跟随者收到一个AppendEntries请求，其中包括已经存在于其日志中的日志条目，那么它将忽略新请求中的这些条目。

5.6 Timing and availability

One of our requirements for Raft is that safety must not depend on timing: the system must not produce incorrect results just because some event happens more quickly or slowly than expected. However, availability (the ability of the system to respond to clients in a timely manner) must inevitably depend on timing. For example, if message exchanges take longer than the typical time between server crashes, candidates will not stay up long enough to win an election; without a steady leader, Raft cannot make progress.

我们对Raft的要求之一是安全性不能依赖于时间：系统不能因为某些事件发生得比预期快或慢而产生错误的结果。然而，可用性（系统及时响应客户的能力）必须不可避免地取决于时间。例如，如果消息交换的时间超过了服务器崩溃之间的典型时间，那么候选人就不会保持足够长的时间来赢得选举；没有一个稳定的领导者，Raft就无法取得进展。

Leader election is the aspect of Raft where timing is most critical. Raft will be able to elect and maintain a steady leader as long as the system satisfies the following *timing requirement*:
领袖选举是Raft中时间最关键的方面。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导者：

$$broadcastTime \ll electionTimeout \ll MTBF$$

In this inequality *broadcastTime* is the average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses; *electionTimeout* is the election timeout described in Section 5.2; and *MTBF* is the average time between failures for a single server. The broadcast time should be an order of magnitude less than the election timeout so that leaders can reliably send the heartbeat messages required to keep followers from starting elections; given the randomized approach used for election timeouts, this inequality also makes split votes unlikely. The election timeout should be a few orders of magnitude less than MTBF so that the system makes steady progress. When the leader crashes, the system will be unavailable for roughly the election time out; we would like this to represent only a small fraction of overall time.

在这个不等式中，broadcastTime是一台服务器向集群中的每台服务器并行发送RPC并接收其响应所需的平均时间；electionTimeout是第5.2节中描述的选举超时；MTBF是单台服务器的平均故障间隔时间。广播时间应该比选举超时少一个数量级，这样领导者就可以可靠地发送心跳信息，以防止追随者开始选举；考虑到选举超时使用的随机方法，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。当领导者崩溃时，系统将在大约选举超时的时间内不可用；我们希望这只占总体时间的一小部分。

The broadcast time and MTBF are properties of the underlying system, while the election timeout is something we must choose. Raft’s RPCs typically require the recipient to persist information to stable storage, so the broadcast time may range from 0.5ms to 20ms, depending on storage technology. As a result, the election timeout is likely to be somewhere between 10ms and 500ms. Typical server MTBFs are several months or more, which easily satisfies the timing requirement.

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通常要求接收者将信息持久化到稳定的存储中，所以广播时间可能在0.5ms到20ms之间，这取决于存储技术。因此，选举超时可能是在10ms到500ms之间。典型的服务器MTBF是几个月或更长时间，这很容易满足时间要求。

6 Cluster membership changes

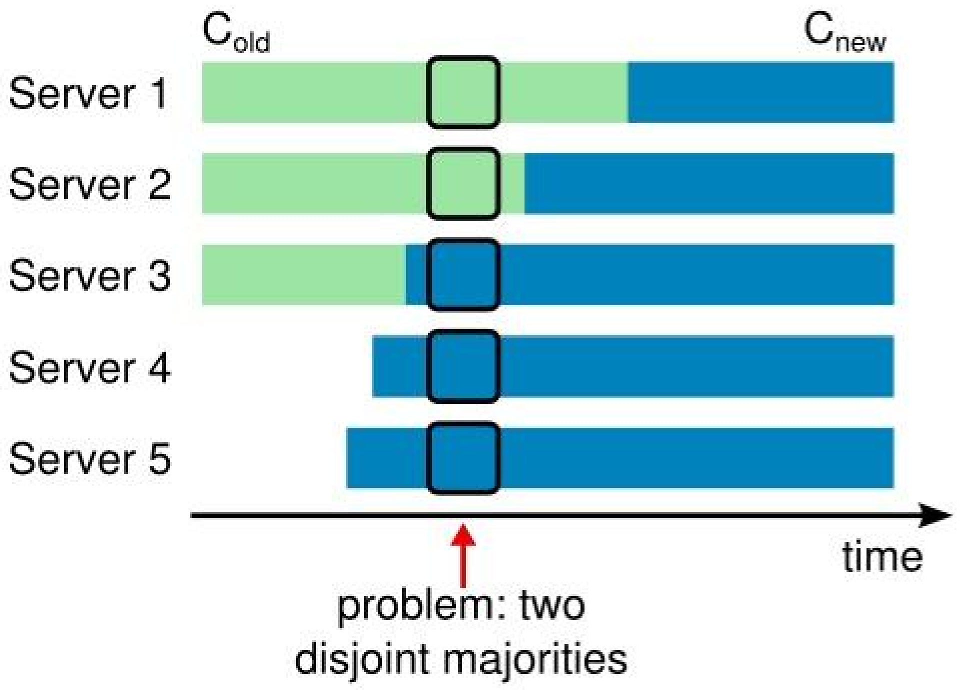


Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).
图10：直接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时间切换。在这个例子中，集群从三个服务器增长到五个。不幸的是，有一个时间点，两个不同的领导者可以在同一任期内当选，一个是旧配置的多数（C_{old}），另一个是新配置的多数（C_{new}）。

