

# Paxos



分布式研究小院

openACID

discover design distribute

可靠分布式系统基础：paxos的直观解释

# 背景

多个节点一起完成一件事情.

分布式中唯一的一个问题:对某事达成一致.

Paxos: 分布式系统的核心算法.

# 目录

1. 问题

2. 复制策略

3. Paxos 算法

4. Paxos 优化

# 问题

对系统的需求:

持久性要达到: 99.99999999%

我们可以用的基础设置:

磁盘: 4% 年损坏率

服务器宕机时间: 0.1% 或更长

IDC间丢包率: 5% ~ 30%

# 解决方案(可能)

## 多副本

$x < n$  个副本损坏不会丢数据

## 多副本的数据丢失风险:

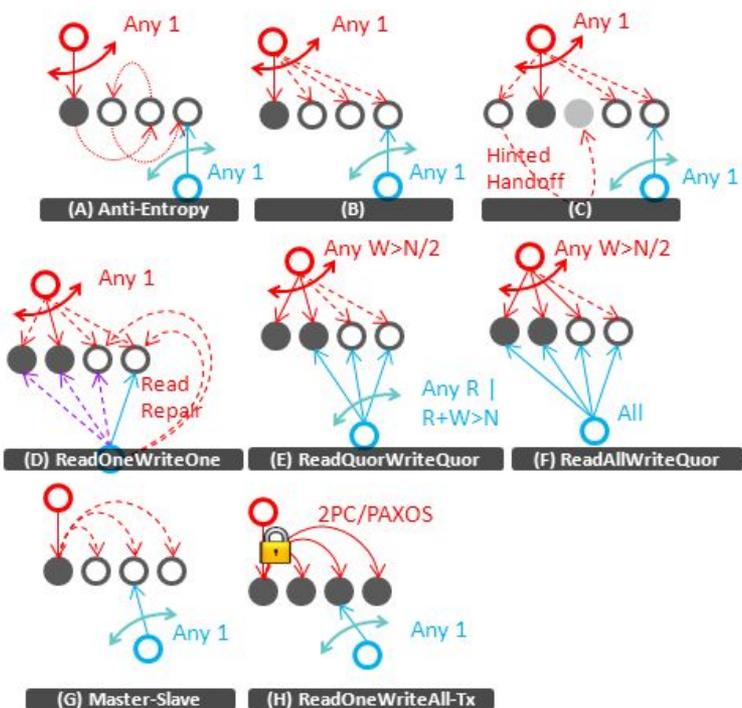
1 副本:  $\sim 0.63\%$

2 副本:  $\sim 0.00395\%$

3 副本:  $< 0.000001\%$

$n$  副本:  $\sim x^n$  /\*  $x$  = 单副本损坏率 \*/

# 解决方案.

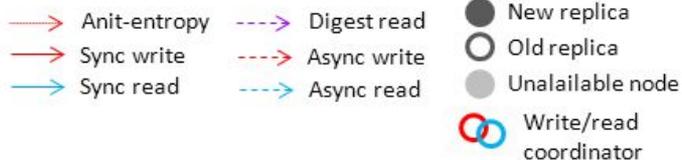


## 如何实施‘复制’?

除了副本数之外, 还有:

可用性  
原子性  
一致性

...



# 基础的复制算法

主从异步复制

主从同步复制

主从半同步复制

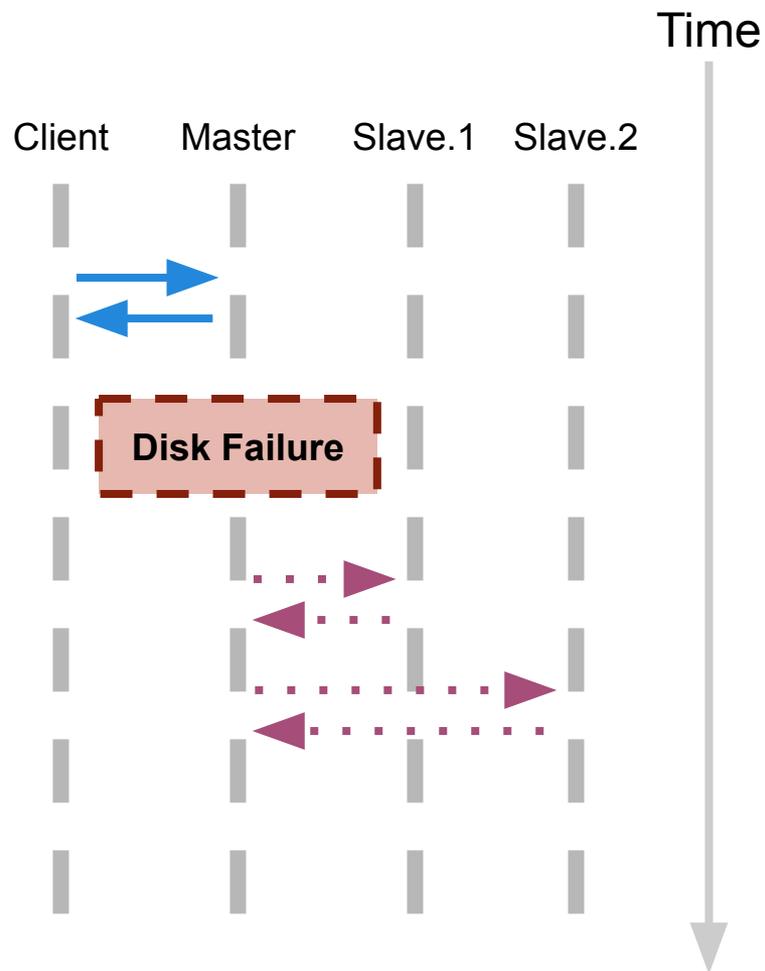
多数派写(读)

# 主从异步复制

如Mysql的binlog复制.

1. 主接到写请求.
2. 主写入本磁盘.
3. 主应答'OK'.
4. 主复制数据到从库.

如果磁盘在复制前损坏：  
→ 数据丢失.



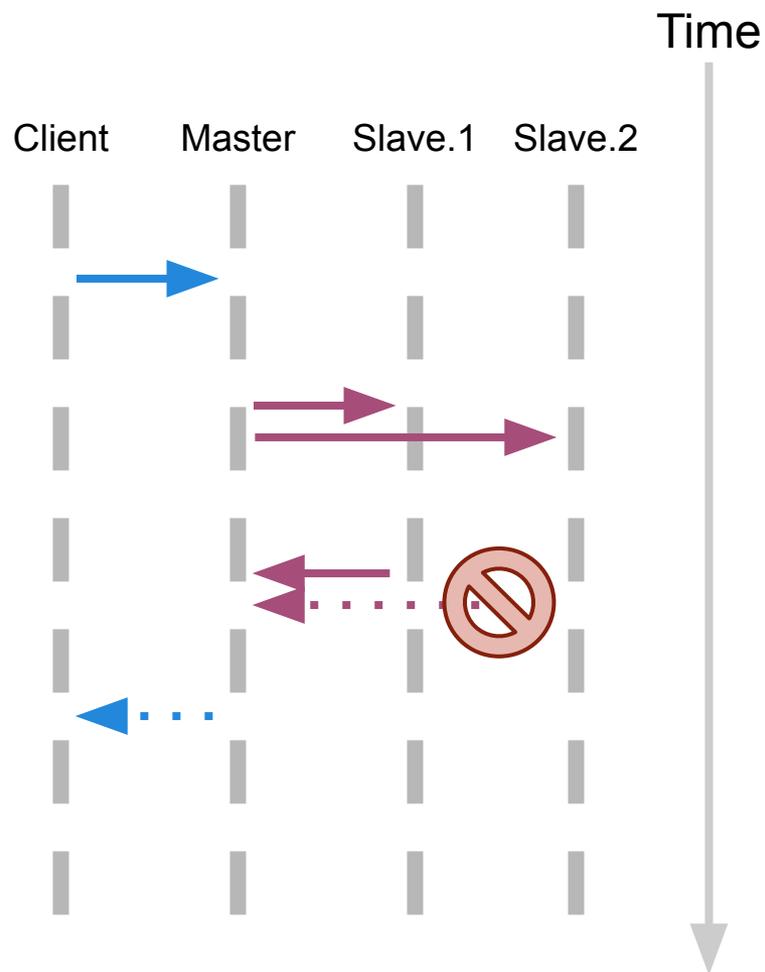
# 主从同步复制

1. 主接到写请求.
2. 主复制日志到从库.
3. 从库这时可能阻塞...
4. 客户端一直在等应答'OK', 直到所有从库返回.

