寻找一种易于理解的一致性算法 (扩展版)

Diego Ongaro and John Ousterhout
Stanford University

摘要—Raft 是一种基于日志复制的一致性算法。Raft 的结果等价于 (multi-)Paxos 并且和 Paxos 一样的有效，但却和 Paxos 的结构完全不同。正是 Raft 不同于 Paxos 的结构使得 Raft 更容易理解，并且更易于工程实现。Raft 通过将一致性关键点进行拆解，包含选主、日志复制和安全三部分，另外通过减少状态机的状态使得 Raft 更容易理解。通过学生学习 Raft 和 Paxos 的分析，结果显示 Raft 相比于 Paxos 更容易理解。Raft 创新的使用了一种新的机制来处理集群中机器的变更。

I. 简介

一致性算法可以使得若干机器像一个一致集群一样的工作，即使在集群成员变更或者宕机的情况下，整个集群仍然可以正常工作。正因为如此一致性算法在大规模分布式系统中扮演着至关重要的角色。过去的十年，paxos 统治了整个一致性算法。大多数一致性算法的实现都是基于 Paxos 或者受到 Paxos 的影响，同时一致性算法课程大多也是讲授 Paxos。

不幸的是，Paxos 晦涩难懂，尽管很多人努力尝试让 Paxos 更容易理解。除此之外，在实际系统中，实现者必须根据系统作出大量的调整。结果，系统开发者和学生都为了理解 Paxos 在苦苦挣扎。

我们也是苦苦的与 Paxos 斗争，后来我们希望找到一个既容易实现又利于学生学习的一致性算法。所以我们寻找一致性算法的首要目标就是易于理解：我们可以定义一种比 Paxos 利于实现和学习的一致性算法？进一步我们希望算法可以让系统开发者有直观感受和理解。我们认为算法能工作很重要，更重要的是让算法实现者直观到理解它为什么可以工作。

为了设计 Raft 算法，我们使用了特定的技术来使算法更容易理解，包括分解法（Raft 分解为选主、日志复制和安全）和状态空间简化（相较于 Paxos，Raft 减少了未定状态的数量）。我们分析两个学校 43 名同学的学习情况，结果显示 Raft 比 Paxos 更容易理解：学习过两种算法后，他们中的 33 位同学可以更好的解答 Raft 中的问题相比于 Paxos。

Raft 和存在的一些一致性算法是相似的，但是也有它自己新颖的特点：

- **Strong leader**：Raft 使用了一种比其它一致性算法更强的 leader。比如，日志记录只能从 leader 复制到其它机器，这样可以简化日志复制和降低理解难度。
- **Leader election**：Raft 使用了随机定时器来选择 leader。通过在心跳上增加一些小的改动，可以简单快速的解决冲突。
- **Membership changes**：Raft 使用了一种联合一致性的方法，使得集群中的机器发生变更的时候，整个集群也可以正常的工作。联合一致性配置是两个不同配置的大多数机器的重叠。

我们相信 Raft 不管是在教学或者实现方面都是优于其它一致性算法。我们详细的描述可以让开发者更简单的进行系统的实现。另外已经有几个公司公开了自己的实现，Raft 的安全性是验证和证明的，同样它也和其它一致性算法一样有效。

文章接下来会在第二节讲述复制状态机，第三节会讨论 Paxos 的优缺点，第四节会描述易于理解方法，第五-八节会描述 Raft 一致性算法，第九节会评估 Raft，第十节讨论我们的相关工作。

II. 复制状态机

通常提到一致性算法都会提到复制状态机 [37]。在复制状态机中，集群中所有的机器有相同的日志副本，即使集群中某些集群宕机，整个集群仍然可以正常工作。复制状态机通常被用来解决分布式系统的容错问

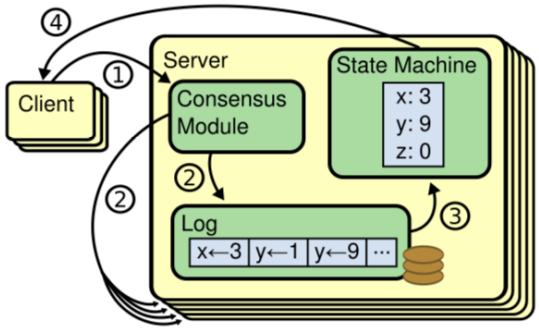

Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

题。例如：例如 GFS、HDFS 和 RAMCloud 等大规模系统中，通常使用分离的状态机来管理选主和存储配置信息。Chubby 和 ZooKeeper 中也使用了复制状态机。

复制状态机通常是通过复制日志来实现的，如图 1 所示。每个机器的 log 文件中包含了一系列的命令，这些命令将会被按顺序执行到状态机。每个机器的日志包含相同的命令并且命令都有相同的顺序，如果状态机按相同的顺序执行命令。由于状态机初始化状态是相同的，并且按照相同顺序执行相同的命令，那么必然状态机有相同的输出。

一致性算法的工作就是保证各个状态机一致。机器上的一致性模块接受到来自客户端的命令，并把命令追加到自己的日志中。然后和其它机器上的一致性模块进行通信来保证最终所有机器都按相同的顺序存储了请求，即使其中的一些机器宕机不能工作。一旦命令被正确的复制，每个机器都按照日志顺序将命令执行到状态机，然后将输出返回给客户端。最终，所有的机器呈现一个高度可靠的状态机。

实际系统的一致性算法具有一下的特性：

- 在非拜占庭条件下必须是安全的，不会返回错误的结果。例如：网络延时、分区、丢包、重复和乱序等。
- 在集群中大多数机器可以正常工作的情况下，整个集群必须是可用的。一个 5 机器的集群可以容忍两台机器的宕机。我们假定机器宕机后可以从稳定存储介质恢复并重新加入到集群中。
- 它们不依赖时间来保证日志的的一致性：错误的时钟和巨大的消息延迟，在极端情况下，会导致集群的不可用。
- 在正常情况下，集群中的机器在一轮 RPC 下就可以完成一次日志的复制，个别较慢的机器不能影响整个系统的性能。

III. Paxos 存在的问题

过去十年，Leslie Lamport 的 Paxos 算法等同于一致性算法。许多学校会教授 Paxos 算法，许多算法也是依据 Paxos 进行实现的。Paxos 是第一个能对单次提议达成一致的协议，就好像单条日志复制。我们把达成单次一致的协议称为 *single-decree Paxos*。Paxos 通过一系列单次一致的实例来达到对一系列的提议达成一致，这被称作 *multi-Paxos*。Paxos 算法可以保证安全和存活，同时也支持集群成员变更。Paxos 的正确性是被证明的，并且在正常情况下是有效的。

不幸的是，Paxos 有个显著的缺点。第一个缺点就是 Paxos 晦涩难懂。很多人投入大量的经历，但是只有很少的人真正的理解 Paxos。已经有很多学者尝试以更容易理解的方式解释 Paxos。这些解释都是集中在 *single-decree Paxos*，但是仍然存在很大的挑战。通过调研我们发现很多人不满 Paxos，即使很多很有经验的研究者。为了设计我们的算法，我们阅读了很多简单的解释，并尝试设计可用的算法，整个过程花费大约 1 年的时间。

我们认为 Paxis 晦涩难懂的原因是因为 Paxos 选择了 *single-decree Paxos* 作为基本的子问题导致的。*single-decree Paxos* 是紧凑且精巧的，它的两个阶段是不能独立分开的，并且通过简单的直觉进行理解和解释。正因为如此，很难形成 *single-decree Paxos* 为什么可以正常工作的直觉。*multi-Paxos* 更是增加了额外的复杂度和精密度。我们相信整个问题可以达到一致是通过多次的决定，并且可以被拆分进行直观的理解。

Paxos 存在第二个问题是它不能简单的转化为工程实现，一方面是大家没有对 *multi-Paxos* 算法达成一致。Lamport 大神只是详细的描述了 *single-decree Paxos*，但是很多细节还是缺失的。还存在很多基于 Paxos 的变种和分支。Chubby 是基于 Paxos 进行实现的，但是它们的实现细节并没有公开。

另外，Paxos 的结构决定它很难用来实现实际的系统，这是 *single-decree Paxos* 不能分解导致的必然结果。举个例子，通过单独的选择日志记录的集合并把它们聚合在一个顺序日志中，并不能带来太多的收益，但

