什么是共识

- **共识（Consensus） & 一致性（Consistency ）**

共识（Consensus）这个术语很多时候会与一致性（Consistency）术语放在一起讨论。严谨地讲，两者的含义并不完全相同。

一致性的含义比共识宽泛，在不同场景（基于事务的数据库、分布式系统等）下意义不同。

分布式系统场景下：
一致性指的是多个副本对外呈现的状态。顺序一致性、线性一致性，描述了多节点对数据状态的共同维护能力。
共识，则特指在分布式系统中多个节点之间对某个事情达成一致看法的过程。因此，达成某种共识并不意味着就保障了一致性。

实践中，要保障系统满足不同程度的一致性，往往需要通过共识算法来达成。

共识算法解决的是分布式系统对某个提案（Proposal），大部分节点达成一致意见的过程。提案的含义在分布式系统中十分宽泛，如多个事件发生的顺序、某个键对应的值、谁是主节点……等等。可以认为任何可以达成一致的信息都是一个提案。

对于分布式系统来讲，各个节点通常都是相同的确定性状态机模型（又称为状态机复制问题，State-Machine Replication），从相同初始状态开始接收相同顺序的指令，则可以保证相同的结果状态。因此，系统中多个节点最关键的是对多个事件的顺序进行共识，即排序。

问题与挑战

无论是在现实生活中，还是计算机世界里，达成共识都要解决两个基本的问题：

- 首先，如何提出一个待共识的提案？如通过令牌传递、随机选取、权重比较、求解难题等；
- 其次，如何让多个节点对该提案达成共识（同意或拒绝），如投票、规则验证等。

不同节点之间通信存在延迟（光速物理限制、通信处理延迟），并且任意环节都可能存在故障（系统规模越大，发生故障可能性越高）。
如通信网络会发生中断、节点会发生故障、甚至存在被入侵的节点故意伪造消息，破坏正常的共识过程。

一般地，把出现故障（Crash 或 Fail-stop，即不响应）但不会伪造信息的情况称为“非拜占庭错误（Non-Byzantine Fault）”或“故障错误（Crash Fault）”；伪造信息恶意响应的情况称为“拜占庭错误”（Byzantine Fault），对应节点为拜占庭节点。显然，后者场景中因为存在“捣乱者”更难达成共识。

此外，任何处理都需要成本，共识也是如此。当存在一定信任前提（如接入节点都经过验证、节点性能稳定、安全保障很高）时，达成共识相对容易，共识性能也较高；反之在不可信的场景下，达成共识很难，需要付出较大成本（如时间、经济、安全等），而且性能往往较差（如工作量证明算法）。

常见算法

根据解决的场景是否允许拜占庭错误情况，共识算法可以分为 Crash Fault Tolerance (CFT) 和 Byzantine Fault Tolerance（BFT）两类。

对于非拜占庭错误的情况，已经存在不少经典的算法，包括 Paxos（1990 年）、Raft（2014 年）及其变种等。这类容错算法往往性能比较好，处理较快，容忍不超过一半的故障节点。

对于要能容忍拜占庭错误的情况，包括 PBFT（Practical Byzantine Fault Tolerance，1999 年）为代表的确定性系列算法、PoW（1997 年）为代表的概率算法等。确定性算法一旦达成共识就不可逆转，即共识是最终结果；而概率类算法的共识结果则是临时的，随着时间推移或某种强化，共识结果被推翻的概率越来越小，最终成为事实上结果。拜占庭类容错算法往往性能较差，容忍不超过 1/3 的故障节点。

此外，XFT（Cross Fault Tolerance，2015 年）等最近提出的改进算法可以提供类似 CFT 的处理响应速度，并能在大多数节点正常工作时提供 BFT 保障。

Algorand 算法（2017 年）基于 PBFT 进行改进，通过引入可验证随机函数解决了提案选择的问题，理论上可以在容忍拜占庭错误的前提下实现更好的性能（1000+ TPS）。

理论界限

科学家都喜欢探寻问题最坏情况的理论界限。那么，共识问题的最坏界限在哪里呢？很不幸，在推广到任意情况时，分布式系统的共识问题无通用解。

这似乎很容易理解，当多个节点之间的通信网络自身不可靠情况下，很显然，无法确保实现共识（例如，所有涉及共识的消息都丢失）。那么，对于一个设计得当，可以大概率保证消息正确送达的网络，是不是一定能获得共识呢?

理论证明告诉我们，**即便在网络通信可靠情况下，一个可扩展的分布式系统的共识问题通用解法的下限是——没有下限（无解）。**

Raft共识算法

Raft简介

Raft是一种新型易于理解的分布式一致性复制协议，由斯坦福大学的Diego Ongaro和John Ousterhout提出，作为RAMCloud项目中的中心协调组件。Raft是一种Leader-Based的Multi-Paxos变种，相比Paxos、Zab、View Stamped Replication等协议提供了更完整更清晰的协议描述，并提供了清晰的节点增删描述。它在容错和性能上与Paxos等效。不同之处在于它被分解为相对独立的子问题，并且干净地解决了实际系统所需的所有主要部分。

Raft作为复制状态机，是分布式系统中最核心最基础的组件，提供命令在多个节点之间有序复制和执行，当多个节点初始状态一致的时候，保证节点之间状态一致。系统只要多数节点存活就可以正常处理，它允许消息的延迟、丢弃和乱序，但是不允许消息的篡改（非拜占庭场景）。

复制状态机

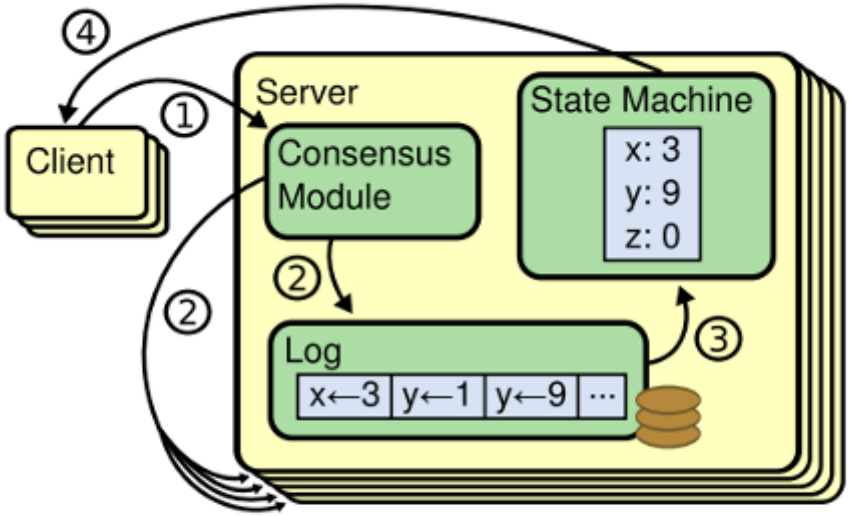
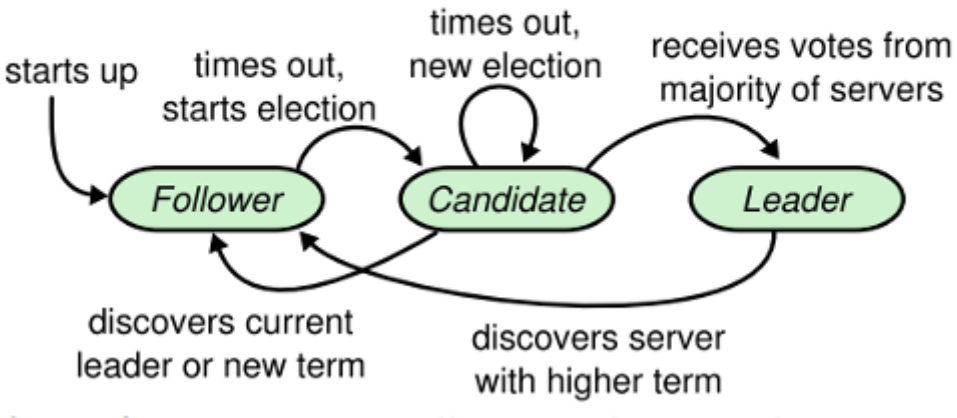


