

Paxos原理及简单实现

胡勇民：SY1806613

刘璨：SY1806704

胡继文：SY1806718

Consensus Problem

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

□ 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，只能有一个结论；
- ◆ 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；

Paxos

□ Paxos协议(Lamport):

- ◆ 一类协议的统称，常见如：basic-paxos、multi-paxos、raft 等。

□ Paxos协议解决的问题:

- ◆ 在不可靠信道中，多个参与者达成一致观点，即将所有节点都写入同一个值或者一个命令序列，且被写入后不再更改。

□ 基本假设:

- ◆ 节点间采用消息通信
- ◆ 采用异步(twisted库)、非拜占庭模型(不存在消息篡改)
- ◆ 消息可能丢失、重复、乱序(网络不可靠)

Basic-Paxos(basic concepts)

□ 三个角色:

- ◆ Proposer: 提出提案, 提案信息包括提议编号(number)和提议(value)
- ◆ Acceptor: 对提议进行决策, 即是否 accept value
- ◆ Learner: 把通过的确定性取值同步给其他未确定的 Acceptors
- ◆ Note: 每个参与者可担任多个角色

□ 算法保证一致性的基本语义:

- ◆ 提议(value)只有在被 proposers 提出后才能被批准
- ◆ 在一次 Paxos 算法的执行实例中, 只批准(chosen)一个 value
- ◆ Learners 只能获得被批准(chosen)的 value



Basic-Paxos(basic concepts)

基于上述语义，可推导出四个约束，用以实现一致性

- ❑ P1 : 一个acceptor必须 accept 第一次收到的 proposal;
- ❑ P2^a: 一旦一个具有value v 的提案被批准 (chosen), 那么之后任何acceptor 再次 accept 的 proposal 必须具有value v ; (后者认同前者)
- ❑ P2^b: 一旦一个具有value v 的提案被批准(chosen), 那么以后任何proposer 提出的 proposal 必须具有value v ;
- ❑ P2^c: 若一个编号为 n 的提案具有value v , 那么存在一个多数派, a) 要么其中所有 acceptor 都没有 accept 编号小于 n 的任何提案; b) 要么大多数 acceptor 批准的所有编号小于 n 的提案中编号最大的那个提案具有value v ;



Basic-Paxos(Algorithm)

□ 算法变量定义:

- ◆ Proposal Number: 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。单一递增, 大小代表了优先级(**作用?**); 另外参考Chubby论文中提议编号的给定方法, 假设有 n 个 Proposer, 每个编号为 n_r , 则 Proposal 编号的任何值 S 都应该大于它已知的最大值, 并且满足:

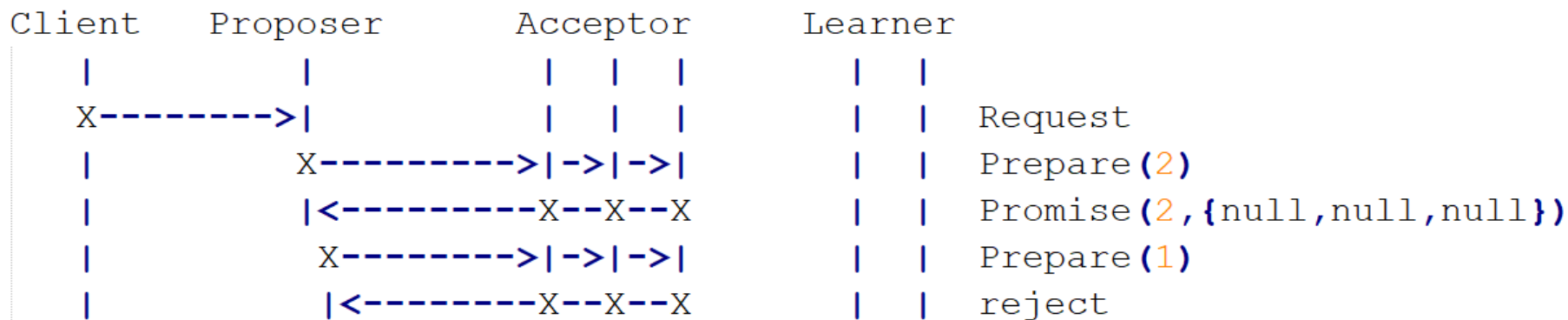
$$S = k * n + n_r$$

- ◆ Value: 提议的值, 可以是某个操作(设置某个变量的值...);
- ◆ Instance: 由用户请求转化的 Paxos 实例;