是却增加了额外的复杂度。设计一种只是按照严格顺序进行追加的日志是简单和更为有效的。此外 Paxos 采用了对称的点对点通信的方法作为它的核心（尽管是出于一种基于弱领导的性能优化）。上面的优化仅仅对于单次决定是有意义的，但是实际系统很少采用这种方法。如果有一些的提议需要被决定，首先选举出一个 leader 然后由 leader 进行协调，这样是更为简单和快速的。

现实中的系统仅有很少和 Paxos 中描述的系统相似。每个实际系统都是从 Paxos 开始，发现实现中有很多不能解决的难题，然后提出一个新的结构并实现。这是耗时且容易犯错的，然后 Paxos 的晦涩难懂进一步使情况恶化。Paxos 的范式可能是一种证明一致性理论的好方法，但是实际的实现中 Paxos 不具备很大的价值。下面来自 Chubby 开发者的评论更正说明这一点：

实际系统的实现和 Paxos 所描述的算法中存在巨大的鸿沟... 最终实现的系统将是基于一个未被证明的协议

因为这些问题的存在，我们认为 Paxos 并不适合系统实现和教学。在大规模软件系统中一致性至关重要，我们决定尝试是否可以设计一种和 Paxos 有相同特性且易于理解的算法。Raft 正是我们试验的产物。

IV. 为了理解而设计

设计 Raft 时，我们有几个目标：它对于系统的开发必须是完整和易于实现的，这样可以减少开发者大量不必要的设计工作；它必须在所有条件下是安全的，并在典型条件下是可用的；另外对于正常操作它必须是有效的。但我们最重要的目标是（也是最大的挑战）是易理解性。它必须是大多数读者可以舒适的理解的。此外，它必须可以让系统开发者有直观的感受，这样系统开发者可以在实际的实现中可以更好的进行扩展。

下面会列举一些在设计 Raft 算法过程中，怎样从很多候选的方法中进行选择的考虑点。我们还是根据候选方法的可理解性来进行评估：每种候选的方法是否易于解释的?（例如，它的状态机空间复杂? 它是否有暗含的精巧设计?）是不是可以让一个读者很容易的理解它的潜在表达。

我们承认这样的分析存在很大主观成分，不可否认的是我们使用了两种通用的技术：第一个技术是众所周知的问题分解法，我们尽可能将问题分解为独立的可以被解决，易于解释和理解的子问题。例如：在 Raft 中我们利用问题分解法分为 4 个独立的问题：选主、日志复制、安全和成员变更。

第二个技术是通过减少需要考虑的状态来简化状态空间。这样使得系统更加一致并尽可能消除不确定的状态。特别的，Raft 不允许日志中出现空洞，并且禁止这样的情况发生。空洞会导致集群中的日志产生不一致。尽管我们尽力来消除不确定状态，但是在一些情况下，增加不确定状态可以更好的帮助理解算法。特别是，随机化方法会引进不确定状态，但是它可以简化状态空间通过以相同的方式来处理所有可能的选择（"choose any; it doesnt matter"）。我们使用了随机化的方法来简化 Raft 的选主算法。

V. Raft 一致性算法

Raft 是一种管理如第二节所描述的状态机的算法。图 2 以紧凑的格式总结了算法中的术语和定义。图 3 列举了算法的关键特性；这些图中的定义和术语将会在接下来的章节进行详细的讲解。

Raft 实现一致性是首先选择一个确定的 leader，然后 leader 负责管理日志复制。leader 接受来自客户端的请求并追加到本地日志，然后把日志复制到其它的机器并告诉其它机器什么时候可以安全的将日志应用到状态机。集群存在一个 leader 的好处可以简化日志复制的管理。例如，leader 可以决定日志的追加，而不需要经其它机器的同意。整个集群的数据流向也是从 leader 流向其它机器。如果 leader 宕机或者网络断开，其它的机器可以重新选举一个新的 leader。

使用选主的方法，Raft 可以将一致性问题分解为 3 个相对独立的子问题，下面的子章节将对这些问题进行讨论：

- **Leader election**：第 5.2 节将讨论，当一个 leader 宕机后，一个新的 leader 必须被选举。
- **Log replication**：leader 必须响应客户端的请求，并把日志复制到整个集群来保证其它机器的日志和自己的相同。
- **Safety**：图 3 中的状态机的安全是 Raft 优先保证的：如果任意一台机器将一条特定的日志应用到自己的状态机，那么其他的机器就不能应用一条不同的日志到自己的状态机。在 5.4 节描述了 Raft 是如何保证这个特性的；解决这个问题的方案就是在选举是增加额外的规则约束（5.2 节）。

在讲述完一致性算法后，这一章节还会讨论可用性的问题和时间在系统中的角色。

State

所有机器需要持久化的状态:
(在 RPC 响应之前，需要更新稳定存储介质)

currentTerm	server 存储的最新任期（初始化为 0 且单调递增）
votedFor	当前任期接受到的选票的候选者 ID（初值为 null）
log[]	日志记录每条日记记录包含状态机的命令 和从 leader 接受到日志的任期。(索引初始化为 1)

所有机器的可变状态:

commitIndex	将被提交的日志记录的索引（初值为 0 且单调递增）
lastApplied	已经被提交到状态机的最后一个日志的索引（初值为 0 且单调递增）

leader 的可变状态:
(每次选举后重新初始化)

nextIndex[]	每台机器在数组占据一个元素，元素的值为下条发送到该机器的日志索引 (初始值为 leader 最新一条日志的索引 +1)
matchIndex[]	每台机器在数组中占据一个元素，元素的记录将要复制给该机器日志的索引的。

AppendEntries RPC

被 leader 用来复制日志 (5.3 节)；同时也被用作心跳 (5.2 节)

Arguments:

term	leader 任期
leaderId	用来 follower 重定向到 leader
prevLogIndex	前继日志记录的索引
prevLogItem	前继日志的任期
entries[]	存储日志记录
leaderCommit	leader 的 commitIndex

Results:

term	当前任期，leader 用来更新自己
success	如果 follower 包含索引为 prevLogIndex 和任期为 prevLogItem 的日志

接受者的实现:
1. 如果 leader 的任期小于自己的任期返回 false。(5.1)
2. 如果自己不存在索引、任期和 prevLogIndex、prevLogItem 匹配的日志返回 false。(5.3)
3. 如果存在一条日志索引和 prevLogIndex 相等，但是任期和 prevLogItem 不相同的日志，需要删除这条日志及所有后继日志。(5.3)
4. 如果 leader 复制的日志本地没有，则直接追加存储。
5. 如果 leaderCommit>commitIndex，设置本地 commitIndex 为 leaderCommit 和最新日志索引中较小的一个。

RequestVote RPC

被候选者用来收集选票:

Arguments:

term	候选者的任期
candidateId	候选者编号
lastLogIndex	候选者最后一条日志记录的索引
lastLogItem	候选者最后一条日志记录的索引的任期

Results:

term	当前任期，候选者用来更新自己
voteGranted	如果候选者当选则为 true。

接受者的实现:
1. 如果 leader 的任期小于自己的任期返回 false。(5.1)
2. 如果本地 voteFor 为空，候选者日志和本地日志相同，则投票给该候选者 (5.2 和 5.4)

Rules for Servers

所有机器:
1. 如果 commitIndex > lastApplied：增加 lastApplied，并将日志 log[lastApplied] 应用到状态机。
2. 如果 RPC 请求中或者响应中包含任期 T>currentTerm，参与者 该机器转化为参与者。

参与者 (5.2 节):
1. 响应来自候选者或者 leader 的请求。
2. 如果选举定时器超时时，没有收到 leader 的的追加日志请求或者没有投票给候选者，该机器转化为候选者。

