

LECTURE 6 RAFT

1. 背景

1. raft与其他共识算法

- Raft是一个**共识算法 (Consensus Algorithm)**
- Raft的设计受到Paxos影响。过去Paxos在共识算法上具有主导性地位，但是不好理解，具体实现也很复杂
- Paxos的缺点
 - 难以理解
 - 实现层面存在问题：主要是指multi-Paxos没有公认的算法。有一些论文和idea，但是都不明确或者缺失细节。对复制状态机架构的log一致性维护不太友好。若使用single-decree-Paxos一条一条log entry进行选择，而后拼接log，代价太大，而且编码复杂
 - 使用一种对称P2P的方式（集群当中每一台计算机都可以作为server或client，削弱了leader的作用）。这一方式在理想的，单一请求的场景下比较合适，但与现实需求不符。在多请求场景下，选出一个leader而后进行协调会更合适
- Paxos算法和实际应用的问题很好地体现了学术成果应用到工业界的困难性

2. Replicated state machine architecture 复制状态机架构。GFS、HDFS、VMware FT都属于这一类

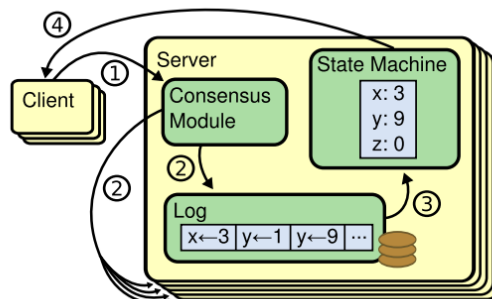


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

3. 拜占庭将军问题：一组拜占庭将军分别各率领一支军队共同围困一座城市。为了简化问题，将各支军队的行动策略限定为进攻或撤离两种。因为部分军队进攻部分军队撤离可能会造成

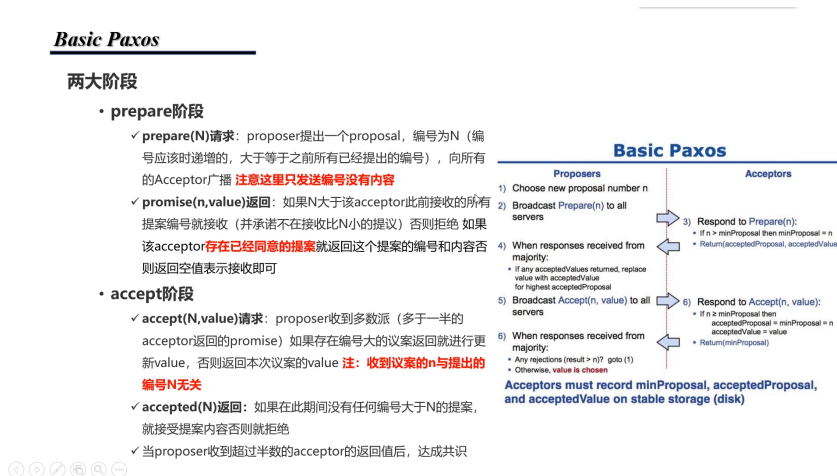
灾难性后果，因此各位将军必须通过投票来达成一致策略，即所有军队一起进攻或所有军队一起撤离。因为各位将军分处城市不同方向，他们只能通过信使互相联系。在投票过程中每位将军都将自己投票给进攻还是撤退的信息通过信使分别通知其他所有将军，这样一来每位将军根据自己的投票和其他所有将军送来的信息就可以知道共同的投票结果而决定行动策略。系统的问题在于，可能将军中出现叛徒，他们不仅可能向较为糟糕的策略投票，还可能选择性地发送投票信息。[wikipedia传送门](#)

- **非拜占庭/拜占庭错误**：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。

4. 共识算法的属性

- 安全性：均可保证在非拜占庭错误之下的安全性
- 正常工作特性/可用性：只要系统的大部分机器都在正常运转并且可以相互通信，则共识算法就是可用的
- 不依赖时间来确保log的一致性（consistency）
- 集群的大多数机器对单轮的RPC响应了就代表一次任务完成，而不需要考虑慢速服务器。慢速服务器对集群性能无影响