一个失联节点造成整个系统不可用.

: 没有数据丢失.

: 可用性降低.



# 主从半同步复制

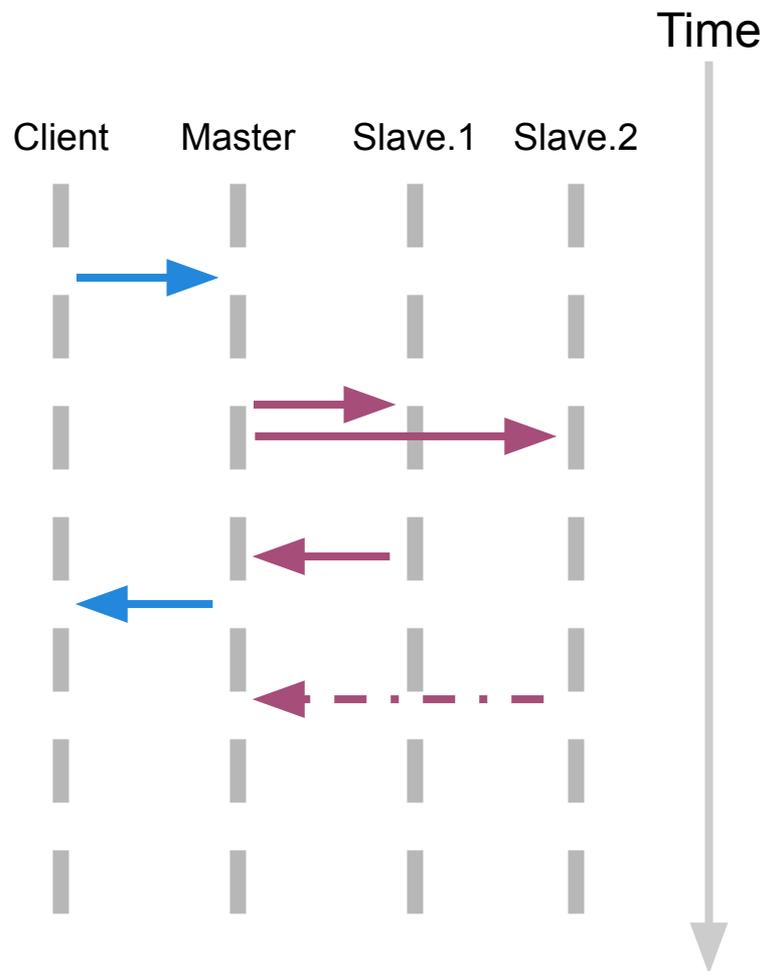
1. 主接到写请求.
2. 主复制日志到从库.
3. 从库这时可能阻塞...
4. 如果  $1 \leq x \leq n$  个从库返回‘OK’,  
则返回客户端‘OK’.

: 高可靠性.

: 高可用性.

: 可能任何从库都不完整

→ 我们需要 多数派写(读)



# 多数派写

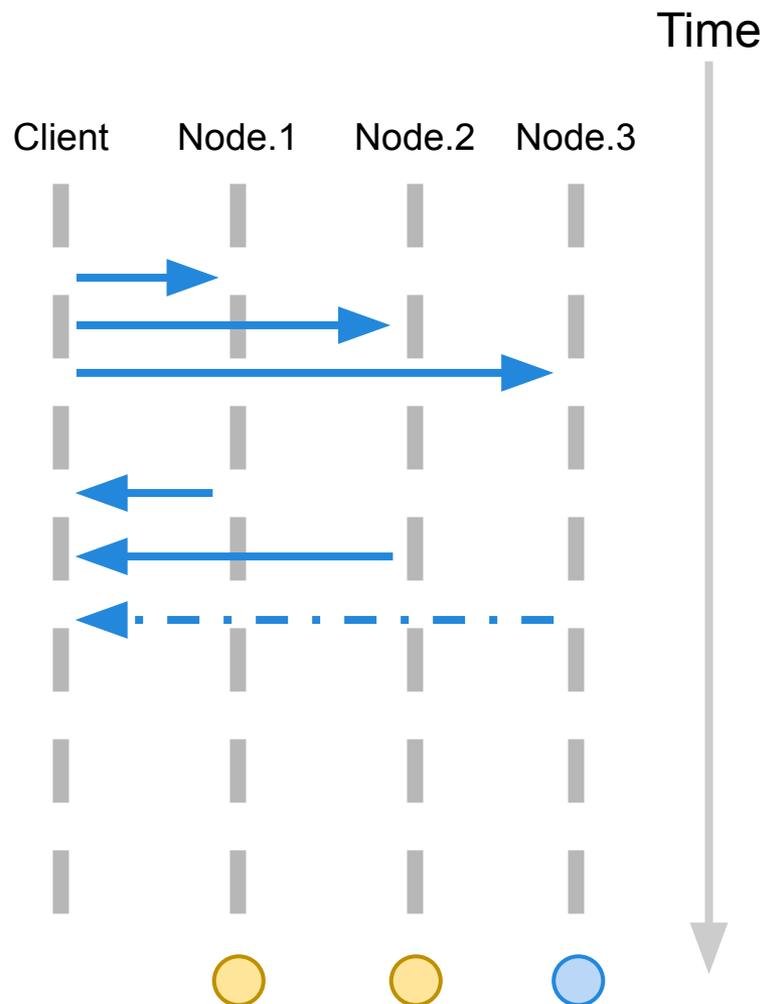
## Dynamo / Cassandra

客户端写入  $W \geq N/2 + 1$  个节点.  
不需要主.

多数派读:

$W + R > N$ ;  $R \geq N/2 + 1$

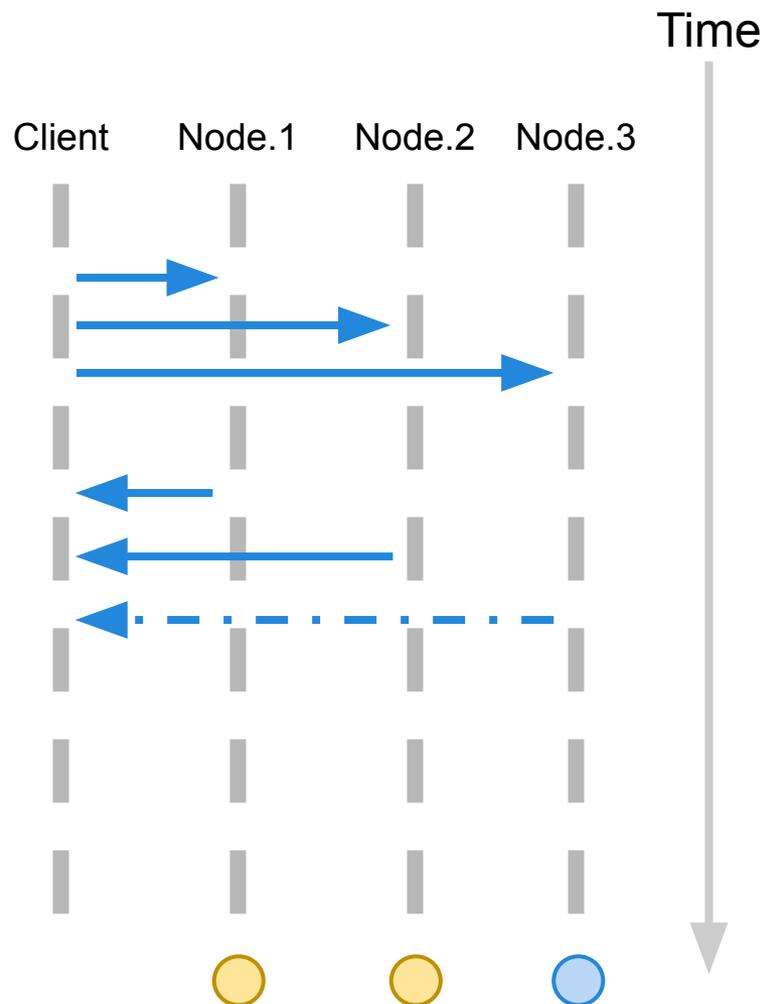
容忍最多  $(N-1)/2$  个节点损坏.



