

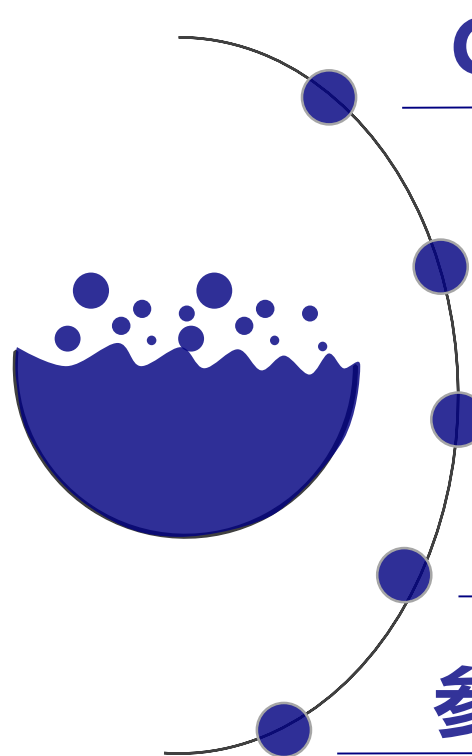
# Paxos/Raft 分布式一致性算法原理 剖析及其在实战中的应用

基础架构事业群-数据库技术-数据库内核  
何登成

---

# 目录 | Contents

---



<b>Consensus Problem</b>
<b>Basic Paxos</b>
<b>Multi-Paxos and Raft</b>
<b>实战分析</b>
<b>参考资料</b>

# Consensus Problem

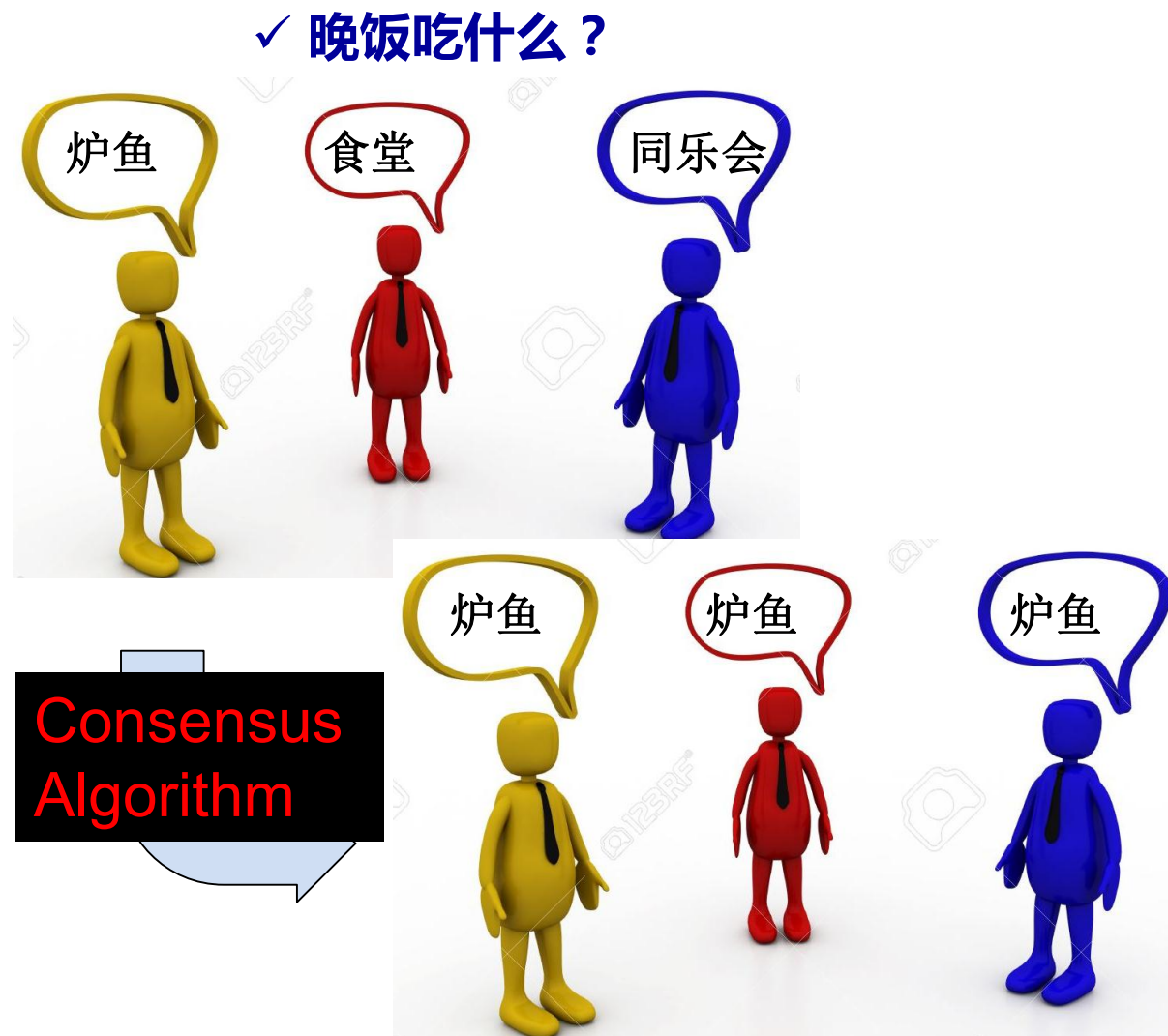
✓ **定义** : The consensus problem requires **agreement** among **a number of processes** (or agents) for **a single** data value.



# Consensus Problem

## ✓ 理解Consensus问题的关键

- ✓ 绝对公平，相互独立：所有参与者均可提案，均可参与提案的决策
- ✓ 针对某一件事达成完全一致：一件事，一个结论
- ✓ 已经达成一致的结论，不可被推翻
- ✓ 在整个决策的过程中，没有参与者说谎



# Consensus Algorithm : Basic Paxos

## ✓ Basic Paxos

- ✓ 一个或多个Servers可以发起提案 ( Proposers )
- ✓ 系统必须针对所有提案中的某一个提案，达成一致
  - ✓ 何谓达成一致？系统中的多数派同时认可该提案

✓ 最多只能针对一个确定的提案达成一致

## ✓ Liveness (只要系统中的多数派存活，并且可以相互通信)

- ✓ 整个系统一定能够达成一致状态，选择一个确定的提案

# Basic Paxos : Components

## ✓ Proposers

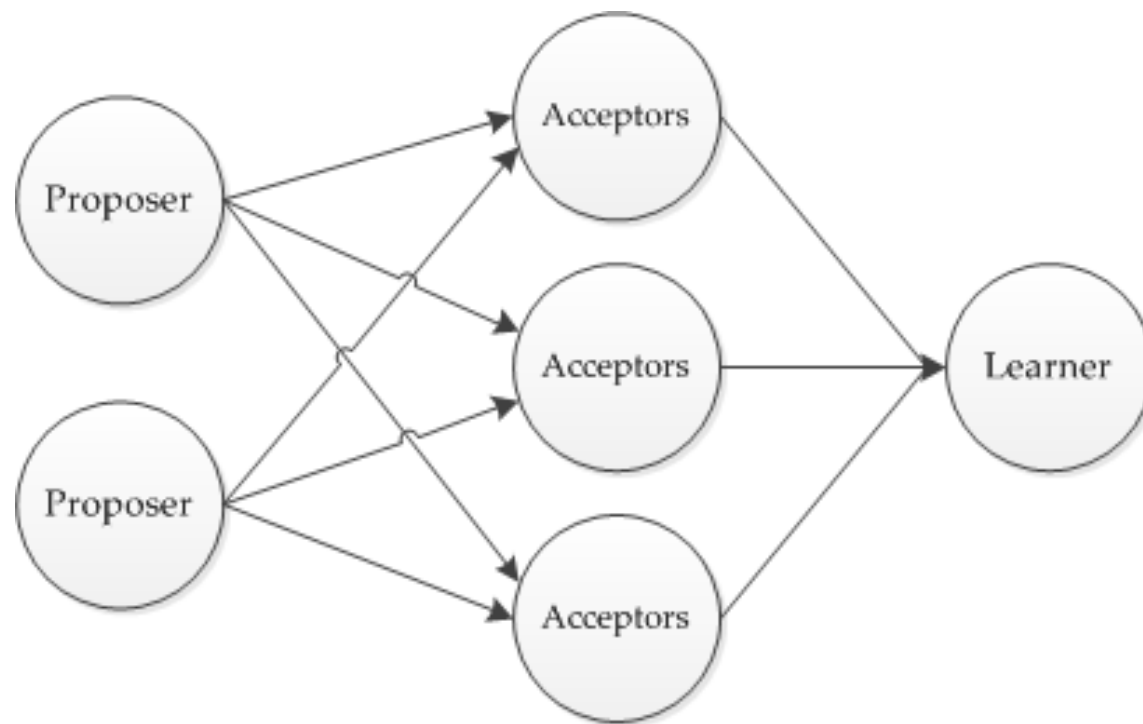
- ✓ Active : 提案发起者 ( value )
- ✓ 处理用户发起的请求

## ✓ Acceptors

- ✓ Passive : 参与决策 , 回应 Proposers的提案
- ✓ 存储accept的提案 ( value ) , 存储决议处理的状态

## ✓ Learners

- ✓ Passive : 不参与决策 , 从 Proposers/Acceptors学习最新达成一致的提案 ( value )



✓ 本文接下来的部分 , 一个Server同时具有 **Proposer**和**Acceptor**两种角色 , **Learner**角色逻辑简单 , 暂时不讨论