5. Paxos算法流程简述



6. Partition（分区）问题：网络被分成了两部分，两部分之间不能进行通信

7. Majority：在多数投票系统（Majority Voting System）中，majority通常需要计算整个集群的majority，而不是依然工作的server的majority。并且，新的leader的majority必须要与前一个leader的majority至少有一台机器是一样的（从数学上很好理解，给两个机器投票的服务器至少是一半，因此必定有重叠）

8. log的作用

- log指出了operation执行的顺序
- follower需要暂存数据：在某entry已经append但是尚未commit的情况下，暂存该entry
- 在某些follower没有收到leader请求的情况下，通过保留的operation操作副本来恢复follower的状态

9. TLA+ specification language：可用于测试分布式系统性能

2. 基本架构

1. 设计思路

- 分治思想：将共识算法分解成为容易解决的小问题
- 减少状态数目，从而减小状态空间
 - 使用了一些随机化策略，尽管会带来不确定性（设计中不希望有不确定性），但是可以更容易理解

2. 服务器被分为leader和其他server

- leader需要被选择出来，当leader故障的时候，需要重选leader

3. leader的作用

- 接受来自其他server的log entries
- 自行决定新的log entry加在log的什么位置
- 相应client请求

4. raft的两个重要环节

- leader选择
- log复制

3. 原理

3.1. leader选择

1. server状态变化：每一个server的状态必定处于follower、candidate与leader其中之一。server状态变化如图所示。

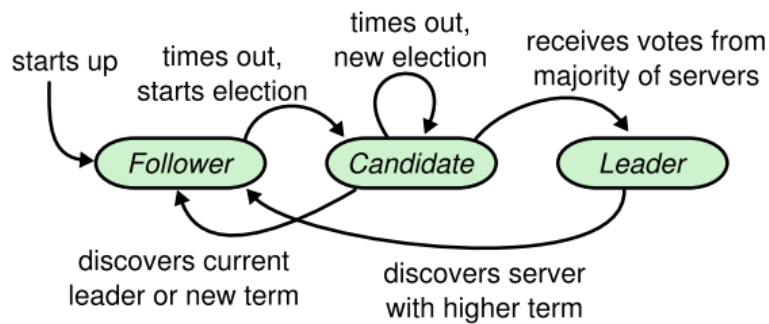


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

- follower不响应client的请求，如果client向follower发起请求，follower会把请求移交给leader。follower只响应其他server的请求，当一直得不到其他server的通信时（超过指定的时间，这一时间称为election timeout），follower会发起leader选举。leader会定期通过AppendEntries RPC发布Heartbeat信息，因此在集群工作正常的情况下，未收到通信也就意味着没有leader
- 一个集群至少需要三台server才能工作
 - 进一步，一个有 $2n + 1$ 台server的集群，可以容许 n 台机器的failure。这种情况下，剩下的 $n + 1$ 台依然能够构成majority投票选出新的leader
- 三个状态的作用：
 - leader：见前文
 - follower：接受leader和candidate的请求。follower本身不会发起请求
 - candidate：选举leader

2. term机制：将时间分成若干个term，term长度任意，由整数进行连续递增编号

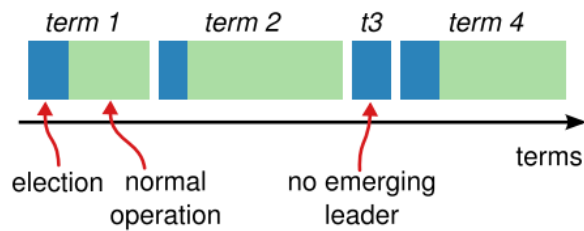


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different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

- 一个任期有leader选举开始，而后leader在此任期内协调相关的操作。若没有选出leader，则任期结束
- 同一时间至多一个leader
- current term
 - 每一台server都有，保存在磁盘上。存储的是当前term的编号
 - server之间通信时会更新current term（如果发现某个server的current term比自己的大，则会把自己的current term更新成更大的）。
- 异常情况：
 - 某台server可能没有观察到谁是leader，甚至没有观察到term
 - leader在发现自己任期到了之后，会转变成为follower状态（有可能leader的通信有问题，没有在term到期时立即切换状态）
 - follower不会响应过期（stale）的leader的请求

3. RPC

- server之间通信使用RPC
- 类别：
 - RequestVote RPCS：由candidate发起
 - AppendEntries RPCS：由leader发起，也用于提供heartbeat信息（不带有log entry信息即可）
 - snapshot RPCS