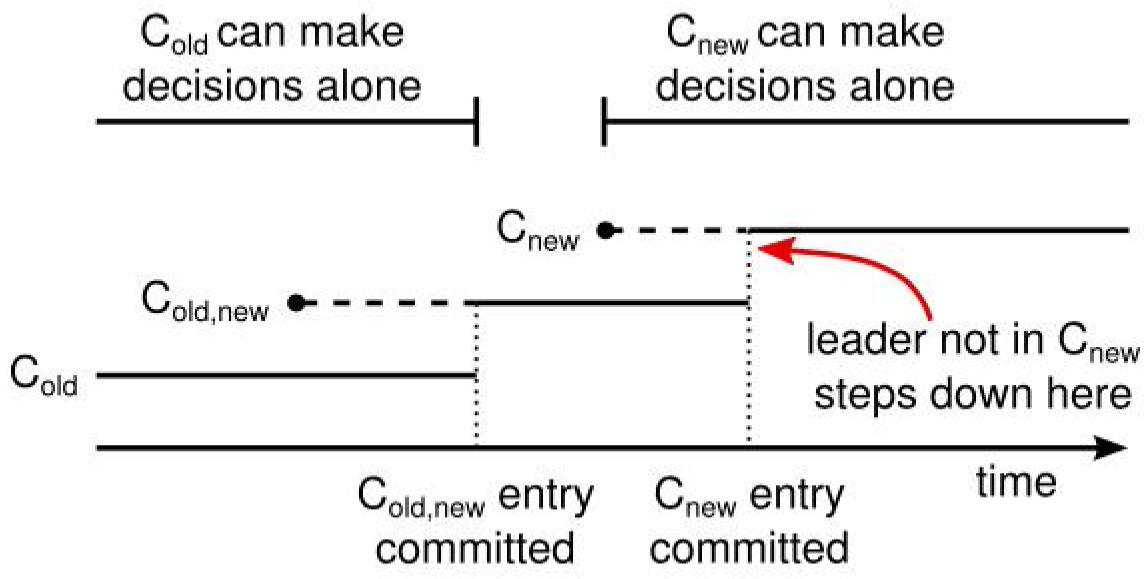

C_{old} can make decisions alone
C_{new} can make decisions alone
C_{new}
$C_{old,new}$
C_{old}
leader not in C_{new} steps down here
$C_{old,new}$ entry committed
C_{new} entry committed
time

Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a majority of C_{old} and a majority of C_{new}). Then it creates the C_{new} entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.
图11：配置变更的时间线。虚线表示已经创建但未提交的配置条目，实线表示最新提交的配置条目。领导者首先在其日志中创建 $C_{old,new}$ 配置条目，并将其提交给 $C_{old,new}$（C_{old} 的大多数和 C_{new} 的大多数）。然后，它创建了C_{new} 条目，并将其提交给多数的C_{new}。在这个时间点上，C_{old} 和C_{new} 都不能独立做出决定。

Up until now we have assumed that the cluster *configuration* (the set of servers participating in the consensus algorithm) is fixed. In practice, it will occasionally be necessary to change the configuration, for example to replace servers when they fail or to change the degree of replication. Although this can be done by taking the entire cluster off-line, updating configuration files, and then restarting the cluster, this would leave the cluster unavailable during the changeover. In addition, if there are any manual steps, they risk operator error. In order to avoid these issues, we decided to automate configuration changes and incorporate them into the Raft consensus algorithm.
到目前为止，我们一直假设集群配置（参与共识算法的服务器集合）是固定的。在实践中，偶尔有必要改变配置，例如在服务器故障时更换服务器或改变复制的程度。虽然这可以通过将整个集群下线，更新配置文件，然后重新启动集群来完成，但这将使集群在转换期间不可用。此外，如果有任何手动步骤，就有可能出现操作错误。为了避免这些问题，我们决定将配置变更自动化，并将其纳入Raft共识算法中。

For the configuration change mechanism to be safe, there must be no point during the transition where it is possible for two leaders to be elected for the same term. Unfortunately, any approach where servers switch directly from the old configuration to the new configuration is unsafe. It isn't possible to atomically switch all of the servers at once, so the cluster can potentially split into two independent majorities during the transition (see Figure 10).
为了使配置变化机制安全，在过渡期间必须没有任何一点可以让两个领导人当选同一任期的情况。不幸的是，任何服务器直接从旧配置切换到新配置的方法都是不安全的。不可能一次性地切换

