议题

Paxos 介绍
Paxos 应用场景
Classical Paxos vs Fast Paxos
Paxos-based MDCC
Q&A

什么是 Paxos？

Paxos算法是Leslie Lamport于1990年提出的一种基于消息传递且具有高度容错特性的一致性算法。

Lamport 虚构一个叫做Paxos的希腊城邦，这个岛按照议会民主制的政治模式制订法律，但是没有人愿意将自己的全部时间和精力放在这种事情上。所以无论是议员，议长或者传递纸条的服务员都不能承诺别人需要时一定会出现，也无法承诺批准决议或者传递消息的时间。但是这里假设没有拜占庭问题（Byzantine failure，即虽然有可能一个消息被传递了两次，但是绝对不会出现错误的消息）；只要等待足够的时间，消息就会被传到。另外，Paxos岛上的议员是不会反对其他议员提出的决议的。

对应于分布式系统，议员对应于各个节点，制定的法律对应于系统的状态。各个节点需要进入一个一致的状态，例如在独立Cache的SMP系统中，各个处理器读内存的某个字节时，必须读到同样的一个值，否则系统就违背了一致性的要求。一致性要求对应于法律条文只能有一个版本。议员和服务员的不确定性对应于节点和消息传递通道的不可靠性。

拜占庭将军问题（Byzantine failures）又称两军问题，是由莱斯利·兰伯特提出的点对点通信中的基本问题。含义是在存在消息丢失的不可靠信道上试图通过消息传递的方式达到一致性是不可能的。因此对一致性的研究一般假设信道是可靠的，或不存在本问题。

Paxos基本概念

Proposer: 写操作发起者。

Acceptor: 存储节点，接受写入； n = 2f + 1。

Quorum(of acceptors) : Acceptor中的多数派。

Round：Paxos 的1次运行，至少包括2个phase：Phase 1 & Phase 2。

Round Number (rnd): 每个Round的唯一标识；单调升；Last-Win。

Value (v): Acceptor 已经接受的值。

Value round number (vrnd): Acceptor接受的value的rnd

最终确定的值: 某个Value被多数(Quorum)个Acceptor 接受才认为Paxos系统确定了这个值

Paxos算法过程

Paxos算法包括2个阶段：

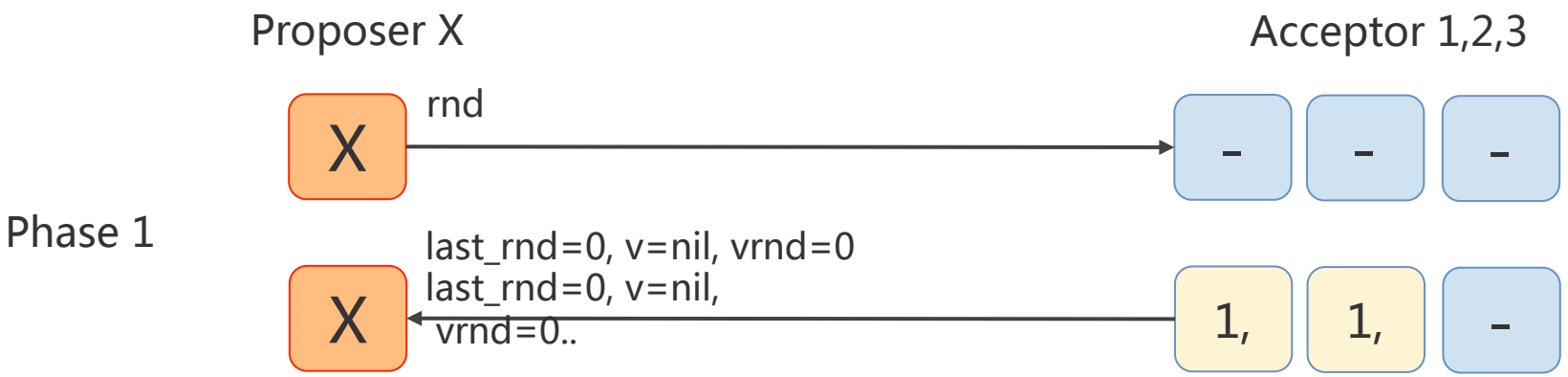
Phase1：prepare

a. Proposer选择一个proposal编号n，发送给acceptor的一个majority。
b. 如果acceptor发现n是它已回复的请求中编号最大的，则会回复它已经接受最大的proposal和对应的value 值（如果有）；同时不会批准编号小于n的proposal。