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

复制状态机：只要给予两个机器相同的初始状态和输入数据，他们就会得到完全相同的输出序列。
而Raft共识算法的目的就是实现一个复制状态机集群，保证每个可用的机器的输出序列是一致的，从而实现数据的高可用性，而在此之上也保证了算法的性能。

Raft的三个子模块

1. 领导人选举



领导人是raft集群中的核心节点，负责整个raft集群对外部请求的响应，以及内部行为的调度，具有十分重要的作用，因此，raft集群中的一个关键问题就是，如何安全地选举出一个领导人。
而这个问题的关键在于，首先要知道，领导人的职责以及它与raft的日志的关系，这样才能清楚我们到底需要什么样的领导人。

日志：日志是raft中的核心数据单元，可以把它理解为raft的具体的指令，也就是我们希望raft集群统一去执行的任务。
这里我们去联想复制状态机的定义，复制状态机是对于一整个集群，输入相同的序列都可以得到相同的输出，或者让集群执行相同的任务，而raft共识算法的目的就是实现一个复制状态机集群，那么我们就发现，日志就是我们对于复制状态机集群的输入序列。

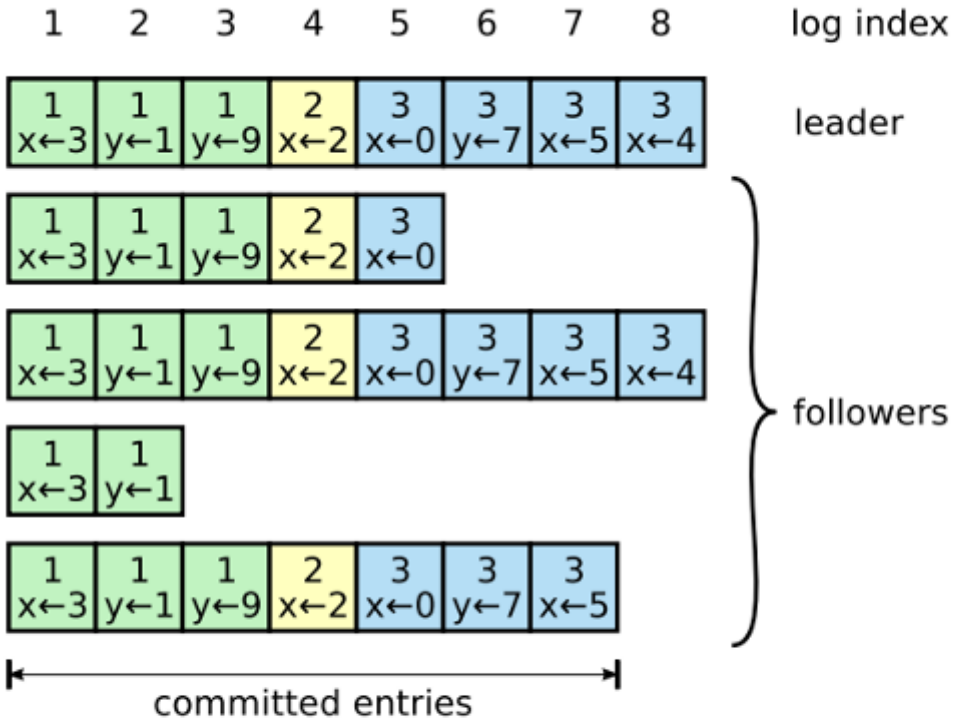
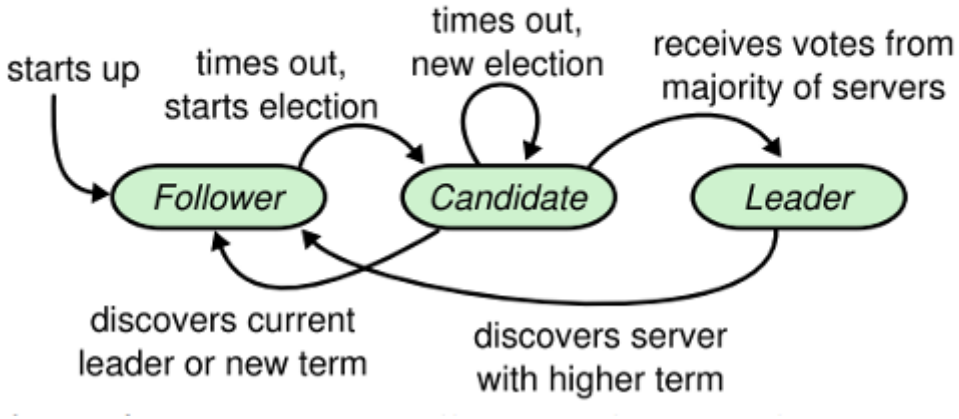
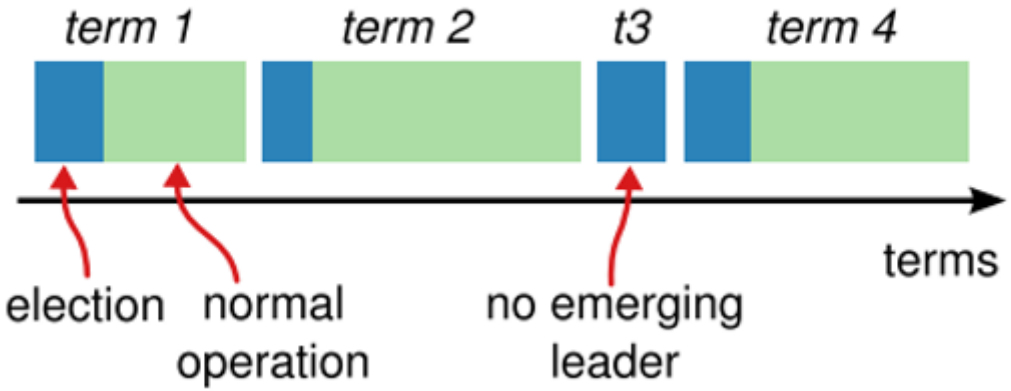