所有的服务器，所以集群有可能在过渡期间分裂成两个独立的多数派（见图10）。

In order to ensure safety, configuration changes must use a two-phase approach. There are a variety of ways to implement the two phases. For example, some systems (e.g., [22]) use the first phase to disable the old configuration so it cannot process client requests; then the second phase enables the new configuration. In Raft the cluster first switches to a transitional configuration we call *joint consensus*; once the joint consensus has been committed, the system then transitions to the new configuration. The joint consensus combines both the old and new configurations:
为了确保安全，配置变更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（如[22]）使用第一阶段禁用旧配置，使其无法处理客户请求；然后第二阶段启用新配置。在Raft中，集群首先切换到一个过渡性配置，我们称之为联合共识；一旦联合共识被提交，系统就会过渡到新的配置。联合共识结合了新旧两种配置：

• Log entries are replicated to all servers in both configurations.
日志条目被复制到两个配置中的所有服务器。

• Any server from either configuration may serve as leader.
两种配置中的任何一个服务器都可以作为领导者

• Agreement (for elections and entry commitment) requires separate majorities from *both* the old and new configurations.
达成协议（选举和日志条目承诺）需要新旧两个组合分别获得多数。

The joint consensus allows individual servers to transition between configurations at different times without compromising safety. Furthermore, joint consensus allows the cluster to continue servicing client requests throughout the configuration change.
联合共识允许单个服务器在不同时间在配置之间转换，而不影响安全。此外，联合共识允许集群在整个配置变化过程中继续为客户端请求提供服务。

Cluster configurations are stored and communicated using special entries in the replicated log; Figure 11 illustrates the configuration change process. When the leader receives a request to change the configuration from C_{old} to C_{new}, it stores the configuration for joint consensus ($C_{old,new}$ in the figure) as a log entry and replicates that entry using the mechanisms described previously. Once a given server adds the new configuration entry to its log, it uses that configuration for all future decisions (a server always uses the latest configuration in its log, regardless of whether the entry is committed). This means that the leader will use the rules of $C_{old,new}$ to determine when the log entry for $C_{old,new}$ is committed. If the leader crashes, a new leader may be chosen under either C_{old} or $C_{old,new}$, depending on whether the winning candidate has received $C_{old,new}$. In any case, C_{new} cannot make unilateral decisions during this period.
集群配置是通过复制日志中的特殊条目来存储和通信的；图11说明了配置改变过程。当领导者收到将配置从C_{old}改为C_{new}的请求时，它将联合共识的配置（图中的$C_{old,new}$）存储为一个日志条目，并使用之前描述的机制复制该条目。一旦某个服务器将新的配置条目添加到其日志中，它就会将该配置用于所有未来的决策（一个服务器总是使用其日志中的最新配置，无论该条目是否被提交）。这意味着领导者将使用 $C_{old,new}$ 的规则来决定 $C_{old,new}$ 的日志条目何时被提交。如果领导者崩溃了，一个新的领导者可能会在C_{old} 或$C_{old,new}$ 下被选择，这取决于获胜的候选人是否已经收到了$C_{old,new}$。在任何情况下，C_{new} 都不能在这期间做出单方面的决定。

Once $C_{old,new}$ has been committed, neither C_{old} nor C_{new} can make decisions without approval of the other, and the Leader Completeness Property ensures that only servers with the $C_{old,new}$ log entry can be elected as leader. It is now safe for the leader to create a log entry describing C_{new} and replicate it to the cluster. Again, this configuration will take effect on each server as soon as it is seen. When the new configuration has been committed under the rules of C_{new}, the old configuration is irrelevant and servers not in the new configuration can be shut down. As shown in Figure 11, there is no time when C_{old} and C_{new} can both make unilateral decisions; this guarantees safety.

一旦$C_{old,new}$被提交，C_{old}和C_{new}都不能在未经对方批准的情况下做出决定，而且领导者完整性属性确保只有拥有$C_{old,new}$日志条目的服务器才能被选为领导者。现在，领导者创建描述C_{new}的日志条目并将其复制到集群中是安全的。同样，这个配置一旦被看到，就会在每个服务器上生效。当新的配置在C_{new}的规则下被提交后，旧的配置就不重要了，不在新配置中的服务器可以被关闭。如图11所示，没有任何时候C_{old}和C_{new}可以同时做出单边决定；这保证了安全。

There are three more issues to address for reconfiguration. The first issue is that new servers may not initially store any log entries. If they are added to the cluster in this state, it could take quite a while for them to catch up, during which time it might not be possible to commit new log entries. In order to avoid availability gaps, Raft introduces an additional phase before the configuration change, in which the new servers join the cluster as non-voting members (the leader replicates log entries to them, but they are not considered for majorities). Once the new servers have caught up with the rest of the cluster, the reconfiguration can proceed as described above.