候选者 (5.2 节):
1. 一旦变为候选者，则开始启动选举:
   1.1 增加 currentTerm
   1.2 选举自己
   1.3 重置选举定时器
   1.4 并行发送选举请求到其他所有机器
2. 如果收到集群大多数机器的选票，则称为新的 leader。
3. 如果接受到新 leader 的追加日志请求，则转化为参与者。
4. 如果选举定时器超时，则重启选举。

leaders:
1. 一旦当选：发送空的追加日志请求（心跳）到其它所有机器；在空闲时间发送心跳，阻止其它机器的选举定时器超时。
2. 如果接受到来自客户端的请求，追加日志记录到本地日志，如果成功应用日志记录到状态机则回应客户端。
3. 如果某个参与者的最新日志索引大于等于本地存储该参与者的最新日志索引：给该参与者发送包含从 nextIndex 开始的日志追加请求。
   3.1 如果成功，更新该参与者的 nextIndex 和 matchIndex。(5.3 节)
   3.2 如果由于日志不一致而失败，减少 nextIndex 并重试。(5.3 节)
4. 如果存在 N > commitIndex（本地待提交日志的索引），majority(matchIndex[i]>= N)（如果参与者大多数的最新日志的索引大于 N），并且这些参与者索引为 N 的日志的任期也等于 leader 的当前任期：commitIndex =N（leader 的待提交的日志索引设置为 N）(5.2 和 5.4 节)。

Election Safty：在一个给定的任期最多只可以选举出一个 leader（5.2 节）。
Leader Append-Only：对于一个 leader 它永远不会重写和删除日志中的日志记录，它只会追加日志记录。（5.3 节）
Log Matching：如果两个日志文件中存在相同索引和任期的日志记录那么两个日志文件所有的日志记录在给定索引情况下是相同的。（5.3 节）
Leader Completeness：如果一条日志在一个给定的任期已经提交，那么这条日志将会出现在所有任期大于给定任期的 leader 的日志中。（5.4 节）
State Machine Safety：如果一个 server 已经将给定索引的日志应用到状态机，别的 server 将不能应用一个相同索引但内容不同的日志记录到自己的状态机。(5.4.3 节)

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

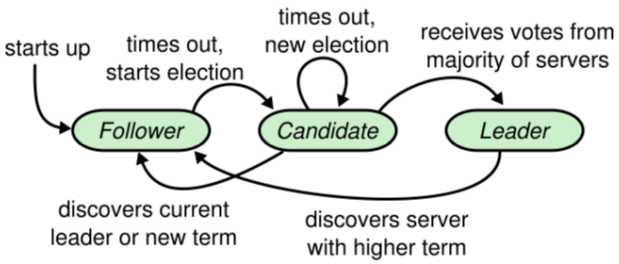

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

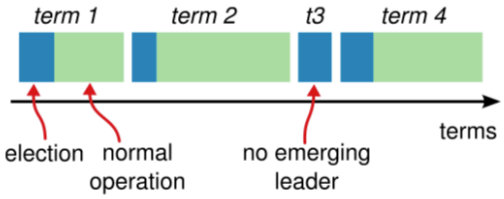

Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different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

A. Raft 基础

一个 Raft 集群一般包含多台机器，5 台机器是一种典型配置。它可以容忍集群中的两台机器不能正常工作。任意时刻，一台 server 会处在三种状态中的一种：leader、follower 和 candidate。正常情况下，集群中包含一个 leader 和参与者。集群中的参与者是被动的，它们不会主动解决问题而是被动的响应 leader 或者参与者的请求。集群中 leader 负责处理所有的客户端请求，如果一个客户端的请求连接到了参与者，这个参与者会将请求重定向到 leader。候选者是在选举中可能成为 leader 的状态，如 5.2 节所描述。图 4 展示了这些状态，并描述了这些状态的迁移。

如图 5 所示，Raft 将时间分为任意长度的间隔，每个间隔是一个任期。每个任期会由一个连续的整数进行表示。每个任期都是从选举开始的，在这个阶段会有一个或者多个候选者参与竞选，如图 5 所示。一旦某个候选者在选举中胜出，这个任期剩下的时间将有这个候选者作为 leader。一些特殊的情况下，一次选举可能出现选举分裂的情况。选举分裂的情况下，当前任期将不会选举出 leader。紧接着一个新的任期将会启动重新进行选举。Raft 通过上面的过程保证每个任期只会选举出至多一个 leader。

在不同的时间，不同的 servers 可能观察到迁移发生在不同的任期。一些特殊的情况下，一个 server 可能观察不到一次选举或者某个任期。Raft 中的任期表现的像逻辑锁，这些逻辑锁可以使得 servers 观察到某个 leader 是否已经过期。每个 server 都用一个变量存储了当前任期，并且这个变量随着时间是单调递增的。当 servers 进行通讯的时候，也会交换当前的任期。如果一个 server 存储的任期小于其他机器存储的任期，那么它将更新自己的任期到其它机器存储的最大任期。如果是一个候选者或者 leader 发现自己的任期已经过期，它们会转变到参与者的状态。如果一个 server 接受到一个请求，这个请求中的任期是过时的，它将直接拒绝该请求。

不同服务间的 Raft 模块是通过 RPC 进行通信的，Raft 的简单版只需要两种类型的 RPC。RequestVote RPCs 是候选在这选举过程中使用的，AppendEntries RPCs 是 leader 进行日志复制和心跳时使用的。在第七节我们增加了传送快照的 RPC。所有的 RPC 请求在规定的时间没有接受到响应可以进行重试，这些请求在发送的时候可以并行进行从而获得更好的性能。

B. 选主

Raft 使用心跳机制来触发选主的过程。当 servers 启动的时候，都是作为参与者。如果一个参与者收到来自 leader 或者候选者的合法请求，它将保持在参与者

的状态。leader 会发送心跳到其它的 server 来授权延长自己的任期。如果一个参与者的选举定时器超时的时候还没有收到任何请求，它可以假设整个集群没有可用的 leader 或者候选者，然后发起新的选举。

一次选举开始时，参与者增加自己本地存储的当前任期然后转变为候选者状态。这个候选者先选举自己，并行的给集群中的其它机器发送 RequestVote RPCs。候选者将会一直保持候选状态直到下面三件事情中的任意一件发生：(a)：候选者本次选举胜出，(b)：另外一台机器确认自己是 leader，(c)：僵持一段时候没有人胜出。上面的三种情况将在下面的段落进行讨论。

一个候选者如果接受到集群中大多数机器在同一个任期的选票，么它将胜出成为 leader。每台机器在一个任期只能投票给一个候选者，按照先到先服务的原则（5.4 节会对这个规则增加一些额外限制）。大多数投票胜出规则可以保证在一个特定的任期至多选出一个 leader（图 3 所描述的选举安全原则）。一旦一个候选者胜出将成为集群的 leader，它将会并行的给集群的其它机器发送心跳来宣示自己胜出，并阻止进行新的选举。

在等待选票的过程中，一个候选者可能接受到来自其它 server 的请求，该请求声明自己已经成为 leader。如果请求中的 leader 的任期大于候选者本地存储的任期，那么当前候选者认为这个 leader 是合法的并转变为参与者状态。如果请求中 leader 的任期小于当前候选者本地存储的任期，那么候选将拒绝这个请求并保持在候选者状态。

第三种可能是是整个集群的所有候选者都没有胜出。如果集群中所有的参与者同一时刻转变为候选者，由于每个机器只能投票给一个候选者，这种情况新会很容易发生选举分裂即没有一个候选者获得半数以上选票。当这种情况发生，所有的候选者的选举定时器将会超时，它们增加自己本地存储的任期并启动新一轮的选举。从上面可以看出如果没有额外的规则约束，选举分裂的情况将极易发生。

Raft 通过随机选举定时器来阻止选举分裂的发生，即使选举分裂发生也可以很快的被解决。选举超时将在 [150,300]ms 之间随机生成，这样就大概率保证集群中会有一个机器会先超时，而避免所有机器同时超时从而降低选举分裂情况发生的概率。如果首先超时的机器将会首先转变为候选者，它将会大概率的选举胜出成为 leader，然后发送心跳续阻止其它机器定时器超时。9.3 节讲述这种机器是有效的。