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

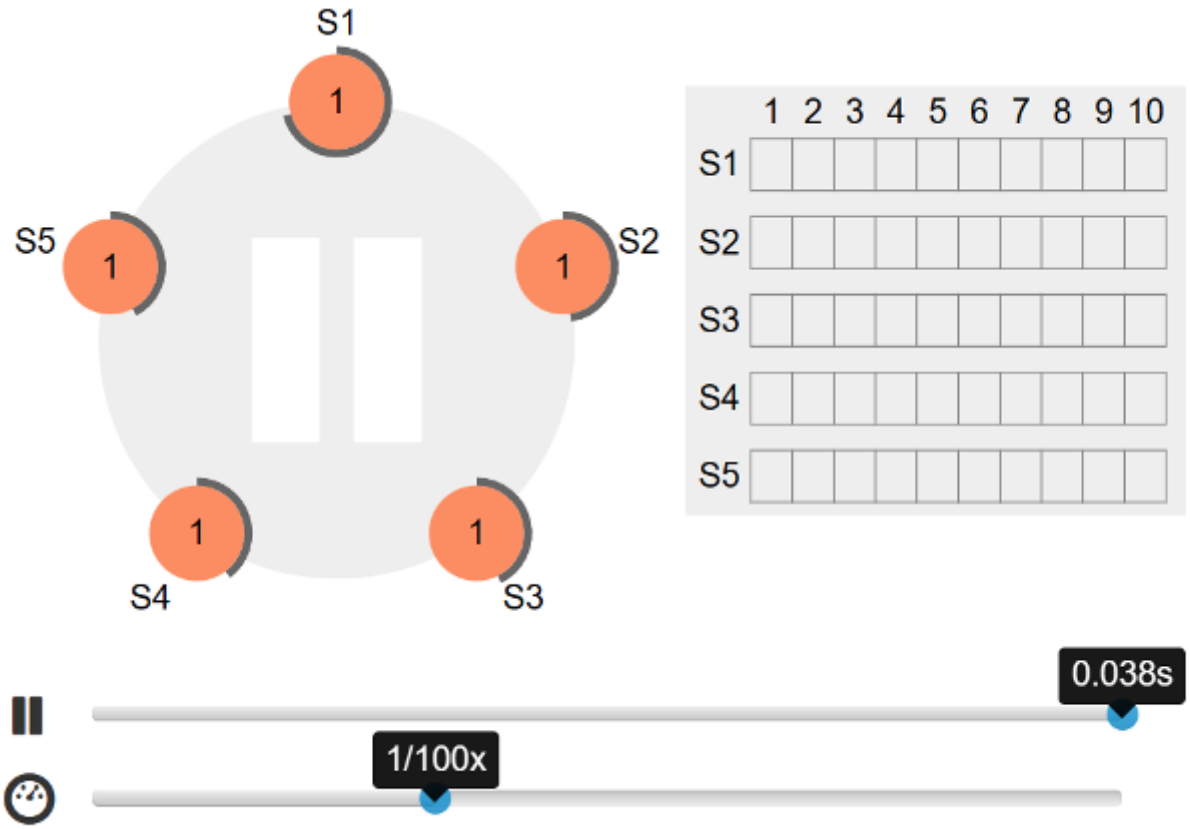
Raft的日志序列：每一个日志都有一个唯一的索引号（理解为数组的下标），以及一个任期号，通过这两个数据，我们可以不需要读取日志中的具体的内容去统一的确定一个唯一的日志。

领导人的职责就是，把日志安全的复制到集群中的每一个节点，然后让每一个节点分别应用这份日志，从而执行并输出结果。

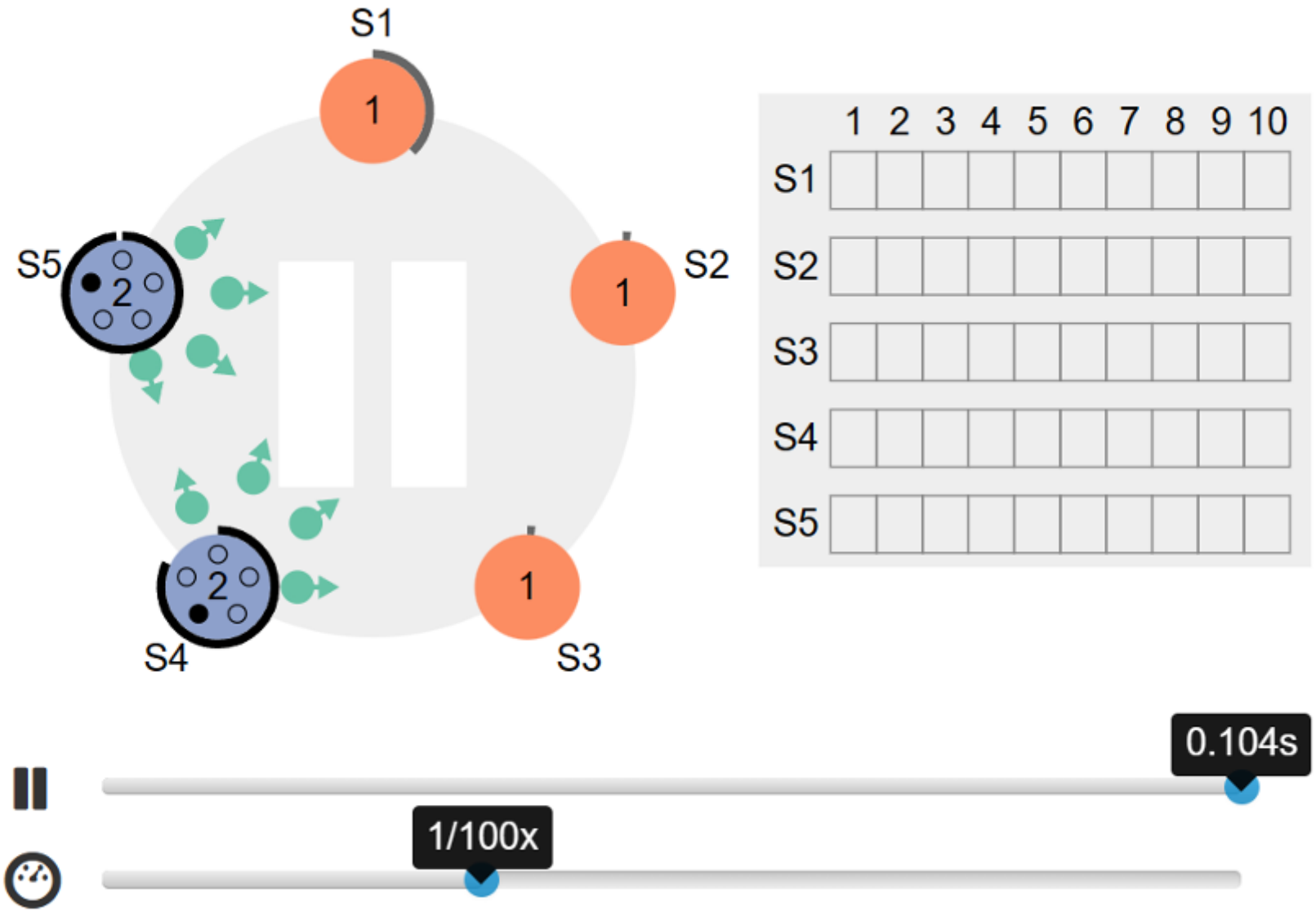




那么，我们就可以来分析领导人选举的问题，以上这张图就是Raft集群中的三个状态
而任期(term)，则是Raft共识算法中的时间单位，一个任期只是一个逻辑时间计数器，而不代表实际的时间的长短。