# 多数派写. 后写入优胜

最后1次写入覆盖先前写入.

所有写入操作需要有1个全局顺序: 时间戳



# 多数派写..

- : 高可靠性.
- : 高可用性.
- : 数据完整性有保证.

够了吗？

# 多数派写... $W + R > N$

一致性:

 最终一致性

事务性:

 非原子更新

 脏读

 更新丢失问题

[http://en.wikipedia.org/wiki/Concurrency\\_control](http://en.wikipedia.org/wiki/Concurrency_control)

# 一个假想存储服务

- 一个有3个存储节点的存储服务集群.
- 使用多数派读写的策略.
- 只存储1个变量“i”.
- “i”的每次更新对应有多多个版本: i1, i2, i3...
- 这个存储系统支持3个命令:

```
get      /* 读最新的 “i” */
```

```
set <n>  /* 设置下个版本的i的值为 <n> */
```

```
inc <n>  /* 对“i”加<n>, 也生成1个新版本 */
```

我们将用这个存储系统来演示多数派读写策略的不足, 以及如何用paxos解决这些问题.

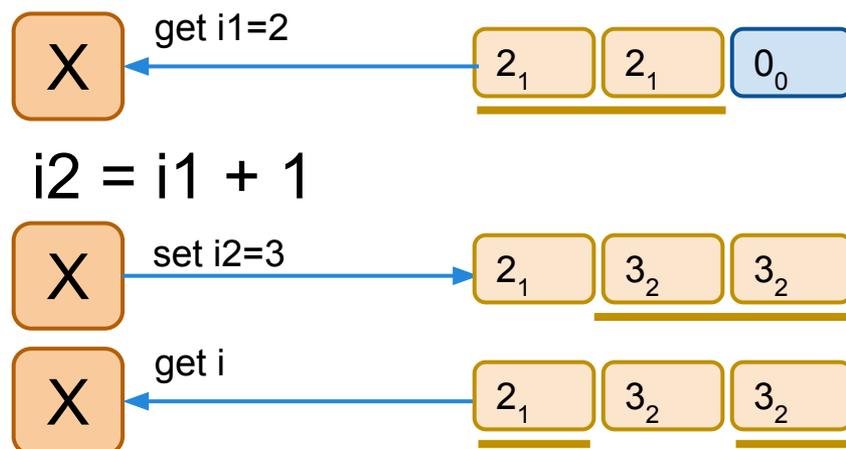
# 一个假想存储服务.实现

命令实现:

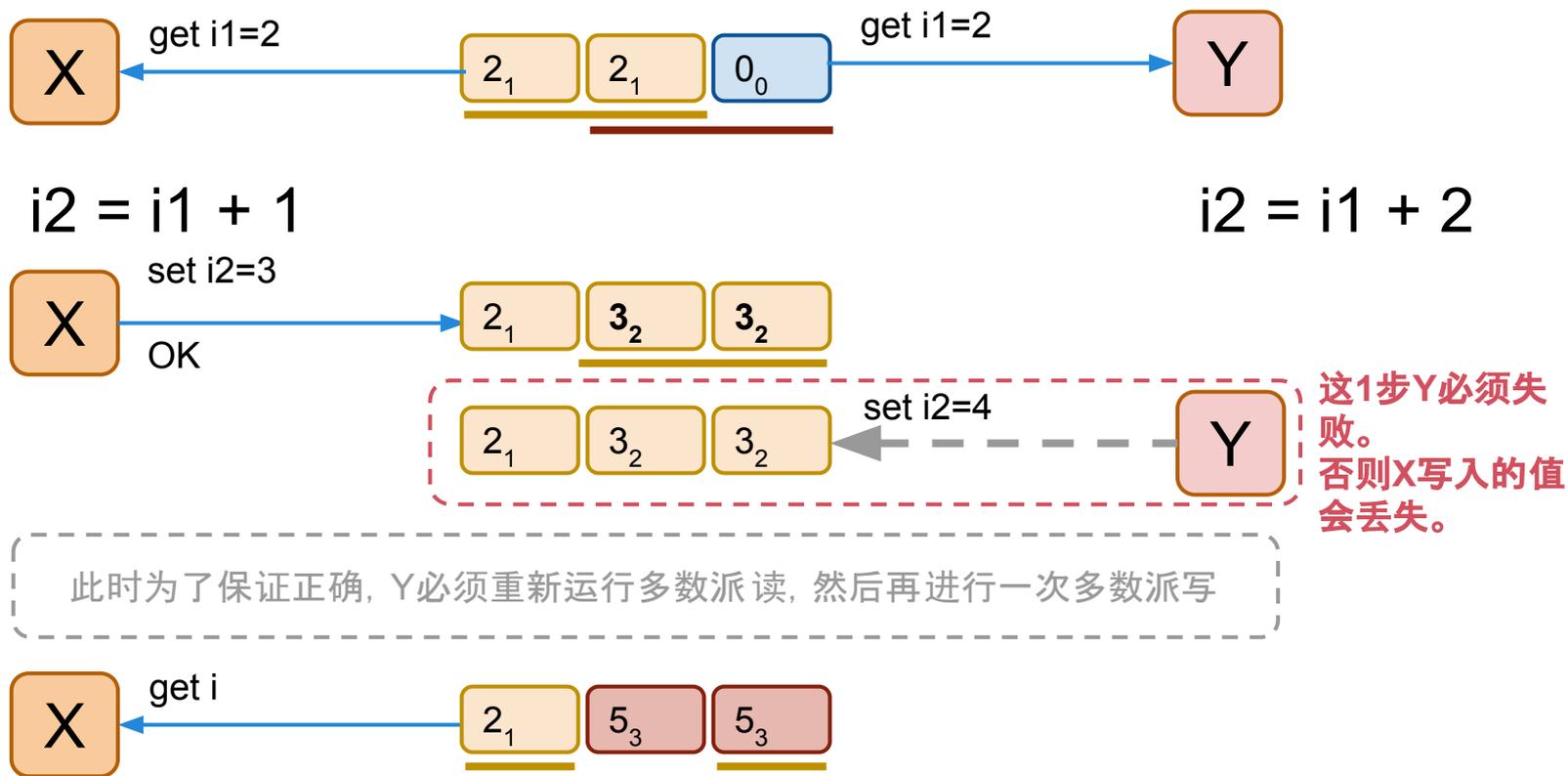
"set" → 直接对应多数派写.

"inc" → (最简单的事务型操作):

1. 通过多数派读, 读取最新的 "i":  $i_1$
2. Let  $i_2 = i_1 + n$
3. set  $i_2$



# 一个假想存储服务..并发问题



我们期待最终X可以读到 $i3=5$ ,  
这需要Y能知道X已经写入了 $i2$ 。如何实现这个机制？

# 一个假想存储服务...

在X和Y的2次“inc”操作后, 为了得到正确的i3:

整个系统里对i的某个版本(i2), 只能有1次成功写入.

