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.

Raft 是一种用于管理复制日志的共识算法。 它产生一个等价于（多）Paxos的结果，和Paxos一样高效，但是它的结构和Paxos不同； 这使得 Raft 比 Paxos 更易于理解，也为构建实用系统提供了更好的基础。 为了增强可理解性，Raft 分离了共识的关键要素，例如领导人选举、日志复制和安全性，并强制执行更强程度的一致性以减少必须考虑的状态数量。 用户研究的结果表明，Raft 比 Paxos 更容易让学生学习。 Raft 还包括一种用于更改集群成员的新机制，该机制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群体工作，即使其中一些成员出现故障，它们也能存活下来。正因为如此，它们在构建可靠的大型软件系统中扮演着关键角色。
Paxos[15,16]在过去的十年里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已经成为教授学生关于共识的主要工具。

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，Paxos很难理解，尽管我们曾多次尝试让它变得更加平易近人。此外，它的体系结构需要复杂的更改来支持实际的系统。因此，系统建设者和学生都在使用Paxos。

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

在与Paxos抗争之后，我们开始寻找一种新的共识算法，可以为系统建设和教育提供更好的基础。我们的方法不同寻常之处在于，我们的主要目标是可理解性:我们能否为实际系统定义一个共识算法，并以一种比Paxos更容易学习的方式描述它?此外，我们希望算法能够促进直觉的发展，而直觉对于系统构建者来说是必不可少的。重要的不仅仅是算法是否有效，还有它为什么有效。

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

这项工作的结果是一个被称为Raft的共识算法。在设计Raft时，我们应用了特定的技术来提高可理解性，包括分解(Raft分离leader选举、日志复制和安全性)和状态空间缩减(相对于Paxos，Raft减少了不确定性的程度和服务器之间可能不一致的方式)。

一项针对两所大学43名学生的用户研究表明，Raft比Paxos更容易理解:在学习了这两种算法后，其中33名学生回答关于Raft的问题比关于Paxos的问题更好。

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:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a new *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

Raft在许多方面与现有的共识算法相似(最显著的是Oki和Liskov的Viewstamped Replication[29, 22])，但它有几个新特点:

强大的领导:Raft使用了比其他共识算法更强大的领导形式。例如，日志条目只从leader流向其他服务器。
这简化了对复制日志的管理，并使Raft更容易理解。
•领袖选举:Raft使用随机计时器来选举领袖。这只为任何共识算法已经需要的心跳增加了少量的机制，同时简单而快速地解决冲突。
•成员变化:Raft的机制用于改变集群中的服务器集合，使用了一种新的联合共识方法，其中两个不同配置的大多数在过渡期间重叠。这允许集群在配置更改期间继续正常运行。

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优于Paxos和其他共识算法，无论是出于教育目的，还是作为实现的基础。它比其他算法更简单、更容易理解;对其进行了完整的描述，足以满足实际系统的需要;它有几个开源实现，并被几家公司使用;它的安全性能已被正式指定和证明;该算法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状态机问题(第2节)，讨论了Paxos的优缺点(第3节)，描述了我们的可理解性的一般方法(第4节)，提出了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

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

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。

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

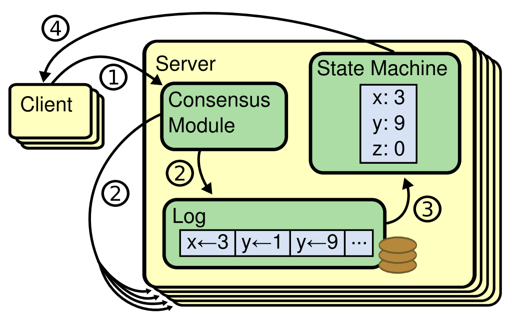

Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

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]。

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].

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，每个状态机都计算相同的状态和相同的输出序列。

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.

保持复制的日志一致是共识算法的工作。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，以确保每个日志最终包含相同顺序的相同请求，即使一些服务器失败。一旦命令被正确复制，每个服务器的状态机按照日志顺序处理它们，并将输出返回给客户端。因此，这些服务器似乎形成了一个单一的、高度可靠的状态机。

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.

复制的状态机通常使用复制的日志来实现，如图1

Consensus algorithms for practical systems typically have the following properties:

在所有非拜占庭条件下，包括网络延迟、分区和数据包丢失、重复和重新排序，它们都能确保安全（永远不会返回不正确的结果）。

只要任何大多数服务器都在运行，并能彼此和客户进行通信，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。服务器被假定为通过停止而失败；它们后来可能从稳定存储的状态中恢复并重新加入集群

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

在普通情况下，只要集群中的大多数对单轮远程过程调用作出反应，一个命令就可以完成；少数缓慢的服务器不需要影响整个系统的性能。

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

3 What's wrong with Paxos?

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

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

We hypothesize that Paxos' opaqueness derives from its choice of the single-decree subset as its foundation. Single-decree Paxos is dense and subtle: it is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. Because of this, it is difficult to develop intuitions about why the 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.

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.

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

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

> There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system.... the final system will be based on an unproven protocol [4].

Because of these problems, we concluded that Paxos does not provide a good foundation either for system building or for education. Given the importance of consensus in large-scale software systems, we decided to see if we could design an alternative consensus algorithm with better properties than Paxos. Raft is the result of that experiment.

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.

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?

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.

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.

5 The Raft consensus algorithm

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是一种算法，用于管理Section 2中描述的replicated log。Figure 2以简明形式总结了该算法以供参考，Figure 3列出了算法的关键属性。这些图的元素将在本节的其余部分中进行分段讨论。

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实现一致性的方法是，首先选择一个杰出的leader，然后让leader全权负责管理replicated log。leader接受来自client的log entries，将其复制到其他server上，并告诉server何时可以安全地将log entries应用于其state machines。拥有一个leader可以简化对replicated log的管理。例如，leader可以决定何处放置新的log，而不需要咨询其他server，数据以一种简单的方式从leader流到其他server。一个leader可能会失败或者与其他服务器断开连接，在这种情况下会选出一个新的leader

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems, which are discussed in the subsections that follow:

在采用leader方法的情况下，Raft将一致性问题分解为三个相对独立的子问题，以下各小节对此进行了讨论：

- **Leader election:** a new leader must be chosen when an existing leader fails (Section 5.2).