对于重新配置，还有三个问题需要解决。第一个问题是，新的服务器最初可能不会存储任何日志条目。如果它们在这种状态下被添加到集群中，可能需要相当长的时间才能赶上，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在配置改变之前引入了一个额外的阶段，在这个阶段，新的服务器作为非投票成员加入集群（领导者将日志条目复制给他们，但他们不被考虑为多数）。一旦新的服务器赶上了集群的其他部分，重新配置就可以按上述方法进行。

The second issue is that the cluster leader may not be part of the new configuration. In this case, the leader steps down (returns to follower state) once it has committed the C_{new} log entry. This means that there will be a period of time (while it is committing C_{new}) when the leader is managing a cluster that does not include itself; it replicates log entries but does not count itself in majorities. The leader transition occurs when C_{new} is committed because this is the first point when the new configuration can operate independently (it will always be possible to choose a leader from C_{new}). Before this point, it may be the case that only a server from C_{old} can be elected leader.

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，一旦它提交了C_{new}日志条目，领导者就会下台（返回到跟随者状态）。这意味着会有一段时间（在它提交C_{new}的时候），领导者在管理一个不包括自己的集群；它复制日志条目，但不把自己算在多数中。领导者过渡发生在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（它将始终有可能从C_{new}中选择一个领导者）。在这之前，可能只有C_{old}的一个服务器可以被选为领导者。

The third issue is that removed servers (those not in C_{new}) can disrupt the cluster. These servers will not receive heartbeats, so they will time out and start new elections. They will then send RequestVote RPCs with new term numbers, and this will cause the current leader to revert to follower state. A new leader will eventually be elected, but the removed servers will time out again and the process will repeat, resulting in poor availability.

第三个问题是，被移除的服务器（那些不在C_{new}中的服务器）会扰乱集群。这些服务器不会收到心跳，所以它们会超时并开始新的选举。然后他们会发送带有新任期编号的RequestVote RPCs，这将导致当前的领导者恢复到追随者状态。一个新的领导者最终将被选出，但被移除的服务器将再次超时，这个过程将重复，导致可用性差。

To prevent this problem, servers disregard RequestVote RPCs when they believe a current leader exists. Specifically, if a server receives a RequestVote RPC within the minimum election timeout of hearing from a current leader, it does not update its term or grant its vote. This does not affect normal elections, where each server waits at least a minimum election timeout before starting an election. However, it helps avoid disruptions from removed servers: if a leader is able to get heartbeats to its cluster, then it will not be

deposed by larger term numbers.
为了防止这个问题，当服务器认为存在一个当前的领导者时，它们就会忽略RequestVote RPCs。具体来说，如果一个服务器在听到当前领袖的最小选举超时内收到RequestVote RPC，它不会更新其任期或授予其投票。这并不影响正常的选举，每个服务器在开始选举之前至少要等待一个最小的选举超时。然而，这有助于避免被移除的服务器的干扰：如果一个领导者能够得到其集群的心跳，那么它就不会被较大的任期数字所废黜。