Basic-Paxos(Algorithm)

□ Prepare 阶段:

- ◆ 当Proposer提出方案 v_x ，需要先向大多数 Acceptor 发出prepare请求(含序列号 $\langle n_x \rangle$);
- ◆ 当Acceptor接收到prepare请求 $\langle n_x \rangle$ 时，检查自身上次回复过的prepare请求 $\langle n_{\max_last} \rangle$ ：若 $n_x > n_{\max_last}$ ，回复 $\langle ok, n_{highest_accepted}, v_{accepted} \rangle$ 并promise不再处理编号小于 n_x 的请求，即 $n_{\max_last} = n_x$ ；否则，reject失败请求需回到上步，修改 n_x 再重新发起提议；(Note: 两种情况)

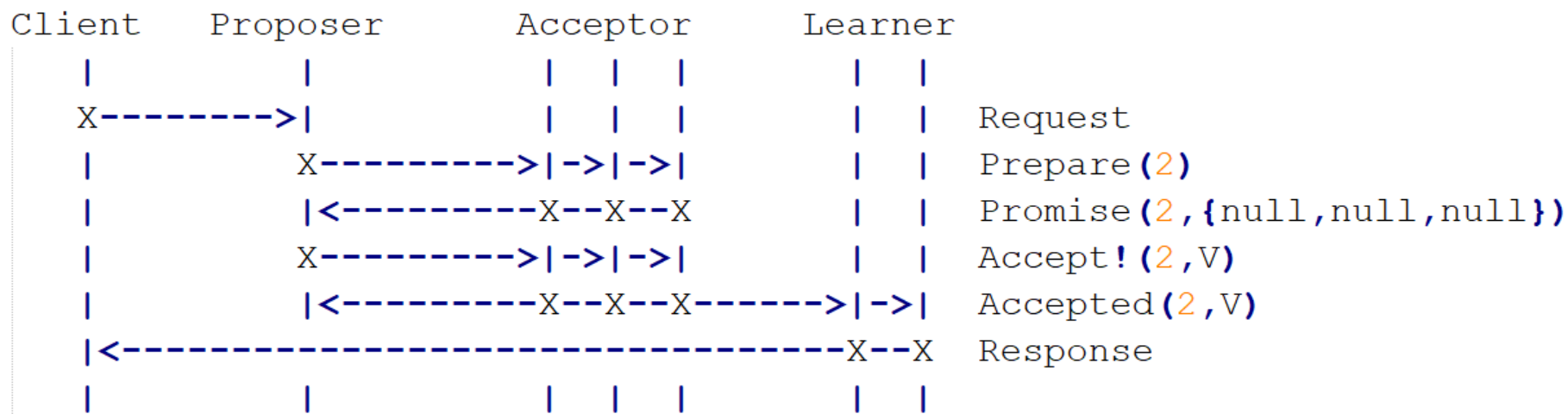


Basic-Paxos(Algorithm)

□ Accept 阶段:

收到半数Acceptors承诺的Proposer, 发Accept请求($\langle n_i, v_i \rangle$) 到所有节点:

- ◆ 若 $n_i \geq n_{\max_last}$, $n_{highest_accepted} = n_{\max_last} = n_i$, $v_{accepted} = v_i$, 并本地持久化, 返回 `accepted` 给 Proposer; 若该 Proposer 收到的 `accepted` 过半, 则 v_i 在本轮 Instance 中被批准(chosen), 否则返回上一阶段;
- ◆ 否则, `reject` 并且返回 n_{\max_last} , 失败请求需返回上一阶段重新开始;





Basic-Paxos(Algorithm)

□ Commit 阶段(可优化):

发出 Accept 请求的 Proposer, 在收到过半 Acceptors 的 accepted 之后, 标志着本次 Accept 成功, 可向所有 Acceptors 追加 Commit 消息, 也即 Learn 的过程

□ 异常情况(持久存储)