  当现有leader失败时，必须选择新的leader

- **Log replication:** the leader must accept log entries term:服务器上的任意时间段，需要选举新的leader

State

持久化

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

server看到的最新term

currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically) 递增

当前已获得投票的候选人编号

votedFor candidateId that received vote in current term (or null if none)

log entries；每个log entry包含state machine的命令，以及leader接收entry时的term

log[] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

易失的

Volatile state on all servers:

已知最新的log entry的index

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

应用于state machine的最新log entry的index

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

Volatile state on leaders:

(Reinitialized after election)

对于每个server，发送到该server的下一个log entry的index

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server，已知要在server上复制的最新log entry的index

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)

如果现有条目与新条目冲突（索引相同但术语不同），请删除现有条目及其后面的所有条目

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)

候选人的日志至少和接收者的日志一样是最新的

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).

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.

Raft一致性算法的简明摘要（不包括server身份变化和log压缩）。左上角的server行为描述为一组独立且重复触发的规则。

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

如果两个log包含具有相同index和term的entry
则该log在从给定index起的所有term中都是相同的

如果一个log entry commit给定的term，那么该entry将出现在所有higher-numbered term的leader的日志中

如果一个服务器在state machine上应用了一个给定index的log entry，那么其他server将永远不会为同一index应用一个不同的log entry

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

Raft保证这些属性在任何时候都为真

from clients and replicate them across the cluster, forcing the other logs to agree with its own (Section 5.3).

leader必须接收来自client的log entries，并在整个集群中复制它们，强制其他log与其自己的log一致

- **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.

Raft的关键安全属性是Figure 3中的State Machine Safety Property。如果任何server已经将一个特定的log entry应用到它的State Machine，那么其他任何server都不能对相同的log index应用不同的命令。Section 5.4描述了Raft如何确保此属性；该解决方案涉及对Section 5.2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.

5.1 Raft basics

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集群包含多个server，一般是5个，它允许系统容忍两次故障。在任何给定时间，每个server都处于三种状态之一:leader、follower、candidate。
在正常运行中，只有一个leader，其他所有的服务器都是follower。follower是被动的:他们自己不发出任何请求，而只是回应leader和candidate的requests。leader处理所有的client requests(如果client联系了一个follower, follower将它重定向到leader)

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)

Raft将时间划分为任意长度。term用连续整数进行编号。每个term以选举开始，其中一名或多名candidate试图成为leader。如果candidate赢得了选举，那么他将在剩下的任期内担任leader。在某些情况下，选举会导致split vote。在这种情况下，任期将以没有leader结束,新任期(进行新选举)即将开始.Raft确保在一个给定的任期内最多有一个leader

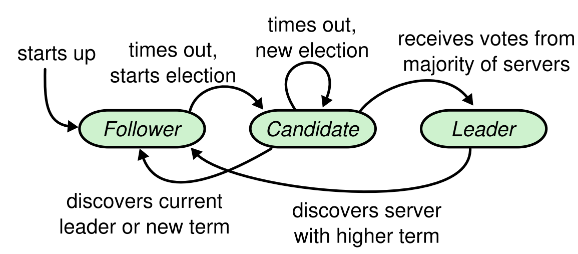

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

follower只响应来自其他server的请求。如果follower没有收到任何信息，它就成为candidate，并发起选举。一个follower从全体成员中获得多数选票成为新的leader。leader通常会一直工作到失败为止。

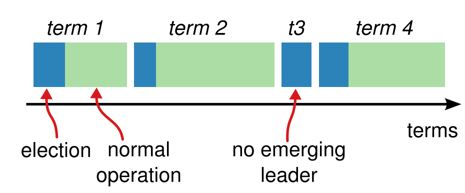

Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

will begin shortly. Raft ensures that there is at most one leader in a given term.

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server可能会在不同的时间观察到term之间的转换,在某些情况下，server可能不会观察到选举甚至整个term。term在Raft中充当logical clock，它们允许server检测过时的信息，比如过时的leader。每个server存储了一个当前term编号，它随着时间单调地增加。当前term在server通信时进行交换;如果一个server的当前term小于另一个server的，那么它将其当前term更新为较大的值。如果一个candidate或leader发现它的term过时了，它立即恢复到follower状态。如果server接收到一个过期term编号的请求，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(rpc)进行通信，基本一致性算法只需要两种类型的rpc。RequestVoteRPCs在选举期间由candidate发起，AppendEntriesRPCs由leader发起,以复制log entry并提供一种心跳的形式。section7添加了用于在server之间传输快照的第三个RPC。如果server没有及时收到响应，则会重试rpc，并行发出rpc以获得最佳性能。

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leader的选举。当server启动时，它们开始作为follower。只要server从leader或candidate接收到有效的rpc，它就保持在follower状态。leader定期向所有follower发送心跳(没有log entry的AppendEntries rpc)，以维护它们的权限。如果follower在一段称为选举超时的时间内没有收到任何通信，那么它会假设没有可行的leader，并开始选举一个新的leader

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，follower递增其当前term，并转变为candidate状态。然后它为自己投票，同时并行地向集群中的每个其他server发出RequestVoteRpc。一个candidate一直在这个状态，直到发生以下三件事中的一件:(A)它赢得了选举，(b)另一个服务器成为了ldeader，或(c)一段时间过去了没有选举成功。这些结果将在下文各段分别讨论。

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candidate在同一term内获得整个集群中大多数server的选票，那么它就赢得了选举。每个server在一个term内最多只能投票给一个candidate，以先到先得的方式(注意:5.4节增加了一个额外的投票限制)。多数票原则确保最多一个candidate可以在一个特定的term内赢得选举。一旦一个candidate赢得选举，他就成为leader。然后，它向所有其他server发送心跳信息，以建立其权威，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.

