

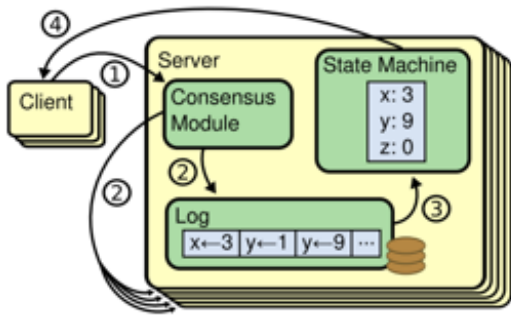
Raft一致性算法论文理解

- Raft 新特性
- 复制状态机
- Paxos 算法的不足
- Raft 设计原则
- Raft 一致性算法
- Raft中的RPC通信
- 角色&状态转换
- 任期逻辑时钟
- 领导人选举流程
- 选举活锁(多个节点超时同时选主)
- 日志复制
- 安全性
- Follower和Candidate崩溃
- 时序和可用性
- 集群成员变化
- 日志压缩
- 客户端交互
- 参考

Raft 新特性

Strong Leader	更强的领导形式 例如日志条目只会从领导者发送到其他服务器，这很大程度上简化了对日志复制的管理
Leader Election	使用随机定时器来选举领导者 用最简单的方式减少了选举冲突的可能性
Membership Change	新的联合一致性 (joint consensus) 方法

复制状态机



1. 复制状态机通过日志实现

- 每台机器一份日志
- 每个日志条目包含一条命令
- 状态机按顺序执行命令

2.应用于实际系统的一致性算法一般有以下特性

- 确保安全性
- 高可用性
- 不依赖时序保证一致性
- 一条命令能够尽可能快的在大多数节点对一轮RPC调用响应时完成

Paxos 算法的不足

- 算法复杂度高, 较难理解
- 工程复杂度高, 难以在实际环境中实现

Raft 设计原则

- 概念分解
 - Leader election
 - Log replication
 - Membership changes
- 通过减少状态数量将状态空间简化
 - 日志不允许出现空洞, 并且 raft 限制了日志不一致的可能性
 - 使用随机化时钟简化了领导选举的算法

Raft 一致性算法

State (状态)

在所有服务器上持久存储的(响应RPC之前稳定存储的)

currentTerm	服务器最后知道的任期号(从0开始递增)
votedFor	在当前任期内收到选票的候选人Id(如果没有就为null)
log[]	日志条目, 每个条目包含状态机要执行的命令以及从Leader收到日志时的任期号

在所有服务器上不稳定存在的

commitIndex	已知被提交的最大日志条目索引
lastApplied	已被状态机执行的最大日志条目索引

在Leader服务器上不稳定存在的

nextIndex[]	对于每一个follower, 记录需要发给他的下一条日志条目的索引
matchIndex[]	对于每一个follower, 记录已经复制完成的最大日志条目索引

AppendEntries RPC (日志复制)

由leader通过RPC向follower复制日志, 也会用作heartbeat

入参

term	Leader任期号
leaderId	Leader id, 为了能帮助客户端重定向到Leader服务器
prevLogIndex	前一个日志的索引
prevLogTerm	前一个日志所属的任期
entries[]	将要存储的日志条目列表(为空时代表heartbeat, 有时候为了效率会发送超过一条)
leaderCommit	Leader已提交的日志条目索引

返回值

term	当前的任期号, 用于leader更新自己的任期号
success	如果其他follower包含能够匹配上prevLogIndex 和prevLogTerm的日志, 那么为真

接收日志的follower需要实现的

1. 如果term < currentTerm, 不接受日志并返回false
2. 如果索引prevLogIndex处的日志的任期号与prevLogTerm不匹配, 不接受日志并返回false
3. 如果一条已存在的日志与新的冲突(index相同但是term不同), 则删除已经存在的日志条目和他之后所有的日志条目
4. 添加任何在已有日志中不存在的条目
5. 如果leaderCommit > commitIndex, 则设置commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC (投票请求)

入参

term	候选人的任期号
------	---------

candidateId	发起投票请求的候选人id
lastLogIndex	候选人最新的日志条目索引
lastLogTerm	候选人最新日志条目对应的任期号

返回值

term	目前的任期号, 用于候选人更新自己
voteGranted	如果候选人收到选票, 那么为true

接收日志的follower需要实现的

1. 如果term < currentTerm, 那么拒绝投票并返回false
2. 如果votedFor为空或者与candidateId相同, 并且候选人的日志和自己一样新或者更新, 那么就给候选人投票并返回true