# Basic Paxos : Acceptors如何决策？

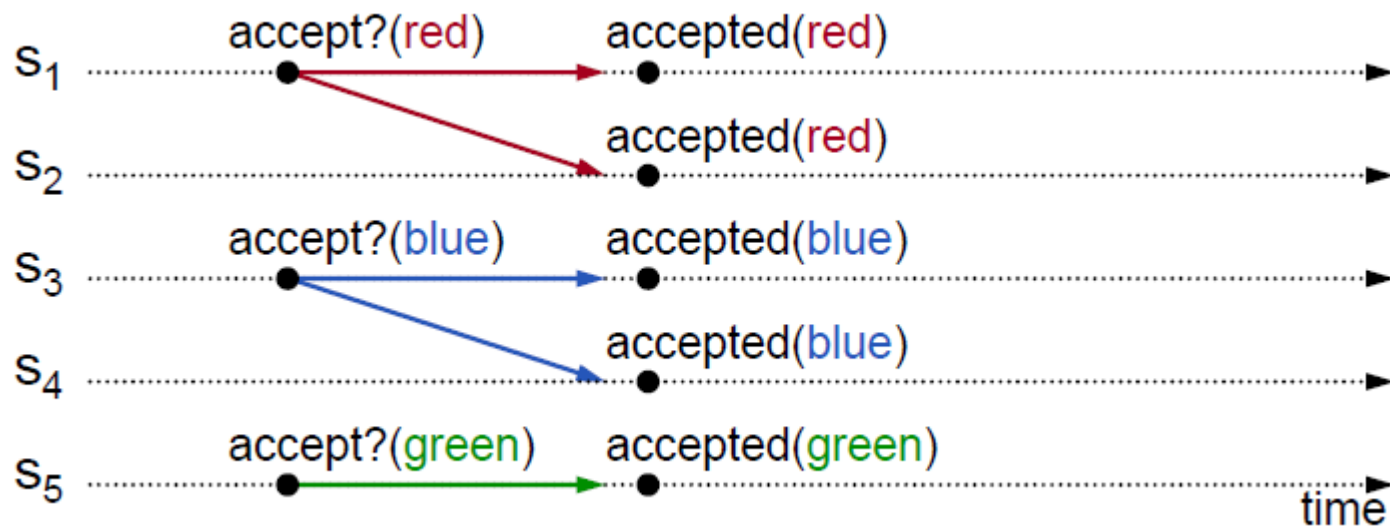
- ✓ **Accept First**

- ✓ Acceptor仅仅接受其收到的第一个value

- ✓ **Accept Last**

- ✓ Acceptor接受其收到的所有value，新的覆盖老的

# Accept First : Split Votes

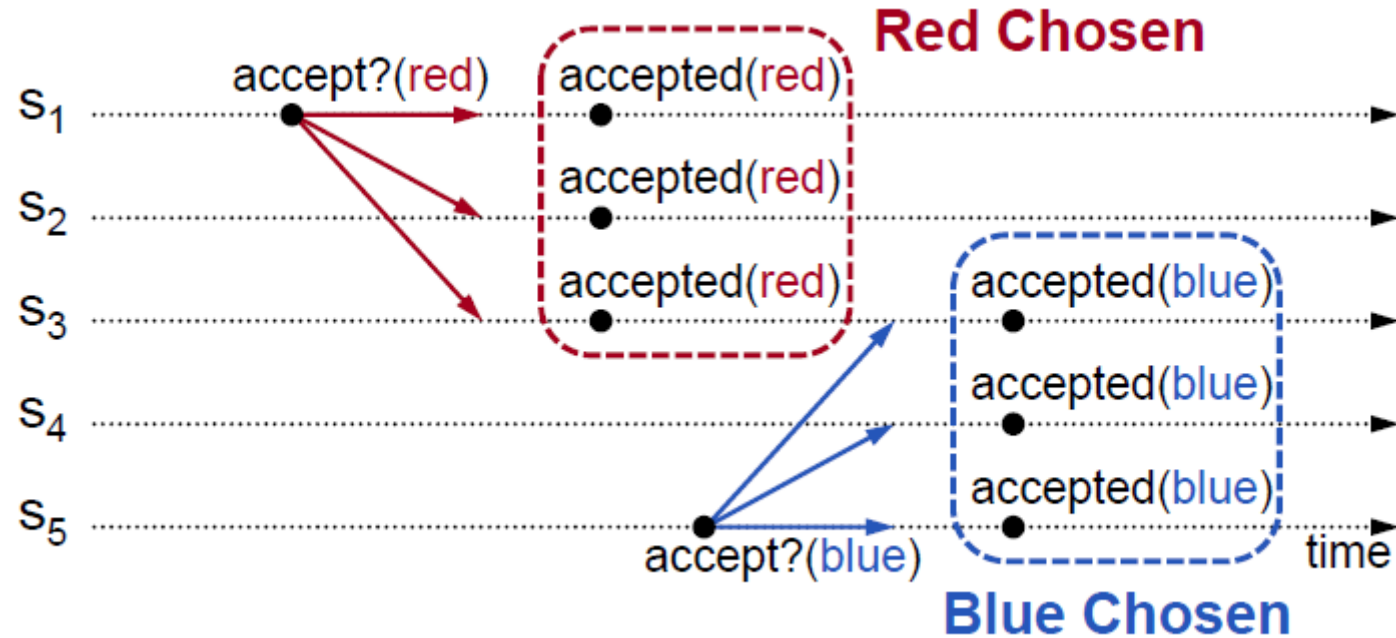


✓对于并发的proposals，同时accept多个value，违背原则：

- ✓ 系统中的多数派同时认可该提案
- ✓ 最多只能针对一个确定的提案达成一致



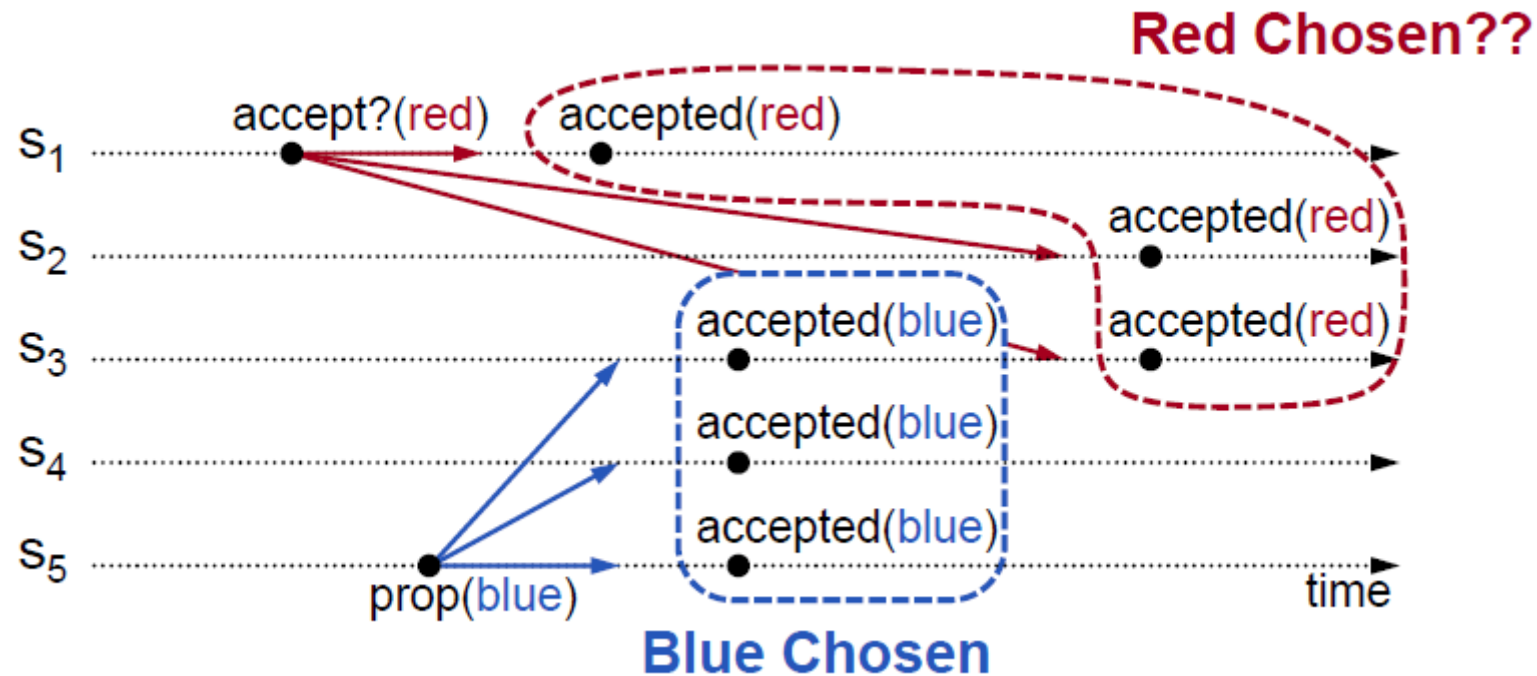
# Accept Last : Conflicting Choices



✓ 对于并发的proposals，先后accept多个value，违背原则：

✓ 最多只能针对一个确定的提案达成一致

# Accept Last : Conflicting Choices ( 2 )



✓ 对于并发的proposals，先后accept多个value，违背原则：

✓ 最多只能针对一个确定的提案达成一致

# Basic Paxos : Acceptors如何决策？

- ✓ **Accept First**

- ✓ Acceptor仅仅接受其收到的第一个value

- ✓ **Accept Last**

- ✓ Acceptor接受其收到的所有value，新的覆盖老的

✓ **Accept First , Accept Last都有问题。那Basic Paxos如何解决这些问题？如何实现Consensus？**

# Basic Paxos : 一阶段至两阶段

## ✓ 一阶段Accept

- ✓ **Accept First , Accept Last**都不行
- ✓ 参考分布式事务，两阶段提交（2PC）？

## ✓ 两阶段Accept

- ✓ 第一阶段：Propose阶段。Proposers向Acceptors发出Propose请求，Acceptors针对收到的Propose请求进行Promise承诺
- ✓ 第二阶段：Accept阶段。收到多数派Acceptors承诺的Proposer，向Acceptors发出Accept请求，Acceptors针对收到的Accept请求进行接收处理
- ✓ 第三阶段（可优化）：Commit阶段。发出Accept请求的Proposer，在收到多数派Acceptors的接收之后，标志着本次Accept成功。向所有Acceptors追加Commit消息

# Basic Paxos : Proposal ID

## ✓ Propose阶段

- ✓ **Proposal ID** : 唯一标识所有的Propose。包括：不同Proposers发出的Propose，同一Proposer发出的不同Propose。

## ✓ Proposal Number ( 唯一性 )

### Proposal ID

Round Number	Server Id
--------------	-----------

- ✓ **Round Number** : 每个节点存储本节点曾经**看到过**的最大值：maxRound
- ✓ **Server Id** : 系统中每个节点的唯一标识
- ✓ **Propose** : 获取当前节点的maxRound，自增1，然后加上本节点的Server Id，作为本次的Proposal ID

## ✓ Proposal ID ( 特性 )

- ✓ Proposal ID**全局唯一递增**，大小代表了优先级。**优先级的作用？**

# Basic Paxos : Propose阶段 ( 文字版 )

## ✓ Proposer 发送 Propose

- ✓ Proposer 生成全局唯一且递增的Proposal ID , 向集群的所有机器发送 Propose , 这里无需携带提案内容 , 只携带Proposal ID即可

## ✓ Acceptor 应答 Propose

- ✓ Acceptor 收到Propose后 , 做出 “**两个承诺 , 一个应答**”

### ✓ 两个承诺

- ✓ 第一 , 不再应答 Proposal ID 小于等于 ( **注意 : 这里是  $\leq$**  ) 当前请求的 Propose
- ✓ 第二 , 不再应答 Proposal ID 小于 ( **注意 : 这里是  $<$**  ) 当前请求的 Accept请求

### ✓ 一个应答

- ✓ 返回已经 Accept 过的提案中 Proposal ID 最大的那个提案的Value和accepted Proposal ID , 没有则返回空值

# Basic Paxos : Accept阶段 ( 文字版 )

## ✓ Proposer 发送 Accept

- ✓ **“提案生成规则”** : Proposer 收集到多数派的Propose应答后，从应答中选择存在提案Value的并且同时也是Proposal ID最大的提案的Value，作为本次要发起 Accept 的提案。如果**所有应答的提案Value均为空值**，则可以自己随意决定提案Value。然后携带上当前Proposal ID，向集群的所有机器发送 Accept请求

## ✓ 应答 Accept

- ✓ Acceptor 收到 Accpet 请求后，检查不违背自己之前作出的“两个承诺” 情况下，持久化当前 Proposal ID 和提案Value。最后 Proposer 收集到多数派的Accept应答后，形成决议

# Basic Paxos : 细节 ( 算法版 )

## Proposers

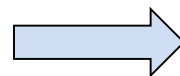
- 1) Choose new proposal number  $n$
- 2) Broadcast Prepare( $n$ ) to all servers



- 4) When responses received from majority:

If any **acceptedValues returned**, replace value with acceptedValue for highest acceptedProposal

- 5) Broadcast Accept( $n$ , value) to all servers



- 6) When responses received from majority:

Any rejections (result  $> n$ )? goto (1)

Otherwise, value is chosen

✓ **Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage**

## Acceptors

- 3) Respond to Prepare( $n$ ):

If  $n > \text{minProposal}$  then  $\text{minProposal} = n$

Return(acceptedProposal, acceptedValue)



- 6) Respond to Accept( $n$ , value):

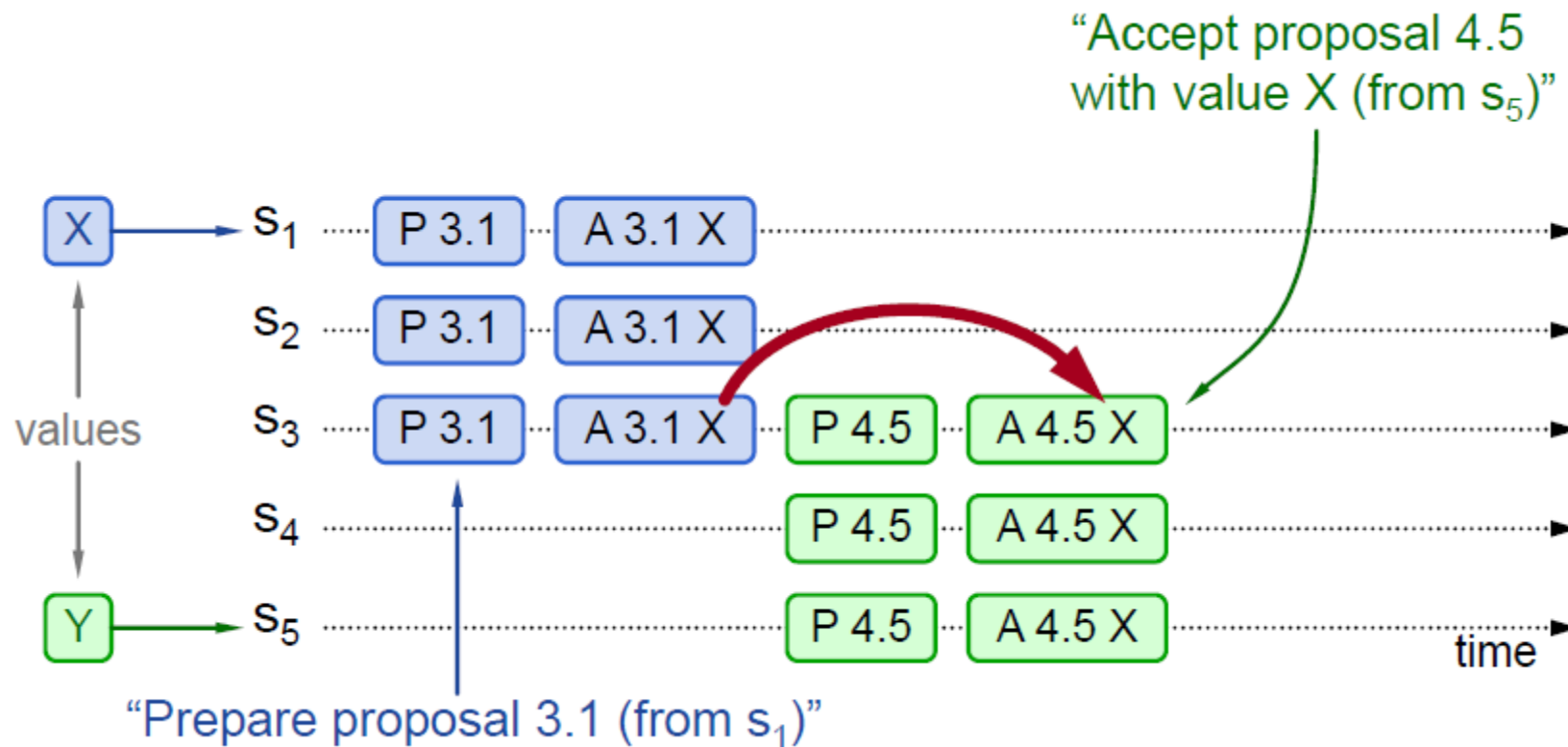
If  $n \geq \text{minProposal}$  then acceptedProposal =  
 $\text{minProposal} = n$  acceptedValue = value

Return(minProposal)





# Basic Paxos : Examples ( 1 )

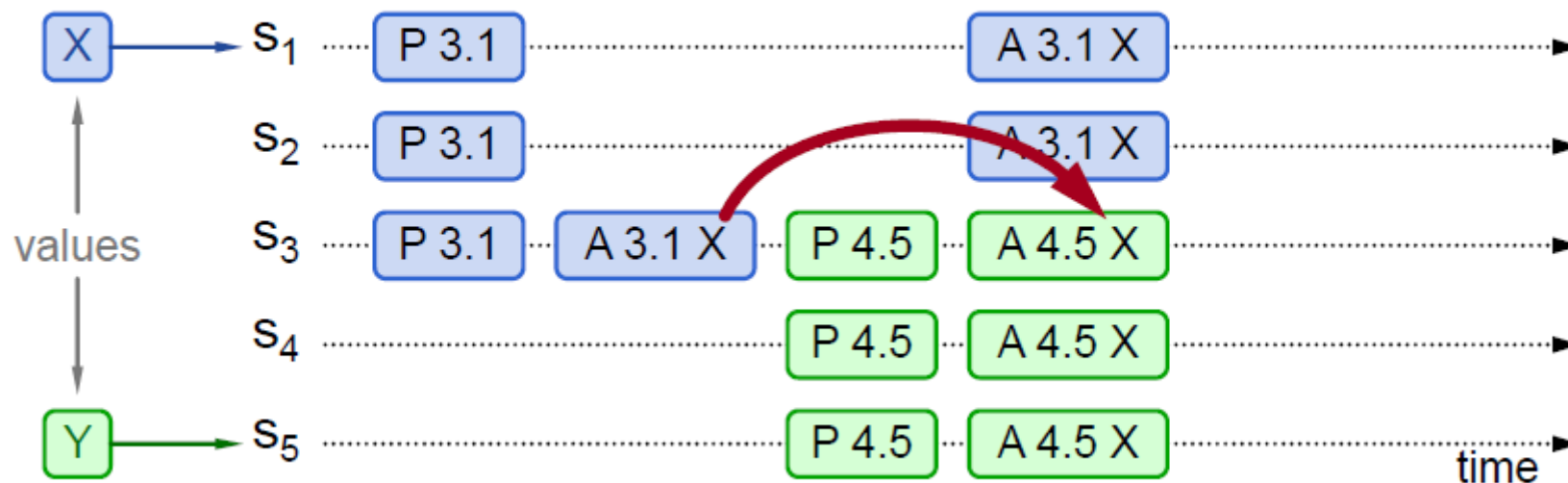


## ✓P 3.1

✓ Proposal ID : round number(3), server id(1)