Phase2：accept

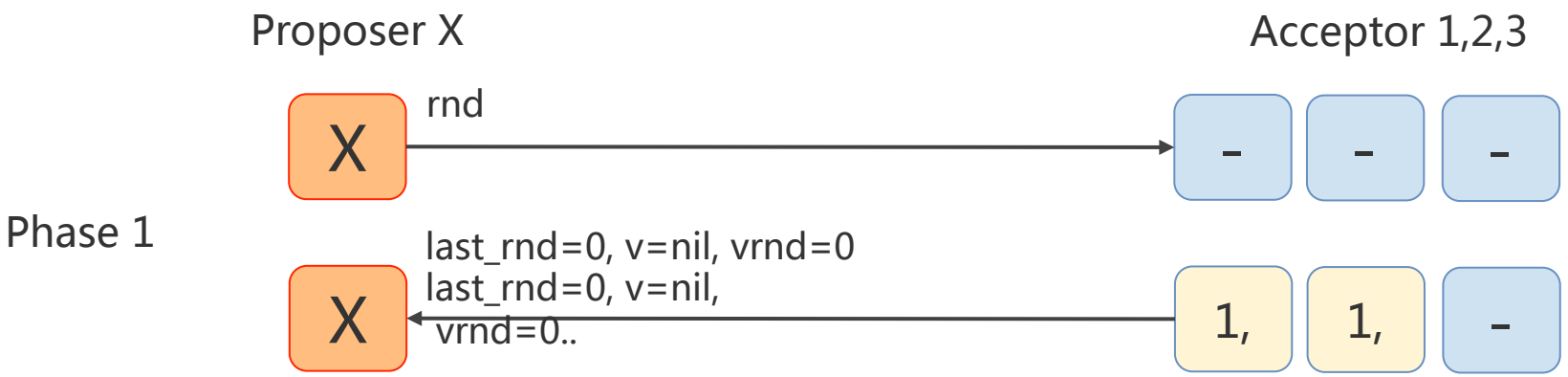
a. 如果proposer接收到了majority的回应，它发送一个accept消息到acceptor的majority（可以与 prepare的majority不同）。
b. Acceptor接收到accept消息后check，如果没有比n大的回应，则接受对应的value，否则拒绝或不回应。