在等待投票时，candidate可能会从另一个server收到一个自称为leader的AppendEntries RPC。如果leader的term(包括其RPC)至少和candidate的当前term一样大，那么candidate承认leader是合法的，并返回到follower状态。如果小于candidate的当前term，那么candidate拒绝RPC并继续在candidate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 RequestVote RPCs. However, without extra measures split votes could repeat indefinitely.

第三种可能的结果是，一个candidate既没有赢得选举，也没有输掉选举:如果许多follower同时成为candidate，选票可能被spilt，因此没有一个candidate获得多数。当这种情况发生时，每个candidare都将超时并通过递增term启动另一轮RequestVoteRpc来开始新的选举。然而，如果没有额外的措施，spilt vote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来确保很少出现spilt vote，并且能够快速解决。为了防止spilt vote，选举超时是在固定的时间间隔(例如，150 - 300毫秒)中随机选择的。这将分散server，因此在大多数情况下只有一个server会超时;它赢得了选举，并在其他server超时之前发送心跳。同样的机制也用于处理splite vote。每个candidate在选举开始时重新启动它的随机选举超时器，并在开始下一个选举之前等待该超时过去;这降低了在新选举中再次出现spilt vote的可能性。第9.3节显示了这种方法能够迅速选出一个leader

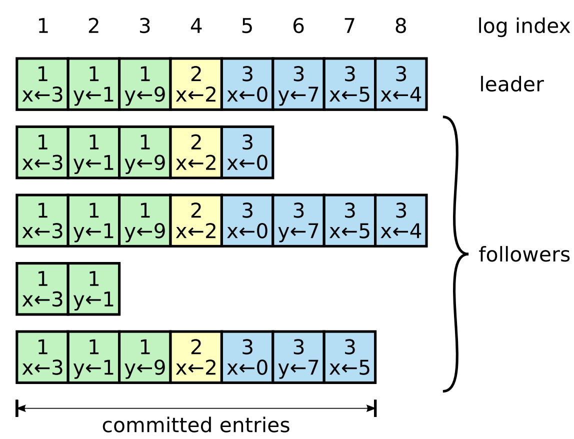

Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

log由entry组成，entry按顺序编号。每个entry包含创建它的条件(每个框中的数字)和state machine的命令。如果某个entry应用于state machine是安全的，则认为该entry已提交。

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.

选举就是一个例子，说明可理解性如何引导我们在设计选择中做出选择。最初我们计划使用一个排名系统:每个candidate被分配一个唯一的排名，用于在竞争的candidate之间进行选择。如果一个candidate发现了另一个级别更高的candidate，它就会回到follower状态，这样级别更高的candidate更容易赢得下次选举。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server失败，一个排名较低的server可能需要超时并再次成为candidate，但如果它过早地这样做，就会重置选举leader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角落案例。最终我们得出结论，随机重试的方法更明显，更容易理解。

5.3 Log replication

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 issues 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 AppendEntries RPCs indefinitely (even after it has responded to the client) until all followers eventually store all log entries.

一旦leader被选出，它就开始处理client request。每个client request都包含一个由replicated state machine执行的命令。leader将命令作为new entry附加到其log中，然后并行地向每个其他server发出AppendEntries RPCs以复制该entry。当entry被安全复制后(如下所述)，leader将entry应用到其state machine，并将执行结果返回给client。如果follower崩溃或运行缓慢，或者如果网络数据包丢失，leader无限期地重试AppendEntries RPC(即使它已经响应了client)，直到所有follower最终存储所有log entry。

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 iden-

log的组织方式如Figure 6所示。每个log entry存储一个state machine命令，以及leader接收entry时的term number。log entry中的term number用于检测log之间的不一致性，并确保Figure 3中的一些属性。每个log entry也有一个整数索引，用来标识其在log中的位置。

tifying its position in the log.

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).

leader决定何时将log entry应用到state machine是安全的;这样的entry称为committed。Raft保证提交的entry是持久的，最终将由所有可用的state machine执行。一个log entry一旦被创建的leader在大多数server上复制(例如图6中的entry7)，就会被commit。这也会在leader的log中提交之前的所有entry，包括之前leader创建的entry。第5.4节讨论了在leader变更后应用这一规则时的一些微妙之处，它也表明了这种承诺的定义是安全的。leader跟踪它知道要提交的highest index，并将该索引包含在未来的AppendEntries rpc中(包括心跳)，以便其他server最终发现。一旦follower知道一个log entry被提交，它就将这个entry应用到它的本地state machine(按log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:

我们设计了raft log机制来维护不同server上log之间的高水平一致性。这不仅简化了系统的行为，使其更易于预测，而且是确保安全的一个重要组成部分。Raft维护以下属性，它们共同构成了图3中的Log Matching属性:

- If two entries in different logs have the same index and term, then they store the same command.
- 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log中的两个entry具有相同的index和term，则它们存储相同的命令

如果不同日志中的两个entry具有相同的index和term，那么log中的所有前面的entry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.

第一个属性源于这样一个事实:在给定的term中，一个leader最多创建一个给定log index的entry，并且log entry永远不会改变它们在log中的位置。第二个属性由AppendEntries执行的一个简单的一致性检查来保证。当发送AppendEntries RPC时，leader会将新entry的上一个entry的index和term信息包含进去。如果follower在其log中没有找到具有相同index和term的entry，那么它将拒绝新的entry。一致性检查归纳步骤:log的初始空状态满足“ Log Matching ”属性，对log进行扩展时，一致性检查保留“ Log Matching ”属性。因此，每当AppendEntries成功返回时，leader知道follower的log与自己的log在新entry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.