推广为:

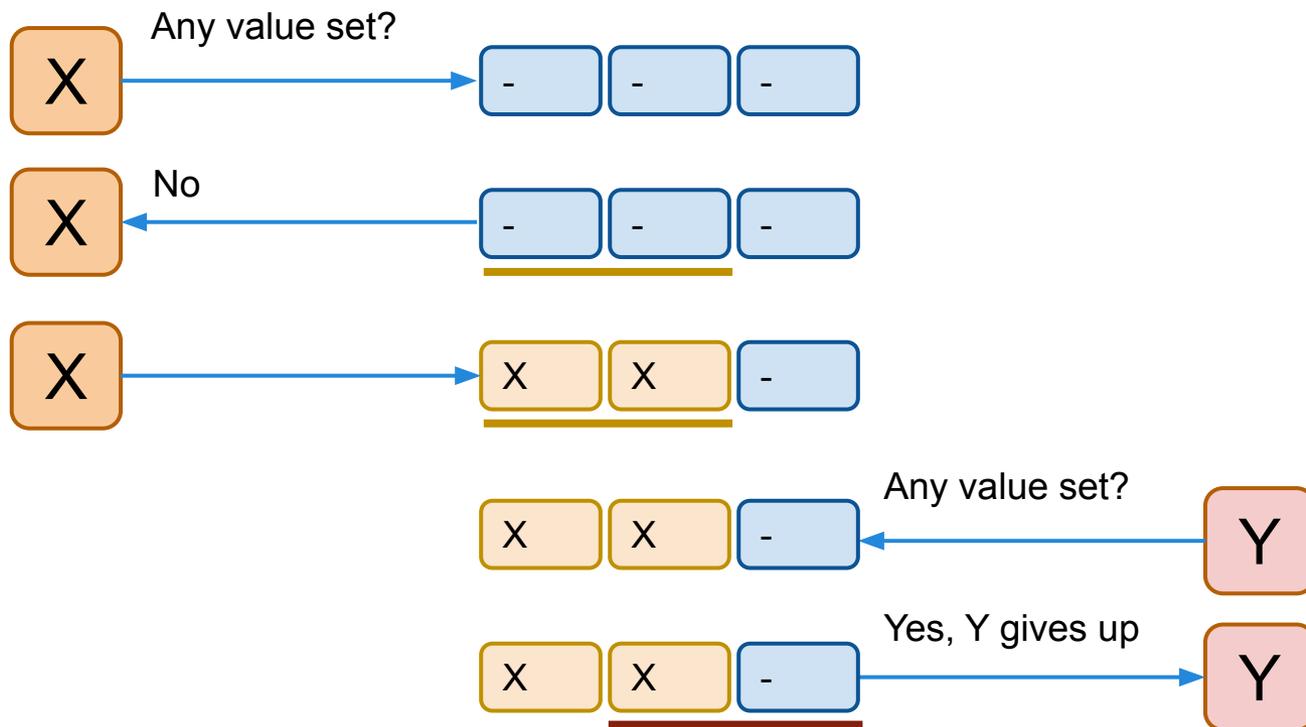
在存储系统中, 一个值(1个变量的1个版本)在被认为确定(客户端接到OK)之后, 就不允许被修改().

如何定义“被确定的”?

如何避免修改“被确定的”值?

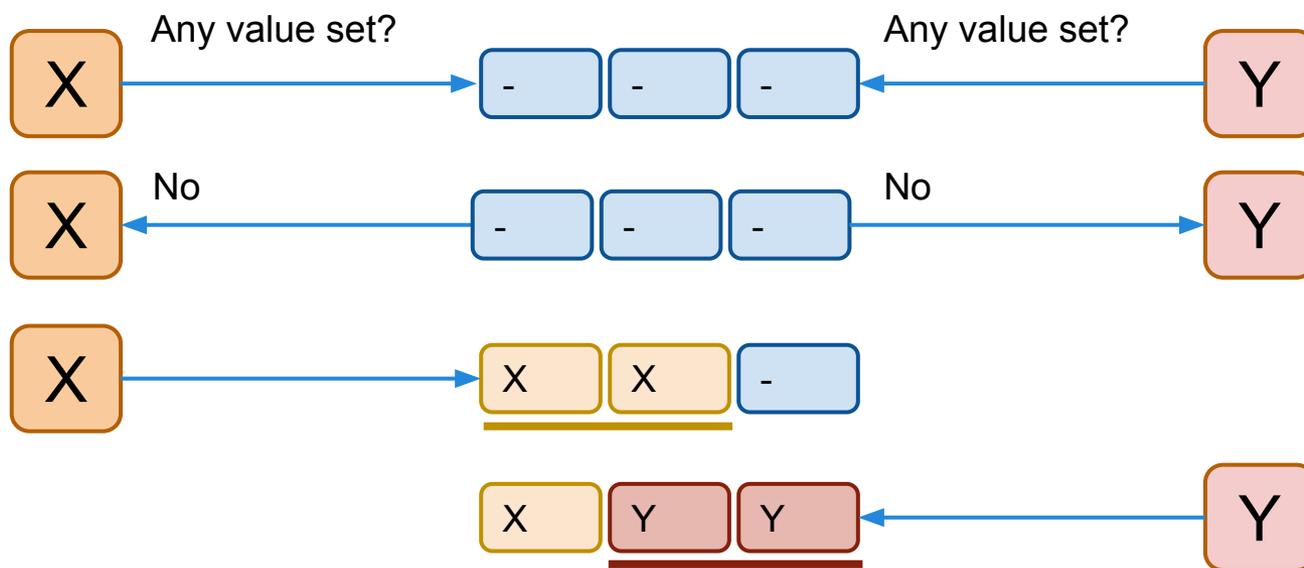
# 如何确定一个值

方案: 每次写入一个值前, 先运行一次多数派读, 来确认是否这个值(可能)已经被写过了.



# 如何确定一个值.并发问题

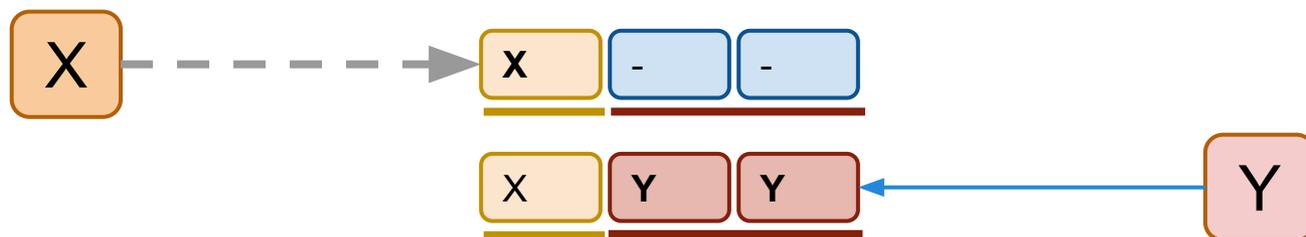
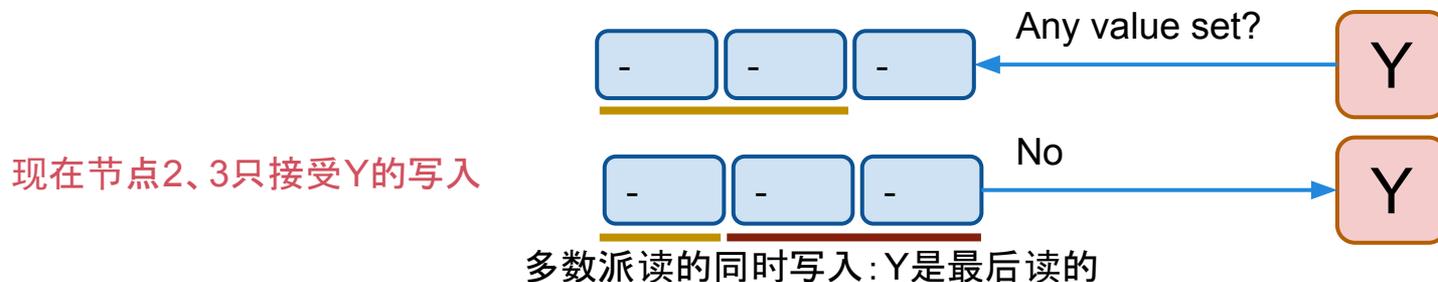
但是, X和Y可能同时以为还没有值被写入过, 然后同时开始写。



更新丢失

# 如何确定一个值..

方案改进:让存储节点记住谁最后1次做过“写前读取”,并拒绝之前其他的“写前读取”的写入操作。



# 如何确定一个值...

使用这个策略, 一个值( $i$ 的每个版本)可以被安全的存储.

Leslie Lamport 写了个这个算法的paper.