Paxos算法流程图



Acceptor:
记录Proposer发来的rnd，表示可以接受这个round的phase 2请求
只接受大于last_rnd的rnd

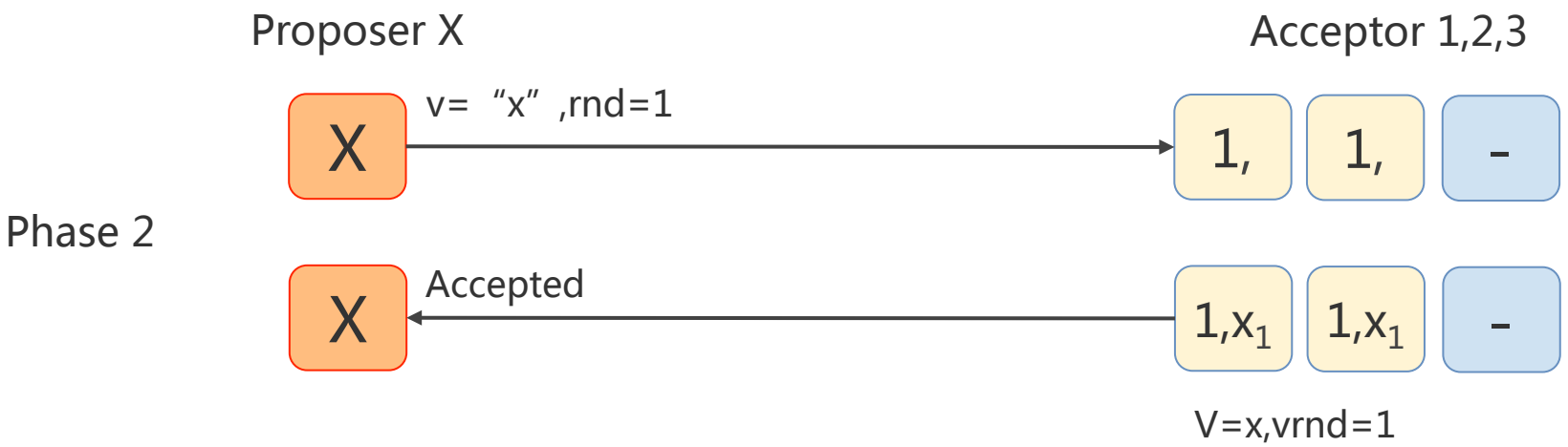
Paxos算法流程图



Proposer:
检查每个Acceptor返回的last_rnd，如果last_rnd比自己的rnd更大，则放弃此轮round；

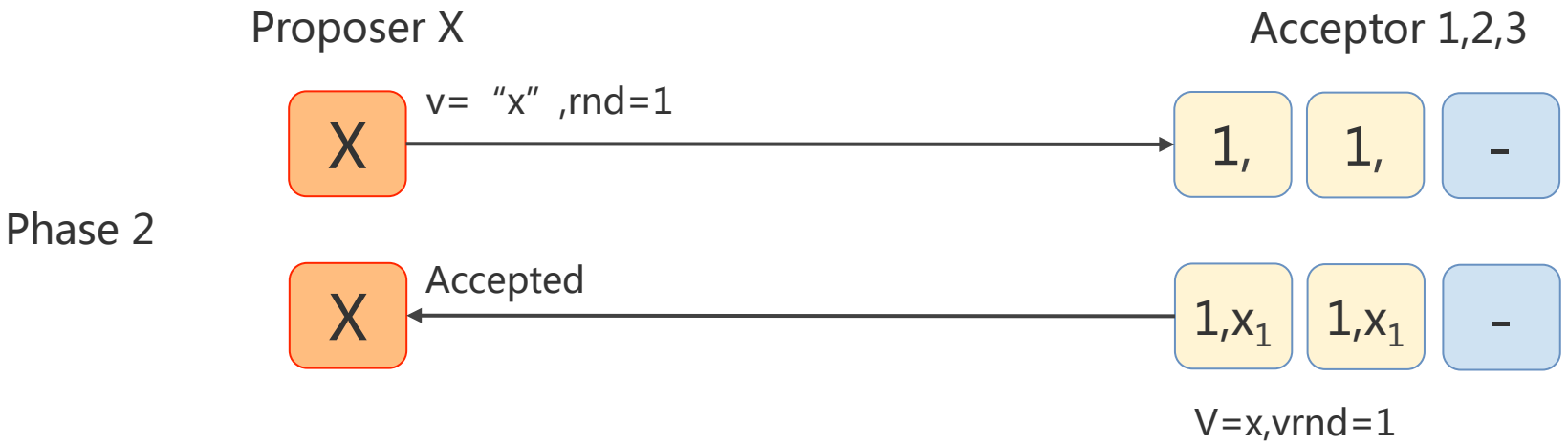
检查Acceptor返回的所有v和vrnd，如果所有v都是空，Proposer可以任意决定Phase 2写入的值；否则选择最大vrnd的v。