7 Log compaction

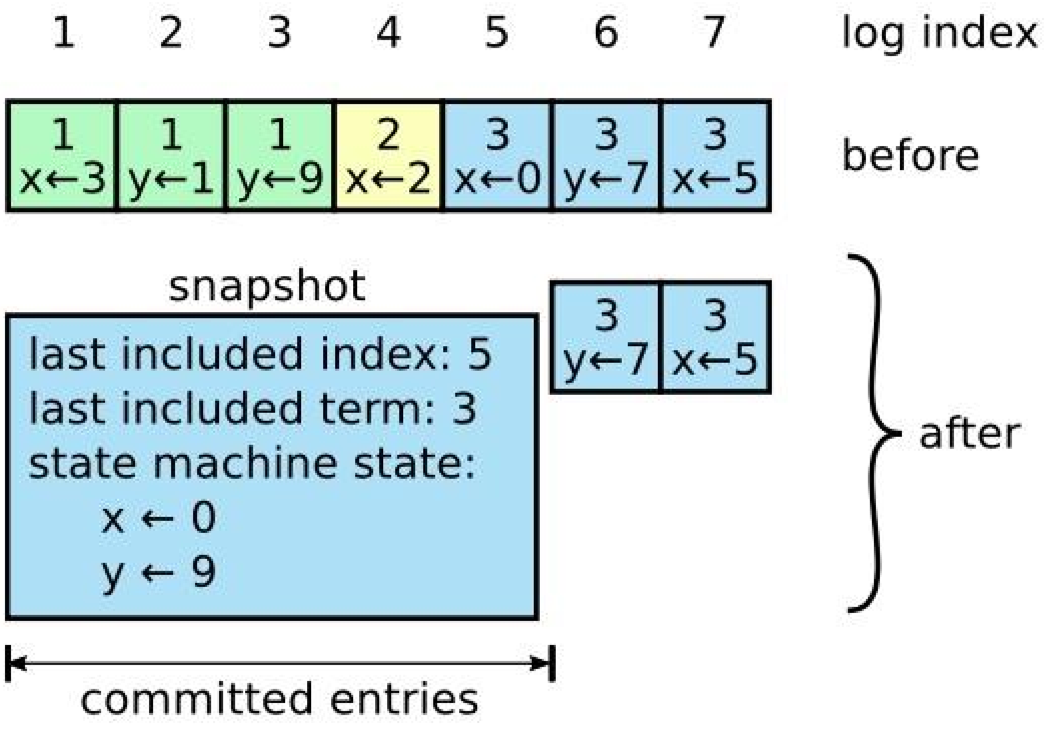


Figure 12: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables x and y in this example). The snap shot's last included index and term serve to position the snap shot in the log preceding entry 6.
图12：一个服务器用一个新的快照替换了其日志中的已提交条目（索引1到5），该快照只存储了当前状态（本例中的变量x和y）。快照最后包含的索引和任期用于定位快照在日志第6条之前的位置。

Raft's log grows during normal operation to incorporate more client requests, but in a practical system, it can not grow without bound. As the log grows longer, it occupies more space and takes more time to replay. This will eventually cause availability problems without some mechanism to discard obsolete information that has accumulated in the log.
Raft的日志在正常运行期间不断增长，以纳入更多的客户端请求，但在实际系统中，它不可能无限制地增长。随着日志的增长，它占据了更多的空间，需要更多的时间来重放。如果没有某种机制来丢弃积累在日志中的过时信息，这最终会导致可用性问题。

Snapshotting is the simplest approach to compaction. In snapshotting, the entire current system state is written to a *snapshot* on stable storage, then the entire log up to that point is discarded. Snapshotting is used in Chubby and ZooKeeper, and the remainder of this section describes snapshotting in Raft.
快照是最简单的压缩方法。在快照中，整个当前系统状态被写入稳定存储的快照中，然后丢弃截至该点的整个日志。快照在Chubby和ZooKeeper中使用，本节的其余部分将介绍Raft中的快照。

Incremental approaches to compaction, such as log cleaning [36] and log-structured merge trees [30, 5], are also possible. These operate on a fraction of the data at once, so they spread the load of compaction more evenly over time. They first select a region of data that has accumulated many deleted and overwritten objects, then they rewrite the live objects from that region more compactly and free the region. This requires significant additional mechanism and complexity compared to snapshot ting, which simplifies the problem by always operating on the entire data set. While log cleaning would require modifications to Raft, state machines can implement LSM trees using the same interface as snapshotting.

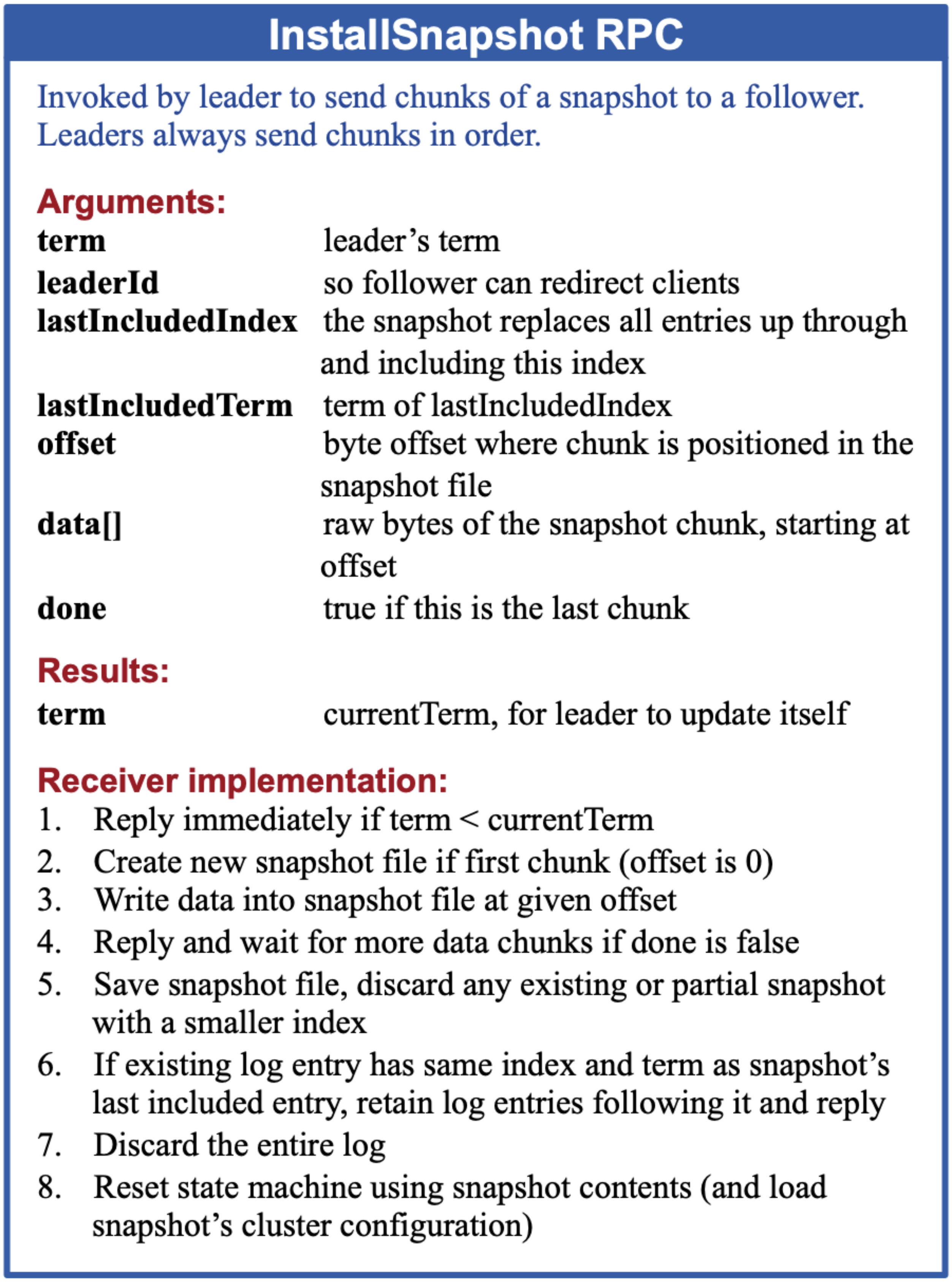
递增的压缩方法，如日志清理[36]和日志结构的合并树[30，5]，也是可能的。这些方法一次性对部分数据进行操作，因此它们将压缩的负荷更均匀地分散在时间上。它们首先选择一个积累了许多被删除和被覆盖的对象的数据区域，然后将该区域的活跃对象重写得更紧凑，并释放该区域。与快照处理相比，这需要大量的额外机制和复杂性，因为快照处理总是对整个数据集进行操作，从而简化了问题。虽然日志清理需要对Raft进行修改，但状态机可以使用与快照相同的接口实现LSM树。