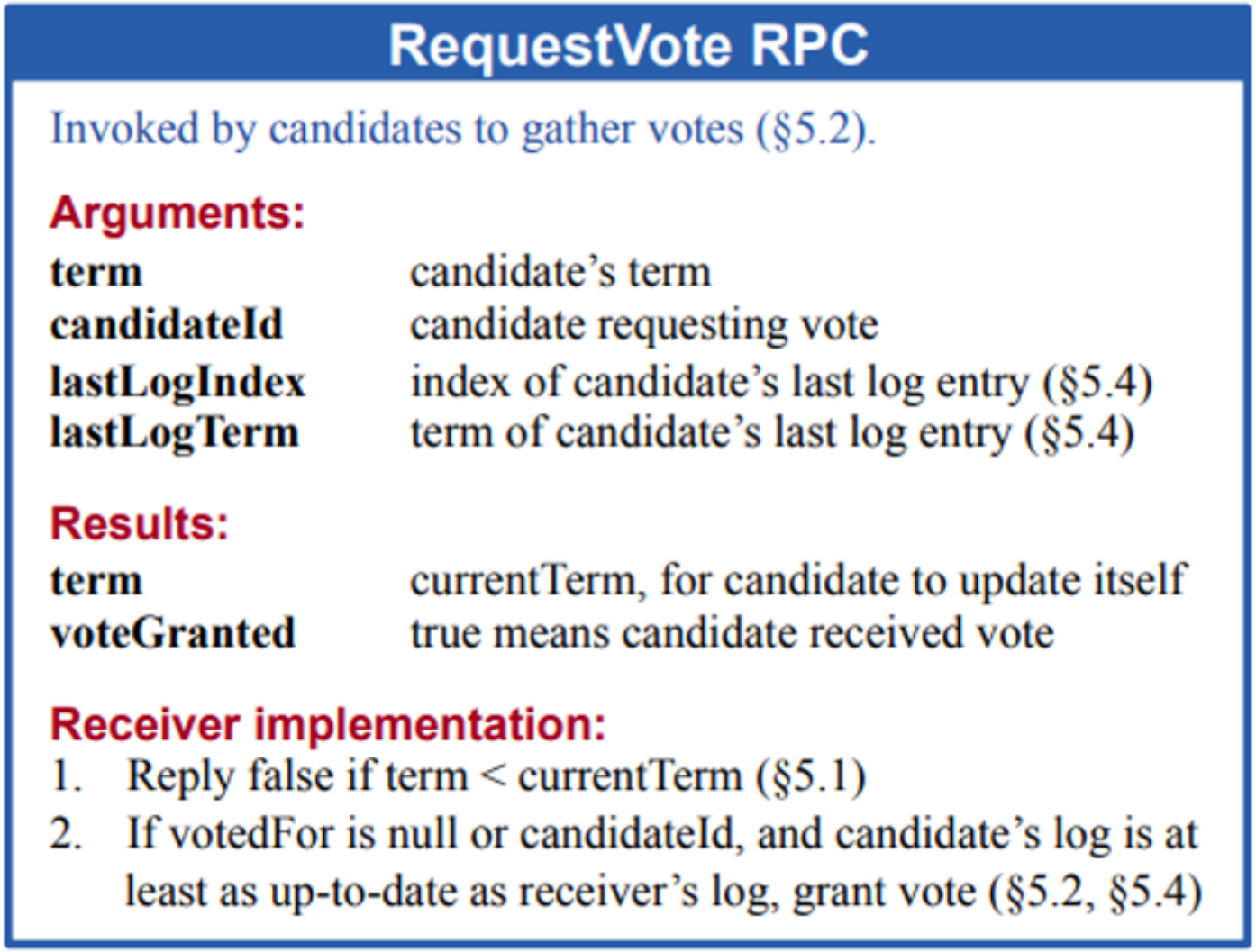
起初 所有的节点都处于跟随者(Follower)的状态，任期都是1，每一个Follower都存在一个随机的计时器，在这个随机的计时器的倒计时结束前，如果他都没有收到领导人(Leader)的心跳，他就会认为当前集群是没有领导人的，因此就会转变为候选者(Candidate)状态。



如图，S5和S4 两者同时在随机计时器结束后都没有收到来自领导人的心跳，因此他的状态转为了候选者，自身的任期号加一，两者都发起了领导人选举(Leader Election)。

在每次选举中，每个候选者都会给自己投一票，而拒绝给其他候选人投票，跟随者在一个任期的选举中只能按照先到先得的原则投一票。

当候选人得到了超过集群半数的选票以后，他便可以当选Leader。
而如果在这一轮任期的选举中，所有候选人都没有得到足够的选票，那么在计时器超时后，候选人会重置得到的选票，并且将自身的任期号加一，发起下一轮任期的选举。

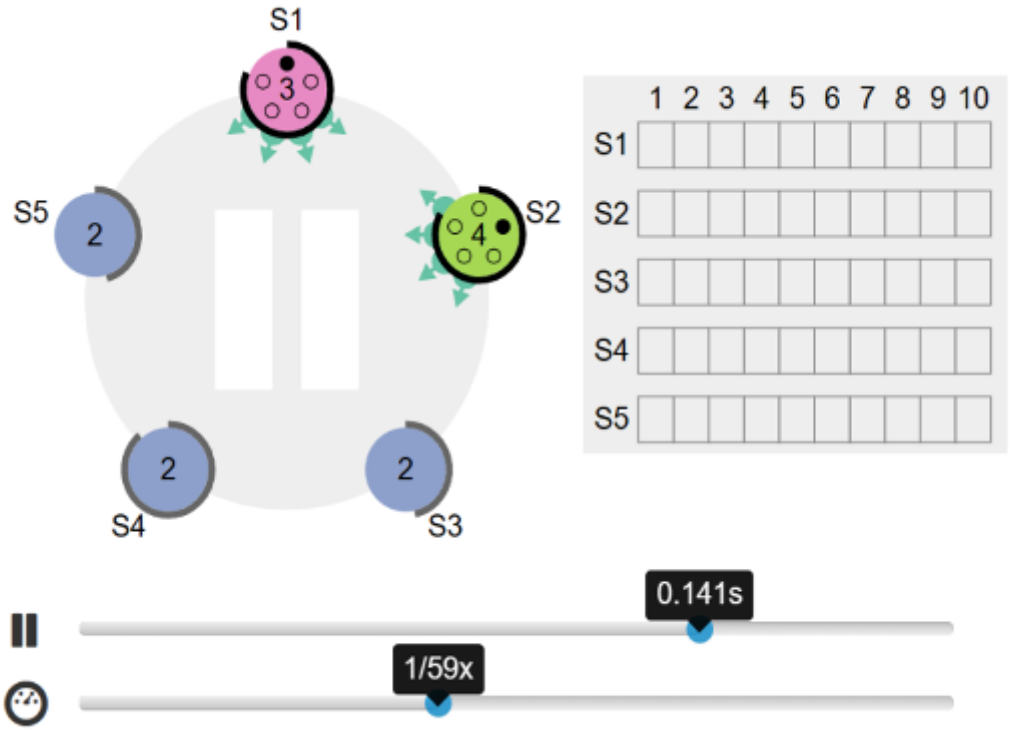


RPC是分布式系统中最重要的通信方式之一，几乎所有开源的Raft项目都是基于RPC的通信实现的。
而在请求投票RPC(Request Vote RPC)中，我们可以看到几个参数：候选人自己的任期号，候选人对应的服务器的编号，以及日志的相关数