Paxos算法流程图



Proposer:
向Acceptor Quorum写入自己决定值v

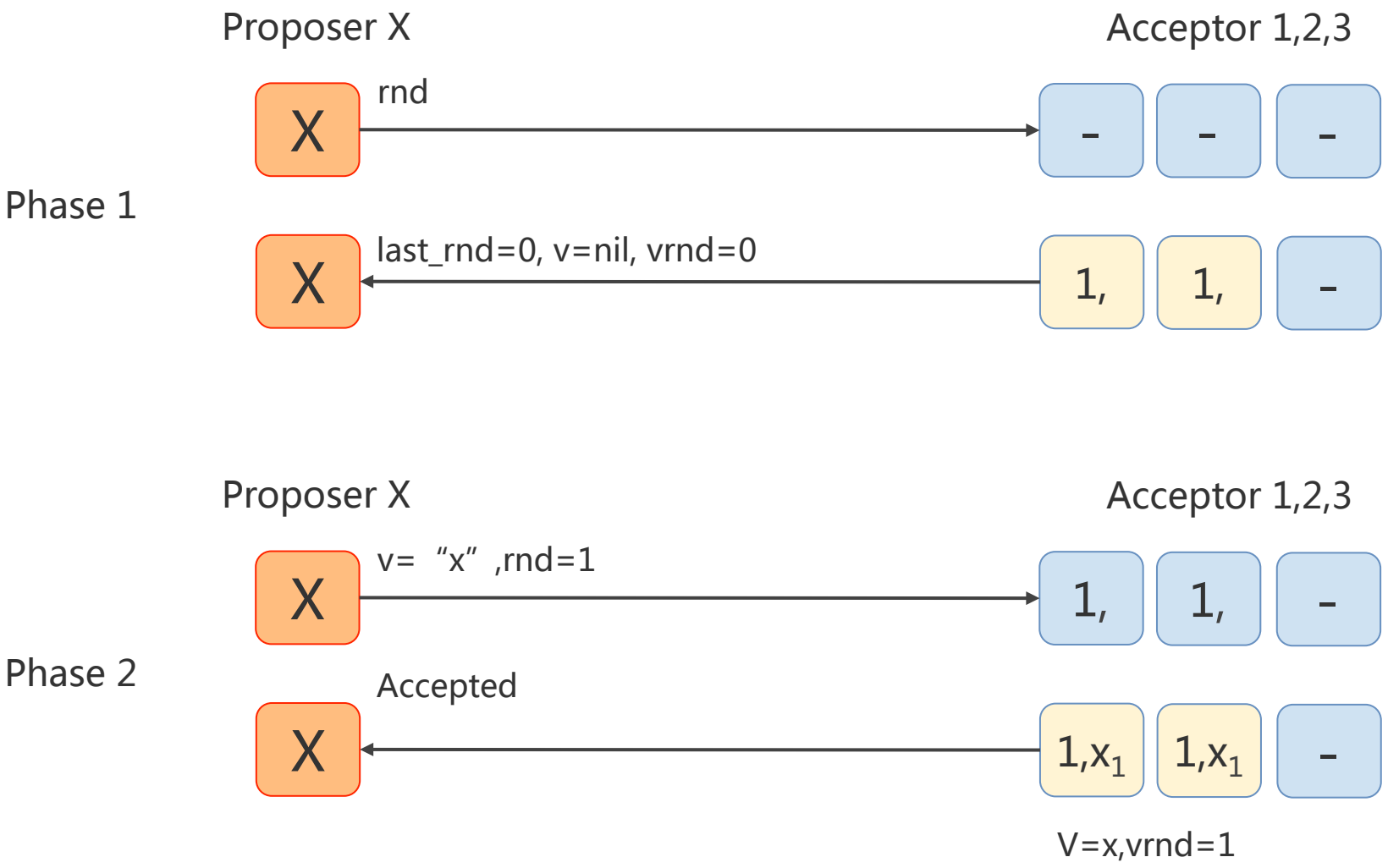
Paxos算法流程图