# Paxos



**I WANT YOU**

# Paxos是什么

- 一个可靠的存储系统: 基于多数派读写.
- 每个paxos实例用来存储一个值.
- 用2轮RPC来确定一个值.
- 一个值‘确定’后不能被修改.
- ‘确定’指被多数派接受写入.
- 强一致性.

# Paxos

## Classic Paxos

1个实例(确定1个值)写入需要2轮RPC.

## Multi Paxos

约为1轮RPC, 确定1个值(第1次RPC做了合并).

## Fast Paxos

没冲突: 1轮RPC确定一个值.

有冲突: 2轮RPC确定一个值.

# Paxos: 执行的条件

存储必须是可靠的:

没有数据丢失和错误

/\* 否则需要用Byzantine Paxos \*/

容忍:

消息丢失(节点不可达)

消息乱序

# Paxos: 概念

**Proposer:** 发起paxos的进程.

**Acceptor:** 存储节点, 接受、处理和存储消息.

**Quorum(Acceptor的多数派):**  $n/2+1$  个Acceptors.

**Round:** 1轮包含2个阶段: Phase-1 & Phase-2

每1轮的编号 (rnd):

单调递增; 后写胜出; 全局唯一(用于区分Proposer);

# Paxos: 概念.

Acceptor看到的最大rnd (**last\_rnd**):

Acceptor记住这个值来识别哪个proposer可以写。

Value (**v**): Acceptor接受的值.

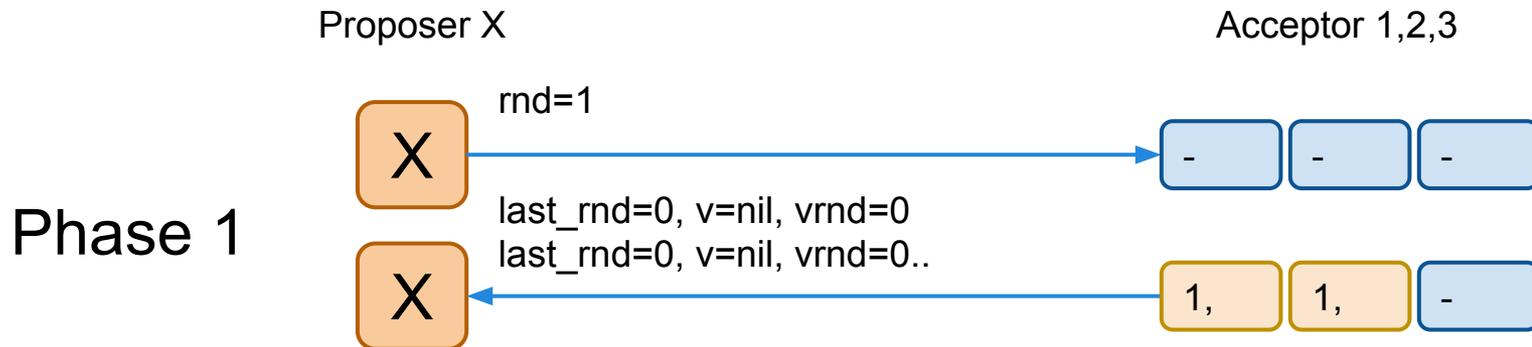
Value round number (**vrnd**):

Acceptor接受的v的时候的rnd

值‘被确定的’定义:

有多数(多于半数)个Acceptor接受了这个值.

# Paxos: Classic - phase 1



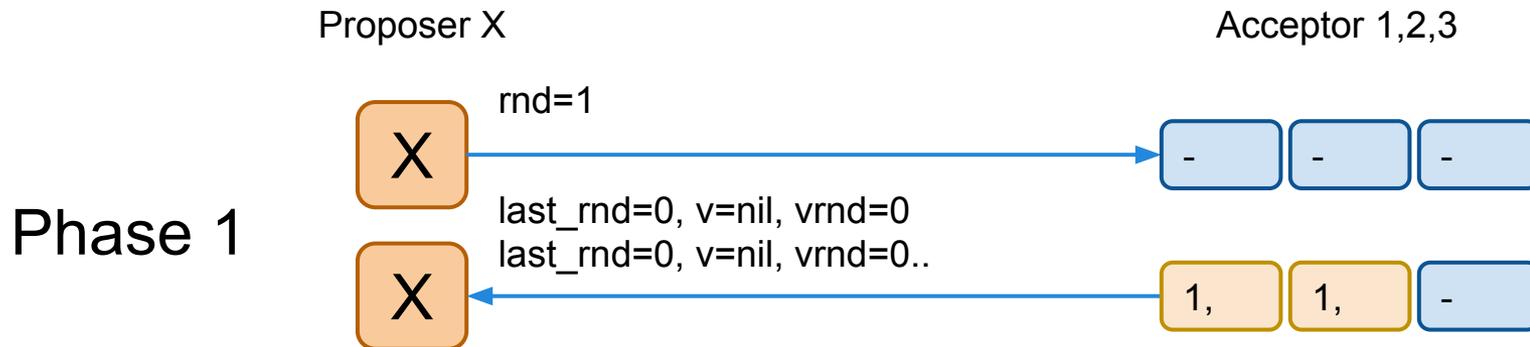
当Acceptor收到phase-1的请求时：

- 如果请求中**rnd**比Acceptor的**last\_rnd**小，则拒绝请求
- 将请求中的**rnd**保存到本地的**last\_rnd**.

从此这个Acceptor只接受带有这个**last\_rnd**的**phase-2**请求。

- 返回应答，带上自己之前的**last\_rnd**和之前已接受的**v**.

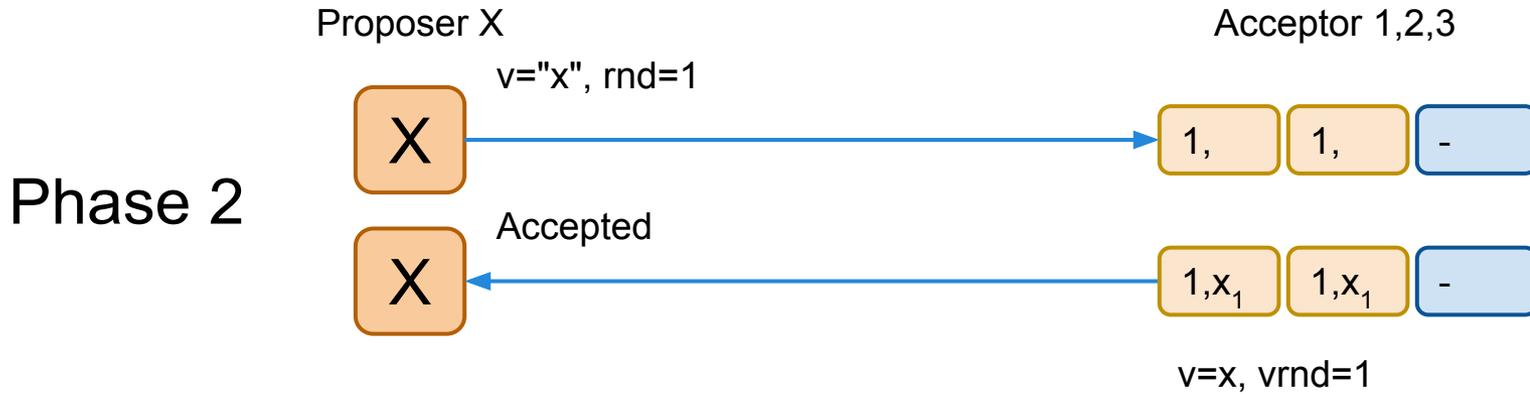
# Paxos: Classic - phase 1.