- server并发地发布RPC。这样性能更好（投票时一个一个询问，比一次性发出投票信息更慢）

4. election流程

1. follower递增current term，转变到candidate状态
2. 为自己投票，并且发布RequestVote RPCs 到其他server
3. 当以下三种情况之一发生时，转变为follower状态：
 - 赢得选举：赢得多数票则称为leader。一个server只能投一票，这保证了一个任期至多只有一个leader。在多个candidate的情况下，每个server使用先来先服务的策略。成为leader之后，leader向所有的server发送heartbeat消息宣布自己是leader
 - 另一个server宣布自己是leader：当收到的heartbeat消息的current term大于等于自己的term时，退回follower状态，承认当前leader；否则拒绝RPC，并且继续维持在candidate状态
 - 超时：当有多个candidate，但是没有candidate获得多数选票时（这一现象称为split votes），所有的candidate会超时，而后进入下一轮election。显然如果没有一些额外的机制，split votes可能会重复出现

5. randomized election timeouts

- election timeouts会在一个固定的区间之内进行选择
- 可用在follower超时超过election timeouts之后成为candidate发起election
- 也可用在split vote的情况下等待所有的server超时超过election timeouts

6. 投票规则：

- A 比 B up-to-date：A最后一条log entry的term比B最后一条log entry的term大；或者在AB最后一条log entry term一样大时，A的log更长
- follower只会给比自己更up-to-date的candidate投票
- 为什么不选择log最长的candidate？

S1	5	6	7
S2	5	8	
S3	5	8	

这种情况下，S1的log更长，但是index为6、7的log entry显然未提交。如果S1成为leader，反而会擦除已经提交的8

3.2. log复制

AppendEntries RPC:

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

1. log结构：log由若干条log entry组成。每一个log entry都绑定了一个term index。当log entry被创建时，current term就会被写入log entry。同时，每个log entry也会有个log index
2. 工作流程：client向server发出请求，leader先将新的log entry应用到自己的log中，然后发送AppendEntries RPCs到其他followers。当log entry在所有（或绝大多数）的server上都被复制，leader就会将log entry应用到所有的复制状态机上开始执行，而后向client返回数据。当有follower故障或者执行过慢时，leader会一直重试RPC，直到所有的server都已经含有该log entry为止（只要求log entry，没有要求要更新）
3. committed：当一条log entry可以被安全地（安全的条件：绝大多数的机器都收到了log entry，**并且leader收到了majority关于AppendEntries的返回信息**。是否安全由leader决定！而不是单纯地指majority有这一条entry）应用在复制状态机上时，就认为这条log entry是已提交的。已提交的log entry是持久化的（durable），并且最终会被应用到所有的server上进行执行。简而言之，committed = 大于半数机器都收到了log entry**并且返回成功给leader**，可以开始执行。
4. 当前的log entry已提交时，所有log index更小的log entry都会自动变成已提交状态（不管是由哪个leader创建。这一条可以根据Log Matching Property得到）
5. **Log Matching Property**：两个来自两个不同的log的log entry，若它们的log index与term完全相同，则这两条log entry包含相同的指令，并且在这两条指令之前的log entry是完全相同的
6. consistency check：leader会追踪所有的log entry中log index最大的、且已提交的log entry。leader也会把该log index附加在以后所有的AppendEntries RPCs中，用于指示follower执行。同时，leader也会加上当前log entry的前一条的term 和 index（PreLogIndex与PreLogTerm）。在follower收到RPC时，会检查新log entry前一条的term和index，如果是自

己log的最后一条，就返回成功，否则就拒绝，并且返回错误。起始时，所有server的log都是空的，而后某一次向log 中添加entry时，若所有的RPC都返回true，则当前server的log一定是一致的（数学归纳法）

7. 可能的机制：

- 当follower落后leader太多时，可能需要减缓leader执行的速度。follower会给leader发送message告知当前执行进度

4. 容错

4.1. log不一致

当遇到某个follower的log与leader不一致时，会比对follower的log，找到第一条不一样的log entry，然后将后面所有的log entry通过一个RPC传递一条log entry的形式，进行覆盖。找到第一条不一样的log entry这一行为通过leader维护的nextIndex数组来实现。nextIndex一开始被维护为leader的log中即将写入的log entry的index。若发现某个follower的RPC返回错误，则递减nextIndex，直到RPC成功为止。