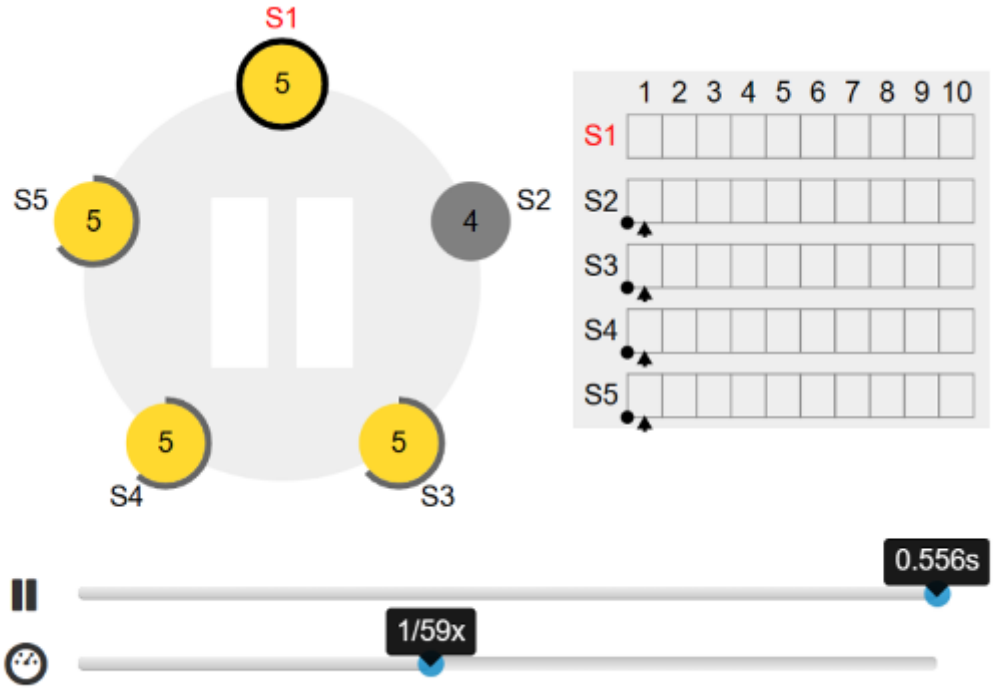
据，我们先不对日志的信息做解析，这个留到日志复制的时候再去解析。

在Raft中，任期号无疑是一个很重要的时间单位，对于任期号，我们有如下规则：
当双方利用rpc通信时，如果发现对方的任期号比自己大，那么自己的状态就必须马上转换为跟随者，并将自己的任期号调节为更大的一方。

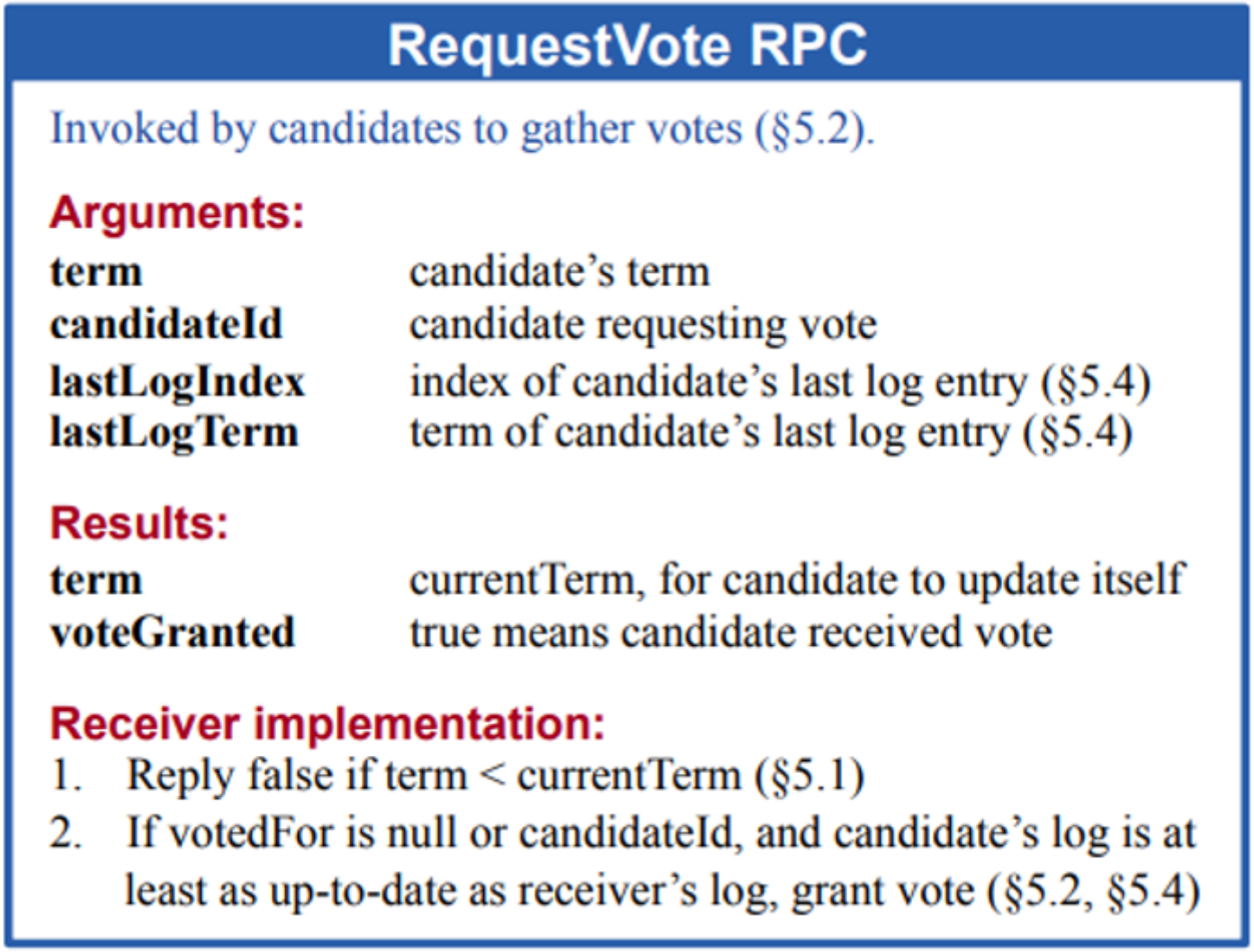
所谓安全性，就是防止一个问题，也就是这个集群会不会出现两个领导人，如果出现了两个领导人该如何防止呢?



在这种情况下，出现了两个不同任期的候选者，由于每个任期，每个跟随者都有一票的投票资格，所以S1和S2是有可能同时得到过半的选票而成为领导人的，但是S2如果向S1发投票rpc的话，由于S2的任期比S1要大，S1会马上从候选人状态转变为跟随者，并将任期调整为4，或者说如果S1成为了任期3的领导人，由于接到了S2的任期4的rpc，他也会马上从领导人变回一个跟随者。



而这种情况下，如果S2在任期4担任领导人，而他发生了网络故障，导致他一直认为自己是任期4的领导人，而S1通过超时，重新选举成为了任期5的领导人，那么当S2网络恢复正常以后，也会向其他节点发送心跳，而由于他的任期号比较小，所以他会马上从领导人变回跟随者。



通过分析选举的过程，我们可以发现，只要集群中仍然存在过半的正常互相通信的节点，我们就仍然可以正常选举出领导人，并维护整个集群的正常运行。

而任期号的限制则保证了在任意一个任期，只会存在一个领导人，这就避免了分布式系统中最棘手的“脑裂”问题（也就是存在两个稳定运行的领导人从而干扰了整个集群的正常运作）。