服务器要遵守的规则

- **所有服务器:**
 - 如果commitIndex > lastApplied, 那么将lastApplied自增并把对应日志log[lastApplied]应用到状态机
 - 如果RPC请求或响应包含一个term T大于currentTerm, 那么将currentTerm赋值为T并立即切换状态为follower
- **Follower:**
 - 无条件响应来自candidate和leader的RPC
 - 如果在选举超时之前没收到任何来自leader的AppendEntries RPC或RequestVote RPC, 那么自己转换状态为candidate
- **Candidate:**
 - 转变为candidate之后开始发起选举
 - currentTerm自增 --> 重置选举计时器 --> 给自己投票 --> 向其他服务器发起RequestVote RPC
 - 如果收到了来自大多数服务器的投票, 转换状态成为leader
 - 如果收到了来自新leader的AppendEntries RPC(Heartbeat), 转换状态为follower
 - 如果选举超时, 开始新一轮的选举
- **Leader:**
 - 一旦成为leader, 想其他所有服务器发送空的AppendEntries RPC(Heartbeat), 并在空闲时间重复发送以防选举超时

- 如果收到来自客户端的请求，向本地日志追加条目并向所有服务器发送AppendEntries RPC，在收到大多数响应后将该条目应用到状态机并回复响应给客户端
- 如果leader上一次收到的日志索引大于一个follower的nextIndex，那么通过AppendEntries RPC将nextIndex之后的所有日志发送出去；如果发送成功，将follower的nextIndex和matchIndex更新，如果由于日志不一致导致失败，那么将nextIndex递减并重新发送
- 如果存在一个 $N > \text{commitIndex}$ 和半数以上的 $\text{matchIndex}[i] \geq N$ 并且 $\log[N].\text{term} == \text{currentTerm}$ ，将commitIndex赋值为N

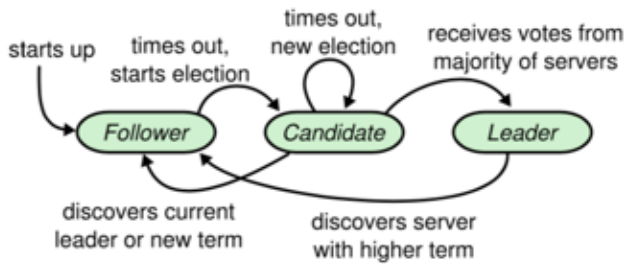
一致性算法总结

Election Safety	选举安全原则：一个任期内最多允许有一个leader
Leader Append-Only	只增加日志原则：Leader只会增加日志条目，永远不会覆盖或删除自己的日志
Log Matching	日志匹配原则： 如果两个日志在相同的索引位置上并且任期号相同， 那么就可以认为这个日志从头到这个索引位置之间的条目完全相同
Leader Completeness	领导人完整性原则： 如果一个日志条目在一个给定任期内被提交， 那么这个条目一定会出现所有任期号更大的leader中
State Machine Safety	状态机安全原则： 如果一个服务器已经将给定索引位置上的日志条目应用到状态机， 那么所有其他服务器不可能在该索引位置应用不同的日志条目

Raft中的RPC通信

- RequestVote RPC
- AppendEntries RPC
 - 日志条目
 - 心跳
- InstallSnapshot RPC

角色&状态转换



- Follower: 都是被动的，不会发送任何请求，只是响应来自leader和candidate的请求
- Leader: 处理来自客户端的请求，如果一个客户端与follower进行通信，follower会帮助客户端重定向到leader
- Candidate: 用来选举一个新的leader

任期逻辑时钟



时间被划分为一个个任期(term)，每一个任期的开始都是leader选举，在成功选举之后，一个leader会在任期内管理整个集群，如果选举失败，该任期就会因为没有leader而结束，这个转变会在不同的时间的不同的服务器上被观察到

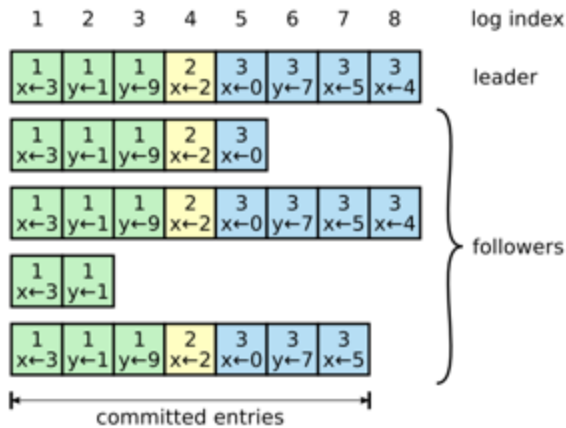
领导人选举流程

- follower --> candidate (选举超时触发)
 - candidate --> leader
 - 赢得了选举
 - candidate --> follower
 - 另一台服务器赢得了选举
 - candidate --> candidate
 - 一段时间内没有任何一台服务器赢得选举