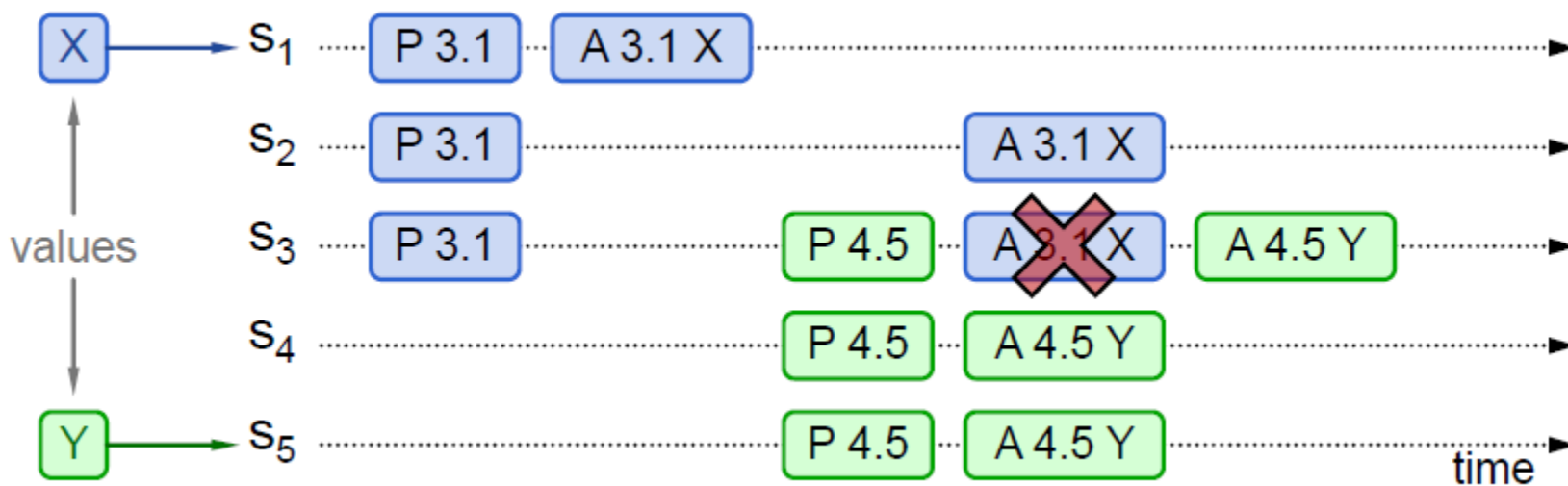
✓ P 3.1达成多数派，其Value(X)被Accept，然后P 4.5学习到Value(X)，并Accept

## Basic Paxos : Examples ( 2 )



✓ P 3.1 没有被多数派 Accept ( 只有 S<sub>3</sub> Accept ) , 但是被 P 4.5 学习到 , P 4.5 将自己的 Value 由 Y 替换为 X , Accept ( X )

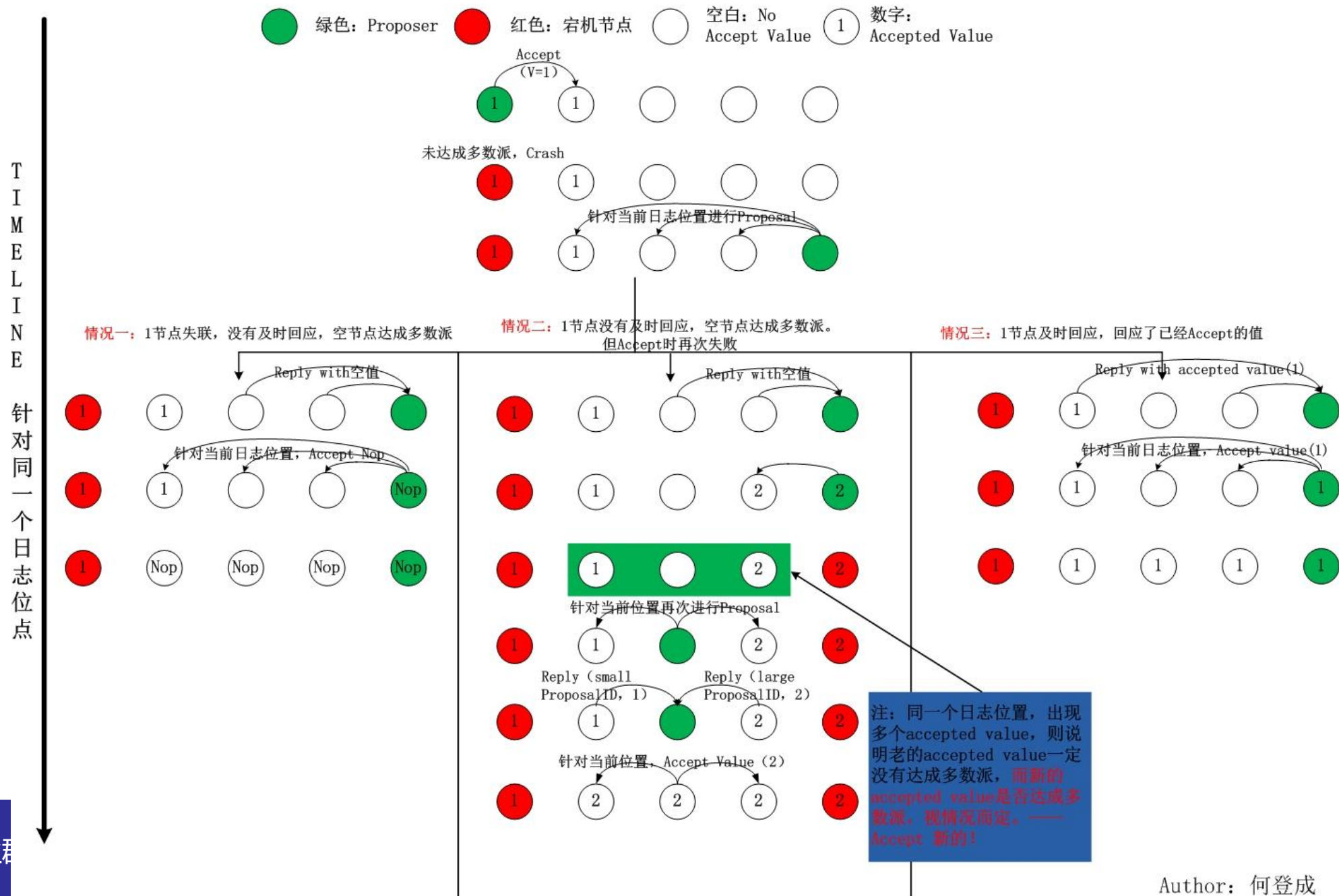
# Basic Paxos : Examples ( 3 )



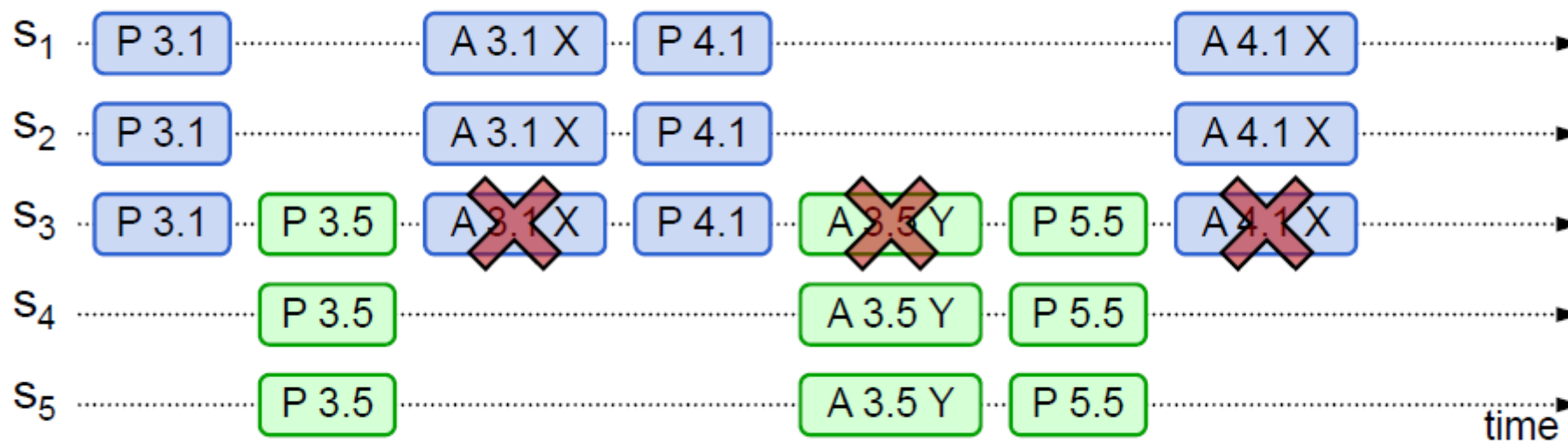
✓ P 3.1 没有被多数派 Accept ( 只有 S1 Accept ) , 同时也没有被 P 4.5 学习到。由于 P 4.5 Propose 的所有应答 , 均未返回 Value , 则 P 4.5 可以 Accept 自己的 Value ( Y )

✓ 后续 P 3.1 的 Accept ( X ) 会失败 , 已经 Accept 的 S1 , 会被覆盖

## Paxos一个异常场景的分析



# Basic Paxos : Livelock ( 活锁 )



## ✓回顾两个承诺之一

- ✓ 不再应答 Proposal ID 小于等于当前请求的 Propose。意味着：需要应答 Proposal ID 大于当前请求的 Propose

## ✓两个Proposers交替Propose成功，Accept失败，形成活锁 ( Livelock )

# Basic Paxos : 阶段总结

## ✓ Now , what we learn ?

- ✓ 学会了Basic Paxos , 能用Basic Paxos解决Consensus问题 , **确定一个提案 , 确定一个取值**

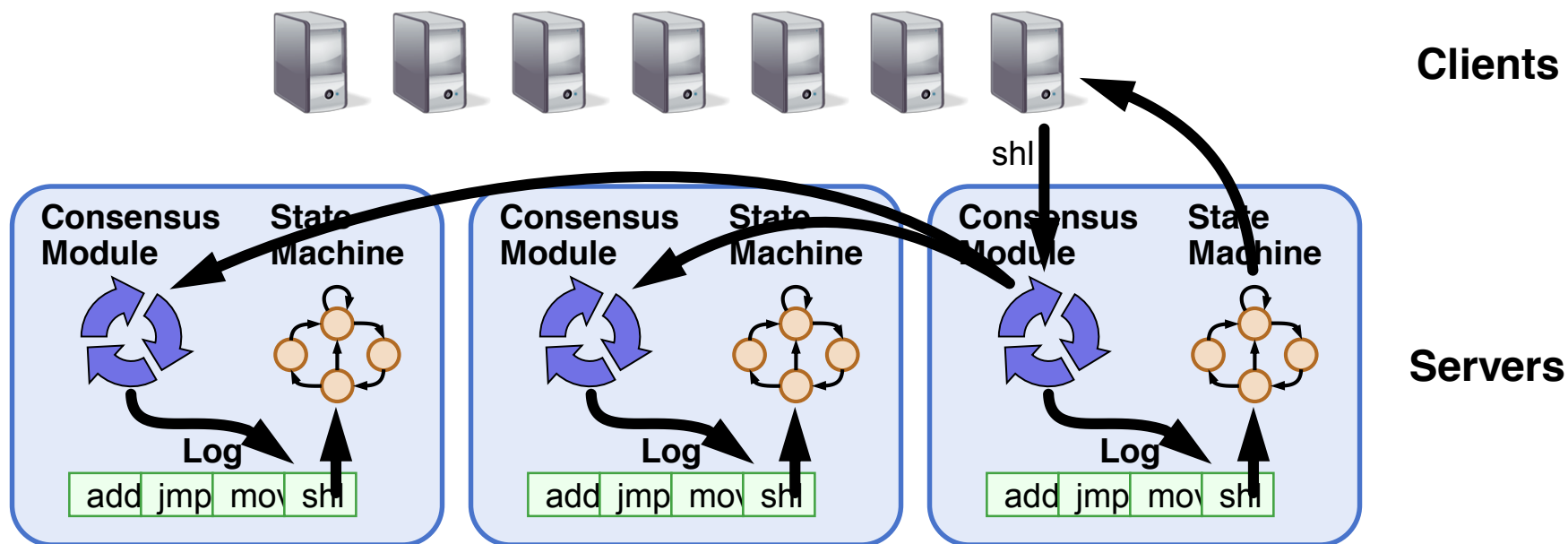
## ✓ Now , what we can do ?

- ✓ 决定本届美国总统选谁 ( **两个Proposers** : 民主党、共和党 ; **两个Value** : Hillary、Trump ; **但我们好像是Learner角色...** )
- ✓ 决定今晚吃什么 ?
- ✓ 一主多从架构 , 主宕机 , 决定选哪个从当新主

## ✓ Now , what we can' t do ?

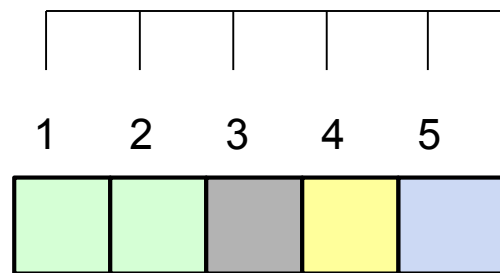
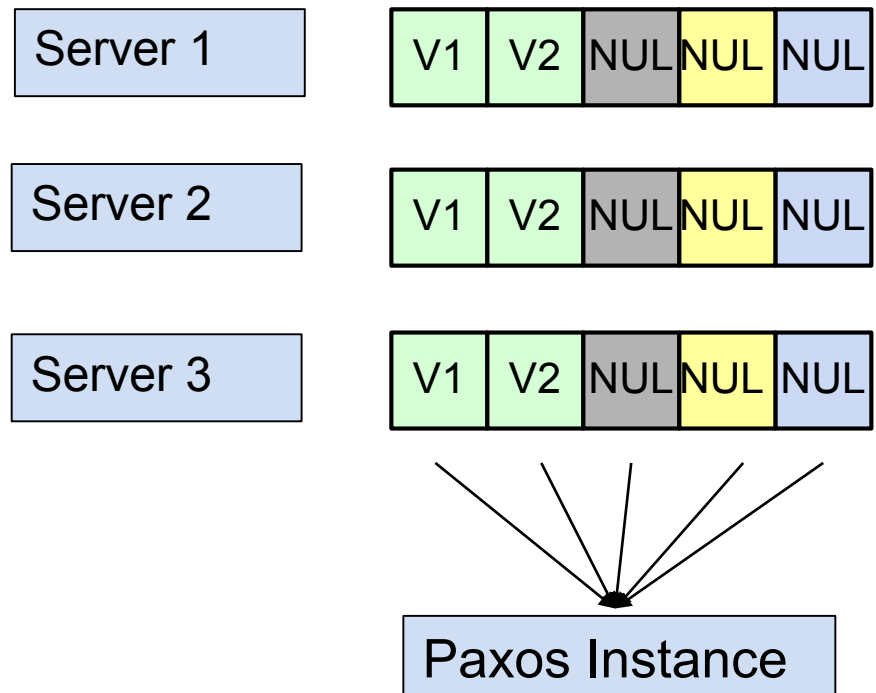
- ✓ 如果想确定连续多个提案 , 确定连续多个值 , Basic Paxos搞不定了...