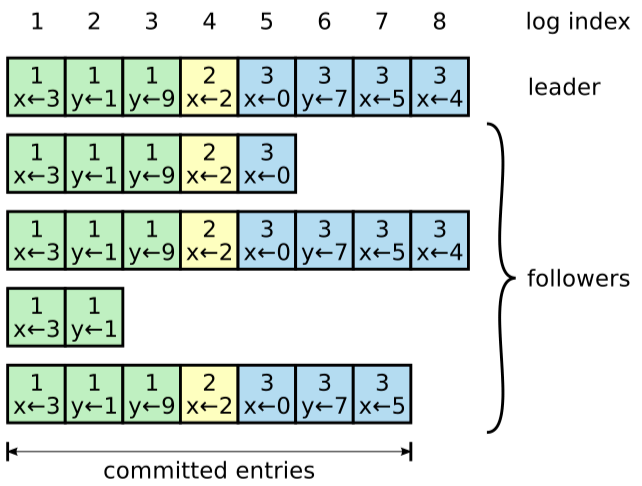

Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

选主的设计就是我们为了易懂而权衡的一种结果。最初我们准备使用一种给每个候选者加权的排名系统。这样当候选发现发现有别的候选者排名高于自己，它将转变为参与者从而保证排名高的候选者可以在下一轮以大概率胜出。但是排名的方法设计的更精巧而不是易懂，如果一个高排名的机器宕机，那么低排名的机器要等待超时后才能再次成为候选者，但是这样会延长选主过程。我们为了解决上面的问题对排名方法进行了很多次的调整，每次调整都会有新的问题产生。最终我们选择了随机重试方法，这种方法更为直观和易于理解。

C. 日志复制

一旦一个候选者成为 leader，它将开始处理客户端的请求。客户端的每个请求包含了一条需要执行到状态机的命令。leader 将命令追加到自己的日志记录，同时并发 AppendEntries RPCs 请求来进行日志复制。当日志安全的复制之后，leader 将日志应用到自己的状态机并将结果返回给客户端。如果集群中有参与者宕机、处理速度慢、网络丢包等情况发生，leader 将重试 AppendEntries RPCs 请求直到日志被安全的复制。

日志的格式如图 6 所示。每条日志记录包含了 leader 的任期和状态机命令。任期是为了检测日志之间的不一致，从而保证图 3 所示的某些性质。同时每条日志记录也会有一个索引记录其在日志中的位置。

leader 来决定将一条日志应用到状态机的安全时机。这样的一条日志被称为 ***committed***。Raft 来保证

日志的持久化并且所有已提交的日志将会都会被应用到状态机。一旦 leader 将一条日志成功的复制到集群的大多数机器，那么这条日志就是已提交状态（如图 6 的 entry7）。如果当前日志记录已提交，那么由前任 leader 或者当前 leader 创建的前继日志记录都会被提交。5.4 会讨论当 leader 变更后的应用日志的一些问题，同时也会定义什么是安全的提交。leader 维护了即将被提交的日志记录的索引，并把这个索引放在未来的 AppendEntries RPCs 请求中。当参与者从请求获知已提交的索引，它会将本地该索引的日志应用到状态机。

我们设计 Raft 来保证不同机器之间日志的高度一致，不仅仅是为了可以简化和预测系统的行为，更重要的是保证系统的安全。Raft 保证了下面的几个性质，这些性质共同组成了图 3 中的 ***Log Matching Property***:

- 如果两个日志的两条日志记录有相同的索引和任期，那么这两条日志记录中的命令必然也是相同的。
- 如果两个日志的两条日志记录有相同的索引和任期，那么这两个日志中的前继日志记录也是相同的。

在一个给定的任期，leader 创建的日志索引是递增不重复的，一旦日志某条日志创建后是不会改变它在日志中位置。上面的事实保证了第一个性质的成立。每次当 leader 发送 AppendEntries RPCs 请求的时候，请求中会包含当前正在复制的日志记录的直接前继的任期和索引，如果参与者在自己的日志中没有发现有相同任期和索引的日志记录，它将直接拒绝请求。上面描述的一致性检测保证第二个性质的成立。一致性检测的步骤如下：初始化时候是满足 Log Matching Property 的；当有追加日志的时候进行一次一致性检测来保护 Log Matching Property。这样当 leader 接受到返回成功的 AppendEntries RPCs 请求时，说明了参与者与自己的日志是相同的。

正常情况下，leader 和参与者的日志都是相同的，日志一致性检测也不失败。当 leader 崩溃的时候就会导致日志的不一致，例如旧的 leader 没有将自己的日志记录安全的复制到其它机器。这些不一致可能聚合多个 leader 和参与者的崩溃。图 7 描述参与者的日志和新 leader 日志不一致的情况。一个参与者可能缺失了 leader 有的日志记录，它也可能多出了 leader 没有的日志，或者上面的两种情况同时发生。缺失或多余的日志可能存在在多个任期。

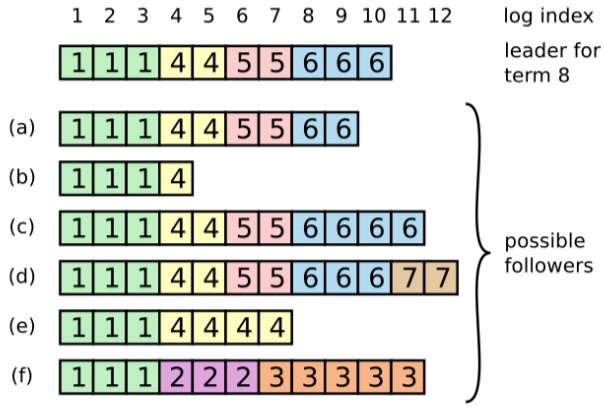

Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example,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

Raft 是通过强制参与者只能复制 leader 的日志来解决不一致。这就意味者参与者的日志和 leader 的日志发生冲突的时候，参与者的日志将会重写或者删除。5.4 将会讨论在额外附加的约束下上面的过程是安全的。

如果一致性检测失败后，为了保证参与者和自己的日志一致，leader 需要先确认参与者和自己一致的最后一条日志记录。然后通知参与者删除这条日志记录后面的日志，并将这条日志记录之后的日志复制给参与者。leader 为每个参与者维护了 *nextIndex*，这个索引记录了 leader 将复制给该参与者的日志索引。当一个 leader 选举生效后，它将初始化 *nextIndex* 为它自己日志记录中最后一条日志记录的索引（图 7 索引 11 所示）。如果 leader 的日志和参与者的日志不一致那么下一轮的 AppendEntries RPCs 进行 AppendEntries 一致性检测的时候就会发现。如果检测发生不一致，leader 将会减少 *nextIndex* 并重试。经过多次重试 leader 就会确定参与者和自己一致的日志索引，然后通知参与者删除后面不一致的日志，然后复制自己的日志给参与者。经过上面的过程 leader 和参与者的一致就可以恢复一致。

上面过程中每次 *nextIndex* 减少 1 进行重试效率是存在问题的，但是也是可以优化的。一旦参与者进行日志一致性检测发现不一致之后，在响应 leader 请求

中包含自己在这个任期存储的第一条日志。这样 leader 接受到响应后，就可以直接跳过所有冲突的日志（其中可能包含了一致的日志）。这样就可以减少寻找一致点的过程。但是在实际中我们怀疑这种优化的必要性，因为失败发生的概率本来就很低，也不会同时存在大量不一致的日志记录。

使用上面的机制，一个 leader 生效的时候就不需要进行额外的操作来恢复日志的一致性。它只需按照正常的流程，日志的不一致经过多次 AppendEntries RPCs 一致性检测后会自动收敛。leader 也不需要重写和删除本地日志（图 3 的日志只追加特性）。

上面所描述的日志复制机器满足了第 2 节所期望的一致性特性：Raft 可以在集群只有半数以上存活的情况下接受、复制和应用新的日志记录；正常情况下只需要一轮 RPCs 可以将日志记录复制到集群的大多数；单个速度慢的参与者不会影响整个集群的性能。

D. 安全性

之前的章节描述了 Raft 的选主和日志复制。但是目前所描述的机制还不能安全的保证日志是按照相同顺序被应用到状态机。例如：在 leader 提交了若干条日志后，某个参与者宕机并被选为新的 leader。新的会 leader 重写其它机器的日志，结果导致不同的状态机有不同的命令序。

我们通过添加额外约束控制某些机器能被选举为 leader 来完善 Raft 算法。这个约束保证当选的 leader 包含了前任所有提交的日志（图 3 的 Leader Completeness Property）。通过约束我们可以精准的控制日志的提交。这节剩下的部分我们会给出 Leader Completeness Property 一个简略的证明，并解释它如何保证复制状态机的正确性。