在正常的操作过程中，leader和follower的log保持一致，所以AppendEntries一致性检查从不失败。但是，leader crash可能会使log不一致(旧的leader可能没有完全复制其log中的所有entry)。这些不一致性会导致一系列leader和follower崩溃。图7说明了follower的log与新leader的不同之处。follower可能缺少leader上存在的entry，也可能有额外的leader上不存在的entry，或者两者都有。log中缺少的和无关的entry可能会跨越多个term。

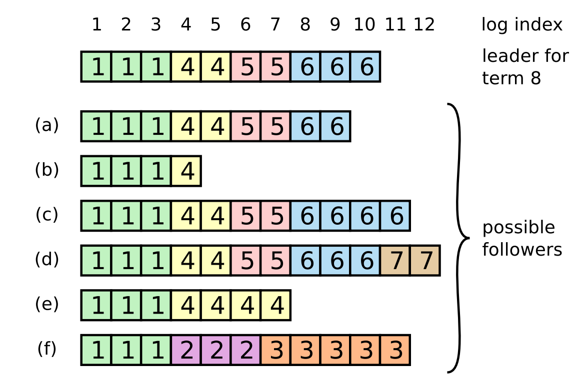

Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, 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.

当有leader时候，任何情况(a – f)都有可能出现在follower log中。每个框代表一个log entry；方框里的数字是它的term。follower可能缺少entry(a – b)，可能有额外的未committed entry(c – d)，或者两者都有(e – f)。例如，如果该server是term 2的leader，向其log中添加了几个entry，然后在committed任何entry之前崩溃，则可能发生场景(f);它很快重新启动，成为term 3的leader，并在其log中增加了一些entry；在committed第2项或第3项中的任何entry之前，server再次崩溃，并持续好几个term。

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，leader通过强制follower复制自己的log来处理不一致性。这意味着follower log中冲突的entry将被leader log中的entry覆盖。Section5.4将说明如果再加上一个限制，这是安全的。

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follower的log与自己的保持一致，leader必须找到两个log一致的最新log entry，删除follower log中在该点之后的所有entry，并将该点之后leader的所有entry发送给follower。所有这些操作都是为了响应AppendEntries rpc执行的一致性检查。leader为每个follower维护一个next index，这是leader将发送给那个follower的下一个log entry的索引。当一个leader第一次掌权时，它将所有的next Index初始化到它的log中最后一个之后的索引(Figure7中的11)。如果follower的log与leader的不一致，AppendEntries一致性检查将在下一个AppendEntries RPC中失败。拒绝之后，leader递减下一个Index并重试AppendEntries RPC。最终，next Index将到达leader和follower log匹配的点。当这种情况发生时，AppendEntries将成功，它将删除follower的log中任何冲突的entry，并从leader的log中添加entry(如果有)。一旦AppendEntries成功，follower的log和leader的一致，并且在剩下的term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

如果需要，可以对协议进行优化，以减少被拒绝的AppendEntries RPC的数量。例如，当拒绝AppendEntries Request时，follower可以存储包含冲突entry的term以及它为该term存储的第一个index。有了这个信息，leader就可以递减nextIndex来绕过该term中所有冲突的entry;每个有冲突entry的term都需要一个AppendEntries RPC，而不是每个entry一个RPC。在实践中，我们怀疑这种优化是必要的，因为失败很少发生，也不太可能有很多不一致的entry

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.

有了这种机制，leader在掌权时不需要采取任何特殊行动来恢复log一致性。它只是开始正常的操作，当AppendEntries一致性检查失败时，log自动收敛。leader从不覆盖或删除自己log中的entry(Figure3中的leader Append-Only属性)

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 AppendEntries consistency check. A leader never overwrites or deletes entries in its own log (the Leader Append-Only Property in Figure 3).

这个日志复制机制展示了Section 2中描述的理想的一致性属性:只要大多数server都是正常的，Raft就可以接受、复制和应用新的log entry;通常情况下，一个新entry可以用一轮rpc复制到集群的大部分;单一的慢follower不会影响性能。

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.

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.

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.

5.4.1 Election restriction

在任何基于leader的一致性算法中，leader最终必须存储所有committed的log entry。在一些一致性算法中，比如Viewstamped Replication，即使一个leader最初不包含所有committed的entry，也可以被选出来。这些算法包含了额外的机制来识别丢失的entry，并将它们传递给新leader，无论是在选举过程中还是之后不久。不幸的是，这导致了相当多的额外机制和复杂性。Raft使用了一种更简单的方法，它保证从选举的那一刻起，每个新leader上committed的所有entry都存在，而不需要将这些entry转移给leader。这意味着log entry只向一个方向流动，从leader到follower，leader从不覆盖他们log中的现有entry。

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

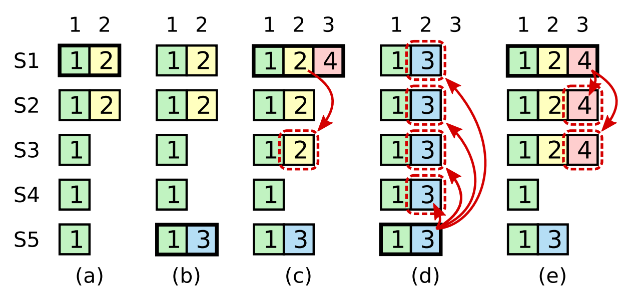

Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at 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.

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.

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voting process来阻止candidate赢得选举，除非其log包含所有已committed的entry。candidate必须联系到集群中的大多数才能被选中，这意味着每个committed的entry必须至少出现在其中的一个server上。如果候选人的log至少和大多数的其他log一样是最新的(下面精确地定义了“最新的”)，那么它将保存所有committed的entry。RequestVote RPC实现了这个限制:RPC包含关于候选人log的信息，如果投票者自己的log比候选人的log更最新，那么投票者拒绝其投票。

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.