# Consensus : From Basic to Multi-Paxos



- ✓ **Consensus Module** : 从Basic Paxos的决策单个Value，到连续决策多个Value。同时确保每个Servers上的顺序完全一致
- ✓ **相同顺序执行命令，所有Servers状态一致**
- ✓ **只要多数节点存活，Server能提供正常服务**

# Multi-Paxos



每一个Paxos Instance，都有一个唯一标识：**instance Id** (paxos) or **log Index** (Raft)

此文后面部分，均称之为：**log Index**

## ✓ ( Basic ) Multi-Paxos

- ✓ 针对每一个Paxos Instance使用完整的Paxos算法，确保系统中的每一个Server，Server上的每一个Paxos Instance都完全一致



# ( Basic ) Multi-Paxos的缺点、横空出世的Raft

## ✓ ( Basic ) Multi-Paxos : 缺点

- ✓ 难以理解、难以正确实现 ( gap between theory and practice )
  - ✓ Google Paxos Made Live : Despite the existing literature on the subject, building a production system turned out to **be a non-trivial task**...
- ✓ 多点可写 ( Proposers ) , 性能较差
  - ✓ 每个写入两个RPCs ; 多点可写 , 写入冲突
  - ✓ Google MegaStore : **Limiting that rate to a few writes per second** per entity group yields insignificant conflict rates. For apps whose entities are manipulated by a small number of users at a time, this limitation is generally not a concern...

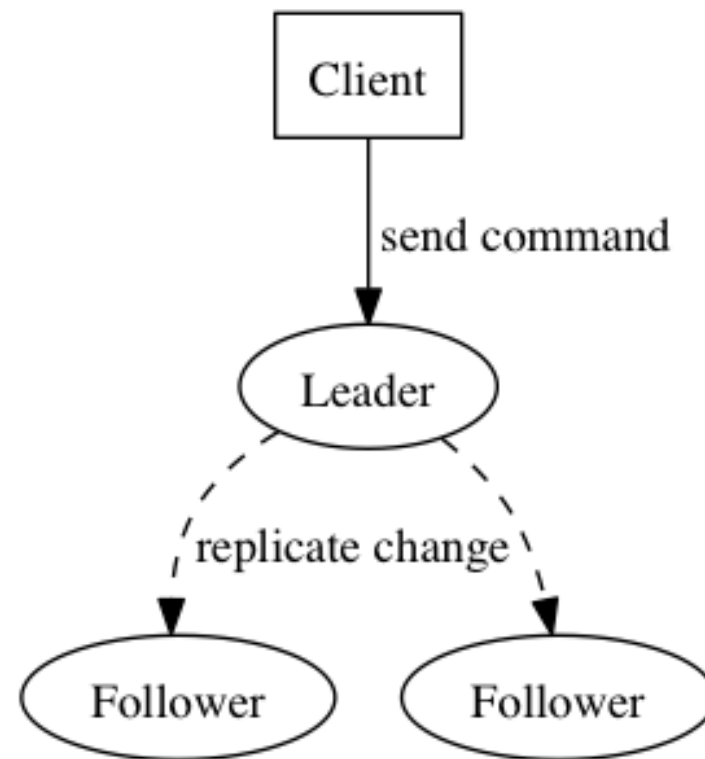
## ✓ Raft ( **Leader based Consensus Algorithm** )

- ✓ Diego Ongaro , Stanford University : In Search of an Understandable Consensus Algorithm ( 2014 )

# Raft ( 特点 )

## ✓Raft ( Leader Based )

- ✓ 系统存在一个Leader角色 ( Proposer ) , 接受Clients发过来的所有读写请求
- ✓ Leader负责与所有的Followers ( Acceptors ) 通信, 将提案/Value/变更复制到所有Followers, 同时收集多数派Followers的应答
- ✓ 少数派宕机, 不会影响系统整体的可用性
- ✓ Leader日常维护与所有Followers的心跳
- ✓ Leader宕机, 会触发系统自动重新选主, 选主期间系统对外不可服务

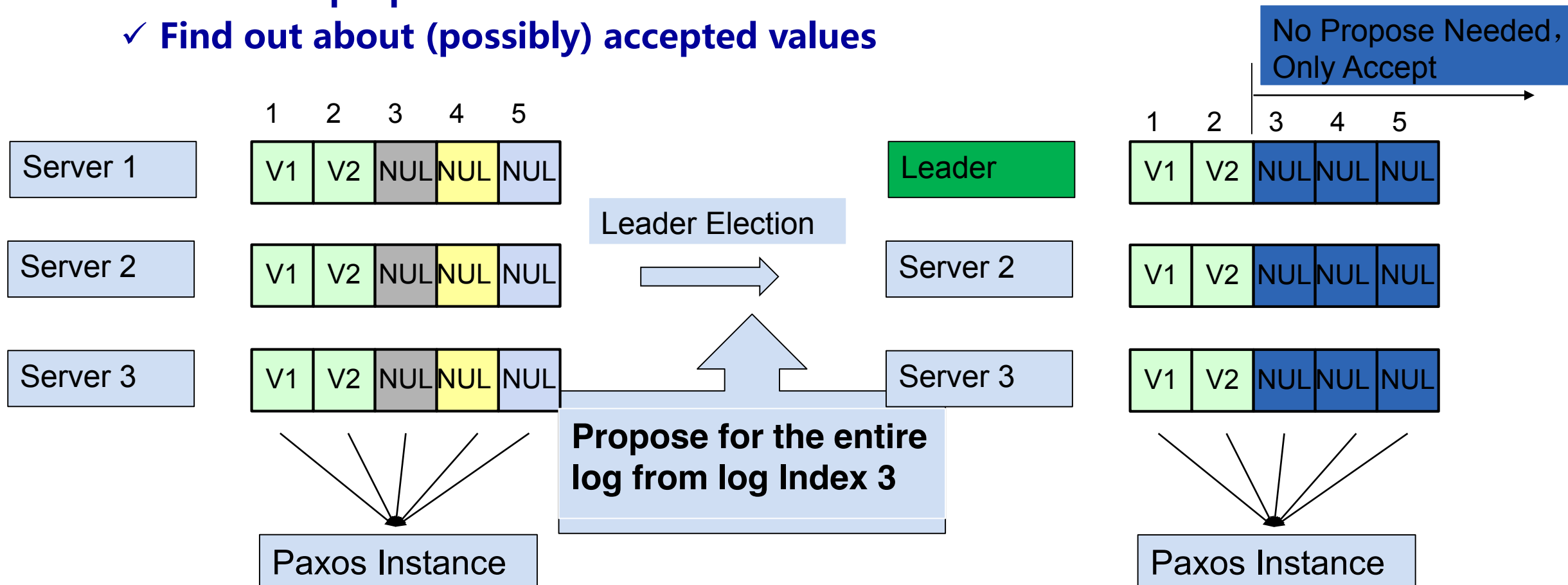


# 关于Leader角色的解读（从Paxos的视角）

## ✓ Paxos Propose的意义

✓ Block old proposals

✓ Find out about (possibly) accepted values



# Raft基本概念

## ✓Server状态

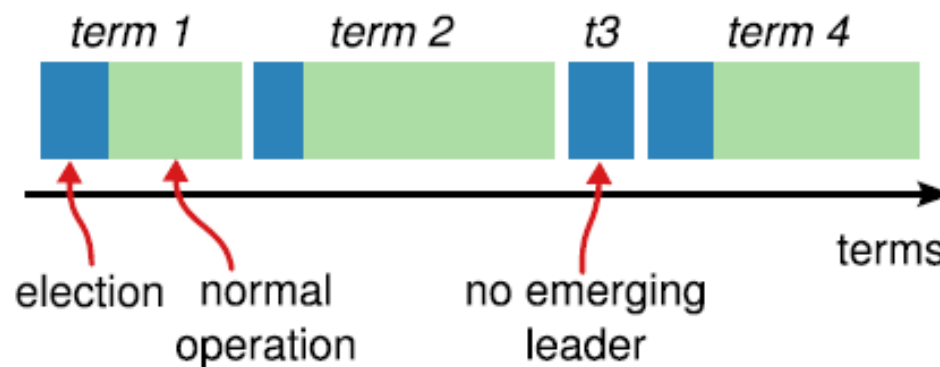
- ✓ Follower : **Completely Passive**. 接收来自Leader或者是Candidate的message , 并响应
- ✓ Leader : 处理所有来自客户端的请求 , 以及日志的复制 ( Log Replication )
- ✓ Candidate : 用于选主的中间状态

## ✓Message类型

- ✓ RequestVote
- ✓ AppendEntries ( HeartBeat )
- ✓ InstallSnapshot

## ✓Terms

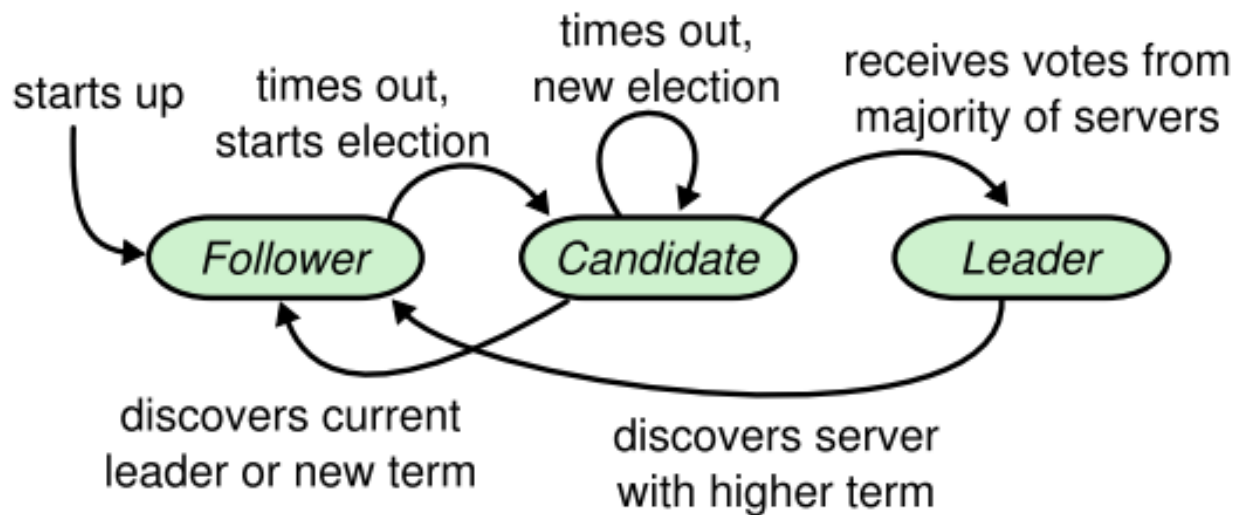
- ✓ 时间轴被划分为多个Terms
- ✓ Term ID : 每个Term的唯一标识 , 按时间轴递增
- ✓ 新Term随着Leader选举而开始
- ✓ 每个Term , 最多只有一个Leader ( 可能没有Leader , split vote )



# Raft功能分解

- ✓ **Leader选举 ( Leader Election )**
- ✓ **日志同步 ( Log Replication )**
- ✓ **Commit Index推进 ( Advance Commit Index )**
- ✓ **崩溃恢复 ( Crash Recovery )**
- ✓ **成员变更 ( Membership Change )**

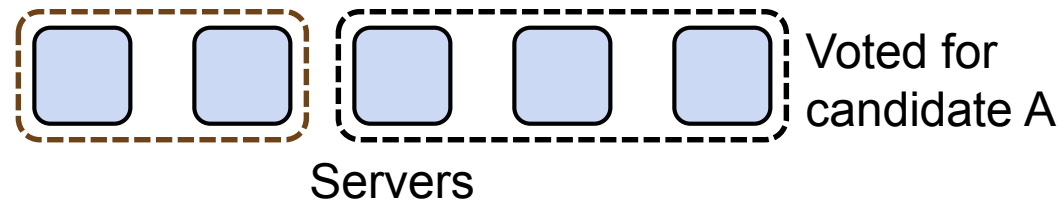
# Raft——Leader Election



## ✓ Leader Election关键点

- ✓ **超时驱动**：Heartbeat Message / Election Timeout
- ✓ **随机开始**：降低选举碰撞（Vote Split）导致的无主概率

B can't also  
get majority



- ✓ **选举动作**：current term++，然后发出RequestVote RPC
- ✓ **Safety**：同一Term，最多只会选出一个Leader，可以没有Leader

# Raft——Leader Election ( 注意事项 )

## ✓ Raft Leader Election 注意事项

- ✓ Leader Election与各节点的**绝对时间无关**，但是跟相对时间有关
  - ✓ 意味着：Servers间的绝对时间可以不相同，但是时间的流失速度需要基本一致（偏差不大）

## ✓ 影响Raft选举成功率的几个时间相关参数

- ✓ RTT ( Round Trip Time ) : 网络延时
- ✓ Heartbeat Timeout : 心跳间隔
- ✓ Election Timeout : Leader与Followers间通信超时触发选举时间
- ✓ MTBF ( Meantime Between Failure ) : Servers连续常规故障时间间隔

**RTT << Heartbeat Timeout < Election Timeout ( ET ) << MTBF**

- ✓ 随机选主触发： $s_{et} = \text{Random} ( ET, 2ET )$  ;
- ✓ 选举碰撞概率： $P(\Delta s_{et} \leq RTT)$