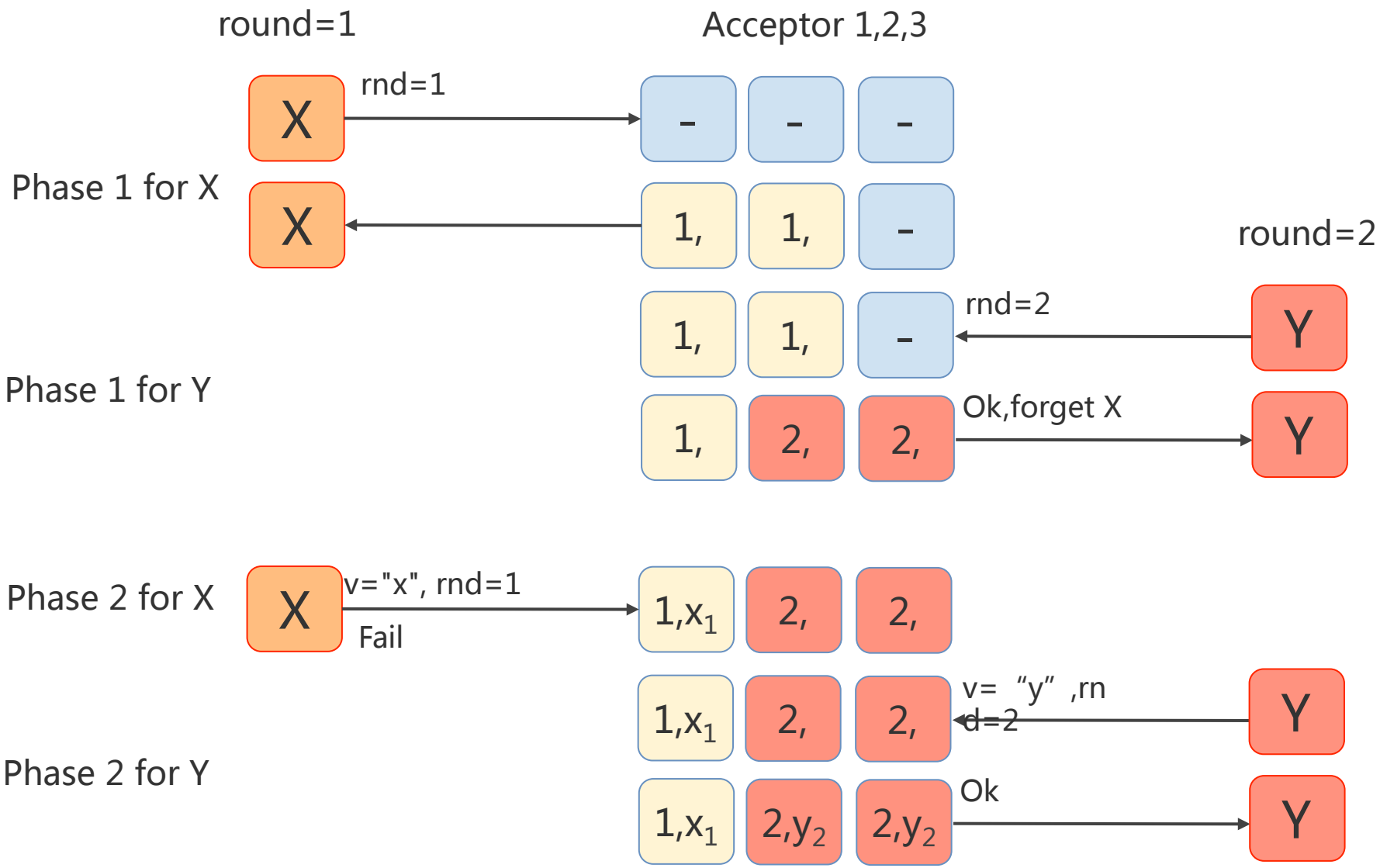
Acceptor:
只接受rnd等于last_rnd的Phase 2请求
last_rnd==rnd保证2个phase中间没有其他Proposer介入