5.4.2 Committing entries from previous terms

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. Fig-

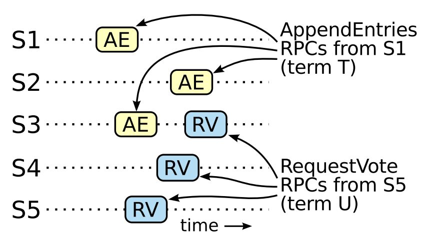

Figure 9: If S1 (leader for term T) commits a new log entry from its term, and S5 is elected leader for a later term U, then there must be at least one server (S3) that accepted the log entry and also voted for S5.

ure 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.

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.

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 rereplicates 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).

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$_{\text{T}}$) 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$_{\text{U}}$) does not store the entry.

1. The committed entry must have been absent from leader$_{\text{U}}$'s log at the time of its election (leaders never delete or overwrite entries).
2. leader$_{\text{T}}$ replicated the entry on a majority of the cluster, and leader$_{\text{U}}$ received votes from a majority of the cluster. Thus, at least one server ("the voter") both accepted the entry from leader$_{\text{T}}$ and voted for leader$_{\text{U}}$, as shown in Figure 9. The voter is key to reaching a contradiction.
3. The voter must have accepted the committed entry from leader$_{\text{T}}$ *before* voting for leader$_{\text{U}}$; otherwise it would have rejected the AppendEntries request from leader$_{\text{T}}$ (its current term would have been higher than T).
4. The voter still stored the entry when it voted for leader$_{\text{U}}$, since every intervening leader contained the entry (by assumption), leaders never remove entries, and followers only remove entries if they conflict with the leader.
5. The voter granted its vote to leader$_{\text{U}}$, so leader$_{\text{U}}$'s log must have been as up-to-date as the voter's. This leads to one of two contradictions.
6. First, if the voter and leader$_{\text{U}}$ shared the same last log term, then leader$_{\text{U}}$'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$_{\text{U}}$ was assumed not to.
7. Otherwise, leader$_{\text{U}}$'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$_{\text{U}}$'s last log entry must have contained the committed entry in its log (by assumption). Then, by the Log Matching Property, leader$_{\text{U}}$'s log must also contain the committed entry, which is a contradiction.
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.
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).

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.

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.

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.

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.

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*:

$$\mathit{broadcastTime} \ll \mathit{electionTimeout} \ll \mathit{MTBF}$$

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 timeout; we would like this to represent only a small fraction of overall time.

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.

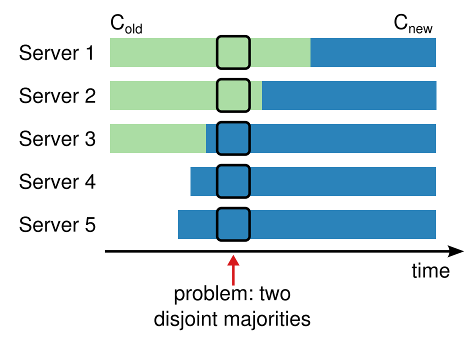

Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

6 Cluster membership changes

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.

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).

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:

- Log entries are replicated to all servers in both configurations.

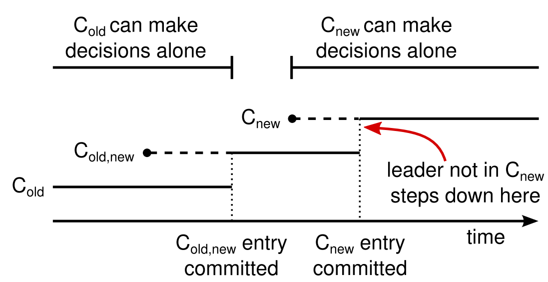

Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a 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.

- Any server from either configuration may serve as leader.
- Agreement (for elections and entry commitment) requires separate majorities from *both* the old and new configurations.

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.

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.

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.

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.

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.

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.

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.

7 Log compaction

Raft's log grows during normal operation to incorporate more client requests, but in a practical system, it cannot 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.

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

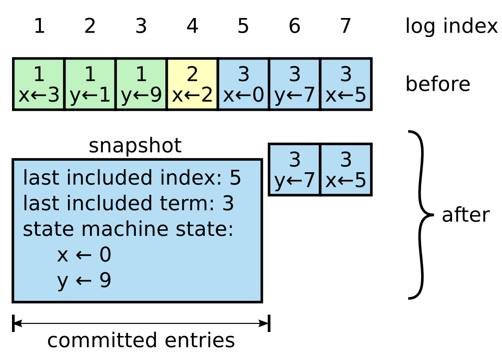

Figure 12: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables *x* and *y* in this example). The snapshot's last included index and term serve to position the snapshot in the log preceding entry 6.

that point is discarded. Snapshotting is used in Chubby and ZooKeeper, and the remainder of this section describes snapshotting in Raft.

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 snapshotting, 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.

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.

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.

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)

Figure 13: A summary of the InstallSnapshot RPC. Snapshots are split into chunks for transmission; this gives the follower a sign of life with each chunk, so it can reset its election timer.

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 snapshot 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.

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.

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.

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.

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).

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.

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.

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.

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).

9 Implementation and evaluation

We have implemented Raft as part of a replicated state machine that stores configuration information for RAMCloud [33] and assists in failover of the RAMCloud coordinator. The Raft implementation contains roughly 2000 lines of C++ code, not including tests, comments, or blank lines. The source code is freely available [23]. There are also about 25 independent third-party open source implementations [34] of Raft in various stages of development, based on drafts of this paper. Also, various companies are deploying Raft-based systems [34].

The remainder of this section evaluates Raft using three criteria: understandability, correctness, and performance.