# Raft——Log Replication

## ✓ Raft日志格式

✓ Raft Log ( TermID ,  
LogIndex , LogValue )

✓ TermID , LogIndex用来标识日志的唯一性

## ✓ Log Replication关键点

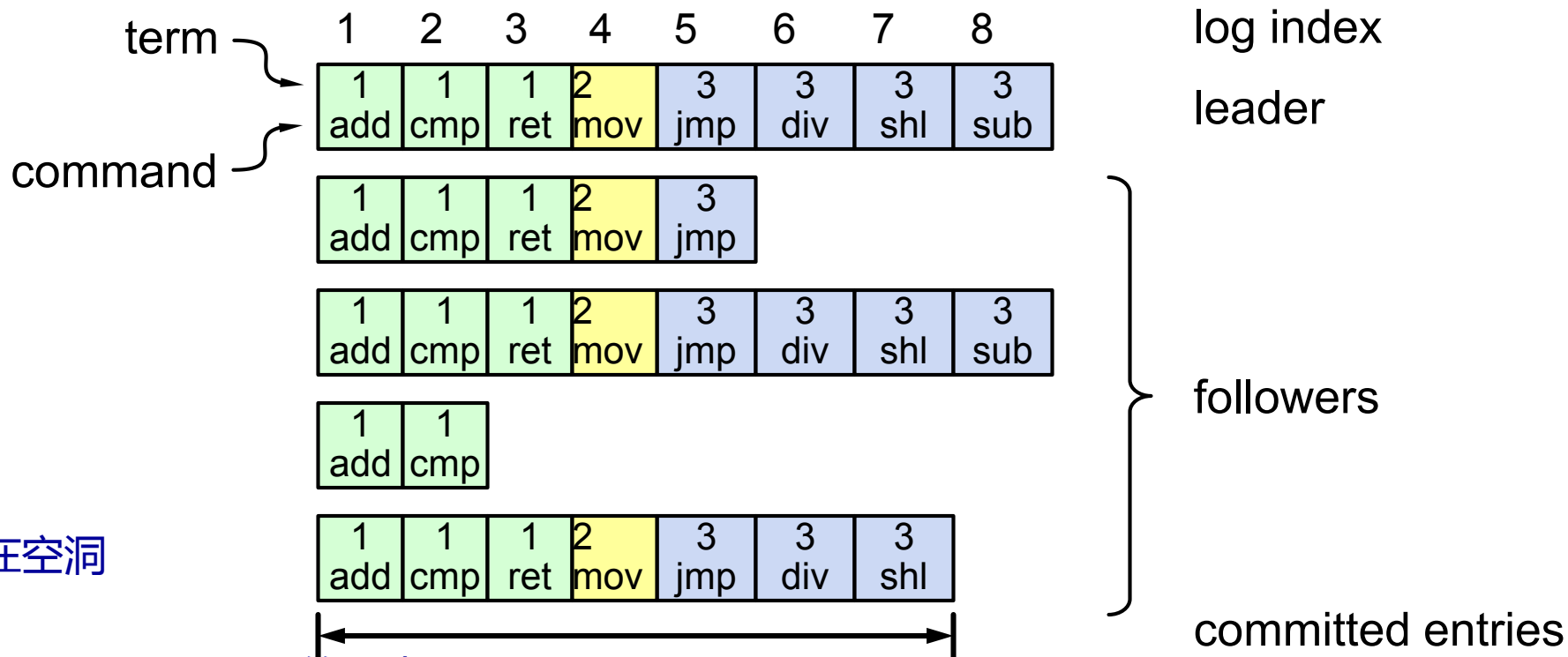
✓ 连续性：Raft日志不能存在空洞

✓ 有效性：

✓ 不同节点，拥有相同Term和LogIndex的日志  
Value一定相同

✓ Leader上的日志都是有效的

✓ Followers日志是否有效，通过与Leader日志对比来判断。How ?

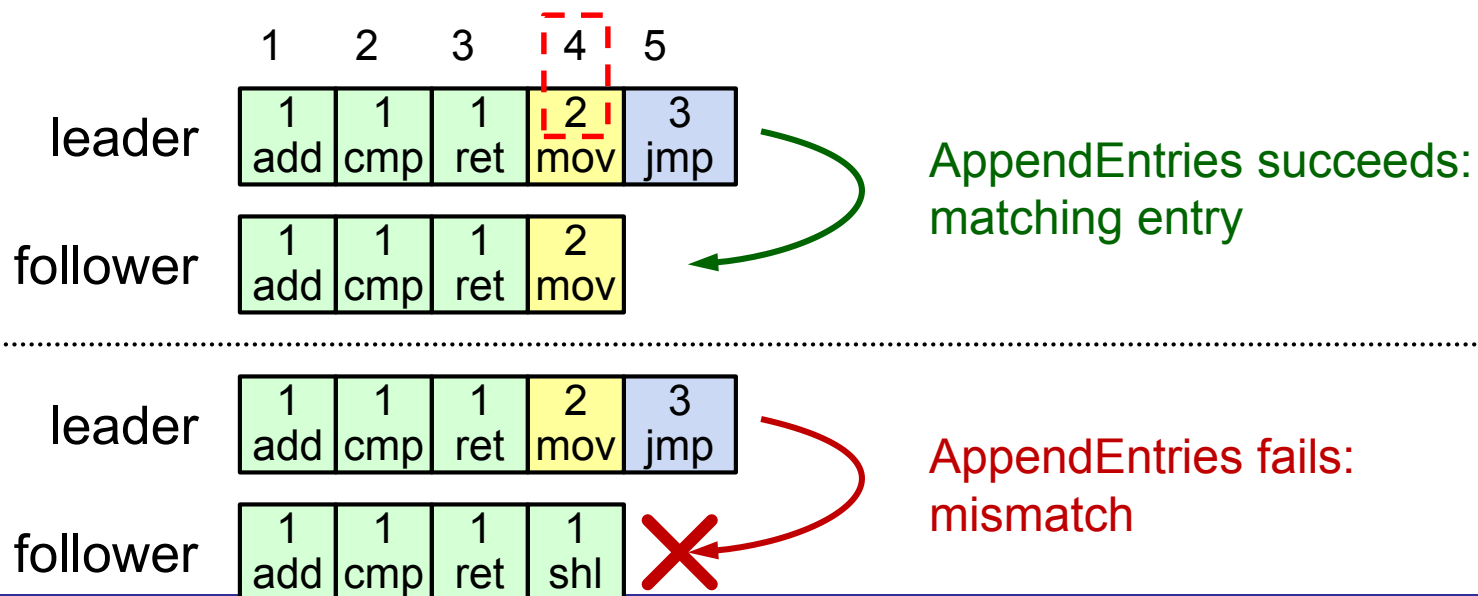




# Raft——Folowers日志有效性检查

## ✓ Followers日志有效性检查

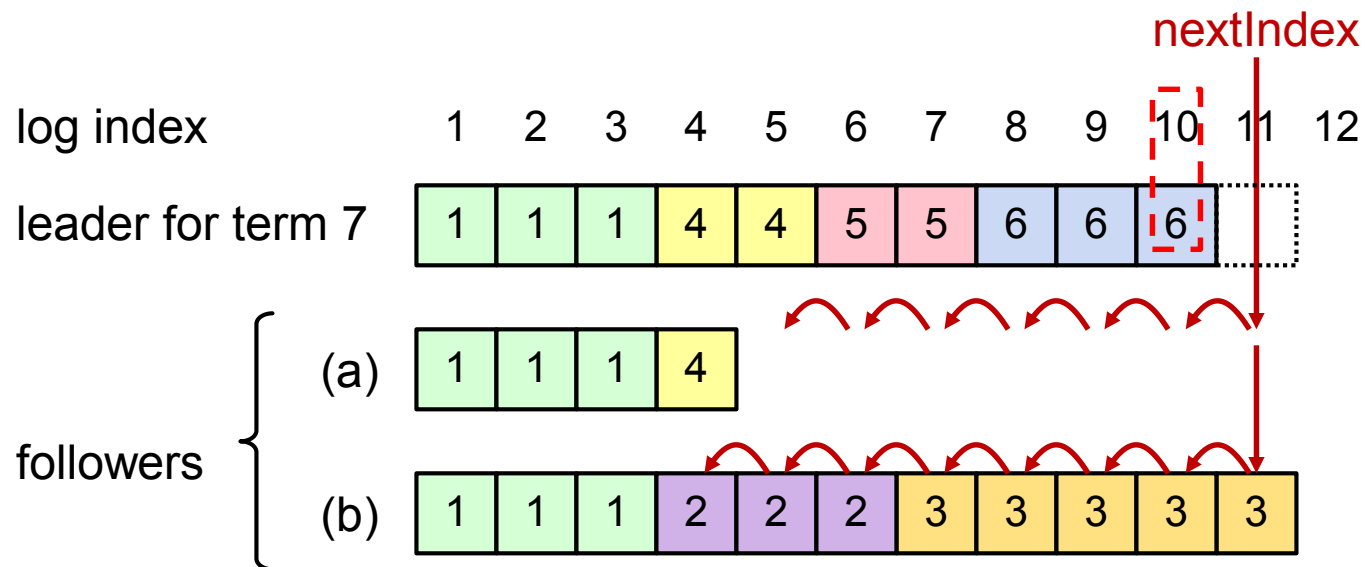
- ✓ AppendEntries RPC中，除了待复制的日志之外，还带有前一个日志的唯一标识 ( prevTermID , prevLogIndex )
- ✓ Follower在收到AppendEntries RPC之后，检查 ( prevTermID , prevLogIndex ) 与本地的日志是否Match：Match则接受，不match则拒绝
- ✓ 递归推导，证明Follower上 ( prevTermID , prevLogIndex ) 之前的所有日志，与Leader均相同