Figure 12 shows the basic idea of snapshotting in Raft. Each server takes snapshots independently, covering just the committed entries in its log. Most of the work consists of the state machine writing its current state to the snapshot. Raft also includes a small amount of metadata in the snapshot: the *last included index* is the index of the last entry in the log that the snapshot replaces (the last entry the state machine had applied), and the *last included term* is the term of this entry. These are preserved to support the AppendEntries consistency check for the first log entry following the snapshot, since that entry needs a previous log index and term. To enable cluster membership changes (Section 6), the snapshot also includes the latest configuration in the log as of last included index. Once a server completes writing a snapshot, it may delete all log entries up through the last included index, as well as any prior snapshot.

图12显示了Raft中快照的基本思路。每台服务器独立进行快照，只覆盖其日志中已提交的条目。大部分的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：最后包含的索引是快照所取代的日志中最后一个条目的索引(状态机应用的最后一个条目），最后包含的任期是这个条目的任期。这些被保留下来是为了支持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任期。为了实现集群成员的变化(第6节），快照还包括日志中的最新配置，即最后包含的索引。一旦服务器完成写入快照，它可以删除所有的日志条目，直到最后包含的索引，以及任何先前的快照。

Although servers normally take snapshots independently, the leader must occasionally send snapshots to followers that lag behind. This happens when the leader has already discarded the next log entry that it needs to send to a follower. Fortunately, this situation is unlikely in normal operation: a follower that has kept up with the leader would already have this entry. However, an exceptionally slow follower or a new server joining the cluster (Section 6) would not. The way to bring such a follower up-to-date is for the leader to send it a snapshot over the network.

尽管服务器通常是独立进行快照，但领导者偶尔必须向落后的跟随者发送快照。这种情况发生在领导者已经丢弃了它需要发送给跟随者的下一个日志条目。幸运的是，这种情况在正常操作中不太可能发生：一个跟上领导者的追随者已经有了这个条目。然而，一个特别慢的跟随者或一个新加入集群的服务器（第6节）就不会有这样的情况。让这样的追随者跟上的方法是，领导者通过网络向它发送一个快照。



InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.
由领导者调用，向跟随者发送快照的分块。领导者总是按顺序发送分块。

Arguments:	
term	leader's term 领导人的任期号
leaderId 领导人的 Id	so follower can redirect clients 以便于跟随者重定向请求
lastIncludedIndex	the snapshot replaces all entries up through and including this index 快照会替换所有的条目，直到并包括这个索引
lastIncludedTerm	term of lastIncludedIndex 快照中包含的最后日志条目的任期号
offset	byte offset where chunk is positioned in the snapshot file 分块在快照文件中位置的字节偏移量
data[]	raw bytes of the snapshot chunk, starting at offset 快照分块的原始字节，从偏移量开始
done	true if this is the last chunk 如果这是最后一个分块则为 true