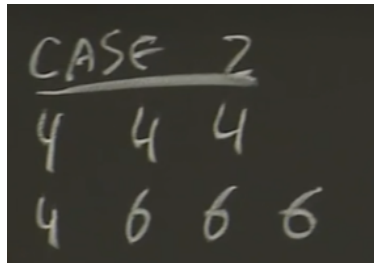
- 这一过程可以进行小优化：follower可以返回不一致的log entry对应的term号，以及该term对应的第一条log entry的index。（根据log matching Property，如果在follower该term的第一条log entry与leader的一致，则说明不一致log entry发生在这一term；反之重复操作即可。）而后RPC一次传送一个term的log entry，而非一次传送一条。但作者认为必要性不强。几种情况如下图所示

- 情况一

CASE 1				
S1	4	5	5	
S2	4	6	6	6

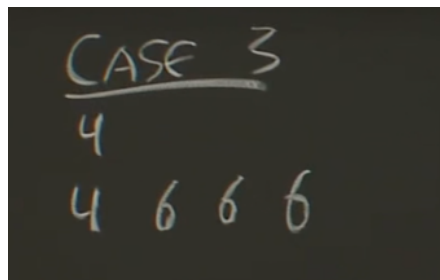
S2是leader。此时S1与S2发生不一致的位置为index=3（假设log index从1开始）的位置（此处的6是新的log entry，leader已经插入到自己的log，要给其他follower发布AE）。但是leader没有term5，因此就定位到S1中term5的开头重试RPC（让nextIndex[S1]=2即可）

- 情况二



此时不一致位置发生在index=3的位置。但是S2此时有term4，因此S2需要定位到自己的term4的最后一条的下一条（即第一个6），将nextIndex[S1]设置为第一个6即可

○ 情况三



冲突的位置S1完全缺失，此时令nextIndex[S1]=2即可（长度相关）

- 为什么在log不一致时，可以擦除某些log entry？因为这些log entry必定是uncommitted，不可能被执行。注意，若一条log entry是committed，则必定会在leader的log中（the Leader Completeness Property），则在此处应当不会发生不一致。

由此，log一致性可以作为AppendEntries失败时的动作而自动维护

4.2. the Leader Append-Only Property

leader从不改写或者回退它的log

4.3. follower/candidate failure

- leader会不停地发送RPC，直到收到为止
- log会被写入磁盘，因此server可以通过磁盘中的log 来恢复状态
- server刚启动时（启动时必定处于follower状态），不允许执行任何操作。因为此时follower并不知道自己的log中哪些entry已经committed

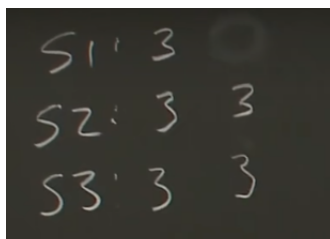
5. 一些tricky的特殊情况

- 单向网络：此时leader只能对外发送信息，而不能接受信息。原始的raft似乎无法解决这一问题。此时heartbeat信息会阻止其他follower进行选举，但是由于leader收不到follower的反

馈，导致append的entries无法commit，因此无法执行、推进并且应答clients。一种可能的办法是：当leader在超出一段时间没有收到心跳信息时，认为自己出现了故障，应该下台，停止发送heartbeat信息

5.1. log不一致的例子

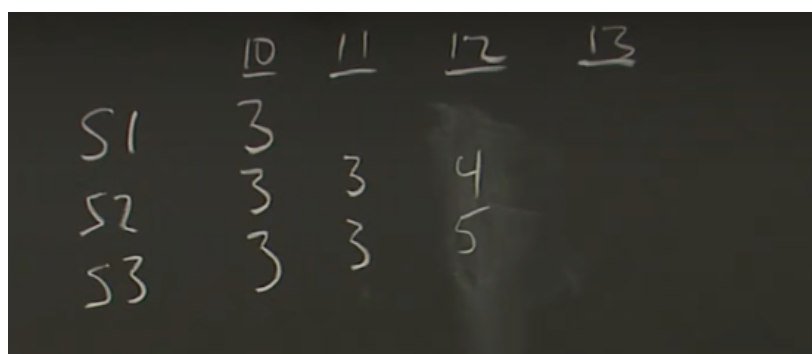
- 例1



S1	3	
S2	3	3
S3	3	3

S2/S3可能是leader，假设S2是leader，先给S3发送AE。而后在给S1发送AE前，S3故障

- 例2



	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>
S1	3			
S2	3	3	4	
S3	3	3	5	

term=3时，S2或者S3是leader。若S2是leader，则再给S1发送index=11，term=3的log entry时故障，而后又恢复。然后S2又被选举为leader（S2此时满足at least up-to-date的条件，可以当选），而后在发出term=4，index=12的log entry时故障。最后s3被选为leader（S3可以得到S1与自己的投票。得不到S2的投票），而后S3又在发送term=5，index=12的log entry之前故障

- 例3

5.2.