# Raft——Folowers日志修复

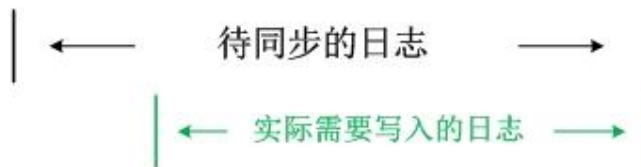
## ✓ Followers日志修复

✓ Followers上，不满足有效性检查的日志，如何修复？



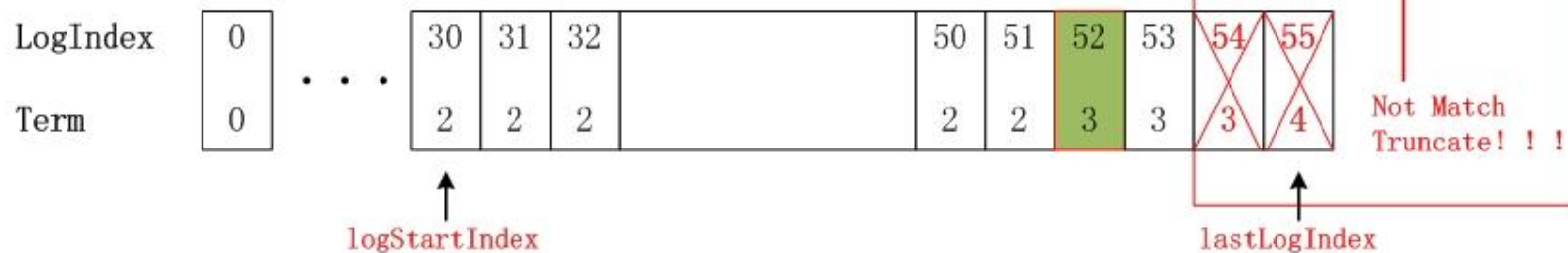
# Raft日志同步——Follower处理逻辑

## Request Log Append

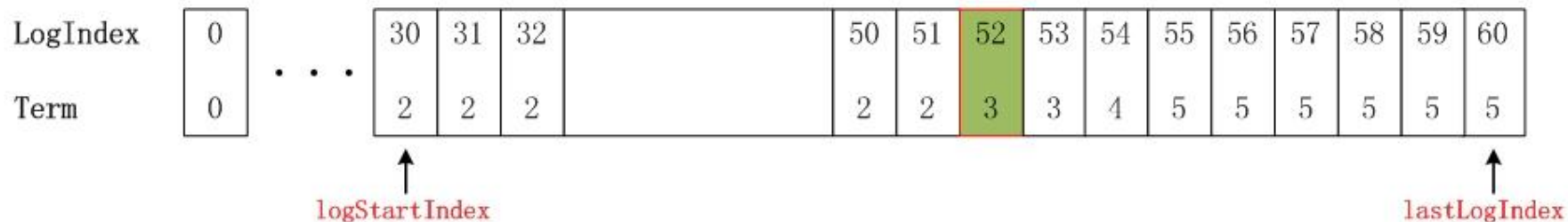


LogIndex	52	53	54	55	56	57	58	59	60
Term	3	3	4	5	5	5	5	5	5

## Follower Log

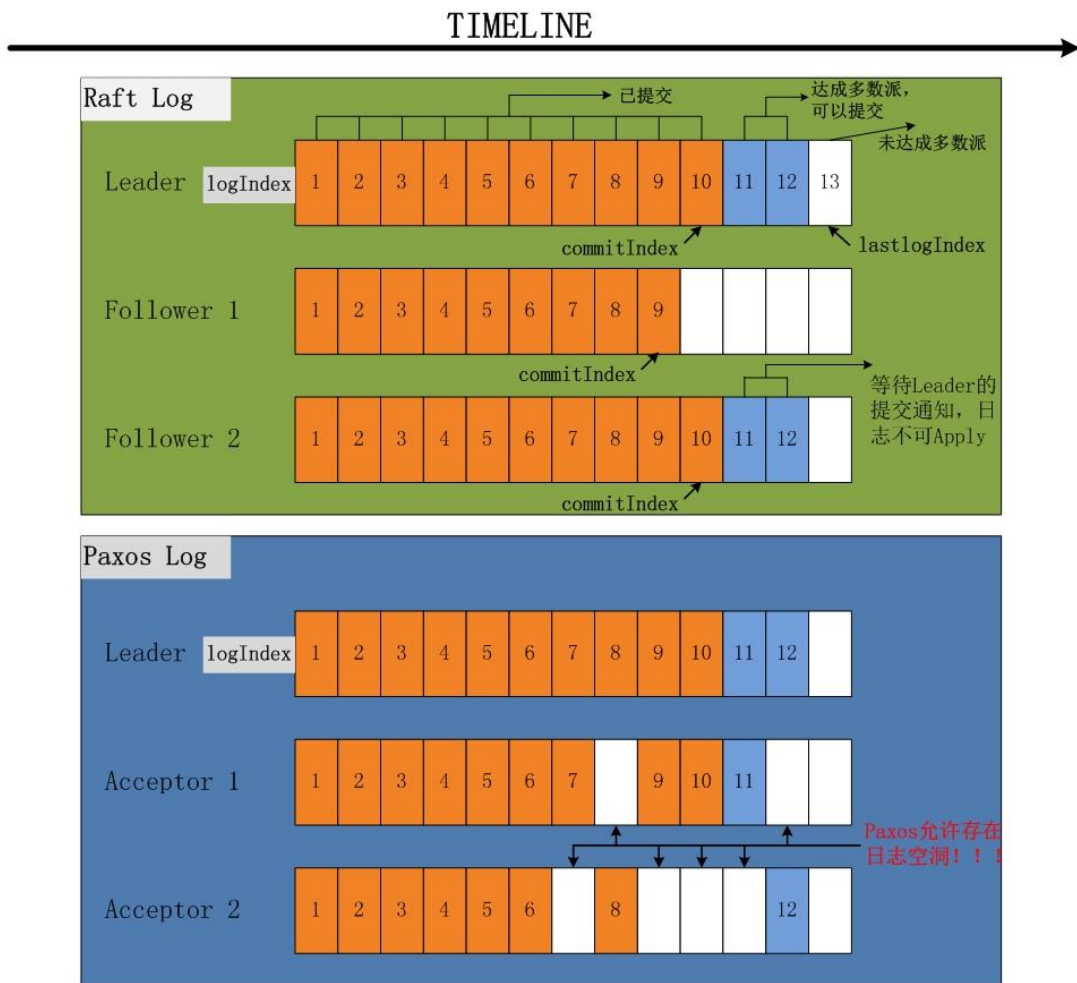


## Follower Log After Log Sync



# Log Replication——Raft vs Multi-Paxos

## Log Different between Paxos and Raft: Normal



✓ Multi-Paxos与Raft，在日志同步上最大的不同

✓ Raft需要日志连续，Multi-Paxos允许日志空洞存在

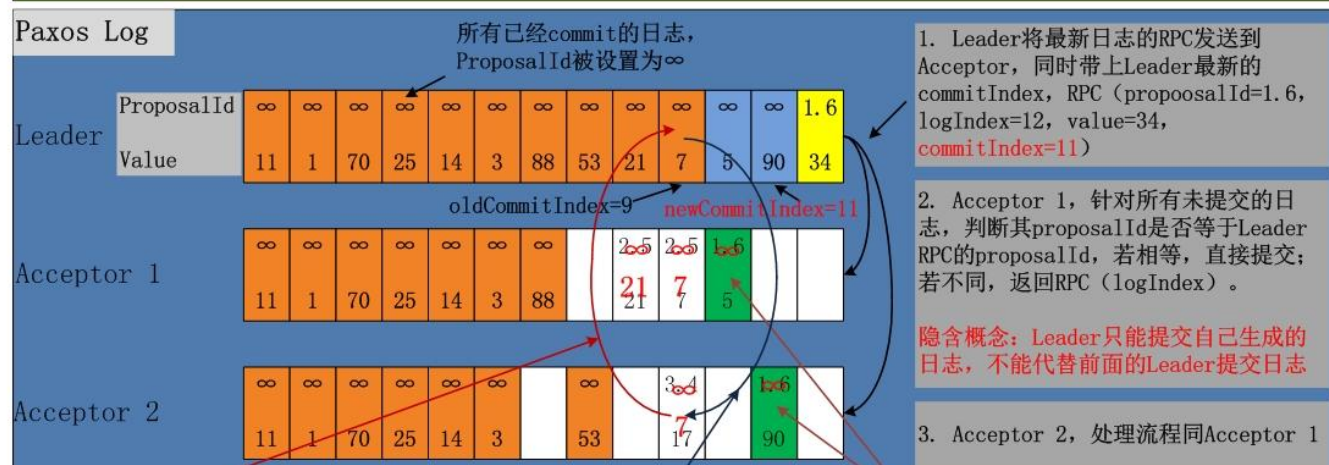
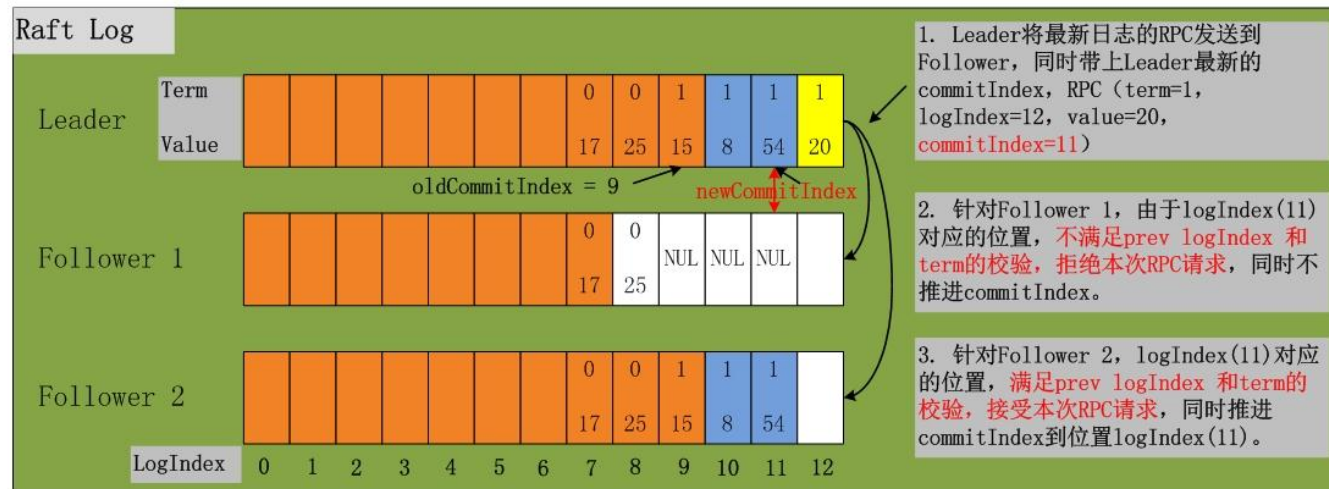
# Raft——Commit Index推进

## ✓CommitIndex ?

- ✓ 日志被复制到Followers之后，先持久化，并不能马上被应用
- ✓ 只有Leader知道日志是否达成多数派，是否可以应用（提交）
- ✓ 所谓的CommitIndex，就是已经达成多数派，可以应用的最新日志位置
- ✓ Followers记录当前的CommitIndex，所有小于等于CommitIndex的日志可以应用到状态机，所有大于CommitIndex的日志不能应用
- ✓ CommitIndex ( TermID , LogIndex )

## ✓CommitIndex推进 ?

- ✓ 在当前日志达成多数派之后，Leader在下一个AppendEntries RPC中，将当前CommitIndex信息发送到所有的Followers
- ✓ Followers日志有效性检查，接受AppendEntries RPC，则同时更新本地的CommitIndex。若检查不通过，拒绝AppendEntries RPC，则同样不更新本地CommitIndex



4. Acceptor 2的9号日志, 其proposalId(3.4)与Leader最新RPC中的proposalId(1.6)不同, 返回RPC(logIndex=9)。

5. Leader再次发出Success(logIndex, Value) RPC (9, 7), Acceptor 2收到Success RPC之后, 修改proposalId为∞, value从17改为7, 标识提交成功。  
隐含概念: Leader上已经提交的日志一定是有效的, 但Acceptor已accept但未提交的可能不一定有效, 用Leader日志覆盖即可。

proposalId相同, 设置为提交。修改proposalId为∞

3. Acceptor 2, 处理流程同Acceptor 1

## ✓ Raft vs Multi-Paxos

- ✓ Raft: 日志连续的特性, 确保CommitIndex推进逻辑非常简单
- ✓ Paxos: 日志空洞, ComitIndex处理逻辑较复杂

# Raft——AppendEntries RPC小结

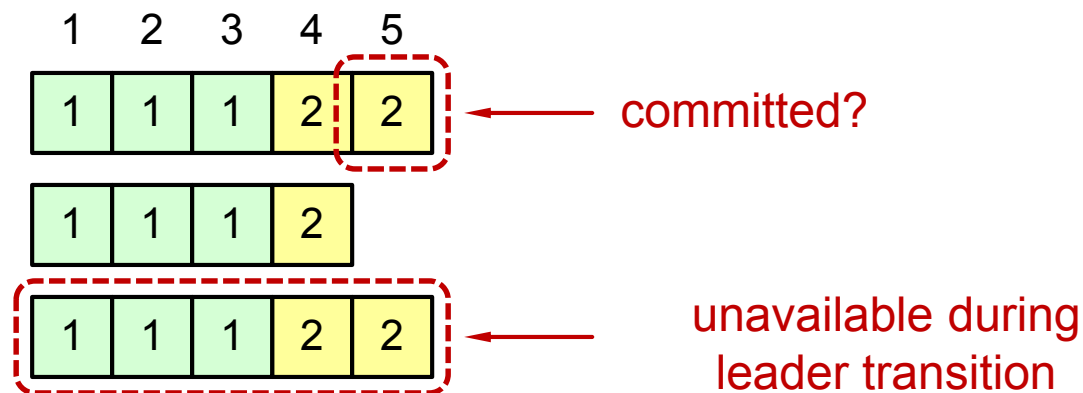
## ✓AppendEntries RPC

- ✓ 完整信息：( currentTerm , logEntries , prevTerm , prevLogIndex , commitTerm , commitLogIndex , currentTimeStamp )
- ✓ **currentTerm , logEntries**：需要同步的日志
- ✓ **prevTerm , prevLogIndex**：Follower日志有效性检查
- ✓ **commitTerm , commitLogIndex**：当前日志的最新提交位点
- ✓ **currentTimeStamp**：心跳时间



# Raft——Leader Crash

## ✓Leader Crash



## ✓New Leader选取原则

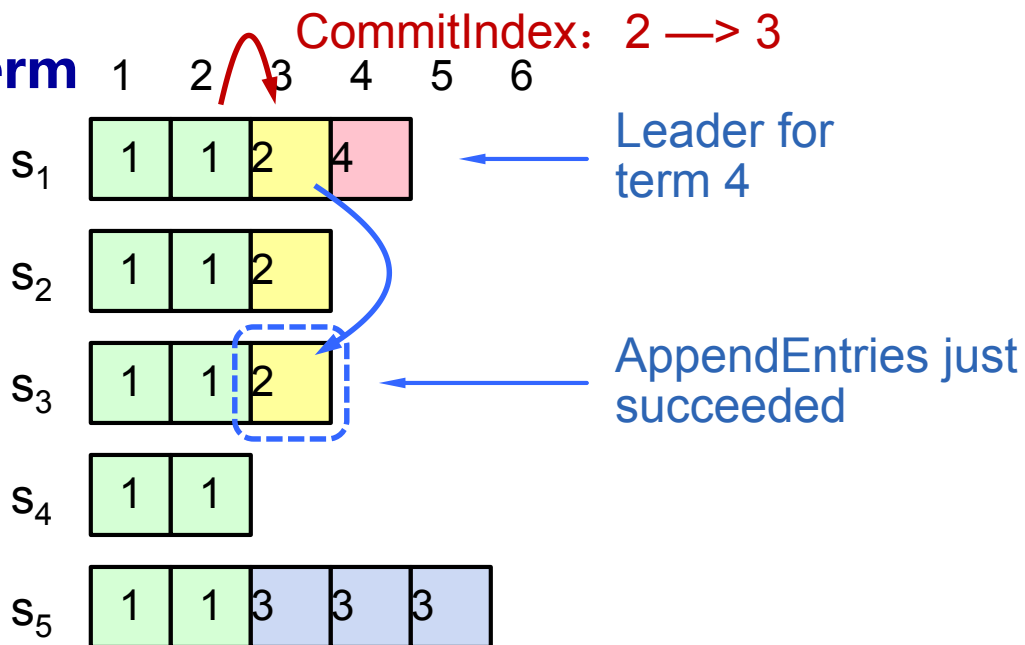
- ✓ **最大提交原则** : During elections, choose candidate with log most likely to contain all committed entries
- ✓ Voting server V denies vote if its log is "more complete" :  
 $(\text{lastTerm}_V > \text{lastTerm}_C) \parallel$   
 $(\text{lastTerm}_V == \text{lastTerm}_C) \ \&\& \ (\text{lastIndex}_V > \text{lastIndex}_C)$



# Raft——Leader Crash ( 一个特殊场景 )

## ✓ Committing Entry from Earlier Term

- ✓ Term 1 Leader :  $s_1$
- ✓ Term 2 Leader :  $s_2$
- ✓ Term 3 Leader :  $s_5$
- ✓ Term 4 Leader :  $s_1$



## ✓ New Leader提交原则

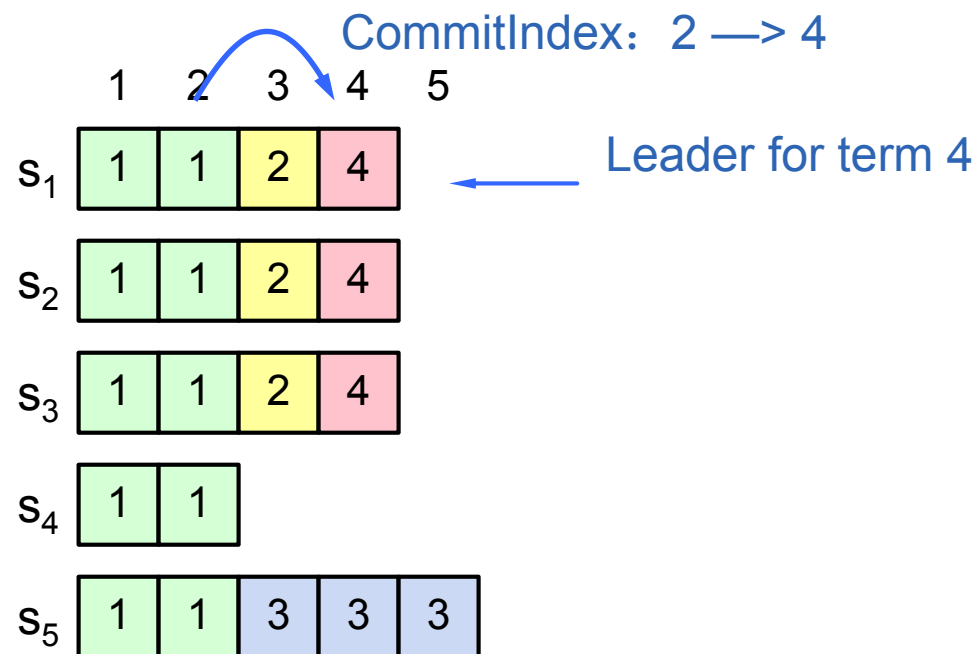
- ✓ New Leader只能提交属于自己Term的日志，不能主动帮之前的Term提交 ( Advance CommitIndex )
- ✓ Why ?