Results:	
term	currentTerm, for leader to update itself 当前任期号，便于领导人更新自己

Receiver implementation:
1. Reply immediately if term < currentTerm 如果term < currentTerm 就立即回复
2. Create new snapshot file if first chunk (offset is 0) 如果是第一个分块（offset 为 0）就创建一个新的快照
3. Write data into snapshot file at given offset 在指定偏移量写入数据
4. Reply and wait for more data chunks if done is false 如果 done == false，则回复并等待更多的数据
5. Save snapshot file, discard any existing or partial snapshot with a smaller index

保存快照文件，丢弃具有较小索引的已存或部分快照
6. If existing log entry has same index and term as snapshot’s last included entry, retain log entries following it and reply 如果现存的日志条目与快照中最后包含的日志条目具有相同的索引值和任期号，则保留其后的日志条目并进行回复
7. Discard the entire log 丢弃整个日志
8. Reset state machine using snapshot contents (and load snapshot’s cluster configuration) 使用快照内容重置状态机（并加载快照的集群配置）

Figure 13: A summary of the InstallSnapshot RPC. Snap shots are split into chunks for transmission; this gives the follower a sign of life with each chunk, so it can reset its election timer.
图13：InstallSnapshot RPC的摘要。快照被分割成若干块进行传输；这给追随者提供了每个块的生命迹象，所以它可以重置其选举计时器。

The leader uses a new RPC called InstallSnapshot to send snapshots to followers that are too far behind; see Figure 13. When a follower receives a snapshot with this RPC, it must decide what to do with its existing log entries. Usually the snapshot will contain new information not already in the recipient’s log. In this case, the follower discards its entire log; it is all superseded by the snapshot and may possibly have uncommitted entries that conflict with the snapshot. If instead the follower receives a snap shot that describes a prefix of its log (due to retransmission or by mistake), then log entries covered by the snapshot are deleted but entries following the snapshot are still valid and must be retained.
领导者使用一个叫做InstallSnapshot的新RPC来向落后于它的跟随者发送快照；见图13。当跟随者收到这个RPC的快照时，它必须决定如何处理其现有的日志条目。通常情况下，快照将包含新的信息，而不是在接收者的日志中。在这种情况下，跟随者会丢弃它的整个日志；它都被快照取代了，而且可能有与快照冲突的未提交的条目。相反，如果跟随者收到描述其日志前缀的快照（由于重传或错误），那么被快照覆盖的日志条目将被删除，但快照之后的条目仍然有效，必须保留。

This snapshotting approach departs from Raft’s strong leader principle, since followers can take snapshots without the knowledge of the leader. However, we think this departure is justified. While having a leader helps avoid conflicting decisions in reaching consensus, consensus has already been reached when snapshotting, so no decisions conflict. Data still only flows from leaders to followers, just followers can now reorganize their data.
这种快照方法偏离了Raft的强领导原则，因为跟随者可以在领导不知情的情况下进行快照。然而，我们认为这种背离是合理的。虽然有一个领导者有助于在达成共识时避免冲突的决定，但在快照时已经达成了共识，所以没有决定冲突。数据仍然只从领导者流向追随者，只是追随者现在可以重新组织他们的数据。

We considered an alternative leader-based approach in which only the leader would create a snapshot, then it would send this snapshot to each of its followers. However, this has two disadvantages. First, sending the snapshot to each follower would waste network bandwidth and slow the snapshotting process. Each follower already has the information needed to produce its own snapshots, and it is typically much cheaper for a server to produce a snapshot from its local state than it is to send and receive one over the network. Second, the leader’s implementation would be more complex. For example, the leader would need to send snapshots to followers in parallel with replicating new log entries to them, so as not to block

new client requests.
我们考虑了另一种基于领导者的方法，即只有领导者会创建一个快照，然后它将这个快照发送给其每个追随者。然而，这有两个缺点。首先，向每个追随者发送快照会浪费网络带宽，并减缓快照过程。每个追随者都已经拥有产生其自身快照所需的信息，而且对于服务器来说，从其本地状态产生快照通常比通过网络发送和接收快照要便宜得多。第二，领导者的实现将更加复杂。例如，领导者需要在向追随者复制新的日志条目的同时向他们发送快照，以便不阻碍新的客户请求。

There are two more issues that impact snapshotting performance. First, servers must decide when to snapshot. If a server snapshots too often, it wastes disk bandwidth and energy; if it snapshots too infrequently, it risks exhausting its storage capacity, and it increases the time required to replay the log during restarts. One simple strategy is to take a snapshot when the log reaches a fixed size in bytes. If this size is set to be significantly larger than the expected size of a snapshot, then the disk bandwidth overhead for snapshotting will be small.
还有两个影响快照性能的问题。首先，服务器必须决定何时进行快照。如果服务器快照的频率过高，就会浪费磁盘带宽和能源；如果快照的频率过低，就会有耗尽存储容量的风险，而且会增加重启时重放日志的时间。一个简单的策略是，当日志达到一个固定的字节大小时进行快照。如果这个大小被设定为明显大于快照的预期大小，那么快照的磁盘带宽开销就会很小。