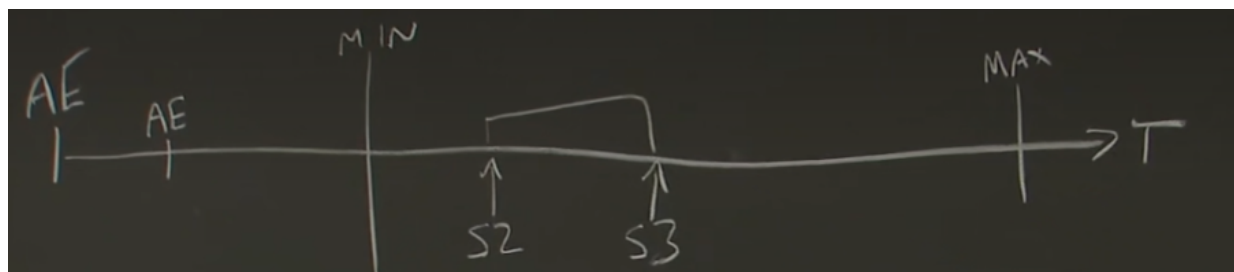
6. 可用性

- 可用性、election timeout、broadcast时间不等式：

$$broadcastTime \ll electionTimeout \ll MTBF$$

heartbeat interval与election Timeout的不等式

- 广播时间：包括了leader发送RPC到收到回复的整个时间段
- MTBF: mean time between failure, 衡量可用性的重要指标



AE即为AppendEntry, 用于发送heartbeat。在Raft中有两个很重要的参数: heartbeat interval和election timeout。election timeout的下限必须比heartbeat interval大, 否则总会出现有一部分server还没来得及收到heartbeat就开始发起选举; broadcastTime必须小于election timeout的原因是: 若election timeout太小, 则candidate会在收到RPC response之前过期, 从而导致没有candidate能成为leader; 如果上图中election timeout的上限过大, 会导致leader failure时raft系统恢复时间过长, 当election timeout接近MTBF时, 系统将无法推进, 即还没有等到有些follower过期进行选举, 下一个failure又产生了。

7. 相关对比

1. MapReduce、GFS、VMware FT都需要单一的实体来决定谁是类似于primary之类的东西
2. 在VMware FT中, 磁盘网络上存在一个Test and Set server。如果是单一的server, 可能会导致故障, 在VMware FT server的primary故障时, 无法决定谁能go live。但是如果构建test and set server的replicated system, 存在需要处理脑裂的问题
 - VMware FT server S1与S2都通过网络与Test and Set server TS1和TS2通信。如果存在网络问题, 导致S1可以与TS1通信, 不能与TS2通信, 并且S2可以与TS2通信不能与TS1通信, 这时就存在brain split的问题。TS1与TS2的状态可能是不一致的, 可能会导致S1与S2同时go live, 也就造成了脑裂现象
3. 解决脑裂:
 - 两种方法
 - 构建一个绝对不会出错的网络系统, 但是需要大量的人力财力进行维护
 - 使用多数投票方法: 也即Raft/Paxos使用的方法

8. Snapshot

用于压缩log

8.1. 结构

last included index与last included term可以用于AE中的 consistency check（需要用到PrevLogIndex与PrevLogTerm）

- last included index：Snapshot取代的最后一条log entry的index
- last included term：Snapshot取代的最后一条log entry的term
- 数据的state

8.2. 机制

1. 每台机器都会定期进行snapshot。这一过程由每台机器自主进行，而不是统一调配。但由于leader也会进行snapshot，会删除掉某些log entry，会使得执行较慢的follower无法执行。因此当leader已经删除了某条log entry，并且这条log entry正是较慢的follower所需要时，leader会给执行较慢的follower发送snapshot。
2. 完成snapshot之后，可以删除snapshot覆盖了的log entry以及之前的snapshot。snapshot覆盖的log entry**必须为已提交**
3. InstallSnapshot RPC：通过此RPC，leader向执行缓慢的follower发送snapshot。若snapshot中含有与follower不一致的log entry，则follower抛弃当前所有的log entry，用snapshot来代替；若snapshot只是follower log的前缀部分，则follower将log中对应的部分删除，用snapshot来代替，保留前缀后面的log entry