1) 选举约束： 在基于 leader 的一致性算法，leader 最终会存储所有已提交的日志记录。在某些一致性算法中，一个日志即使没有存储所有已提交的日志也可以称为 leader。这些算法会通过额外的机制来确定缺失的日志然后传给新的 leader，上面的过程可能发生在选举过程或者选举胜出后。不幸的是，这样会增加额外的复杂度。Raft 使用的方法是每个当选的 leader 必须之前所提交的所有日志，这种方法不需要复制日志到新当选的 leader。这种方法带来的好处是集群中数据的流向只能是从 leader 到参与者，leader 永远仅需要追加即可。

Raft 使用选举过程来保证一个候选者必须包含有所有已提交的日志才能胜出。候选者为了胜出必须联系

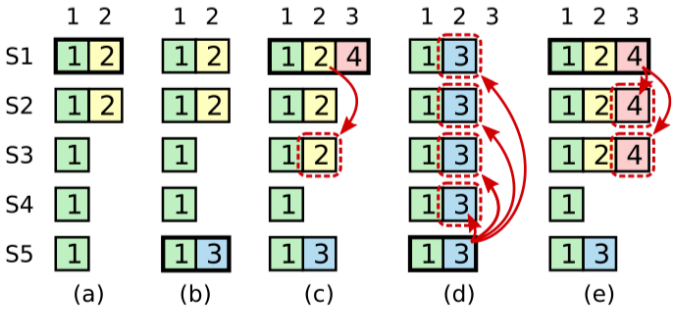

Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at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

集群中的大多数机器，这就意味着每条日志至少出现在机群中的某一台机器上。如果候选者的日志如果比集群其它任意一台机器的日志更新（下面精确定义“更新”），那么它将包含所有已提交的日志。RequestVote RPCs 来实现这个约束：请求中包含了 leader 的日志信息，如果投票者的日志比候选者的日志更新，那么它就拒绝投票。

两个日志文件谁的日志更新是通过比较日志中最后一条日志记录的任期和索引。如果两个日志文件的最后一条日志的任期不相同，谁的任期更大谁的的日志将更新。如果两条日志记录的任期相同，那么谁的索引越大，谁的日志将更新。

2) 提交上一个任期的日志： 如第 5.3 描述，如果当前任期的一条日志已经被复制到集群中的大多数，那么 leader 可以确定这条已经已经处于提交状态。如果上任 leader 在提交日志之前宕机，下一任 leader 将尝试完成日志的复制。然而，尽管上一任期的某条日志已经被复制到了大多数机器，但是新任 leader 还是不能准确断定这条日志是否是已提交。图 8 展示了这样一种情况。

为了消除如图 8 所示的问题。Raft 不能根据上一任期的日志是否被复制到大多数机器来决定是否提交日志。一旦当前任期看到一条日志被提交，由于 Log

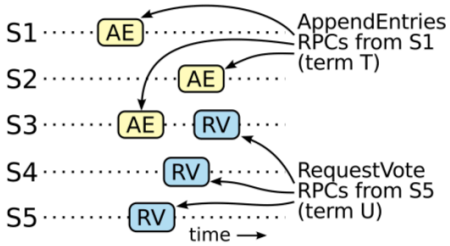

Figure 9: If S1 (leader for term T) commits a new log entry from its term, and S5 is elected leader for a later term U, then there must be at least one server (S3) that accepted the log entry and also voted for S5.

Matching Property 保证，那么这条日志之前的日志已自动被提交。有一些情况下，一个 leader 可以安全的断定一条老的日志是否已提交（例如一条日志已经被存储在所有的机器上），但是 Raft 采取了保守的方法来使得问题简化。

Raft 在提交过程引入的额外复杂度是因为前一个任期已经复制的日志保持了它原来的任期编号。在一些其它的一致性算法中，如果一个新的 leader 复制复制前面任期的日志，那么之前的任期编号必须转换为当前的任期编号。Raft 的方法使日志记录变的简单，同一个任期编号可以跨越时间和日志文件。除此之外，在复制前任的日志记录下，新的 leader 需要更少次数的复制相比其它一致性算法（在日志提交前，其它一致性算法需要复制冗余的日志记录来变换任期编号）。

3) 安全讨论: 给出了完整 Raft 算法，我们可以更精确的讨论 Leader Completeness Property（这里的讨论基于安全的证明，参看 9.2 节）。我们假设 Leader Completeness Property 不成立，使用反证法证明 Leader Completeness Property 成立。假设任期 T 的 leader 在它的任期提交了一条日志，但是这条日志并没有被复制到未来某个任期的 leader。考虑大于 T 的最小任期 U 没有存储这条日志。

1. 当任期 U 的 leader 在它选举的时候这条已提交，日志已不存在于它的日志中（leaders 没有删除或者覆写）。
2. 任期 T 的 leader 已经将日志复制到大多数机器，同时任期 U 的 leader 也接受到集群大多数机器的选票。那么至少存在一台机器即接受到任期 T 的 leader 复制的日志同时也投票给任期 U 的 leader，如图 9 所示。这个投票者是反证法的关键。
3. 这个投票者一定是先接受任期 T 的 leader 复制日志，然后投票给任期 U 的 leader。否则它将拒绝来自任期 T 的 leader 的 AppendEntries 请求（否则它的任期将大于 T，与假设矛盾）。
4. 这个投票者投票给任期 U 的 leader 的时候，已经存储了这条日志，因为这中间任期的 leader 包含这条日志（通过假设），leaders 没有删除过这条日志，参与者只删除自己和 leader 冲突的日志。
5. 投票者将自己的票投给任期 U 的 leader，所以任期 U 的 leader 的日志一定和自己的一样新。这将推出两个矛盾中的一个。
6. 首先，如果投票者和任期 U 的 leader 的最后一条日志记录相同，那么任期 U 的 leader 的日志至少和投票者的日志一样长，所以它的日志包含了投票者的所有日志。这也是一个矛盾，因为投票者包含了已提交的日志，而我们假设任期 U 的 leader 没有这条日志。
7. 否则，任期 U 的 leader 的最后一条日志必须大于投票者。进一步，同时也大于任期 T，因为投票者最后一条日志的任期至少是 T（它存储了来自任期 T 的日志）。创建任期 U 的 leader 最后一条日志的任期 leader，一定包含这条已提交的日志（通过假设）。那么，根据性质 Log Matching Property，任期 U 的 leader 也一定包含这条已提交的日志，这是一个矛盾。
8. 这里完成反证法。所以任期大于 T 的任期一定包含在任期 T 提交的所有日志。
9. Log Matching Property 保证未来的 leader 也包含已经提交的日志，图 8(d)index 2 所示。

证明 Leader Completeness Property，我们可以证明图 3 中的 State Machine Safety Property。这条性质是讲如果一台机器已经应用一条给定索引的日志记录到状态机，其他机器将不会应用一条具有相同索引但是内容不同的日志到状态机。只有在和 leader 的日志相同的情况下并且这条日志已经提交，参与者参会将这条日志记录应用到自己的状态机。考虑任何一台机器应用了一条给定的索引在最小的任期; Log Matching Property 将会保证大于最小任期的 leader 都包含这条指定索引的日志记录。所以 servers 在最新的任期应用这条日志将有相同的值。所以 State Machine Safety Property 成立。

最终，Raft 需要 servers 按照日志索引顺序将应用日志到状态机。结合 State Machine Safety Property 性质，那么所有的 servers 将按照相同的顺序执行相同的日志到状态机。

4) 参与者和候选者宕机: 目前为止我们都是聚焦在 leader 不可用，参与者和候选者采用比 leader 简单的方法处理不可用。如果参与者或者候选者不可用，那么发送给他们的 RequestVote 和 AppendEntries RPCs 将会失败，然后 Raft 采用重试的策略；如果宕机重启，那么重试的请求将会成功。如果一个机器还没有来的及回应就宕机，再它重启之后将会接受和宕机前相同的请求。Raft 请求是可重入的，重试是没有危害的。举个例子，如果候选者接受的 AppendEntries 请求中日志已经被存储们，直接忽略这些请求。

5) 时基和可用性: Raft 安全性一个要求就是不能依赖时间：系统不能由于某些事件发生比期望快或者慢就导致不正确的结果。然而，系统可用性不可避免的会依赖时间。例如：如果一条消息交换的时间长于机器宕机间隔，候选者不能存活足够长的时间去赢得选举；如果没有一个稳定的 leader，Raft 将不能工作。