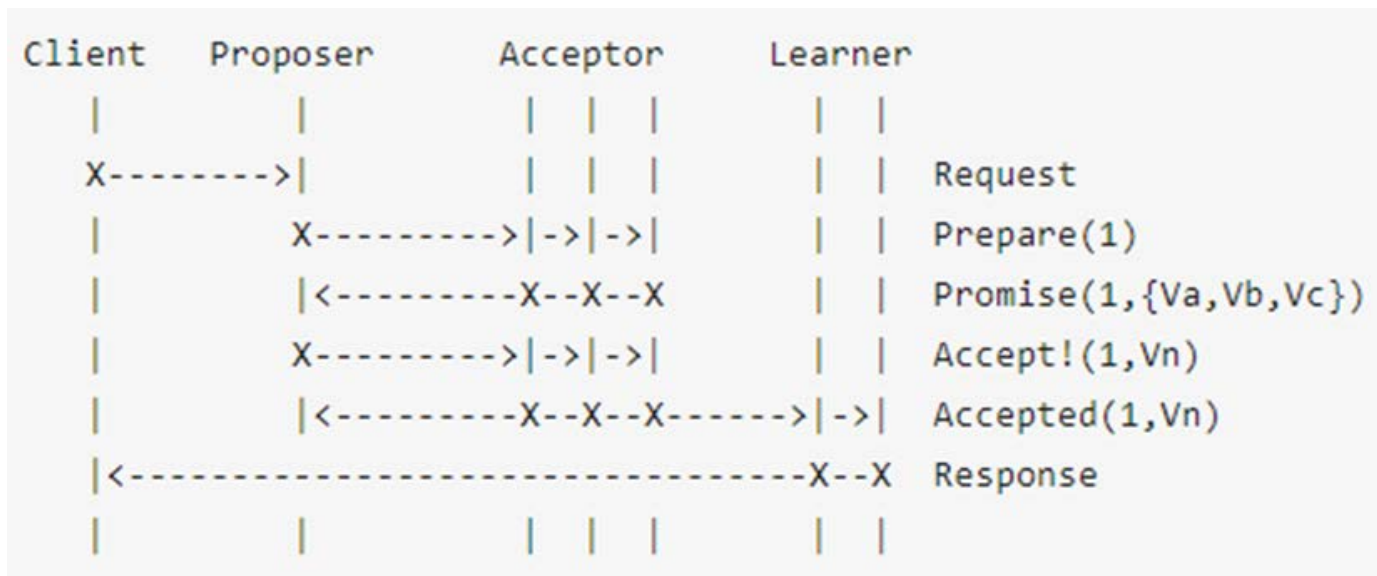
- ◆ 执行过程中有多种异常(宕机, 存储失败等等), 因此所有节点都需实现持久存储, 以做到重启后仍能正确参与 Paxos 处理;
- ◆ Proposer 存储已提交的最大 Proposal 编号、决议编号(Instance id);
Acceptor 存储 n_{\max_last} 、 $n_{highest_accepted}$ 、 $v_{accepted}$ 、Instance id;
Learner 存储 $n_{learned}$ 、 $v_{learned}$;

Basic-Paxos(Liveness problem)

Client	Proposer	Acceptor	Learner
X----->			Request
	X----->	> >	Prepare(1)
	<-----X--X--X		Promise(1,{null,null,null})
	!		!! LEADER FAILS
			!! NEW LEADER (knows last number was 1)
	X----->	> >	Prepare(2)
	<-----X--X--X		Promise(2,{null,null,null})
			!! OLD LEADER recovers
			!! OLD LEADER tries 2, denied
	X----->	> >	Prepare(2)
	<-----X--X--X		Nack(2)
			!! OLD LEADER tries 3
	X----->	> >	Prepare(3)
	<-----X--X--X		Promise(3,{null,null,null})
			!! NEW LEADER proposes, denied
	X----->	> >	Accept!(2,Va)
	<-----X--X--X		Nack(3)
			!! NEW LEADER tries 4
	X----->	> >	Prepare(4)
	<-----X--X--X		Promise(4,{null,null,null})
			!! OLD LEADER proposes, denied
	X----->	> >	Accept!(3,Vb)
	<-----X--X--X		Nack(4)
			... and so on ...