当Proposer收到Acceptor发回的应答：

- 如果应答中的`last_rnd`大于发出的`rnd`: 退出.
- 从所有应答中选择`vrnd`最大的`v`:  
不能改变(可能)已经确定的值
- 如果所有应答的`v`都是空, 可以选择自己要写入`v`.
- 如果应答不够多数派, 退出

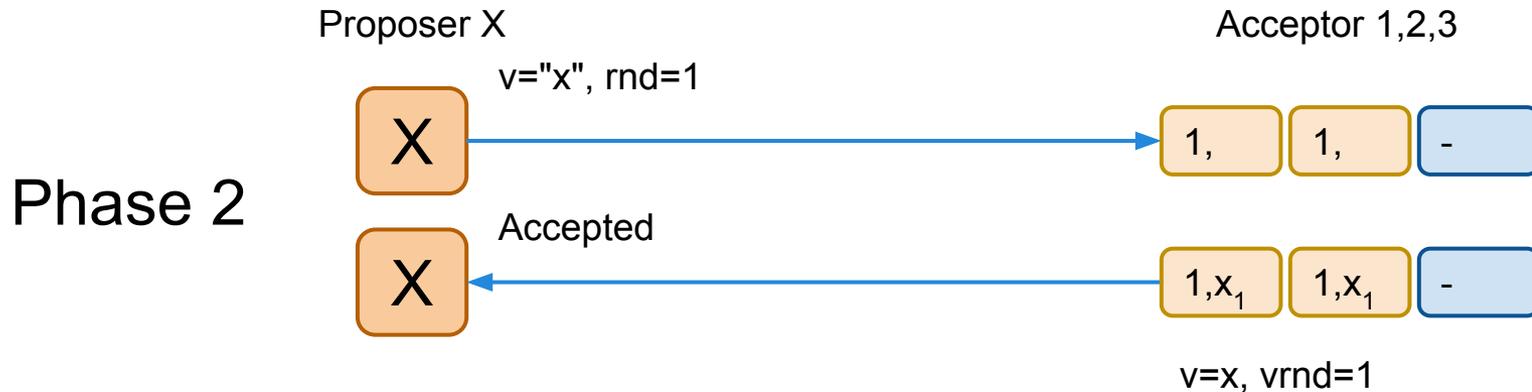
# Paxos: Classic - phase 2



Proposer:

发送phase-2, 带上rnd和上一步决定的v

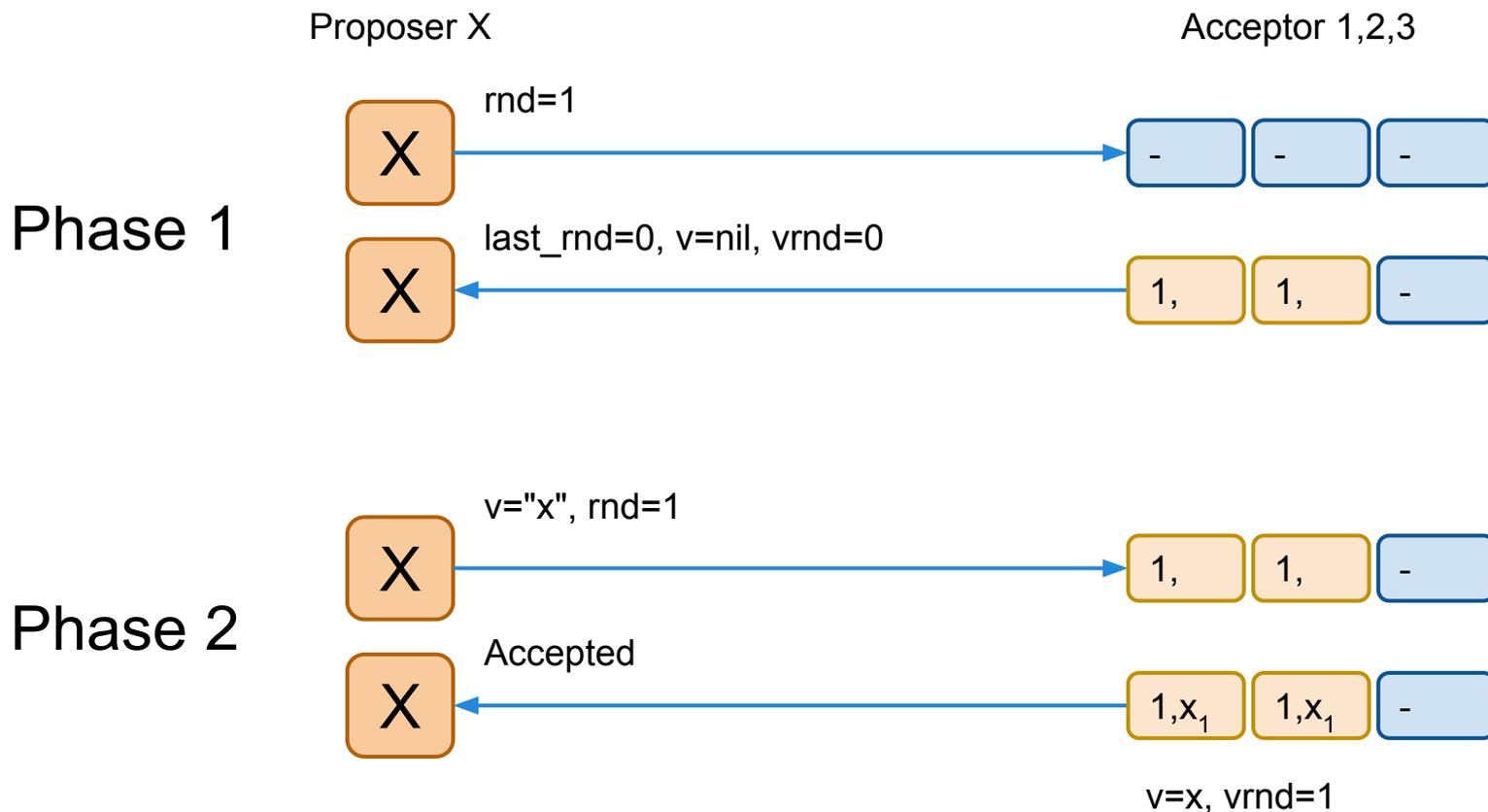
# Paxos: Classic - phase 2.



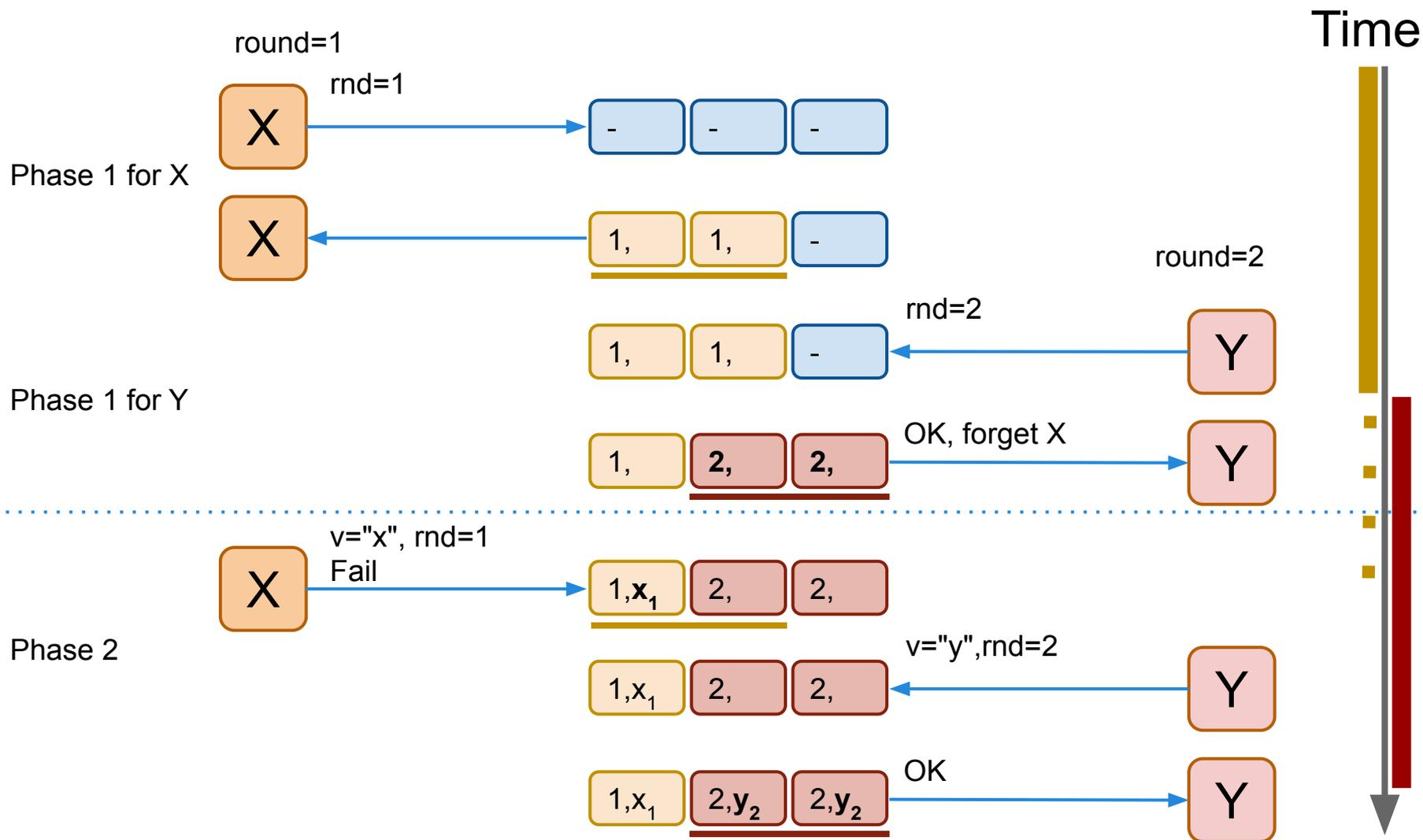
## Acceptor:

- 拒绝rnd不等于Acceptor的last\_rnd的请求
- 将phase-2请求中的v写入本地, 记此v为‘已接受的值’
- last\_rnd==rnd 保证没有其他Proposer在此过程中写入过其他值

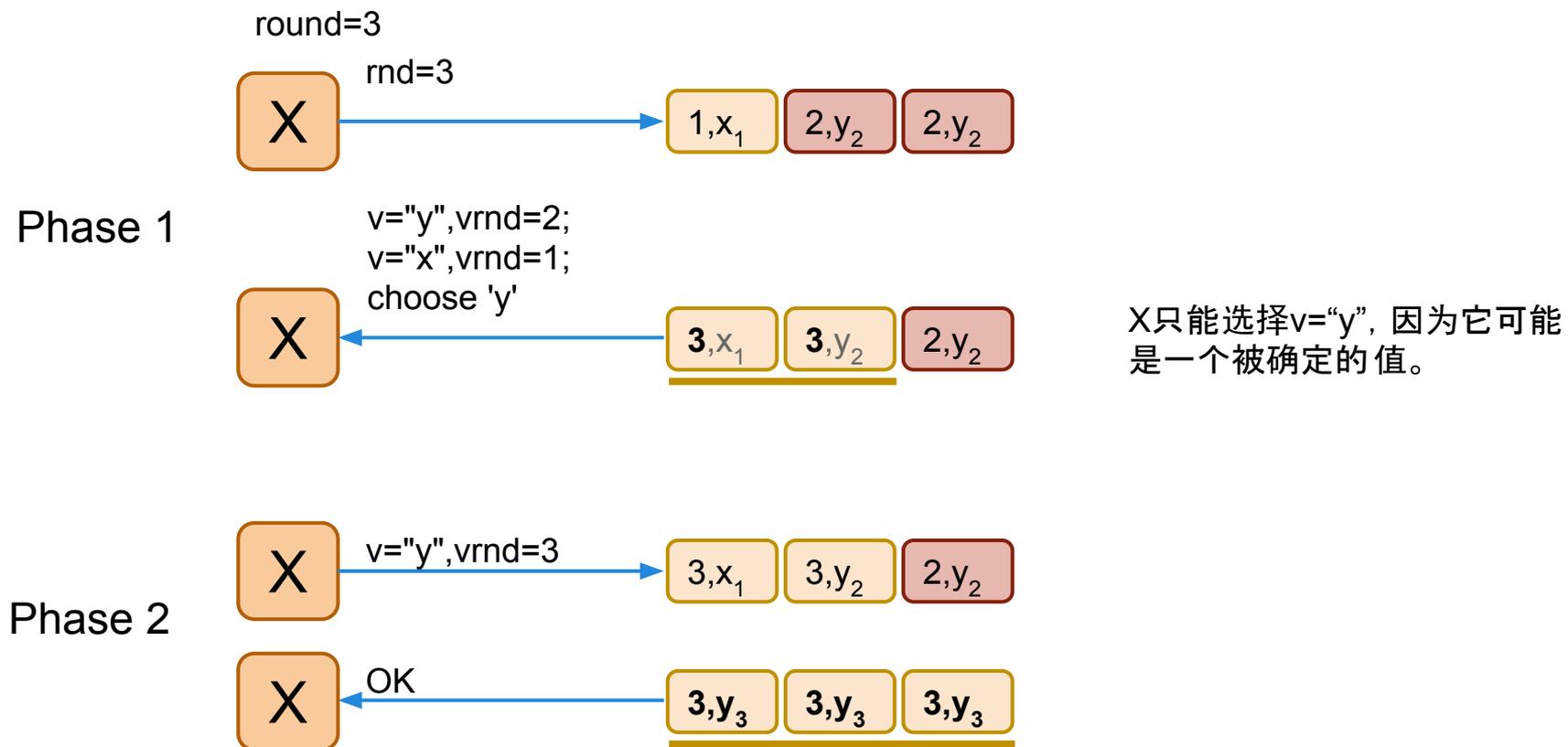
# Paxos: 栗子 1: Classic, 无冲突



# Paxos: 栗子 2.1: 解决并发写冲突



# Paxos: 栗子 2.2: X不会修改确定的v



# Paxos..其他

Learner角色:

- Acceptor发送**phase-3** 到所有learner角色, 让learner知道一个值被确定了.
- 多数场合Proposer就是1个Learner.

Livelock:

多个Proposer并发对1个值运行paxos的时候, 可能会互相覆盖对方的**rnd**, 然后提升自己的rnd再次尝试, 然后再次产生冲突, 一直无法完成

# Multi Paxos

将多个paxos实例的phase-1合并到1个RPC;  
使得这些paxos只需要运行phase-2即可。

应用:

chubby zookeeper megastore spanner

# Fast Paxos

- Proposer直接发送**phase-2**.
- Fast Paxos的**rnd**是0.

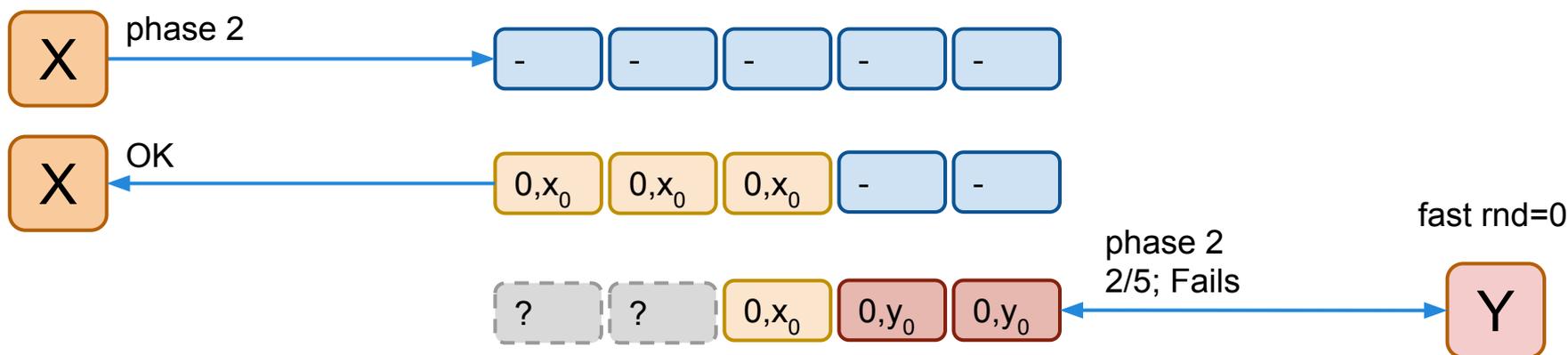
0保证它一定小于任何一个Classic **rnd** , 所以可以在出现冲突时安全的回退到Classic Paxos.