否则(Acceptor接受了更高的rnd)写失败；此时提高rnd,重新执行

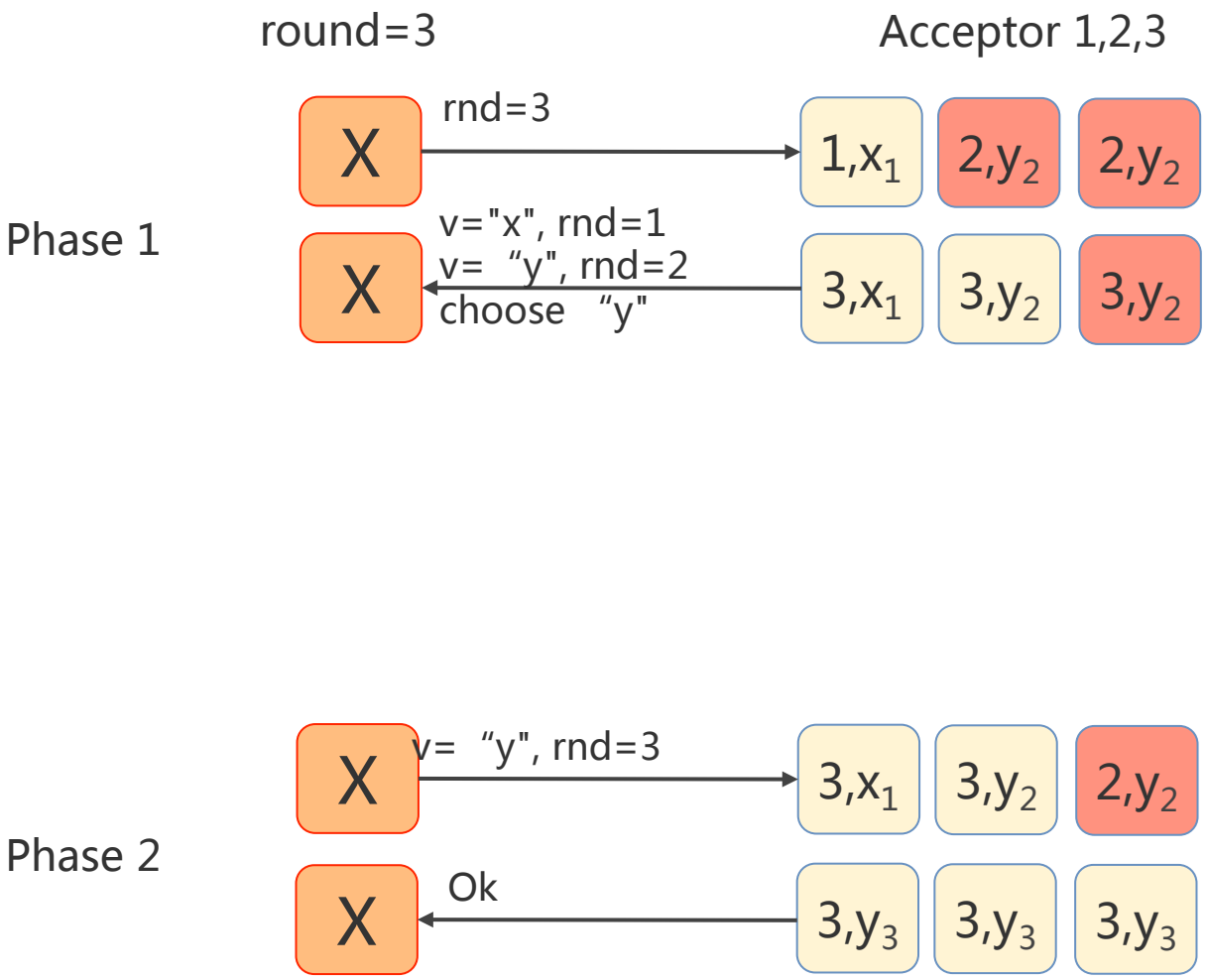
Paxos无冲突情况



Paxos有冲突情况



Paxos响应



Paxos算法应用场景

Paxos算法提出的目的为了解决在分布式系统环境中解决一致性问题。

- NoSQL领域：一致性强调“能读到新写入的”，就是读写一致性。
- 数据库领域：一致性强调“所有数据状态一致”，执行一个事务操作，如果事务成功，所有表中的数据都按照事务中的 SQL进行了操作；如果事务执行失败，所有的数据都回到初始状态。
- 状态机领域：一致性强调在每个初始状态一致的状态机上执行一串命令后状态都必须互相一致，也就是顺序一致性。

Paxos算法的解决方案

针对上述问题，Paxos的解决方案如下：

对分布式系统中各个服务器的状态进行全局编号，如果能编号成功，那么所有操作都按照编号顺序进行，来保证一致性。各个服务器通过投票表决来进行编号，让大多数服务器进行表决来确定哪台服务器是什么编号。而每次表决只能产生一个数据，否则表决就没有任何意义。