Multi-Paxos(Algorithm)

❑ basic paxos的局限性



从上图可以看出，第二阶段的时候client仍需要等着，只有在第二阶段大半acceptor返回accepted后，client才能得到成功的信息。而在高并发情况下可能需要更多的网络来回，极端情况下甚至可能形成活锁。为此有必要对其进行改进。

Multi-Paxos(Algorithm)

❑ Multi-Paxos有两点改进:

1.针对每一个要确定的值，运行一次Paxos算法实例（Instance），形成决议。每一个Paxos实例使用唯一的Instance ID标识。例如，n个进程对同一个值进行写操作，每一次写都是一个实例，所以会产生n个实例，同时写操作也被串行化了，避免不一致性的情况出现。

2.在所有Proposers中选举一个Leader，由Leader唯一地提交Proposal给Acceptors进行表决。这样没有Proposer竞争，解决了活锁问题。在系统中仅有一个Leader进行Value提交的情况下，Prepare阶段就可以跳过，从而将两阶段变为一阶段，提高效率。

Multi-Paxos(Algorithm)

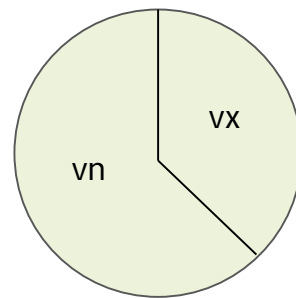
□ 选主状态

由集群中的任意节点拉票发起选主，拉票中带上自己的 v_x ，通过收集集群中半数以上的 v_x ，来更新自己的 v_x 值，得到目前集群通过的最大 $v_x = v_n$ 。

□ 为何选主的时候，半数的 v_x 就可以确定集群最大 v_n ？

选主的时候，半数的 v_x 就可以确定集群最大 v_n

在 v_n 生成过程中，必须达成通知到半数节点的目的， v_n 才能成功生成，所以一定有半数以上的节点知道 v_n



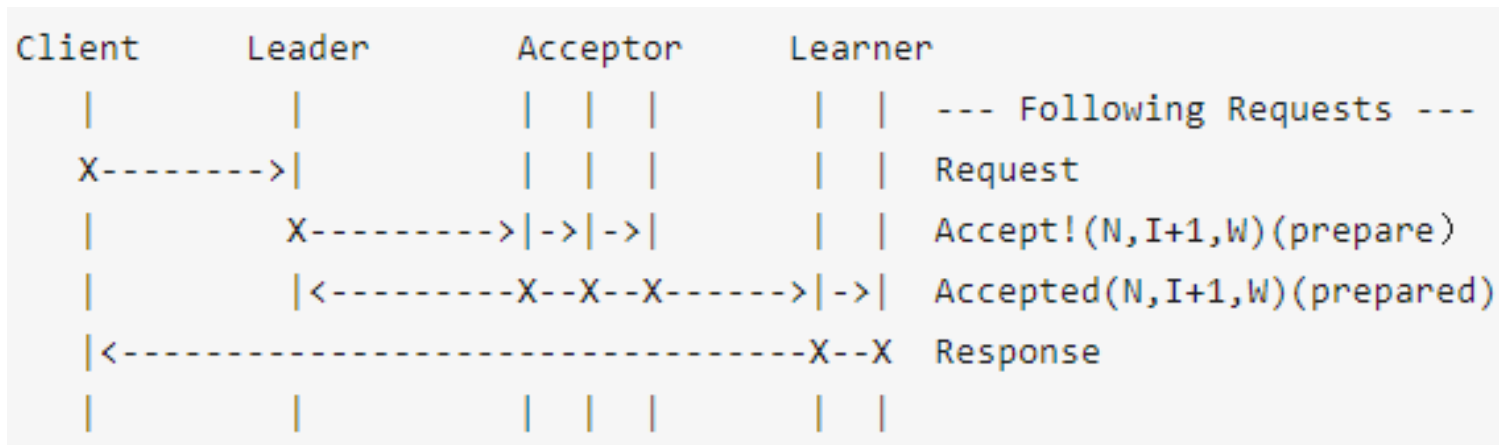
鸽笼原理可知大多数一定包含 v_n

Multi-Paxos(Algorithm)