但是，需要注意的是，上述提到的选举机制，是不能够保证集群的数据安全的，在这里，我们可以看到，请求投票RPC中的参数，有一个 上一个日志的索引号(lastLogIndex)和 上一个日志的任期(lastLogTerm)，这两项参数则是用来保证数据安全的。

2.日志复制

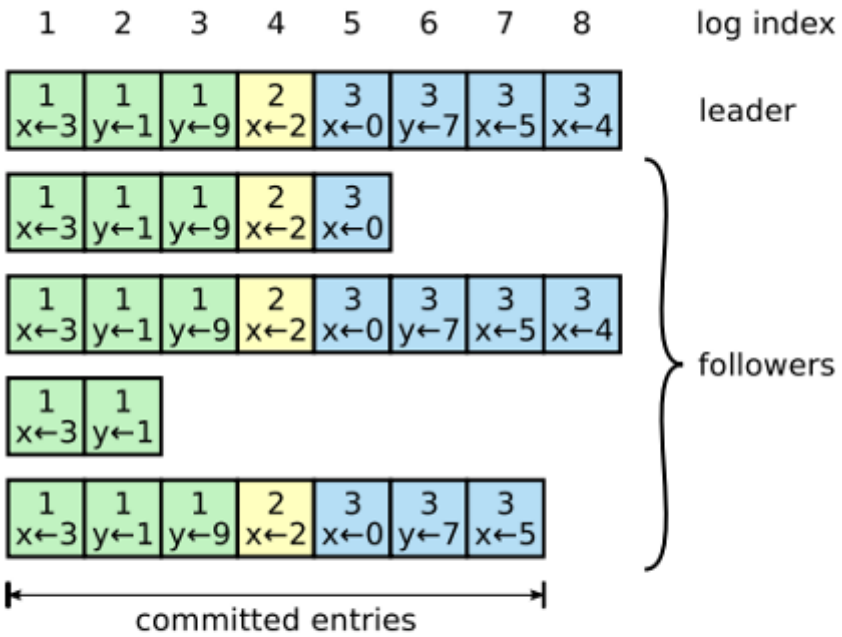


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

我们前面也提到过，当一个节点成为领导人后，他需要将自己的日志，复制到其他的跟随者节点，而这，正是复制状态机的实现关键，只要领导人将自己的日志都正确，安全地复制到了跟随者节点，那么是不是就意味着，不管我们外部向任意一个集群发起指令请求，他都会帮助我们找到领导人，而领导人会帮我们将该请求的日志信息都复制到所有的节点，并执行日志的请求，所以无论我们向集群的任意的节点发起输入，最终得到的输出是不是都是一致的呢?

当然，目前我们提及的Raft共识算法的规则还不足以保证绝对的安全性，我们还需要继续分析这个算法。

AppendEntries RPC

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

Arguments:

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

Results:

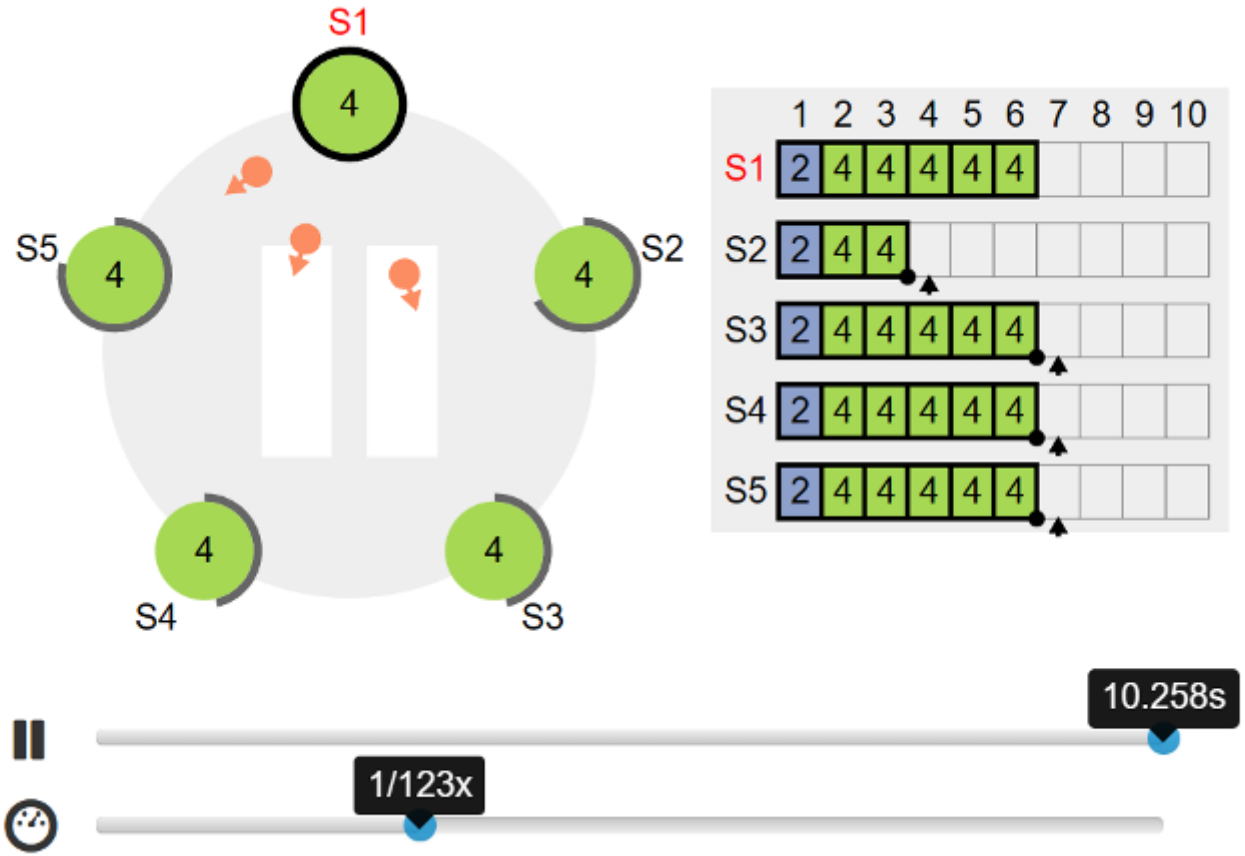
term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

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

首先我们来看一下 日志追加RPC(Append Entries RPC)，这就是Raft集群的领导人会向每个节点发起的心跳，在领导人选举的分析中我们也提到过，只要集群中的节点收到了这个心跳，那么他就不会转变为跟随者。
集群中的跟随者在收到了日志追加RPC以后，会将领导人发过来的日志(entries[]参数)复制到自己的节点，复制完成以后会返回success状态给领导人，表明自己完成了复制任务。
而领导人在收到过半的成功的答复后，就会开始尝试提交当前的日志，然后将结果返回给客户端，表明自己已经执行了这个指令。