Paxos算法的核心和精华就是确保每次表决只产生一个Value。

Fast Paxos算法

自从Lamport在1998年发表Paxos算法后，一直对Paxos进行改进，并在2005年发表了Fast Paxos

Lamport在40多页的论文中不仅提出了Fast Paxos算法，并且还从工程实践的角度重新描述了Paxos，使其更贴近应用场景。从一般的Client/Server来考虑，Client其实承担了Proposer和Learner的作用，而Server则扮演Acceptor的角色

Client/Proposer/Learner：负责提案并执行提案

Coordinator：Proposer协调者，可为多个，Client通过Coordinator进行提案

Leader：在众多的Coordinator中指定一个作为Leader

Acceptor：负责对Proposal进行投票表决

Fast Paxos 算法过程

Fast Paxos算法包括2个阶段：

Phase1：prepare

a. Proposer选择一个proposal编号n,发送给acceptor的一个majority
b. 如果acceptor发现n是它已回复的请求中编号最大的，它会回复它已经接受最大的proposal和对应的value值（如果有）；同时不会批准编号小于n的proposal

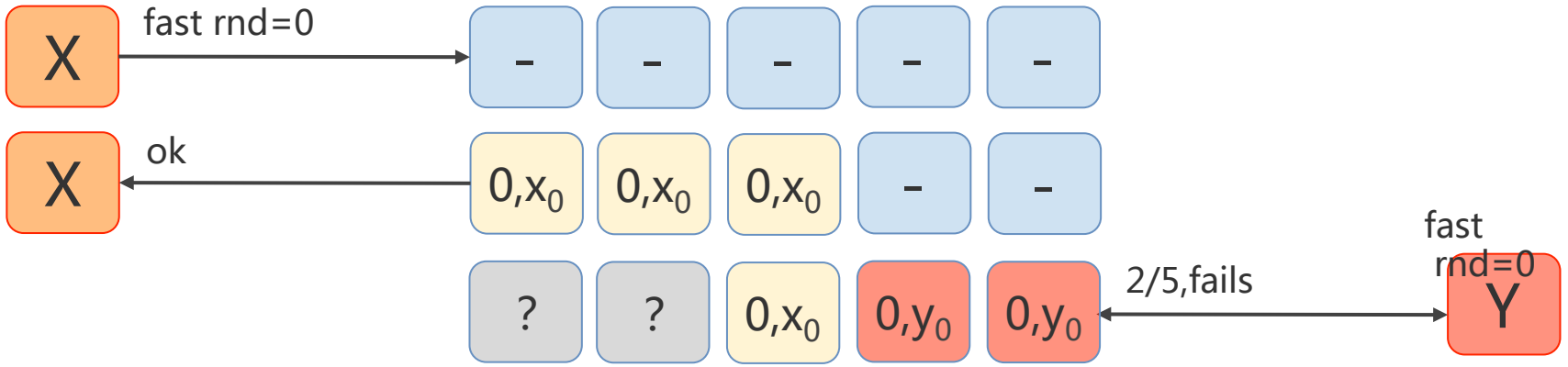
Phase2：accept

a. Leader收集Acceptor的返回值，如果Acceptor无返回值，则自由决定一个；如果有返回值，则选择proposer编号最大的一个
b. Acceptor把表决的结果发送给Learner。

Fast Paxos算法的主要变化在Phase 2a阶段：

- 若Leader可以自由决定一个Value，则发送一条Any消息，Acceptor便等待Proposer提交Value
- 若Acceptor有返回值，则Acceptor需选择某个Value

Fast Paxos算法流程图



X_0是否已经被系统接受了？

当Y只联系到半数Acceptor时,为了保证也能确认X_0是否接受:
要求X_0被接受的标准变为：
在Y联系到的半数Acceptor里，也有半数以上接受了X_0
因此接受X_0的Acceptor > 3/4