8.3. 仅leader创建snapshot的缺点

- 占用网络带宽，减缓snapshot的速度。
- leader的实现会更加复杂

8.4. snapshot对性能的影响

- 何时进行snapshot：太快会过多占用磁盘带宽，太慢会导致恢复时间过长并且存在log耗尽存储空间的风险
 - 解决方案：当log大小超过一个固定的值时就进行snapshot。当这个值远远大于snapshot的大小时，磁盘开销会很小

- snapshot写入的时间较长：使用写时复制技术（利用linux系统中的fork，fork本身是写时复制的，将snapshot数据写到内存之中）

9. client与Raft的交互

1. 寻找leader：client在启动时，随机选择一台server进行通信，如果不是leader，这台server会拒绝client的请求，而后在response中会附带leader的相关信息，从而使得client允许与server通信。假如通信的这台server存在一些问题，导致提供的信息不是leader，则client的请求会失败，超时之后，client会再次随机选择server通信
2. 线性化语义（linearizable semantics）：client要求请求即刻执行，并且只执行一次。有些情况下同一请求会被执行两次，例如leader在返回client的请求之前执行了命令但是崩溃了，而后client retry，则会导致命令执行两次。解决方案是client在命令中附带一个serial number（序列号），当上述故障发生时，raft会追踪每个client最新执行的序列号，并且找到对应的response。
3. 只读操作：只读操作可以不需要log的介入。但是存在风险：当前和client通信的leader已经下台了，有新的leader，但是stale leader并不知晓，导致client读到过时的数据。raft采取了两个预防措施：
 - leader必须时刻清楚那些entries是committed的。尽管the Leader Completeness Property规定了leader一定包含了所有committed entries，但是在leader刚刚上台时，并不知道这些信息。leader会在刚上台时，提交一个含有空命令的entry。（如此一来，该entry前面的所有entry都被认为是已提交的）
 - leader在返回只读操作之前，必须与绝大多数的follower进行通信，保证自己仍然是leader

10. 数据持久化

持久化和非持久化数据仅在server崩溃然后重启时有意义。对于容错容灾而言，可能存在两种现实情况：当集群中的一台计算机故障时，用一台崭新的计算机加入集群，使其跟上整个集群的执行；整个集群的供电中断并回复时，整个集群需要能够恢复到断电前的某个状态并且继续执行。在某台机器启动时必须检查这些持久化数据。

每一次修改持久化数据，都需要将其写入磁盘。但是磁盘IO很慢，代价很昂贵，会成为raft的性能瓶颈。一种方法是使用批量写入的方式，仅在需要回应RPC或者发出RPC时进行写入，但是显然会牺牲正确性。也可以使用固态硬盘，提升IO速度；也可以使用电池来保证机器断电时内存数据不会丢失。

10.1. Log

raft并没有保存应用程序的状态（例如数据库、VMware的Test-and-Set value等），因此log是唯一能够在启动后恢复应用程序状态的数据

10.2. currentTerm

需要持久化地记录当前的term number，否则在发起election阶段，candidate只能从自己的log中找下一个term是什么。可能会出现同一个term有两个leader的情况（未必是同时，但是这种情况也是不允许的）

10.3. voteFor

为了保证一个term只能有一个leader。不持久化voteFor可能会出现：server1作为follower给candidateA投了一票，然后server1崩溃了，重启时，candidateB又发起了投票，server1可能会再给candidateB投票，就无法保证一个server只能投出一票，进而也无法保证一个term只有一个leader

10.4. 为什么lastApplied不需要持久化

raft假定机器重启时，丢失了所有的应用程序状态，因此会从log的最开始一条一条重新执行。这种机制很简单但是很慢。

11. 相关论文

- Paxos 15/16
- Paxos的通俗解释 16/20/21
- Paxos 多次决策的完整化/优化：13/26/39
- Viewstamped Replication 共识算法.VSR是与Paxos同时期提出的多数投票算法
- log 压缩技术：5/30/36
- 测试代码是开源的：23