选举活锁(多个节点超时同时选主)

随机的选举超时时间

日志复制



一旦选出了leader，它就开始接收客户端请求，每个客户端请求都包含一条需要被复制状态机(replicated state machine)执行的命令。leader把这条命令作为新的日志条目追加到自己的日志末尾，然后并行向其他机器发送AppendEntries RPC请求要求复制日志，当半数以上机器复制成功后leader将当前条目应用到它的状态机并向客户端回复执行结果，如果某个follower崩溃或者网络问题丢包，leader会无限重试AppendEntries RPC(甚至在它已经响应客户端以后)直到所有follower都成功复制了所有日志条目

Raft日志机制的特性

- 如果在不同的日志中的两个条目有着相同的索引和任期号，那么他们存储的命令肯定是相同的
 - 源于leader在一个任期里给定的一个日志索引最多创建一条日志条目，同时该条目在日志中的位置也永远不会改变
- 如果在不同的日志中的两个条目有着相同的索引和任期号，那么他们之前的所有日志条目都是完全一样的
 - 源于AppendEntries RPC的一个简单的一致性检查：当发送一个AppendEntries RPC时leader会把新日志之前的一个日志条目的索引位置和任期号都包含在里面，follower会检查是否与自己的日志中的索引和任期号是否匹配，如果不匹配就会拒绝这个日志条目，接下来就是归纳法来证明了
- leader通过强制follower复制它的日志来处理日志的不一致
 - 为了是follower的日志同自己的一致，leader需要找到follower与它日志一致的索引位置并让follower删除该位置之后的条目，然后再讲自己再该索引位置之后的条目发送给follower，这些操作都在AppendEntries RPC进行一致性检查时完成
 - leader为每一个follower维护了一个nextIndex，用来表示将要发送给该follower的下一条日志条目索引，当一个leader开始掌权时，会将nextIndex初始化为它的最新日志条目索引值+1，如果follower在一致性检查过程中发现自己的日志和leader不一致，会在这个AppendEntries RPC请求中返回失败，leader收到响应之后会将nextIndex递减然后重试，最终nextIndex会达到一个leader和follower日志一致的位置，这个时候AppendEntries RPC会成功，follower中冲突的日志条目也被移除了，此时follower和leader的日志就一致了

安全性

- 选举限制
 - 用投票规则的限制来组织日志不全的服务器赢得选举
 - RequestVote RPC限制规则: 拒绝日志没自己新的candidate
 - Leader永远不会覆盖自己的日志条目
 - 日志条目只能从leader流向follower
- 如何提交上一个任期的日志条目
 - 全程保持自己的任期号
- 安全性论证
 - 领导人完整性原则(Leader Completeness)
 - 如果一个日志条目在一个给定任期内被提交, 那么这个条目一定会出现所有任期号更大的leader中
 - 状态机安全原则(State Machine Safety)
 - 如果一个服务器已经将给定索引位置上的日志条目应用到状态机, 那么所有其他服务器不可能在该索引位置应用不同的日志条目

Follower和Candidate崩溃

- 无限重试
- AppendEntries RPC是幂等的

时序和可用性

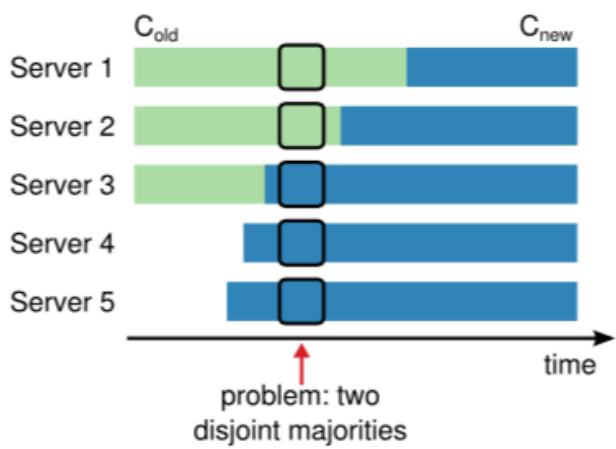
$$\text{broadcastTime} \ll \text{electionTimeout} \ll \text{MTBF}$$

broadcastTime	一台服务器并行的向集群中其他节点发送RPC并且收到它们响应的平均时间
electionTimeout	选举的超时时间
MTBF	是指单个服务器发生故障的时间间隔的平均数