9.1 Understandability

To measure Raft's understandability relative to Paxos, we conducted an experimental study using upper-level undergraduate and graduate students in an Advanced Operating Systems course at Stanford University and a Distributed Computing course at U.C. Berkeley. We recorded a video lecture of Raft and another of Paxos, and created corresponding quizzes. The Raft lecture covered the content of this paper except for log compaction; the Paxos

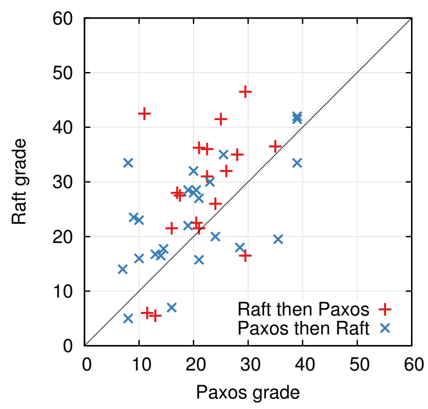

Figure 14: A scatter plot comparing 43 participants' performance on the Raft and Paxos quizzes. Points above the diagonal (33) represent participants who scored higher for Raft.

lecture covered enough material to create an equivalent replicated state machine, including single-decree Paxos, multi-decree Paxos, reconfiguration, and a few optimizations needed in practice (such as leader election). The quizzes tested basic understanding of the algorithms and also required students to reason about corner cases. Each student watched one video, took the corresponding quiz, watched the second video, and took the second quiz. About half of the participants did the Paxos portion first and the other half did the Raft portion first in order to account for both individual differences in performance and experience gained from the first portion of the study. We compared participants' scores on each quiz to determine whether participants showed a better understanding of Raft.

We tried to make the comparison between Paxos and Raft as fair as possible. The experiment favored Paxos in two ways: 15 of the 43 participants reported having some prior experience with Paxos, and the Paxos video is 14% longer than the Raft video. As summarized in Table 1, we have taken steps to mitigate potential sources of bias. All of our materials are available for review [28, 31].

On average, participants scored 4.9 points higher on the Raft quiz than on the Paxos quiz (out of a possible 60 points, the mean Raft score was 25.7 and the mean Paxos score was 20.8); Figure 14 shows their individual scores. A paired t-test states that, with 95% confidence, the true distribution of Raft scores has a mean at least 2.5 points larger than the true distribution of Paxos scores.

We also created a linear regression model that predicts a new student's quiz scores based on three factors: which quiz they took, their degree of prior Paxos experience, and

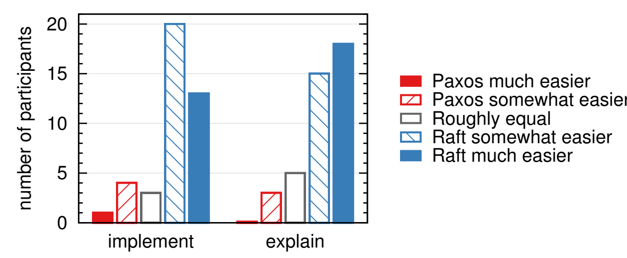

Figure 15: Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

the order in which they learned the algorithms. The model predicts that the choice of quiz produces a 12.5-point difference in favor of Raft. This is significantly higher than the observed difference of 4.9 points, because many of the actual students had prior Paxos experience, which helped Paxos considerably, whereas it helped Raft slightly less. Curiously, the model also predicts scores 6.3 points lower on Raft for people that have already taken the Paxos quiz; although we don't know why, this does appear to be statistically significant.

We also surveyed participants after their quizzes to see which algorithm they felt would be easier to implement or explain; these results are shown in Figure 15. An overwhelming majority of participants reported Raft would be easier to implement and explain (33 of 41 for each question). However, these self-reported feelings may be less reliable than participants' quiz scores, and participants may have been biased by knowledge of our hypothesis that Raft is easier to understand.

A detailed discussion of the Raft user study is available at [31].

9.2 Correctness

We have developed a formal specification and a proof of safety for the consensus mechanism described in Section 5. The formal specification [31] makes the information summarized in Figure 2 completely precise using the TLA+ specification language [17]. It is about 400 lines long and serves as the subject of the proof. It is also useful on its own for anyone implementing Raft. We have mechanically proven the Log Completeness Property using the TLA proof system [7]. However, this proof relies on invariants that have not been mechanically checked (for example, we have not proven the type safety of the specification). Furthermore, we have written an informal proof [31] of the State Machine Safety property which is complete (it relies on the specification alone) and rela-

Concern	Steps taken to mitigate bias	Materials for review [28, 31]
Equal lecture quality	Same lecturer for both. Paxos lecture based on and improved from existing materials used in several universities. Paxos lecture is 14% longer.	videos
Equal quiz difficulty	Questions grouped in difficulty and paired across exams.	quizzes
Fair grading	Used rubric. Graded in random order, alternating between quizzes.	rubric

Table 1: Concerns of possible bias against Paxos in the study, steps taken to counter each, and additional materials available.

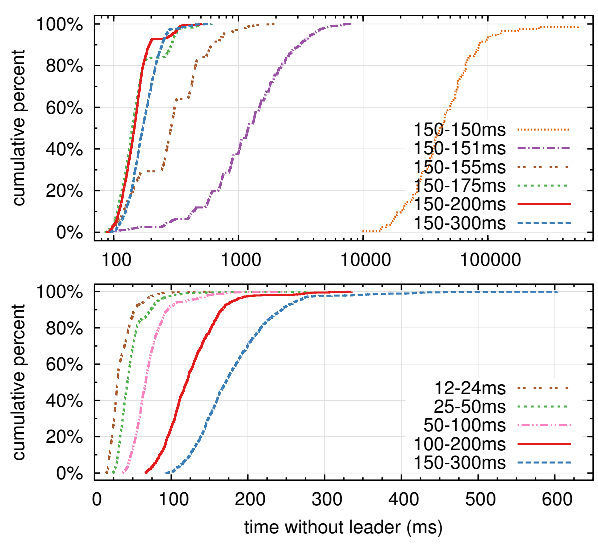

Figure 16: The time to detect and replace a crashed leader. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials (except for 100 trials for "150–150ms") and corresponds to a particular choice of election timeouts; for example, "150–155ms" means that election timeouts were chosen randomly and uniformly between 150ms and 155ms. The measurements were taken on a cluster of five servers with a broadcast time of roughly 15ms. Results for a cluster of nine servers are similar.

tively precise (it is about 3500 words long).