The second performance issue is that writing a snapshot can take a significant amount of time, and we do not want this to delay normal operations. The solution is to use copy-on-write techniques so that new updates can be accepted without impacting the snapshot being written. For example, state machines built with functional data structures naturally support this. Alternatively, the operating system's copy-on-write support (e.g., fork on Linux) can be used to create an in-memory snapshot of the entire state machine (our implementation uses this approach).
第二个性能问题是，写入快照可能需要大量的时间，我们不希望因此而耽误正常的操作。解决方案是使用写时复制技术，这样就可以接受新的更新而不影响正在写入的快照。例如，用功能数据结构构建的状态机自然支持这一点。另外，操作系统的写时拷贝支持（例如Linux上的fork）可以用来创建整个状态机的内存快照（我们的实现采用了这种方法）。

8 Client interaction

This section describes how clients interact with Raft, including how clients find the cluster leader and how Raft supports linearizable semantics [10]. These issues apply to all consensus-based systems, and Raft's solutions are similar to other systems.
本节介绍了客户端如何与Raft互动，包括客户端如何找到集群领导者以及Raft如何支持可线性化语义[10]。这些问题适用于所有基于共识的系统，而且Raft的解决方案与其他系统类似。

Clients of Raft send all of their requests to the leader. When a client first starts up, it connects to a randomly chosen server. If the client's first choice is not the leader, that server will reject the client's request and supply information about the most recent leader it has heard from (AppendEntries requests include the network address of the leader). If the leader crashes, client requests will time out; clients then try again with randomly-chosen servers.
Raft的客户端将其所有的请求发送给领导者。当客户端第一次启动时，它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领导者，该服务器将拒绝客户端的请求，并提供它最近听到的领导者的信息（AppendEntries请求包括领导者的网络地址）。如果领导者崩溃了，客户端的请求就会超时；然后客户端会在随机选择的服务器上再次尝试。

Our goal for Raft is to implement linearizable semantics (each operation appears to execute instantaneously, exactly once, at some point between its invocation and its response). However, as described so far Raft can execute a command multiple times: for example, if the leader crashes after committing the log entry but before responding to the client, the client will retry the command with a new leader, causing it to be executed a second time. The solution is for clients to assign unique serial numbers to every command. Then, the state machine tracks the latest serial number processed for each client, along with the associated response. If it receives a command whose serial number has already been executed, it responds immediately without re-executing the request.

我们对Raft的目标是实现可线性化的语义（每个操作看起来都是瞬时执行的，在其调用和响应之间的某个点上正好一次）。然而，正如目前所描述的那样，Raft可以多次执行一个命令：例如，如果领导者在提交日志条目后但在响应客户端之前崩溃，客户端将用一个新的领导者重试该命令，导致它被第二次执行。解决方案是让客户端为每个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。如果它收到一个序列号已经被执行的命令，它会立即响应，而不重新执行该请求。

Read-only operations can be handled without writing anything into the log. However, with no additional measures, this would run the risk of returning stale data, since the leader responding to the request might have been superseded by a newer leader of which it is unaware. Linearizable reads must not return stale data, and Raft needs two extra precautions to guarantee this without using the log. First, a leader must have the latest information on which entries are committed. The Leader Completeness Property guarantees that a leader has all committed entries, but at the start of its term, it may not know which those are. To find out, it needs to commit an entry from its term. Raft handles this by having each leader commit a blank *no-op* entry into the log at the start of its term. Second, a leader must check whether it has been deposed before processing a read-only request (its information may be stale if a more recent leader has been elected). Raft handles this by having the leader exchange heartbeat messages with a majority of the cluster before responding to read-only requests. Alternatively, the leader could rely on the heartbeat mechanism to provide a form of lease [9], but this would rely on timing for safety (it assumes bounded clock skew).

只读操作可以在不向日志中写入任何内容的情况下进行处理。然而，如果没有额外的措施，这将有返回陈旧数据的风险，因为响应请求的领导者可能已经被它不知道的较新的领导者所取代。可线性化的读取不可以返回陈旧的数据，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，领导者必须拥有关于哪些条目被提交的最新信息。领导者完整性属性保证领导者拥有所有已提交的条目，但在其任期开始时，它可能不知道这些条目是什么。为了找到答案，它需要从其任期内提交一个条目。Raft通过让每个领导者在其任期开始时向日志提交一个空白的无操作条目来处理这个问题。第二，领导者在处理只读请求之前，必须检查它是否已经被废黜（如果最近的领导者已经当选，那么它的信息可能是过时的）。Raft通过让领导者在响应只读请求之前与集群中的大多数人交换心跳信息来处理这个问题。另外，领导者可以依靠心跳机制来提供一种租赁形式[9]，但这要依靠时间来保证安全（它假定了有界的时钟偏移）。