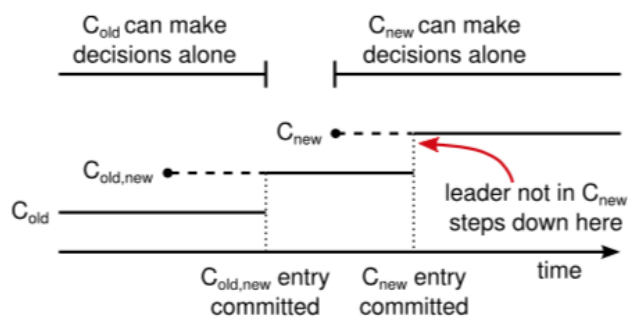
- broadcastTime应该比electionTime小一个数量级, 目的是让leader能够持续发送心跳来阻止follower们开始选举; 根据已有的随机化超时前提, 这个不等式也将瓜分选票的可能性降低到很小
- electionTimeout也要比MTBF小一个几个数量级, 目的是能使系统稳定运行, 当leader崩溃时, 整个集群大约会在electionTimeout的时间内不可用

集群成员变化

下图3节点集群扩展到5节点集群，直接扩展可能导致Server1和Server2构成老集群多数派，Server3，Server4和Server5构成新集群多数派，两者不相交从而导致决议冲突



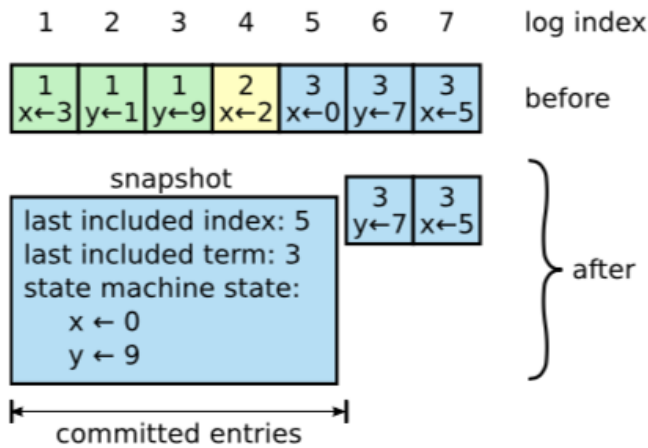
两阶段效果图



Raft采用联合一致性的方式来解决节点变更，先提交一个包含新老节点结合的Configuration，当这条消息commit之后再提交一个只包含新节点的Configuration，具体流程如下：

1. 首先对新节点进行CaughtUp追数据
2. 全部新节点完成CaughtUp之后，向新老集合发送Cold+new命令
3. 如果新节点集合多数和老节点集合多数都应答了Cold+new，就向新老节点集合发送Cnew命令
4. 如果新节点集合多数应答了Cnew，完成节点切换

日志压缩



- 每个服务器独立的创建快照，只包含已被提交的日志
- 快照内容主要包括
 - 状态机的状态
 - Raft的少量元数据(见上图)，保留这些元数据是为了快照后对紧接着的一个AppendEntries进行一致性检查
 - 为了支持集群成员变化，最新的配置(Configuration)也会作为一个条目被保存在快照中

快照分块传输(InstallSnapshot RPC)

- 虽然多数情况都是每个服务器独立创建快照，但是leader有时候必须发送快照给一些落后太多的follower，这通常发生在leader已经丢弃了下一条要发给该follower的日志条目(Log Compaction时清除掉了)的情况下

term	leader任期
leaderId	Leader id, 为了能帮助客户端重定向到Leader服务器
lastIncludedIndex	快照中包含的最后日志条目的索引值
lastIncludedTerm	快照中包含的最后日志条目的任期号
offset	分块在快照中的偏移量
data[]	快照块的原始数据
done	如果是最后一块数据则为真

接收者需要实现的

- 如果term < currentTerm立刻回复
- 如果是第一个分块 (offset为0) 则创建新的快照

- 在指定的偏移量写入数据
- 如果done为false, 则回复并继续等待之后的数据
- 保存快照文件, 丢弃所有已存在的或者部分有着更小索引号的快照
- 如果现存的日志拥有相同的最后任期号和索引值, 则后面的数据继续保留并且回复
- 丢弃全部日志
- 能够使用快照来恢复状态机 (并且装载快照中的集群配置)

客户端交互

- 客户端只将请求发送给领导人原则
- 线性一致读
 - 写raft log走状态机
 - Leader与过半机器交换心跳信息确定自己仍然是leader后可提供线性一致读
 - 利用心跳机制提供租约机制(Lease read), 但是依赖本地时钟的准确性

参考

[braft文档](#)

[raft-paper](#)