✓考虑如下场景：Term 4提交了LogIndex 3 ( Term 2 )，此时 $s_1$ 再次Crash， $s_5$ 被重新选为主。 $s_5$ 重新提交LogIndex 3 ( Term 3 )，Term 2被覆盖，违背Consensus协议原则

# Raft——Leader Crash ( 一个特殊场景：修复 )

## ✓ New Leader提交原则

- ✓ New Leader只能提交属于自己Term的日志，**不能主动**帮之前的Term提交



## ✓ 不能主动提交，被动提交如何？

- ✓ New Leader在写入新日志之后，直接将CommitIndex从LogIndex 2推进到LogIndex 4，**隐式提交**了LogIndex 3
- ✓ 此时，若S1再次Crash，S5不会被选为新主，因为其Term ( 3 ) 小于已经达成多数派的Term ( 4 )

# Raft——Leader Stic

## Raft一个异常场景的分析

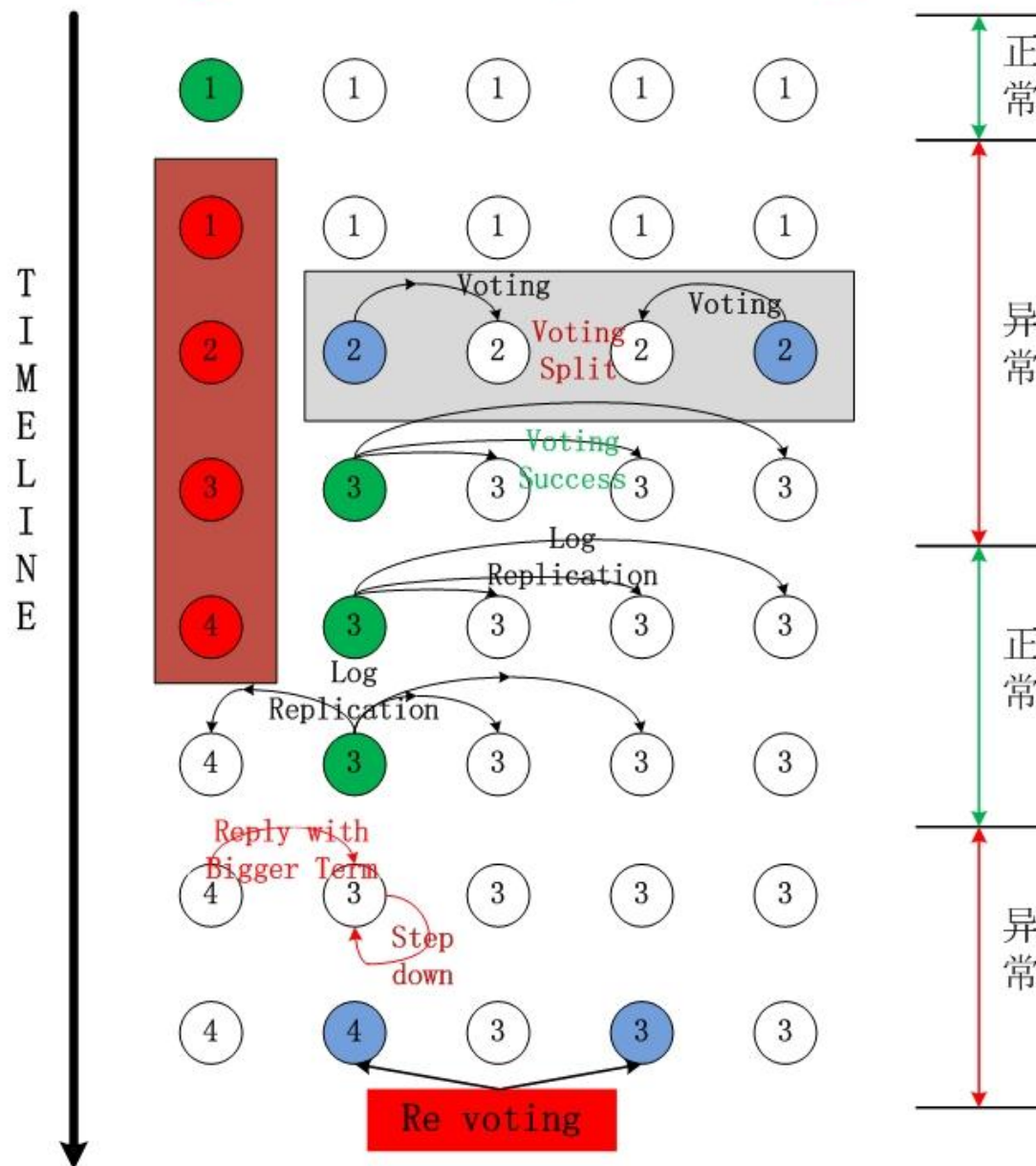
### ✓右图所示

- ✓ 第一阶段：5节点系统正常工作
- ✓ 第二阶段，Leader Network Partition，系统进入重新选主
- ✓ 第三阶段，节点2选主成功，系统恢复正常运行
- ✓ 第四阶段，节点1 Network Partition消除，重新加入。由于其TermID大于当前系统TermID，新主Stepdown，系统需要重新选主

### ✓Leader Stickiness

- ✓ 消除节点加入正常服务集群后，正常服务集群Leader被抢占

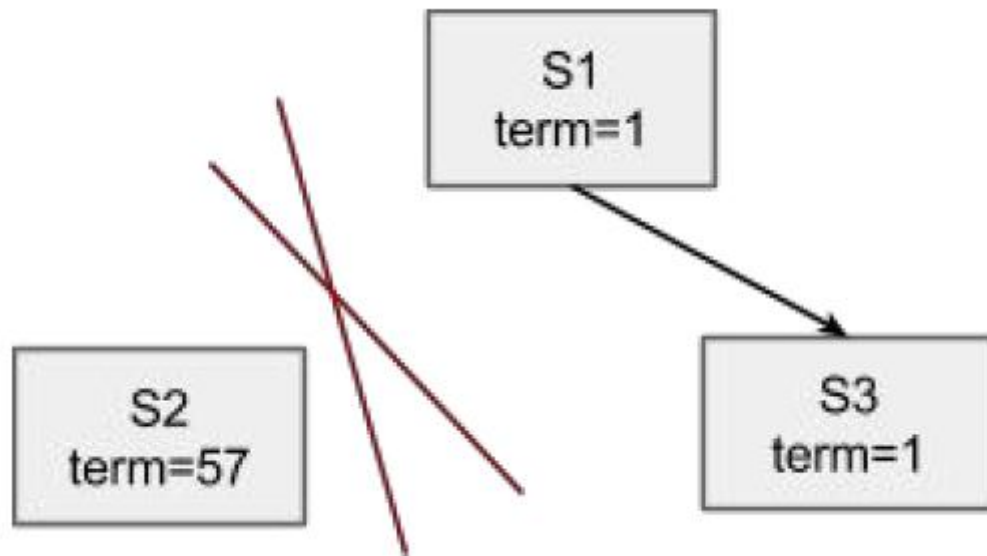
1 绿色：Leader 1 红色：Network Partition 3 蓝色：Candidate 1 数字：TermID



# Raft——Leader Stickiness

## ✓Term Inflation

- ✓ Network Partition节点，由于跟集群失联，不断地试图选择自己为Leader，增加本地的Term ID
- ✓ 待Network Partition消除，节点重新加入集群，其Term已经远远大于正常集群的Term，导致集群无法正常服务，Leader Stepdown

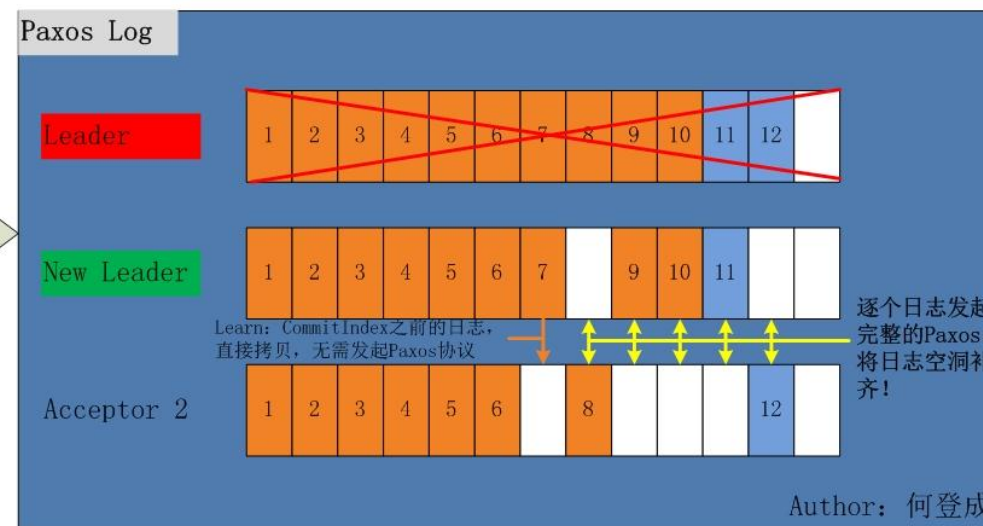
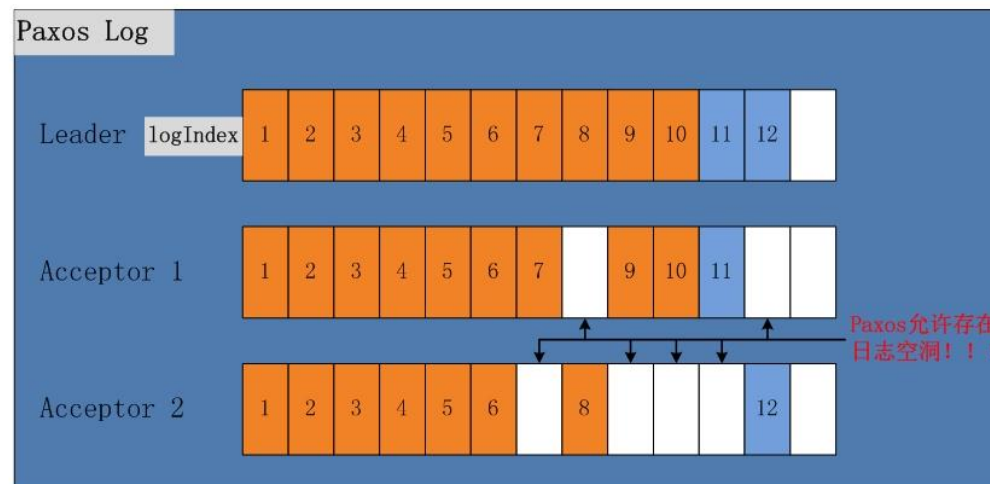
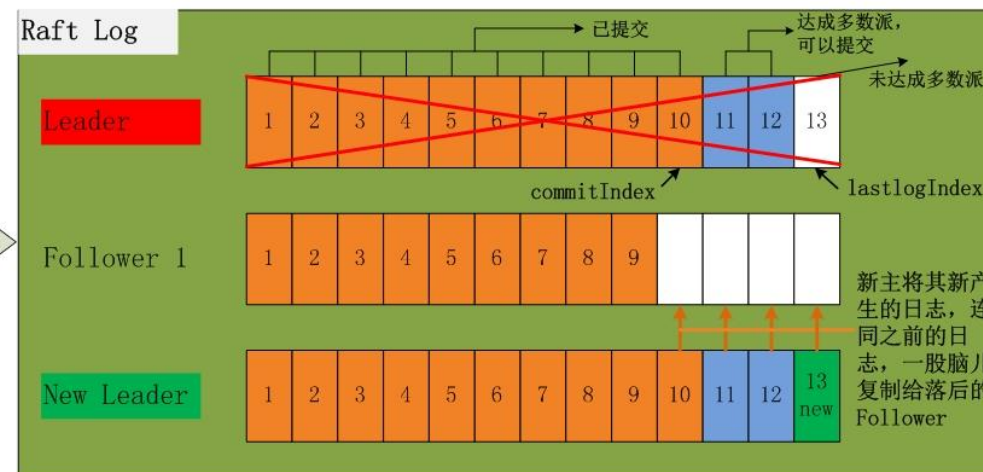
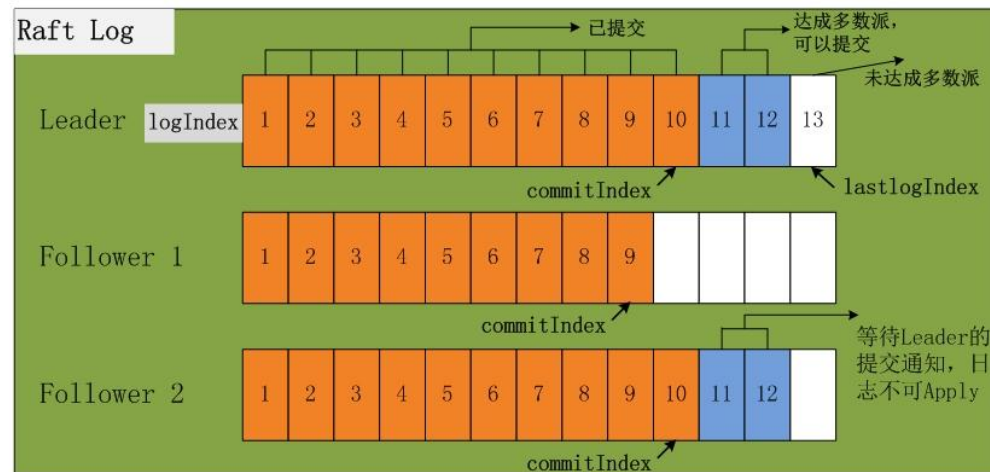


## ✓Pre-Vote

- ✓ 新增Pre-Vote RPC ( currentTerm+1 , lastLogIndex , lastLogTerm )
- ✓ 不修改Server的任何状态
- ✓ Pre-Vote RPC多数派返回成功，增加本地Term，将节点状态转换为Candidate，发出真正的Vote RPC
- ✓ 分区后的节点，Pre-Vote不会成功，Term不会增加。新加入也就不会导致集群无法正常服务