9.3 Performance

Raft's performance is similar to other consensus algorithms such as Paxos. The most important case for performance is when an established leader is replicating new log entries. Raft achieves this using the minimal number of messages (a single round-trip from the leader to half the cluster). It is also possible to further improve Raft's performance. For example, it easily supports batching and pipelining requests for higher throughput and lower latency. Various optimizations have been proposed in the literature for other algorithms; many of these could be applied to Raft, but we leave this to future work.

We used our Raft implementation to measure the performance of Raft's leader election algorithm and answer two questions. First, does the election process converge quickly? Second, what is the minimum downtime that can be achieved after leader crashes?

To measure leader election, we repeatedly crashed the leader of a cluster of five servers and timed how long it took to detect the crash and elect a new leader (see Figure 16). To generate a worst-case scenario, the servers in each trial had different log lengths, so some candidates were not eligible to become leader. Furthermore, to encourage split votes, our test script triggered a synchronized broadcast of heartbeat RPCs from the leader before terminating its process (this approximates the behavior of the leader replicating a new log entry prior to crashing). The leader was crashed uniformly randomly within its heartbeat interval, which was half of the minimum election timeout for all tests. Thus, the smallest possible downtime was about half of the minimum election timeout.

The top graph in Figure 16 shows that a small amount of randomization in the election timeout is enough to avoid split votes in elections. In the absence of randomness, leader election consistently took longer than 10 seconds in our tests due to many split votes. Adding just 5ms of randomness helps significantly, resulting in a median downtime of 287ms. Using more randomness improves worst-case behavior: with 50ms of randomness the worst-case completion time (over 1000 trials) was 513ms.

The bottom graph in Figure 16 shows that downtime can be reduced by reducing the election timeout. With an election timeout of 12–24ms, it takes only 35ms on average to elect a leader (the longest trial took 152ms). However, lowering the timeouts beyond this point violates Raft's timing requirement: leaders have difficulty broadcasting heartbeats before other servers start new elections. This can cause unnecessary leader changes and lower overall system availability. We recommend using a conservative election timeout such as 150–300ms; such timeouts are unlikely to cause unnecessary leader changes and will still provide good availability.

10 Related work

There have been numerous publications related to consensus algorithms, many of which fall into one of the following categories:

- Lamport's original description of Paxos [15], and attempts to explain it more clearly [16, 20, 21].
- Elaborations of Paxos, which fill in missing details and modify the algorithm to provide a better foundation for implementation [26, 39, 13].
- Systems that implement consensus algorithms, such as Chubby [2, 4], ZooKeeper [11, 12], and Spanner [6]. The algorithms for Chubby and Spanner have not been published in detail, though both claim to be based on Paxos. ZooKeeper's algorithm has been published in more detail, but it is quite different from Paxos.
- Performance optimizations that can be applied to Paxos [18, 19, 3, 25, 1, 27].
- Oki and Liskov's Viewstamped Replication (VR), an alternative approach to consensus developed around the same time as Paxos. The original description [29] was intertwined with a protocol for distributed transactions, but the core consensus protocol has been separated in a recent update [22]. VR uses a leader-based approach with many similarities to Raft.

The greatest difference between Raft and Paxos is Raft's strong leadership: Raft uses leader election as an essential part of the consensus protocol, and it concentrates as much functionality as possible in the leader. This approach results in a simpler algorithm that is easier to understand. For example, in Paxos, leader election is orthogonal to the basic consensus protocol: it serves only as a performance optimization and is not required for achieving consensus. However, this results in additional mechanism: Paxos includes both a two-phase protocol for basic consensus and a separate mechanism for leader election. In contrast, Raft incorporates leader election directly into the consensus algorithm and uses it as the first of the two phases of consensus. This results in less mechanism than in Paxos.

Like Raft, VR and ZooKeeper are leader-based and therefore share many of Raft's advantages over Paxos. However, Raft has less mechanism that VR or ZooKeeper because it minimizes the functionality in non-leaders. For example, log entries in Raft flow in only one direction: outward from the leader in AppendEntries RPCs. In VR log entries flow in both directions (leaders can receive log entries during the election process); this results in additional mechanism and complexity. The published description of ZooKeeper also transfers log entries both to and from the leader, but the implementation is apparently more like Raft [35].

Raft has fewer message types than any other algorithm for consensus-based log replication that we are aware of. For example, we counted the message types VR and ZooKeeper use for basic consensus and membership changes (excluding log compaction and client interaction, as these are nearly independent of the algorithms). VR and ZooKeeper each define 10 different message types, while Raft has only 4 message types (two RPC requests and their responses). Raft's messages are a bit more dense than the other algorithms', but they are simpler collectively. In addition, VR and ZooKeeper are described in terms of transmitting entire logs during leader changes; additional message types will be required to optimize these mechanisms so that they are practical.

Raft's strong leadership approach simplifies the algorithm, but it precludes some performance optimizations. For example, Egalitarian Paxos (EPaxos) can achieve higher performance under some conditions with a leaderless approach [27]. EPaxos exploits commutativity in state machine commands. Any server can commit a command with just one round of communication as long as other commands that are proposed concurrently commute with it. However, if commands that are proposed concurrently do not commute with each other, EPaxos requires an additional round of communication. Because any server may commit commands, EPaxos balances load well between servers and is able to achieve lower latency than Raft in WAN settings. However, it adds significant complexity to Paxos.