选主是 Raft 的一部分，其中时间是至关重要的。Raft 可以进行选举和保持一个稳定的 leader，需要系统满足下面的时间要求：

$$broadcastTime \ll electionTimeout \ll MTBF$$

这个不等式中的 broadcastTime 是一个 server 并行发送 RPCs 到每个机器并接受到它们回应的平均时间；electionTimeout 是 5.2 描述的选举超时时间；MTBF 单台机器的平均故障间隔时间。broadcastTime 的数量级必须远远小于 electionTimeout，这样 leader 才可以可靠的发送心跳来阻止参与者启动新的选举；选定超时时间通过使用随机化的方法，这个不等式可以保证选举分裂发生的概率很小；election timeout 也应该在数量级上远远小于 MTBF，这样系统才能稳定的运行。当 leader 宕机，系统最长不可用时间不会超过选举超时时间，我们可以预想到不可用时间占整个运行时间极小的一部分。

broadcastTime 和 MTBF 是下伏系统的固有特性，我们能选择的只有选举超时时间。Raft 的 RPCs 请求往往需求接受者将信息持久化到稳定存储介质，所以 broadcastTime 会在 0.5ms 到 20ms 之间波动。根据上面的分析，选举超时时间应该在 [10ms,500ms]。典型的 MTBF 往往是数月或者更久，可以轻松满足系统的时间需求。

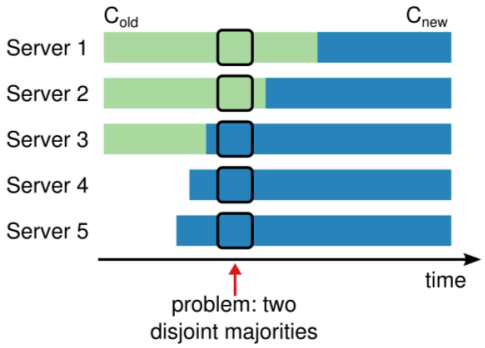

Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one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

VI. 集群成员变更

目前为止我们都是假定集群的配置（参与一致性算法的机器集合）是固定的。在实际系统中，我们有时候可能需要变更配置，例如需要替换宕机的机器或者增加日志的副本数。最笨的方法，我们可以停掉整个集群更换配置并重启集群，但是这将导致整个集群不可用。除此之外，手工操作也会带来其它的风险。最终 Raft 决定支持配置的热变更。

如果要保证配置变更过程是安全的就要保证变更过程中同一个任期不可能有两个 leader 选举胜出。不幸的是，所有直接从旧配置切换到新配置方法都是不安全的。同一时刻切换所有的机器是不可能的，所以变更过程中，集群可以分裂成两个部分（如图 10 所示）。

为了保证安全，配置变更采用两阶段的方法。这里有很多方法可以实现两阶段，例如：有的系统在第一个阶段来停止旧配置，这样就不能响应客户端的请求；然后第二阶段切换到新配置。Raft 集群第一阶段会过渡到迁移配置（我们称之为联合一致性配置 joint consensus）；一旦联合一致性被提交，系统将切换到新配置。joint consensus 配置组合了老配置和新配置。

- 日志记录将被复制新老配置到所有机器。
- 新老配置的机器都可以作为 leader。
- 表决（选举或是日志提交）同时需要老配置中的多数派和新配置中的多数派同意。

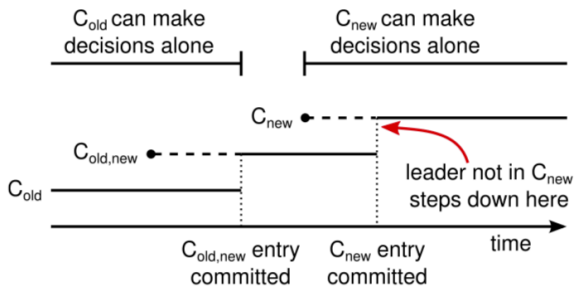

Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a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

联合一致性允许独立的机器变更在不同的配置而不需要保证安全性。进一步，联合一致性可以允许集群在配置变更过程中处理客户端的请求。

集群的配置存储和交换是通过特殊的日志进行的。图 11 展示配置的交换过程。当 leader 接受到请求要将配置从老配置切换到新配置，它将存储联合一致性配置，并通过日志复制将联合一致性日志复制到其它机器。一旦一个参与者接受将这条日志记录并写入自己的日志中，它将使用这个配置来进行后面的决策（一个机器永远使用最新的配置进行决策，不管这条日志记录是否提交）。这就意味这 leader 将使用联合一致性配置的来决策日志是否提交。如果 leader 宕机，一个新的 leader 将会被选举，可能是旧配置也可能是新配置，依赖于候选者是否接受到联合一致性配置。在任何情况下，新配置不能单边作出决定在配置的变更期间。

一旦联合一致性配置已经被提交，不管老配置还是新配置都不可能在不得到对方支持的情况下作出决策，Leader Completeness Property 保证那些使用联合一致性配置的机器可以被选作 leader。那么 ledaer 就可以安全的创建一条日志对于新配置，并将配置复制到其它机器。此外，这个配置一旦被机器接受到就会立即生效。当新的配置在新配置的规则下被提交，那么老配置将变得无关紧要，不在新配置中的机器就可以停机下线了。如图 11 所示，这里存在新配置和老配置同时作出单边决定的时刻。这样就保障了安全性。

配置变更中有三个问题需要说明。第一个问题就是新的机器初始化的时候可能不存储任何日志。如果它们以这样的状态加入集群，需要花费很长的时间后它们的日志才能追赶上集群中其他机器。在这个期间集群是不能提交新的日志。为了避免不可用间隙，Raft 在配置变更前引入了一个新的阶段，在这个阶段新加入的机器没有表决权（leader 将日志复制给它们，但是它们不是投票集合的成员）。一旦新机器的日志追赶上集群的其它机器，配置的变更就可以如前面描述的过程进行。

第二个问题是 leader 可能不是新配置的机器。这种情况下，leader 在提交新配置下的日志记录后就降级到参与者状态。这就意味这存在一时间 leader 管理着一个不包含自己的集群；也就是说它仅仅复制而它自己不是表决集合的成员。leader 的变更发生在新配置被提交，因为是从这个时间点新配置可以单独的运行。在这个点之前，可能都是老配置的机器被选举为 leader。

第三个问题就是删除机器（不在新配置中的机器），这些机器可能中断集群。这些机器接受不到心跳，所以它们就会超时重启新的选举。它们会发送携带新任期编号的 RequestVote RPCs，这将会造成当前 leader 回退到参与者状态。一个新的 leader 虽然会被选举，但是需要移除的机器将会再次超时，这个过程将不断重复，导致集群的频繁不可用。

为了阻止第三个问题的发生，servers 在确定 leader 存在的情况下将不理会 RequestVote RPCs。特殊地，如果一个机器在最小超时时间下接受到来自当前任期 leader 的 RequestVote RPC，它将不更新自己的任期，也不参与这次投票。这不会影响正常的选举，这里每个机器直到等待一个最小超时时间在新一轮的选举之前。然而，这样会阻止待移除的机器扰乱整个集群：如果一个 leader 可以接受到当前集群的心跳，它将不会被更大任期的所废除。

VII. 日志压缩

随着集群的运行，机器的日志将会不断增长。在实际系统中，日志不可能无限增长。日志不断增长将会占据更多的空间，花费更多的时间进行回放。为了保证集群的可用性需要额外的机制来删除日志中过期的日志。

快照是最简单的压缩方法。快照方法下，当前整个系统状态将会写到存储在稳定存储介质的一个快照。这样这个点之前的日志就可以删除。快照被使用在 Chubby 和 ZooKeeper 中，接下来的章节我们将会描述 Raft 中的快照。

使用递增的方法进行压缩，例如日志清理和 LSM tree 都是可以的。递增法压缩往往都是发生在少部分数

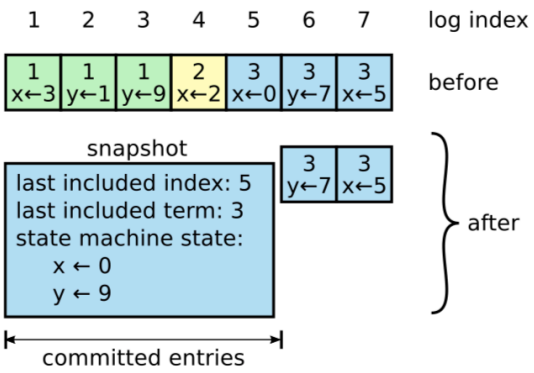

Figure 12: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables *x* and *y* in this example). The snapshot's last included index and term serve to position the snapshot in the log preceding entry 6.