- Acceptor只在**v**是空时才接受Fast **phase-2**请求
- 如果发成冲突, 回退到Classic Paxos, 开始用一个 **rnd** > 0来运行。

但是Fast Paxos 比Classic Paxos高效吗？

# Fast Paxos 的多数派

fast rnd=0



如果Fast的多数派也是  $n/2+1 = 3$ :

当上图中Y发现冲突, 回退到Classic的时候:

Y无法确定哪个值是被确定下来的:  $x_0$  or  $y_0$

解决方法是让未确定的值不占据  $n/2+1$  个节点中的多数派, 因此:

→ Fast 的多数派必须  $> n^{3/4}$ ;

→ Fast Paxos里的值被确定的条件是被  $n^{3/4}+1$  个Acceptor接受.

# Fast Paxos 的多数派.

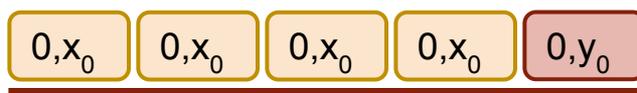
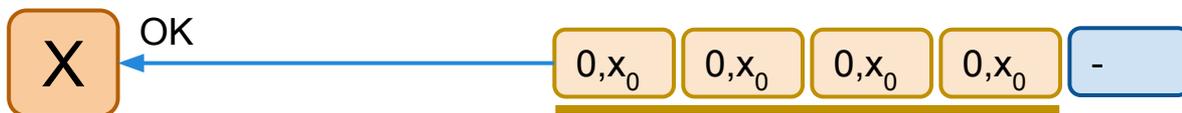
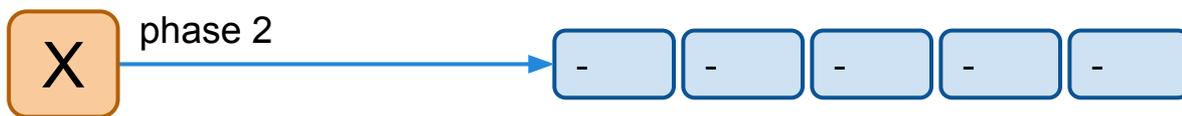
$$Q = n^{*3/4}$$

可用性降低, 因为Fast Paxos需要更多的Acceptor来工作.

Fast Paxos 需要至少5个Acceptors, 才能容忍1个Acceptor不可用.

# Fast Paxos 4/5 Y 发现冲突

fast rnd=0



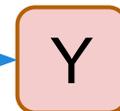
Y在3个Acceptor中看到2个 $x_0$

因此Y必须选择  $x_0$ , 因为  $x_0$ 可能是1个被确定的值.

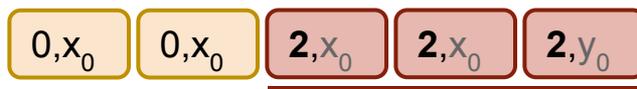
$y_0$  不可能是1个被确定的值, 因为即使剩下的2个Y没有联系到的Acceptor都接受了 $y_0$ , 也不会达到 $(5 \times \frac{3}{4})$  的多数派。

phase 2  
1/5; Fail

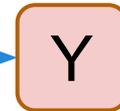
fast rnd=0



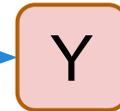
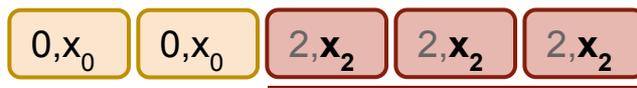
classic rnd=2



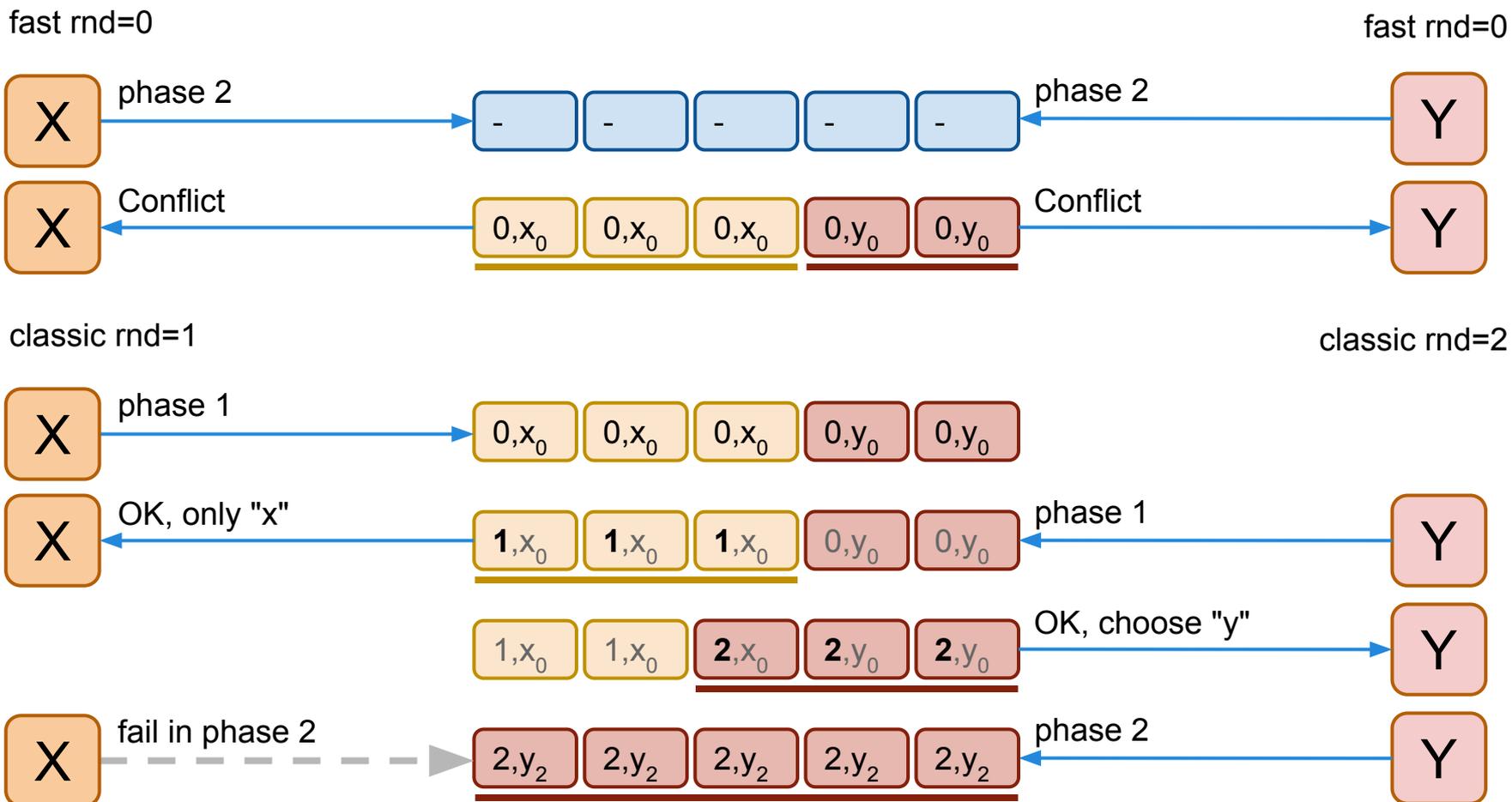
phase 1  
OK, "x"



phase 2  
OK, writes "x"



# Fast Paxos 4/5 X Y 都冲突



# Note

在 **phase-2**, Acceptor可以接受  $\text{rnd} \geq \text{last\_rnd}$  的请求

# Q&A

## Thanks

[drdr.xp@gmail.com](mailto:drdr.xp@gmail.com)

<http://drmingdrmer.github.io>

<https://blog.openacid.com/>

weibo.com: @drdrxp



分布式研究小院

openACID

discover design distribute