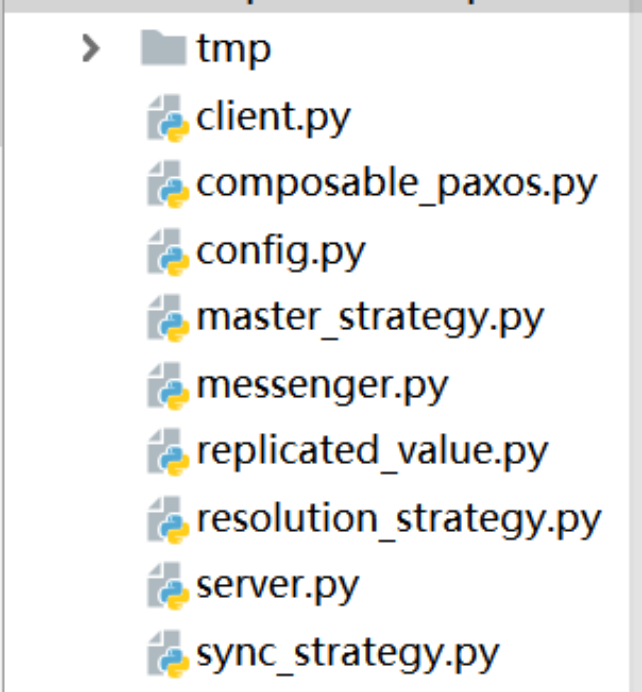
❑ 强leader状态

leader对vn的演变了如指掌，每次把vn的值直接在一阶段中发送给acceptor，和basic paxos的区别：basic paxos一阶段的时候，proposer对vn的值一无所知，要依赖一阶段的结果来算vn

❑ multi-paxos强leader状态的流程图：



paxos实现



```
> tmp
  client.py
  composable_paxos.py
  config.py
  master_strategy.py
  messenger.py
  replicated_value.py
  resolution_strategy.py
  server.py
  sync_strategy.py
```