据上，这将会导致系统需要进行频繁的压缩。它们首先选择一个有很多删除和重复数据的区块，然后将压缩后的内容写入到一个新的区块，并释放旧的区块从而实现日志的压缩。相比快照这需要额外的机制和复杂度来实现，而快照机制每次都是对整个数据集进行操作。如果使用日志清理那么就需要对 Raft 进行修改，状态机提供给 LSM tree 的接口快照同样可以使用。

图 12 展示了 Raft 中快照的基本思想。每个机器独立的进行快照，快照只需要覆盖已经提交的日志。更多的工作是状态机将自己的当前的状态写入到快照中。Raft 包含了少量的 metadata 在快照中：last included index 日志中最后一条被快照取代的日志的索引（也是状态机应用的上一条日志的索引）。last included term 是这条日志对应的任期。这些被持久化用来支持 AppendEntries 的日志一致性检测，如前面讲述的日志一致性检测需要前一条日志的任期和索引。为了支持集群成员变更（第六节），快照也包含了最后一条被快照取代的日志所使用的配置。一旦机器完成快照，它将可以删除 last included index 之前的所有日志，也包括之前的快照。

正常情况下机器都是单独的进行快照，但是某些情况下 leader 也会发送快照到那些落后的机器上。上面的情况发生在 leader 已经删除了下条需要发送给参与者的日志记录。幸运的是，这种情况在正常操作下不会发生：一个参与者保持和 leader 有一样的日志。然而，一个异常慢的参与者或者一个新的机器加入到集群，这些情况下 leader 就需要通过网络发送快照来使其它的机器追赶上自己。

InstallSnapshot RPC

被用来复制快照到参与者：

Arguments：

term	leader 的任期
leaderId	参与者重定向到 leader
lastIncludedIndex	快照替换的最后一条日志记录的索引
lastIncludedItem	快照替换的最后一条日志记录的索引的任期
offset	快照中块的字节偏移量
data[]	数据块，从偏移量开始存储
done	是否是最后一个块文件

Results：

term	当前任期，leader 用来更新自己

接受者的实现：

1. 如果 leader 的任期小于自己的任期返回 false。
2. 创建新的快照文件，如果是第一个块文件（偏移量为 0）。
3. 在快照文件的指定偏移量写入数据。
4. 如果 done 为假，等待更多的块数据。
5. 保存快照文件，删除快照之前的日志。
6. 如果存在快照点之后日志日志，保留这些日志并重放。
7. 删除整个日志文件。
8. 使用快照的状态重置状态机，并加载快照中的配置。

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.

leader 将会使用一种叫做 InstallSnapshot 新的 RPC 来拷贝快照到那些远远落后的机器。如图 13 所示，当一个参与者接受一个包含快照的 RPC，它必须决定对已经存在的日志进行处理。通常快照会包含接受这日志中没有的新信息。这种情况下，参与者删除整个日志并被快照取代，如果参与者重复接受到一个快照，那么快照之前的日志记录可以删除，但是快照之后的日志记录是合法的并需要被保留。

这种快照法有悖 Raft 的强领导原则，因为参与者可以在没有 leader 情况下进行快照。然而，我们认为这样的背离是合理的，存在 leader 是为了避免冲突并达成一致，快照的时候一致性已经达成，所以没有决定是冲突的。集群中的数据流依旧是从 leader 流向参与者，只是参与者可以重组自己的数据。

我们认为基于强领导的方法只有 leader 可以创建快照，然后将快照复制给其它的参与者。然而这里存在两个缺点。第一个，通过网络发送快照将会浪费网络带宽，拖慢快照的处理速度。每个参与者已经具备了产生快照的所有信息，一个参与者通过自己的状态机产生快照的成本是远远低于来自 leader 的快照。第二，这样将会使 leader 的实现变的复杂。例如：leader 可以在并发复制新的日志的时候发送快照，这样将不会阻塞来自客户端的新的请求。

这里有两个方面制约快照的速度。第一，机器需要决定什么时候进行快照。如果一个机器快照的频率过高将会浪费磁盘的带宽和能力；如果快照频率过慢将会加剧容量不足的风险，也会增加重放日志的耗时。一个简单的策略是当日志的总量达到某个阈值的时候进行快照。如果这个阈值设置的比较大，这样快照耗费的磁盘就会比较小。

第二制约性能的问题是快照快照要花费很长的时间，我们不希望拖慢正常的操作，我们可以接受拖慢非正常操作。解决方法就是使用写时复制技术，在进行快照的同时也可以接受新的更新。例如：使用结构化数据设计的状态机天然的支持写时复制。此外，我们可以使用操作系统的写时复制技术来创建基于内存的整个状态机的快照（我们的实现采用了这种方法）。

VIII. 客户端交互

这一章节我们将描述 Raft 是如何和客户端交互的，包含客户端如何发现集群的 leader 以及 Raft 支持线性语义。这些技术在所有基于一致性的系统中都有应用，Raft 的方案和其它系统的相似。

Raft 的客户端将自己的请求发送到 leader。当一个客户端首次启动，它会随机的选择集群的一台机器。如果客户端的首次选择不是 leader，这台机器将拒绝客户端的请求，并会告知自己最近监听到的 leader。如果 leader 宕机，客户端的请求将会超时，客户端可以随机的选择机器进行重试。

Raft 的目标是实现线性语义（每个操作都是立刻被执行的），然而，目前为止我们描述的 Raft 可以重复多次的执行一条命令；例如，如果 leader 提交了日志但是还没有来得及响应客户端就宕机，那么客户端将会换一个 leader 重试之前的命令。解决方法就是客户端给每个命令一个唯一的编号，那么，状态机记录每个客户端处理的最新的编号。一旦接受到一条命令它的序列号已经被执行过，直接响应这个请求但是不重新执行这个请求。

只读操作可以阻止日志的写入。然而没有额外的机制会增加返回脏数据的风险，因为 leader 响应的请求可能被一个新的不知情的 leader 废除。线性读不能返回脏数据，Raft 需要两个额外的措施来保证而不是通过使用日志。第一，一个 leader 必须知道被提交的日志记录的最新信息。Leader Completeness Property 保证 leader 拥有所有已提交的日志，但是任期刚开始的时候，它可能不知道那些是已提交的。为了弄清楚，它需要在自己的任期提交一条日志。Raft 通过让每个 leader 提在自己刚当选的时候提交一个空操作的日志记录到自己的日志。第二，一个 leader 在处理一个只读日志之前必须检测自己是否被废除（如果最近一个新的 leader 已经被选举，那么它的信息可能是过时的），Raft 需要在响应只读请求之前需要 leader 和半数以上的机器交换心跳信息。或者，leader 可以依靠心跳值来提供租约，但是这样会使安全依赖于时间（因为这里假定有界始终偏移）。

IX. 实现和评估

我们实现了 Raft 用来复制 RAMCloud 的配置信息和支持 RAMCloud 失败协调。我们大约使用了 2000 行 C++ 代码。我们实现的代码已经开源，这里还有 25 独立的第三方机构的开源实现，不同的公司也部署了基于 Raft 的系统。下面的章节通过三方面评估 Raft 协议：易理解性、正确性和性能。

A. Understandability

To measure Raft's understandability relative to Paxos, we conducted an experimental study using upper-level un- dergraduate and graduate students in an Advanced Oper- ating Systems course at Stanford University and a Dis- tributed 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 con- tent of this paper except for log compaction; the Paxos lecture covered enough material to create an equivalent replicated state machine, including single-decree Paxos, multi-decree Paxos, reconfiguration, and a few optimiza- tions needed in practice (such as leader election). The quizzes

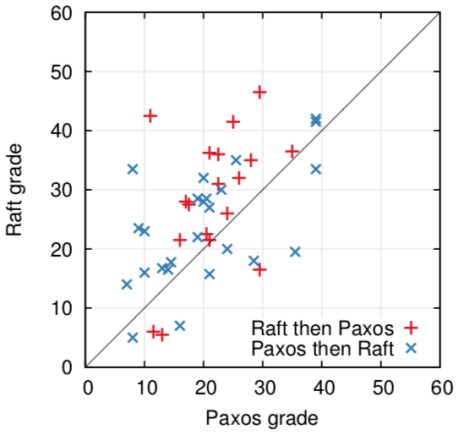

Figure 14: A scatter plot comparing 43 participants' performance on the Raft and Paxos quizzes. Points above the diagonal (33) represent participants who scored higher for Raft.