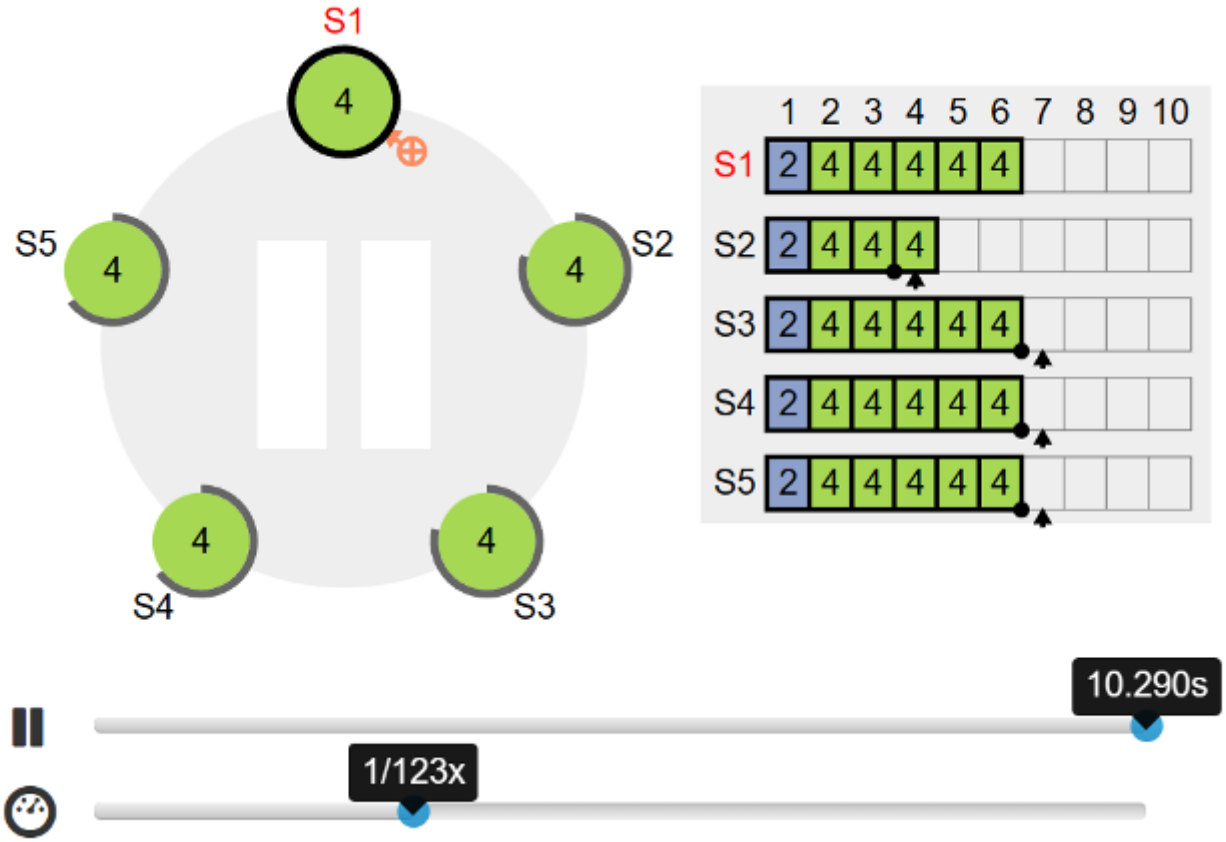
大家可以设想，这个日志复制是不是无条件的呢？当然不是，我们来看下面的一个场景，从而解析 prevLogIndex和prevLogTerm两个参数的作用



在这种情况，S1机器是当前任期4的领导人，而S2机器由于先前发生了崩溃，刚刚恢复，导致他缺失了这个任期不少的日志，在这种情况下，集群中的机器就出现了日志不一致的情况。

因此，在发送心跳的时候，S1会加上自己的上一个日志的任期号和上一个日志的索引号，而跟随者S5会对此执行一致性检查，也就是会检查S1的上一个日志的任期号和索引号，是否跟自己的上一个日志的任期号和索引号一致，假如两者一致，那么就说明在这个索引号前的S5和S1的日志是一样的，然后就可以把日志安全地复制到自己的机器上，但是图中的这种情况是显然不一致的，所以第一次rpc会发送索引6，而第二次rpc就会发送索引5，从而一点一点往前回溯，去找到双方日志序列最后相同的地方。

这就是 prevLogIndex和prevLogTerm的作用，通过这两个参数以及一致性检查机制，raft集群得以保障各个机器上的数据的一致性。



在找到双方最后相同的日志以后，和上图一样，S5就可以把S1发过来的日志直接复制下来，那么一次日志追加的任务就算成功完成了。

我们前面也提到了，领导人在成功地将一项日志复制到过半的节点以后，就可以执行提交了，执行了提交也就意味着这个结果可以返回给客户端了。
可是从日志复制，到日志提交，再到领导人将提交的结果通过心跳告知各个跟随者节点，都是需要时间的，这中间如果领导人或者某个跟随

者发生了崩溃，难道不会导致各个机器的不一致的情况吗?
我们目前的日志复制规则仍然有一些漏洞，会导致各个节点中的日志不一致的情况，我们下面从一个选举场景去入手。

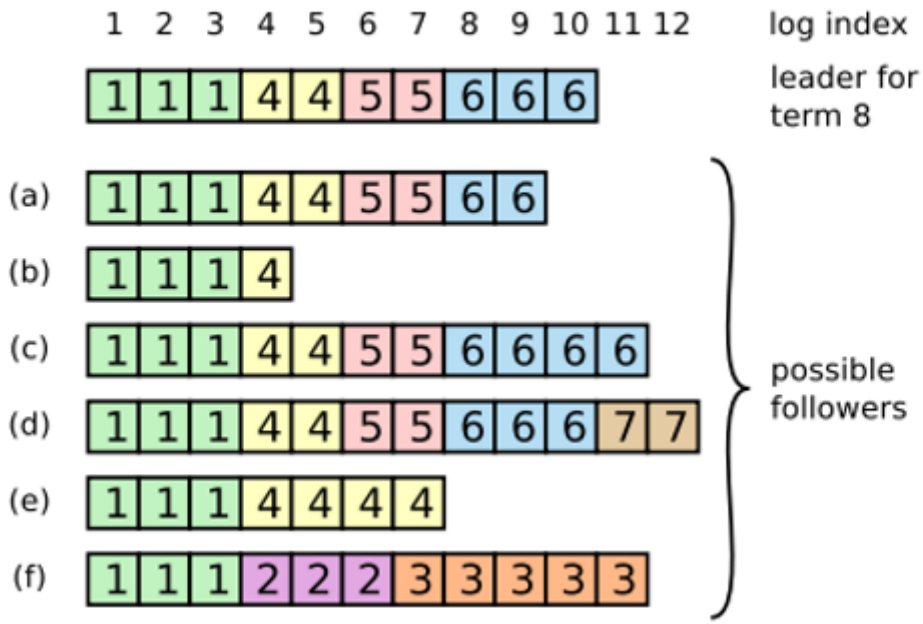


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example,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

这里，由于集群的某些机器发生了多次的崩溃和重启，导致各个节点的日志出现了各种不一致的情况。
那么我们来分析一下，假如在这个场景下，是不是所有的机器都能够当选领导人呢?
我们可以看到，现在索引4 5 6 7 8 9的日志都已经被复制到了过半的节点，所以他们应该也已经正确地提交上去了才对。
那么如果f在这时候当选了领导人，会发生什么后果呢?
f节点跟其他机器的最后相同的日志是索引3的日志，这就意味着它会将自己的索引4到9的日志全部覆盖到其他的节点，那么我们已经正确提交的日志就被覆盖了！这是非常危险的行为。
因为在共识算法里面，提交就代表着我们的集群已经对这一个日志达成了共识，这时候如果把已经提交的日志覆盖，那就是将达成的共识撤销了，而共识算法的设计目的就是不能够撤销已达成的共识，因此，这样的选举无疑是严重的影响安全性的!
那么我们回过头来看，请求投票的RPC的两个参数，正是用来保证安全性而做出的选举限制。