主要有5大功能模块：

composable_paxos: paxos算法的核心逻辑模块

replicated_value: 日志更新维护模块

resolution_strategy: 解决多paxos实例之间冲突问题，保证paxos实例能顺利完成共识

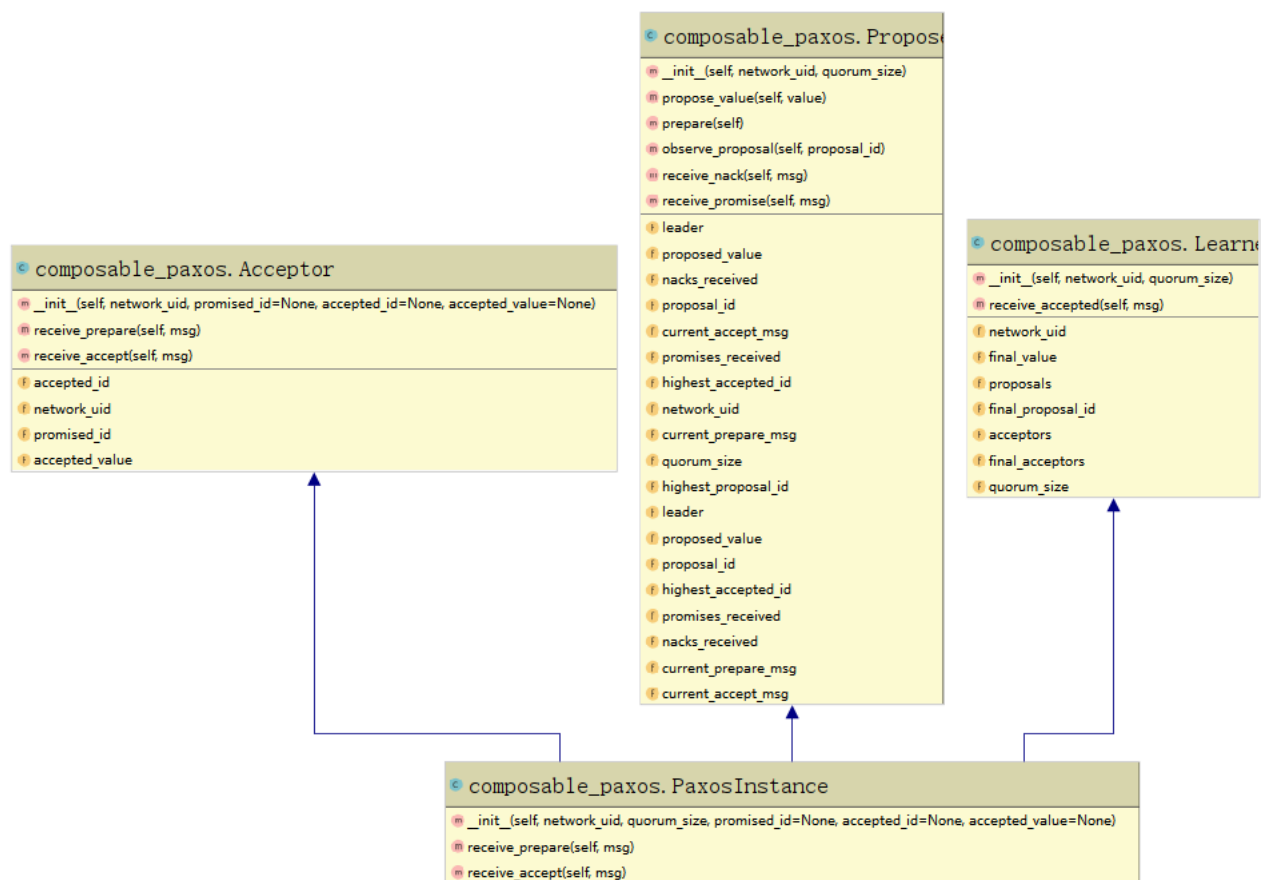
sync_strategy: 同步日志模块

master_strategy: 实现选主功能

paxos实现

composable_paxos: paxos算法的核心逻辑模块

一个paxosInstance要有3个组成部分, proposer、acceptor、learner



paxos实现

```
def receive_prepare(self, msg):  
    if msg.proposal_id >= self.promised_id:  
        self.promised_id = msg.proposal_id  
        return Promise(self.network_uid, msg.from_uid, self.promised_id,  
self.accepted_id, self.accepted_value)  
    else:  
        return Nack(self.network_uid, msg.from_uid, msg.proposal_id,  
self.promised_id)
```

paxos实现

```
def receive_promise(self, msg):  
  
    self.observe_proposal( msg.proposal_id )  
  
    if not self.leader and msg.proposal_id == self.proposal_id and  
msg.from_uid not in self.promises_received:  
  
        self.promises_received.add( msg.from_uid )  
  
        if msg.last_accepted_id > self.highest_accepted_id:  
            self.highest_accepted_id = msg.last_accepted_id  
            if msg.last_accepted_value is not None:  
                self.proposed_value = msg.last_accepted_value  
  
        if len(self.promises_received) == self.quorum_size:  
            self.leader = True  
  
        if self.proposed_value is not None:  
            self.current_accept_msg = Accept(self.network_uid,  
self.proposal_id, self.proposed_value)  
            return self.current_accept_msg
```

replicated_value: 日志更新维护模块

c replicated_value.BaseReplicatedValue

- m __init__(self, network_uid, peers, state_file)
- m set_messenger(self, messenger)
- m save_state(self, instance_number, current_value, promised_id, accepted_id, accepted_value)
- m load_state(self)
- m propose_update(self, new_value)
- m advance_instance(self, new_instance_number, new_current_value, catchup=False)
- m send_prepare(self, proposal_id)
- m send_accept(self, proposal_id, proposal_value)
- m send_accepted(self, proposal_id, proposal_value)
- m receive_prepare(self, from_uid, instance_number, proposal_id)
- m receive_nack(self, from_uid, instance_number, proposal_id, promised_proposal_id)
- m receive_promise(self, from_uid, instance_number, proposal_id, last_accepted_id, last_accepted_value)
- m receive_accept(self, from_uid, instance_number, proposal_id, proposal_value)
- m receive_accepted(self, from_uid, instance_number, proposal_id, proposal_value)

paxos实现

replicated_value: 日志更新维护模块

```
def save_state(self, instance_number, current_value, promised_id,
accepted_id, accepted_value):
```

```
def load_state(self):
```

通过这两个方法来将交互信息存储。

在发送**accept**和**promise**之前就要把数据先存储起来

日志数剧本地化, 持久化

```
def advance_instance(self, new_instance_number, new_current_value,
catchup=False):
```

```
def receive_accepted(self, from_uid, instance_number, proposal_id,
proposal_value):
```

```
self.advance_instance( self.instance_number + 1, proposal_value )
```