Several different approaches for cluster membership changes have been proposed or implemented in other work, including Lamport's original proposal [15], VR [22], and SMART [24]. We chose the joint consensus approach for Raft because it leverages the rest of the consensus protocol, so that very little additional mechanism is required for membership changes. Lamport's α-based approach was not an option for Raft because it assumes consensus can be reached without a leader. In comparison to VR and SMART, Raft's reconfiguration algorithm has the advantage that membership changes can occur without limiting the processing of normal requests; in contrast, VR stops all normal processing during configuration changes, and SMART imposes an α-like limit on the number of outstanding requests. Raft's approach also adds less mechanism than either VR or SMART.

11 Conclusion

Algorithms are often designed with correctness, efficiency, and/or conciseness as the primary goals. Although these are all worthy goals, we believe that understandability is just as important. None of the other goals can be achieved until developers render the algorithm into a practical implementation, which will inevitably deviate from and expand upon the published form. Unless developers have a deep understanding of the algorithm and can create intuitions about it, it will be difficult for them to retain its desirable properties in their implementation.

In this paper we addressed the issue of distributed consensus, where a widely accepted but impenetrable algorithm, Paxos, has challenged students and developers for many years. We developed a new algorithm, Raft, which we have shown to be more understandable than Paxos. We also believe that Raft provides a better foundation for system building. Using understandability as the primary design goal changed the way we approached the design of Raft; as the design progressed we found ourselves reusing a few techniques repeatedly, such as decomposing the problem and simplifying the state space. These techniques not only improved the understandability of Raft but also made it easier to convince ourselves of its correctness.

12 Acknowledgments

The user study would not have been possible without the support of Ali Ghodsi, David Mazières, and the students of CS 294-91 at Berkeley and CS 240 at Stanford. Scott Klemmer helped us design the user study, and Nelson Ray advised us on statistical analysis. The Paxos slides for the user study borrowed heavily from a slide deck originally created by Lorenzo Alvisi. Special thanks go to David Mazières and Ezra Hoch for finding subtle bugs in Raft. Many people provided helpful feedback on the paper and user study materials, including Ed Bugnion, Michael Chan, Hugues Evrard, Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, Robbert van Renesse, Mendel Rosenblum, Nicolas Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia, 24 anonymous conference reviewers (with duplicates), and especially our shepherd Eddie Kohler. Werner Vogels tweeted a link to an earlier draft, which gave Raft significant exposure. This work was supported by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the National Science Foundation under Grant No. 0963859, and by grants from Facebook, Google, Mellanox, NEC, NetApp, SAP, and Samsung. Diego Ongaro is supported by The Junglee Corporation Stanford Graduate Fellowship.

References

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

[2] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI'06, Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp. 335–350.

[3] CAMARGOS, L. J., SCHMIDT, R. M., AND PEDONE, F. Multicoordinated Paxos. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 316–317.

[4] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *Proc. OSDI'06, USENIX Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp. 205–218.

[6] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proc. OSDI'12, USENIX Conference on Operating Systems Design and Implementation* (2012), USENIX, pp. 251–264.

[7] COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ, S., RICKETTS, D., AND VANZETTO, H. TLA$^+$ proofs. In *Proc. FM'12, Symposium on Formal Methods* (2012), D. Giannakopoulou and D. Méry, Eds., vol. 7436 of *Lecture Notes in Computer Science*, Springer, pp. 147–154.

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. SOSP'03, ACM Symposium on Operating Systems Principles* (2003), ACM, pp. 29–43.

[9] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Ssymposium on Operating Systems Principles* (1989), pp. 202–210.

[10] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12* (July 1990), 463–492.

[11] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc ATC'10, USENIX Annual Technical Conference* (2010), USENIX, pp. 145–158.

[12] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proc. DSN'11, IEEE/IFIP Int'l Conf. on Dependable Systems & Networks* (2011), IEEE Computer Society, pp. 245–256.

[13] KIRSCH, J., AND AMIR, Y. Paxos for system builders. Tech. Rep. CNDS-2008-2, Johns Hopkins University, 2008.

[14] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commununications of the ACM 21*, 7 (July 1978), 558–565.

[15] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[16] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[17] LAMPORT, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[18] LAMPORT, L. Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2005.

[19] LAMPORT, L. Fast paxos. *Distributed Computing 19*, 2 (2006), 79–103.

[20] LAMPSON, B. W. How to build a highly available system using consensus. In *Distributed Algorithms*, O. Baboaglu and K. Marzullo, Eds. Springer-Verlag, 1996, pp. 1–17.

[21] LAMPSON, B. W. The ABCD's of Paxos. In *Proc. PODC'01, ACM Symposium on Principles of Distributed Computing* (2001), ACM, pp. 13–13.

[22] LISKOV, B., AND COWLING, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

[23] LogCabin source code. `http://github.com/logcabin/logcabin`.

[24] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART way to migrate replicated stateful services. In *Proc. EuroSys'06, ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), ACM, pp. 103–115.

[25] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: building efficient replicated state machines for WANs. In *Proc. OSDI'08, USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 369–384.

[26] MAZIÈRES, D. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, Jan. 2007.

[27] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proc. SOSP'13, ACM Symposium on Operating System Principles* (2013), ACM.

[28] Raft user study. `http://ramcloud.stanford.edu/~ongaro/userstudy/`.

[29] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. PODC'88, ACM Symposium on Principles of Distributed Computing* (1988), ACM, pp. 8–17.

[30] O'NEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[31] ONGARO, D. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014 (work in progress). `http://ramcloud.stanford.edu/~ongaro/thesis.pdf`.

[32] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proc ATC'14, USENIX Annual Technical Conference* (2014), USENIX.

[33] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. *Communications of the ACM 54* (July 2011), 121–130.

[34] Raft consensus algorithm website. `http://raftconsensus.github.io`.

[35] REED, B. Personal communications, May 17, 2013.

[36] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10* (February 1992), 26–52.

[37] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

[38] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Proc. MSST'10, Symposium on Mass Storage Systems and Technologies* (2010), IEEE Computer Society, pp. 1–10.

[39] VAN RENESSE, R. Paxos made moderately complex. Tech. rep., Cornell University, 2012.