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)

每个候选者在发出选举投票的RPC的时候，都会携带他最后一项日志的任期号和索引号(lastLogIndex and lastLogTerm)，而每个收到rpc的跟随者都会将他与自己的最后一项日志的任期号和索引号比对，如果收到的rpc的最后日志任期号比自己的小，或者如果任期号相同，但是日志的索引号比较小，那么都说明他所具备的日志相比于自己具备的日志是过期的，那么他就会拒绝投票。

那么大家可能有疑问，这里c机器和d机器所拥有的日志是最新的，为什么并没有具备最新日志的机器（当前领导人）能够当选领导人呢？我们以此来分析一下左图的情况，虽然d机器拥有的日志是最新的，但是他并没有成功地将机器复制到过半数的节点，所以我们当前的领导人选

举的时候就可以凭借这个规则成功获取到a b e f的选票，那么也是超过半数的，也能够成功当选领导人，而在当选领导人以后，即使c机器和d机器拥有的日志比较新，那也没有关系，因为他们没有将这份日志复制到过半节点，所以也就没有提交，那么将这些日志覆盖了也是没有关系的。

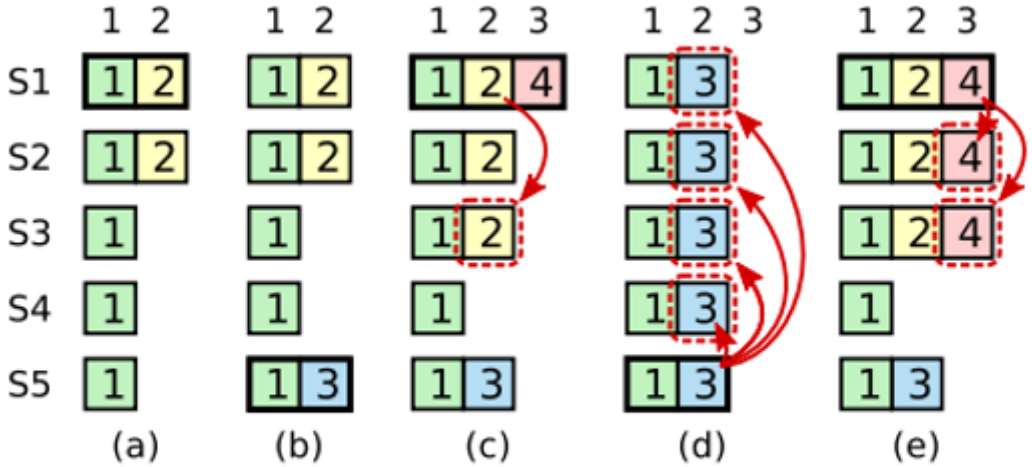


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at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

关于raft的安全性问题，我们还有一个问题没有解决，我们说，领导人将一项日志复制到过半的节点后，就可以执行提交了，那么是不是绝对如此呢？我们来看左边这样一种情况：

S1在任期2当选领导人，然后发生了崩溃，而S5在任期3当选了领导人，而S5又发生了崩溃，S1在任期4再次当选了领导人
这时候，S1会按照规则，先将任期2的日志复制到过半的机器上，那么，这个时候，就可以提交了吗？那么假如我们在此时提交了日志2，会发生什么后果呢?
我们接下来看图d，S1在将任期2的日志复制到各个机器上面以后，结果S1再次发生了崩溃，S5再次担任了领导，那么他也会将索引2 任期3的日志复制到各个机器上，这就把日志2覆盖了！
所以我们需要加一条限制，那就是，一个机器只能提交当前任期的日志，老日志会随着当前任期的日志的提交一并提交。
加上这条限制以后，假如到了图e的场景，假如任期4的领导人S1在提交了索引3任期4的日志后再次发生了崩溃，S5也会由于选举的限制（索引和任期）而无法当选领导人，从而也不会发生已提交的日志被覆盖的情况。

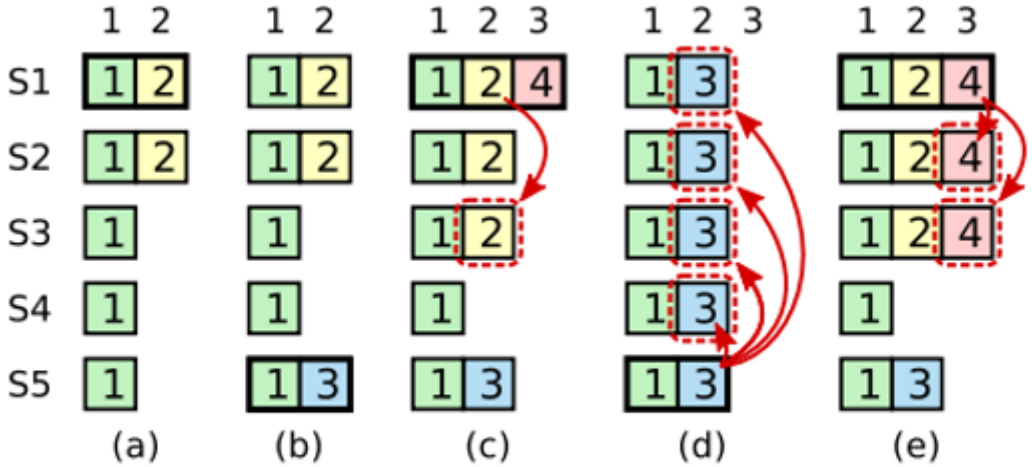


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at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

以上的各个场景中，我们会忽略一个重要的点，那就是在领导人提交日志后，到各个节点得知领导人已经提交日志期间，是有一段时间间隔的，在这个时候，如果领导人已经提交了日志，返回结果给了客户端，但是在通知各个节点成功提交前就发生了崩溃，那么不就出现了这样

一种情况，就是一台机器把一项日志提交了，其他机器都没有提交，这样会影响集群的日志的一致性吗?
答案是不会的，我们可以看左边的图片的e场景，假如S1在提交后发生崩溃，S2和S3的日志都还没有正确地提交，但是，由于这个时候，索引2和3日志已经复制到了过半的节点，所以也只有这两个节点能够当选领导人，而这两个节点当选领导人以后也会将这些日志正确地提交上去，所以在Raft共识算法中，只要有一个单点提交，就意味着，我们可以保证只要后续的集群稳定运行（存在过半可用节点），就一定会将日志正确提交，所以单点提交就意味着集群的整体的提交。

3. 安全性

实际上，Raft的第三个子模块——安全性的问题，是融汇在领导人选举和日志复制的一系列的条条框框里面的，通过以上的限制，我们是有足够的办法来保证集群的数据的安全性的

Raft的简洁优美，易于理解与实现，以及性能高效的特点，是得到了工业实践的证明的，虽然Raft是一个2014年才提出来的架构，但是，在如今的分布式存储系统的架构中，绝大多数系统的首选的算法就是Raft共识算法，比起Paxos，以及其他十多种的非拜占庭式共识算法来说，Raft毫无疑问是优美的，维护成本更低的。
而大名鼎鼎的Etcd，braft，tikv这类开源的分布式存储系统项目，正是基于Raft实现的。

本次系列课程就到此结束。