当前实例收到足够的**accepted**完成一致性认同之后, 通过**instance_number + 1**来产生下一个**instance**

paxos实现

resolution_strategy:处理多paxos实例之间冲突问题, 保证paxos实例能顺利完成共识

```
c resolution_strategy.ExponentialBackoffResolutionStrategyMixin  
m reschedule_next_drive_attempt(self, delay)  
m drive_to_resolution(self)  
m stop_driving(self)  
m advance_instance(self, new_instance_number, new_current_value, catchup=False)  
m propose_update(self, new_value)  
m send_accept(self, proposal_id, proposal_value)  
m receive_accept(self, from_uid, instance_number, proposal_id, proposal_value)  
m receive_nack(self, from_uid, instance_number, proposal_id, promised_proposal_id)
```

paxos实现

resolution_strategy:处理多paxos实例之间冲突问题，保证paxos实例能顺利完成共识

相比于**replicated_value**模块，这里的模块是多个实例并行运行，所以在重写**replicated_value**模块时要先判断相应的**instance_number**

def drive_to_resolution(self):

推进一个**instance**，通过发送一个**prepare**来启动一个**instance**

def receive_nack(self, from_uid, instance_number, proposal_id, promised_proposal_id):

在该方法中，为了解决多实例共同请求**proposal**造成了冲突问题，每次造成冲突的时候要将，**backoff_time**乘二，是指数退避法来避免碰撞。

master_strategy:实现选主功能

c	master_strategy.DedicatedMasterStrategyMixin
m	start_master_lease_timer(self)
m	update_lease(self, master_uid)
m	lease_expired(self)
m	propose_update(self, new_value, application_level=True)
m	load_state(self)
m	drive_to_resolution(self)
m	advance_instance(self, new_instance_number, new_current_value, catchup=False)
m	receive_prepare(self, from_uid, instance_number, proposal_id)
m	receive_accept(self, from_uid, instance_number, proposal_id, proposal_value)
f	_initial_load
f	lease_expiry
f	master_attempt
f	retransmit_task
f	master_uid
f	lease_start
f	lease_window
f	lease_start
f	lease_expiry
f	master_uid
f	master_attempt
f	_initial_load

master_strategy:实现选主功能

主节点有**10秒**的生命周期，其他节点可以探查主节点状况，在主节点挂掉后保证新的主节点

```
def update_lease(self, master_uid):
    self.master_uid = master_uid

    if self.network_uid != master_uid:
        self.start_master_lease_timer()

    if master_uid == self.network_uid:
        renew_delay = (self.lease_start + self.lease_window - 1) - time.time()

        if renew_delay > 0:
            reactor.callLater(renew_delay, lambda :
self.propose_update(self.network_uid, False))
        else:
            self.propose_update(self.network_uid, False)
```


paxos实现

master_strategy:实现选主功能

重写resolution_strategy的方法，考虑存在主节点时如何取得一致性

```
def drive_to_resolution(self):
    if self.master_uid == self.network_uid:
        self.stop_driving()
        if self.paxos.proposal_id.number == 1:
            self.send_accept(self.paxos.proposal_id, self.paxos.proposed_value)
        else:
            self.paxos.prepare()
            self.retransmit_task = task.LoopingCall( lambda :
self.send_prepare(self.paxos.proposal_id) )
            self.retransmit_task.start( self.retransmit_interval/1000.0,
now=False )
    else:
        super(DedicatedMasterStrategyMixin,self).drive_to_resolution()
```

paxos实现

sync_strategy:保证重新连接进网络的服务器同步最新的日志

```
def receive_catchup(self, from_uid, instance_number,
current_value):
    if instance_number > self.instance_number:
        print 'SYNCHRONIZED: ', instance_number, current_value
        self.advance_instance(instance_number, current_value,
catchup=True)
```

在接收到**catchup**之后应该怎么处理。

```
c sync_strategy.SimpleSynchronizationStrategyMixin
m set_messenger(self, messenger)
m receive_sync_request(self, from_uid, instance_number)
m receive_catchup(self, from_uid, instance_number, current_value)
```

Reference

- [1]. Lamport L. Paxos made simple[J]. ACM Sigact News, 2001, 32(4): 18-25.
- [2]. <https://github.com/cocagne/multi-paxos-example/>