## TIMELINE

✓Le



# Raft vs Multi-Paxos : Leader Crash

## ✓Raft

- ✓ 必须选择拥有最多最新日志的Follower，作为新Leader
- ✓ 新Leader上任之后，自己的日志是最全的准确的（根据Leader选取原则），直接覆盖Follower的即可

## ✓Multi-Paxos

- ✓ 可选择任意节点作为新Leader
- ✓ 新Leader上任之后
  - ✓ 获取当前节点的CommitIndex
  - ✓ 获取整个系统中所有节点的MaxIndex
  - ✓ 本节点CommitIndex之前的日志，可直接复制给Acceptors
  - ✓ [CommitIndex, MaxIndex]之间的每一条日志，需要逐个应用Basic Paxos，确定该选择哪一个LogEntries
  - ✓ MaxIndex之后的所有日志，在Leader选举阶段就完成了一次Prepare，后续直接Accept即可

# Raft——Membership Change

## ✓Membership Change

- ✓ **包括**：系统中，节点增加、删除、替换
- ✓ **不包括**：节点Crash、Crash节点重启加入
  - ✓ Raft设计，就是允许少数派节点Crash，不影响系统的可用性

## ✓Membership Change设计目标

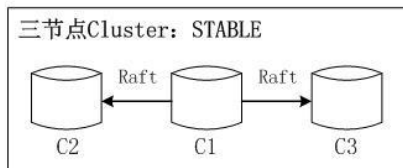
- ✓ Online Membership Change，过程不影响系统的持续可用



# Raft——Men

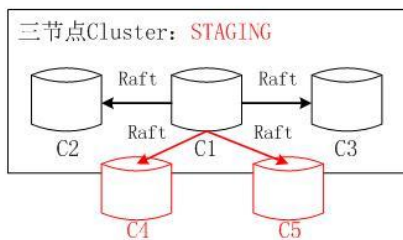
## Raft系统状态转移

源状态



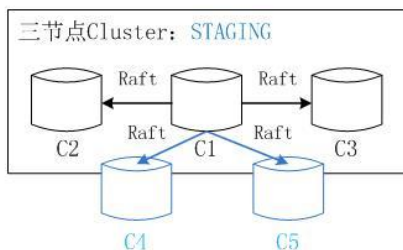
Conf (C1、C2、C3)

Step 1:  
Add 2 Nodes



Conf (C1、C2、C3)  
C4、C5: Uncatch up

Step 2:  
Catch Up

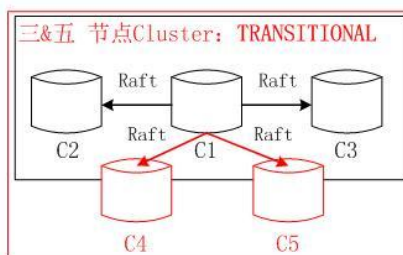


Conf (C1、C2、C3)  
C4、C5: Catch up

### ✓ Step 3 : Transitional Stage

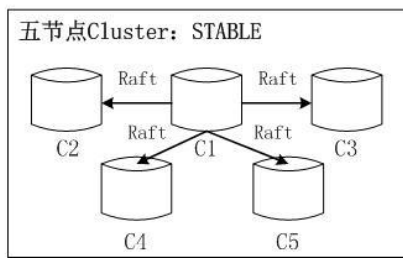
- ✓ Conf\_old、Conf\_new两个 Configuration Group，需要同时达成多数派，Log Replication才算是成功

Step 3:  
Transition



Conf\_old (C1、C2、C3)  
C\_new (C1、C2、C3、C4、C5)

最终状态



Conf (C1、C2、C3、C4、C5)

### ✓ Online Membership Change ( 关键词 )

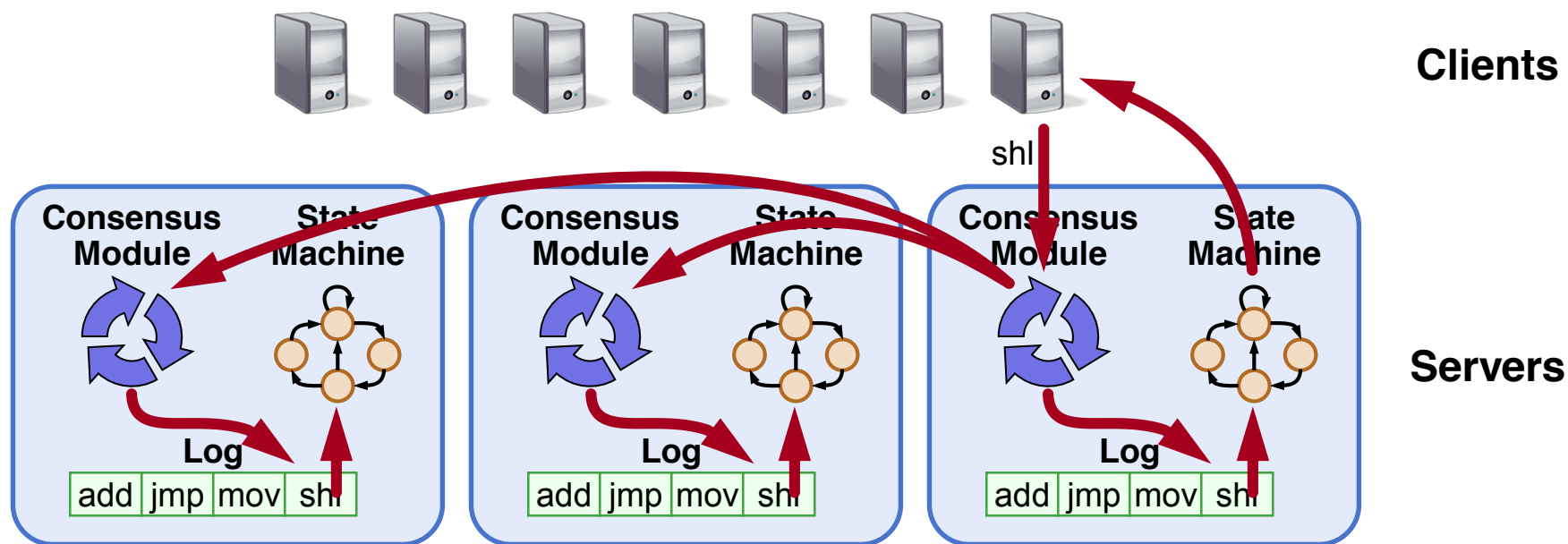
- ✓ 系统状态变化同样通过日志同步
- ✓ 新老配置共存平滑过度
- ✓ 一次可变更多个Member  
( Add/Remove )



## Raft/Multi-Paxos：阶段总结

## ✓ Now , what we learn ?

- ✓ 学会了Raft/Multi-Paxos，能用来解决连续Consensus问题，确定连续多个提案，确保系统各节点状态完全一致
- ✓ 学会了自动化的Leader Election，保证系统在少数派宕机下的持续可用
- ✓ 学会了日志强同步，保证系统在Leader宕机后的零数据丢失
- ✓ 实现所有上面这些功能，都是自封闭的，零外部系统依赖



# Raft vs Multi-Paxos 小结

- ✓ **Paxos的核心**，在Basic Paxos，关于Multi-Paxos缺乏太多的理论论证。
- ✓ **Raft则将Multi-Paxos的功能做了模块划分，并对每个模块做了详细的阐述和理论证明**
  - ✓ Leader Election、Log Replication、Commit Index Advance、Crash Recovery、Membership Change 等
- ✓ **Multi-Paxos和Raft，最本质的区别，在于是否允许日志空洞。Raft必须连续，不允许空洞。Multi-Paxos允许日志空洞存在**
  - ✓ **允许空洞**：Paxos应对复杂网络环境，更为鲁棒（以三副本为例）
    - ✓ 三节点两两不连通：Raft、Paxos均无法工作
    - ✓ 两个Followers与Leader间的网络，短时间内轮询连接震荡：Paxos不受影响，Raft会受部分影响
    - ✓ 除此之外，Raft、Paxos的表现无差别，均可持续正常服务
  - ✓ **不允许空洞**：Raft在Log Replication、Crash Recovery、CommitIndex Advance等模块的算法实现上，更为简单

# Paxos/Multi-Paxos/Raft能干什么？

Google Chubby

Zookeeper

MegaStore

Spanner



ETCD

CockroachDB

MongoDB

OceanBase

...

# Paxos/Multi-Paxos/Raft能干什么？

## ✓Multi-Paxos/Raft功能回顾

- ✓ **系统各节点状态强一致**
- ✓ 系统少数派宕机，不影响可用性（持续可用）
- ✓ 系统本身功能自封闭，零外部系统依赖

## ✓作为大型系统的Build Block

- ✓ MegaStore、Spanner、CockroachDB、TiDB ...

## ✓作为基础系统对外提供服务

- ✓ 服务种类：服务发现、分布式锁服务、配置中心（元数据管理）、集群管理（Leader Election） ...
- ✓ 服务产品：Zookeeper、ETCD ...
- ✓ **所有服务的本质，就是Server状态机中的一个状态变更，通过Paxos/Raft来保证状态变更全局一致和高可用**

# Example : ETCD

✓基于Raft协议实现的，提供服务发现和配置管理的高可用键值系统

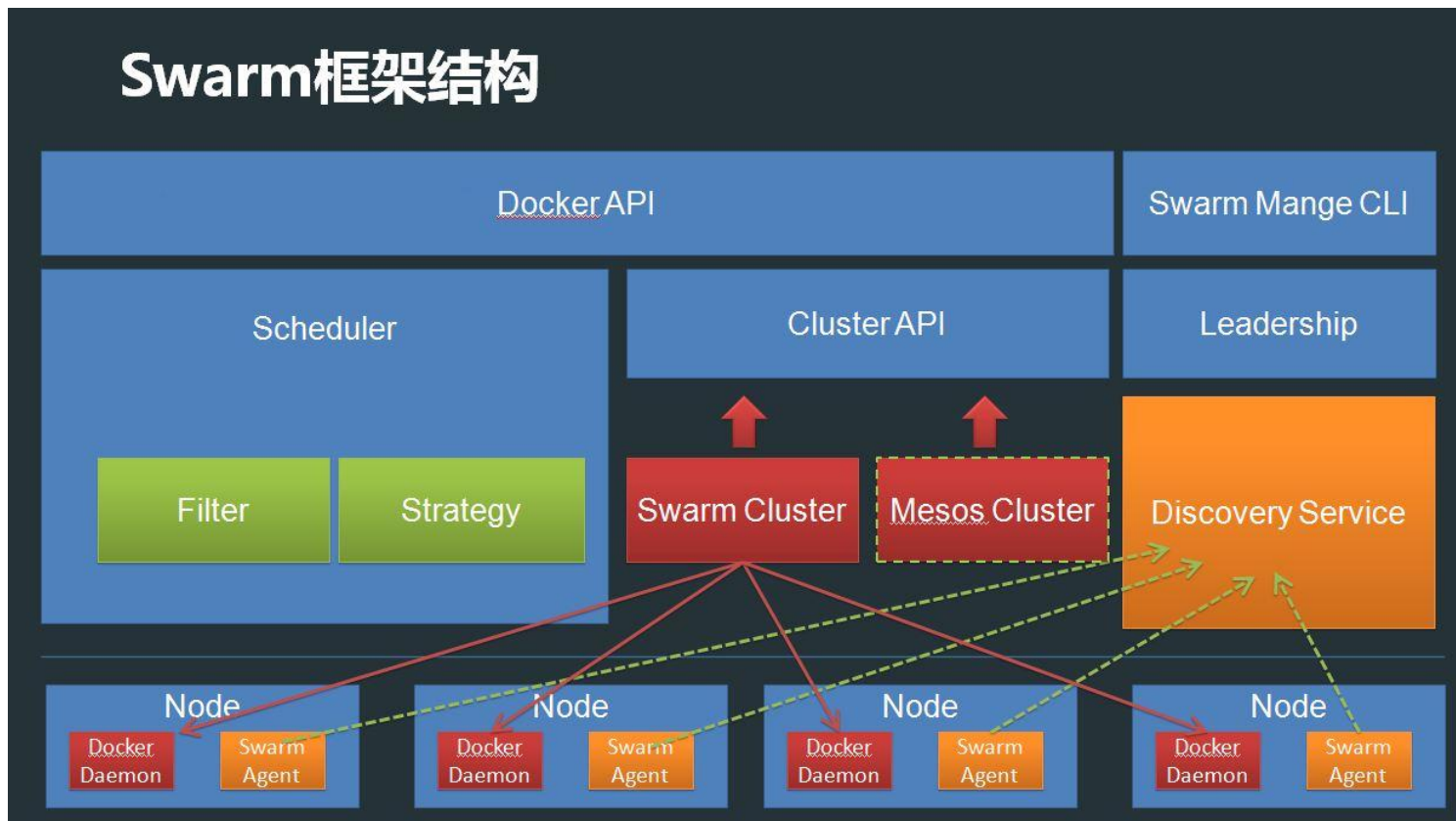


## ✓功能接口

- ✓ Write a key、Read keys
- ✓ Watch key changes
- ✓ Grant leases
- ✓ ...

## ✓使用场景

### Swarm框架结构



# 参考资料

- ✓Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm.
- ✓Tushar Chandra. Paxos Made Live - An Engineering Perspective.
- ✓Jason Baker. Megastore: Providing Scalable, Highly Available Storage for Interactive Services.
- ✓JAMES C. CORBETT. Spanner: Google' s Globally Distributed Database.
- ✓Mike Burrows. The Chubby lock service for loosely-coupled distributed systems.
- ✓Diego Ongaro and John Ousterhout. [Raft: A Consensus Algorithm for Replicated Logs](#).
- ✓John Ousterhout and Diego Ongaro. [Implementing Replicated Logs with Paxos](#).
- ✓李凯. [架构师需要了解的Paxos原理、历程及实战](#).
- ✓The Secret Lives of Data. [Raft Understandable Distributed Consensus](#).
- ✓raft github. [The Raft Consensus Algorithm](#).
- ✓Henrik Ingo. [Four modifications for the Raft consensus algorithm](#).
- ✓Jason Wilder. [Docker Service Discovery Using Etcd and Haproxy](#).
- ✓Jason Wilder. [Open-Source Service Discovery](#).
- ✓线超博. [DockOne技术分享 \( 二十 \) : Docker三剑客之Swarm介绍](#).
- ✓陌辞寒. [Zookeeper和etcd使用场景](#).

# 致谢

谢谢大家！