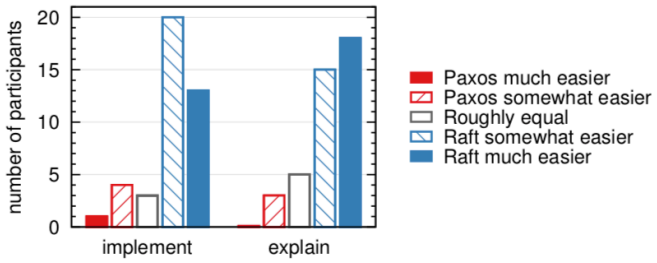

Figure 15: Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

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 deter- mine 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, which they learned the algorithms. The model predicts that the choice of quiz produces a 12.5-point dif- ference 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 sta- tistically 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 over- whelming majority of participants reported Raft would be easier to implement and explain (33 of 41 for each ques- tion). 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].

B. Correctness

We have developed a formal specification and a proof of safety for the consensus mechanism described in Sec- tion 5. The formal specification [31] makes the informa- tion summarized in Figure 2 completely precise using the TLA+ specification language [17]. It

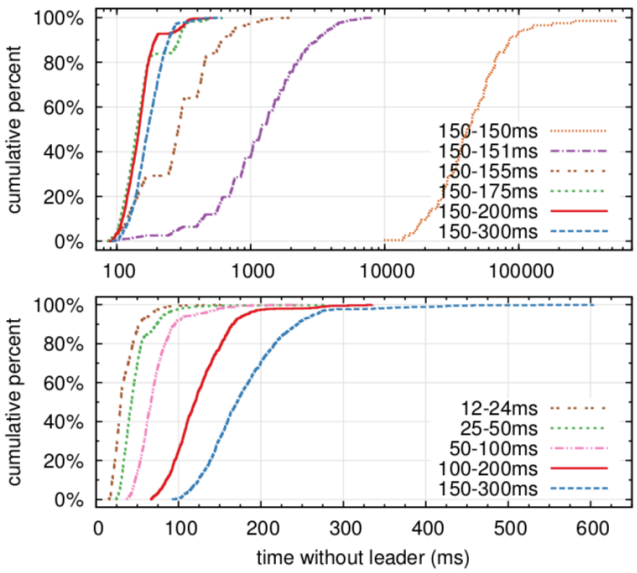

Figure 16: The time to detect and replace a crashed leader. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials (except for 100 trials for "150–150ms") and corresponds to a particular choice of 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).

is about 400 lines long and serves as the subject of the proof. It is also use- ful on its own for anyone implementing Raft. We have mechanically proven the Log Completeness Property us- ing 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-tively precise (it is about 3500 words long).

C. Performance

Raft's performance is similar to other consensus algo- rithms such as Paxos. The most important case for per- formance 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 per- formance. For example, it easily supports batching and pipelining requests for higher throughput and lower la- tency. Various optimizations have been proposed in the literature for other algorithms; many of these could be ap- plied to Raft, but we leave this to future work.

We used our Raft implementation to measure the per- formance 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 Fig- ure 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 en- courage split votes, our test script triggered a synchro- nized 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 crash-ing). 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 time- out.

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 random- ness, leader election consistently took longer than 10 sec- onds 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 $12-24$ms, 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 broad- casting heartbeats before other servers start new elections. This can cause unnecessary leader changes and lower overall system availability. We recommend using a con- servative election timeout such as 150 – 300ms; such time- outs are unlikely to cause unnecessary leader changes and will still provide good availability.

X. Related work

There have been numerous publications related to con- sensus algorithms, many of which fall into one of the fol- lowing categories:

- Lamport's original description of Paxos [15], and at- tempts to explain it more clearly [16, 20, 21].

- Elaborations of Paxos, which fill in missing details and modify the algorithm to provide a better founda- tion for implementation [26, 39, 13].

- Systems that implement consensus algorithms, such as Chubby [2, 4], ZooKeeper [11, 12], and Span- ner [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 de- scription [29] was intertwined with a protocol for distributed trans- actions, 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 or- thogonal to the basic consensus protocol: it serves only as a performance optimization and is not required for achiev- ing consensus. However, this results in additional mecha- nism: 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 con- sensus 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 direc- tions (leaders can receive log entries during the election process); this results in additional mechanism and com- plexity. The published de- scription 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 algo- rithm 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 mem- bership 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 mes- sages are a bit more dense than the other algorithms' , but they are simpler collec- tively. 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 algo- rithm, but it precludes some performance opti- mizations. For example, Egalitarian Paxos (EPaxos) can achieve higher performance under some conditions

with a lead- erless approach [27]. EPaxos exploits commutativity in state machine commands. Any server can commit a com- mand with just one round of communication as long as other commands that are proposed concurrently commute with it. However, if commands that are proposed con- currently do not commute with each other, EPaxos re- quires 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 con- sensus protocol, so that very little additional mechanism is required for membership changes. Lamport's -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 with- out limiting the processing of normal requests; in con- trast, VR stops all normal processing during configura- tion changes, and SMART imposes an -like limit on the number of outstanding requests. Raft's approach also adds less mechanism than either VR or SMART.

XI. Conclusion

Algorithms are often designed with correctness, effi- ciency, and/or conciseness as the primary goals. Although these are all worthy goals, we believe that understandabil- ity is just as important. None of the other goals can be achieved until developers render the algorithm into a prac- tical implementation, which will inevitably deviate from and expand upon the published form. Unless developers have a deep understanding of the algorithm and can cre- ate 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 con- sensus, where a widely accepted but impenetrable algo- rithm, 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 pri- mary design goal changed the way we approached the de- sign 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 tech- niques not only improved the understandability of Raft but also made it easier to convince ourselves of its cor- rectness.

XII. Acknowledgments

The user study would not have been possible with- out the support of Ali Ghodsi, David Mazie‘res, and the students of CS 294-91 at Berkeley and CS 240 at Stan- ford. 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. Spe- cialthanksgotoDavid-Mazie‘resandEzraHochfor finding subtle bugs in Raft. Many people provided help- ful 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, Nico- las 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 ex- posure. This work was supported by the Gigascale Sys- tems Research Center and the Multiscale Systems Cen- ter, two of six research centers funded under the Fo- cus Center Research Program, a Semiconductor Research Corporation program, by

STARnet, a Semiconductor Re- search Corporation program sponsored by MARCO and DARPA, by the National Science Foundation under Grant No. 0963859, and by grants from Facebook, Google, Mel- lanox, NEC, NetApp, SAP, and Samsung. Diego Ongaro is supported by The Junglee Corporation Stanford Gradu- ate Fellowship.

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

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

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

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

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

[6] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KAN- THAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In Proc. OSDI'12, USENIX Conference on Operating Systems Design and 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. Me´ry, Eds., vol. 7436 of Lec- ture Notes in Computer Science, Springer, pp. 147–154.

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

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

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

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

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

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

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

[15] L A M P O RT, L . The part-time parliament. ACM Transac- tions on Computer Systems 16, 2 (May 1998), 133–169.

[16] LAMPORT, L. Paxos made simple. ACM

SIGACT News 32, 4 (Dec. 2001), 18–25.

[17] L A M P O RT, 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] L A M P O RT, L . Fast paxos. (2006), 79–103. Distributed Computing 19, 2

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

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

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

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

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

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

[26] MAZIE‘ RES, D. Paxos made practical. http://www.scs.stanford.edu/ ˜dm/home/papers/paxos.pdf, Jan. 2007.

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

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

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

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

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

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

[33] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIE‘RES, D., MI- TRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. Com- munications of the ACM 54 (July 2011), 121–130.

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

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

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

[37] S C H N E I D E R , F. B . Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Com- puting Surveys 22, 4 (Dec. 1990), 299–319.

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

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