Paxos vs Fast Paxos

Quorum > 3/4 Acceptor 要求更高的系统可用性
如果Classic 可以工作在系统99.99999999%的时间里

Fast 只能工作在系统99.999%的时间里
Fast Paxos需要每个Paxos Group里有5个Acceptor

Multi Datacenter Consistency需要5个IDC
Fast Paxos 部署3个IDC：
要求每次写入都能联系到每1个IDC

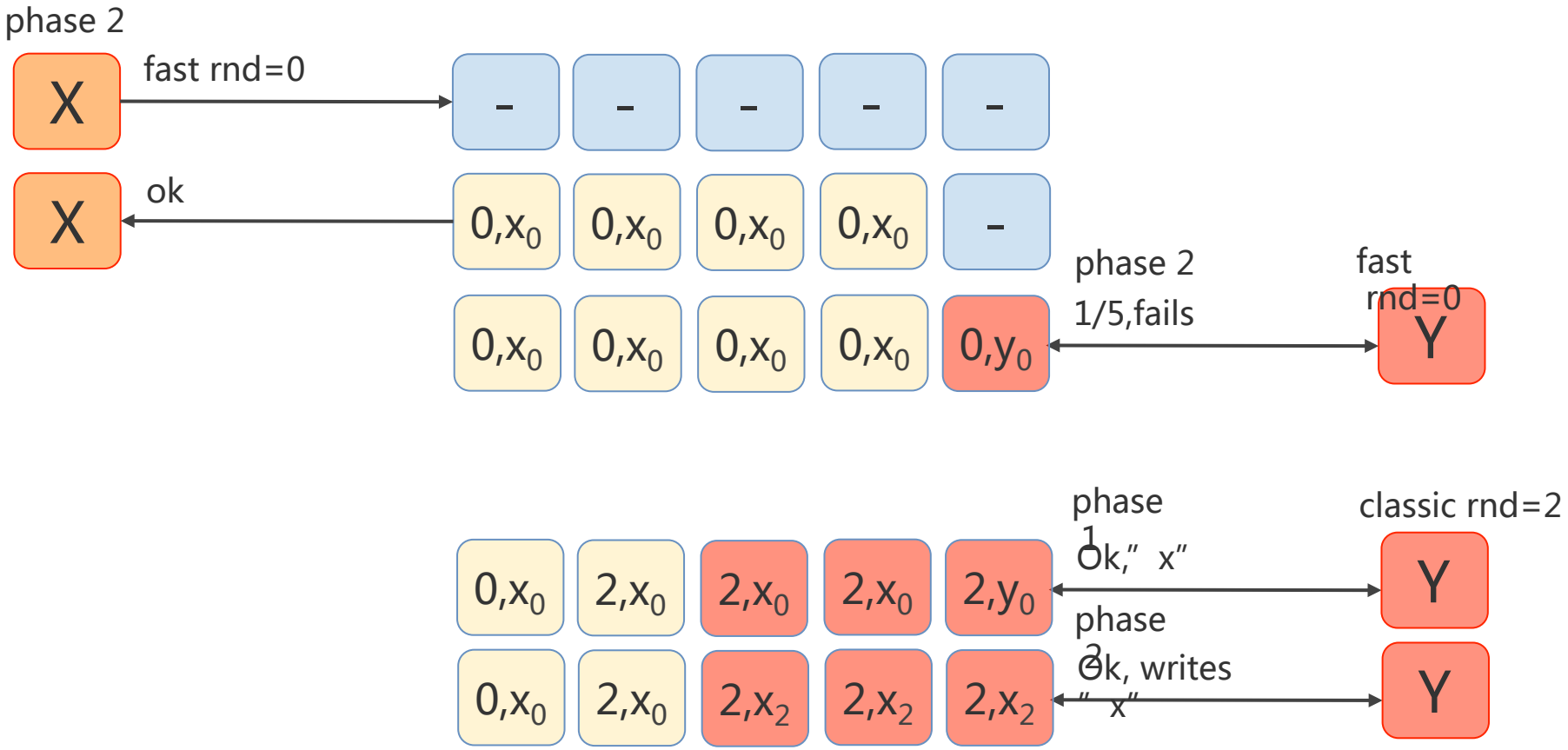
Fast Paxos解决冲突

多个Proposer直接Phase 2, Fast Paxos的rnd=0;

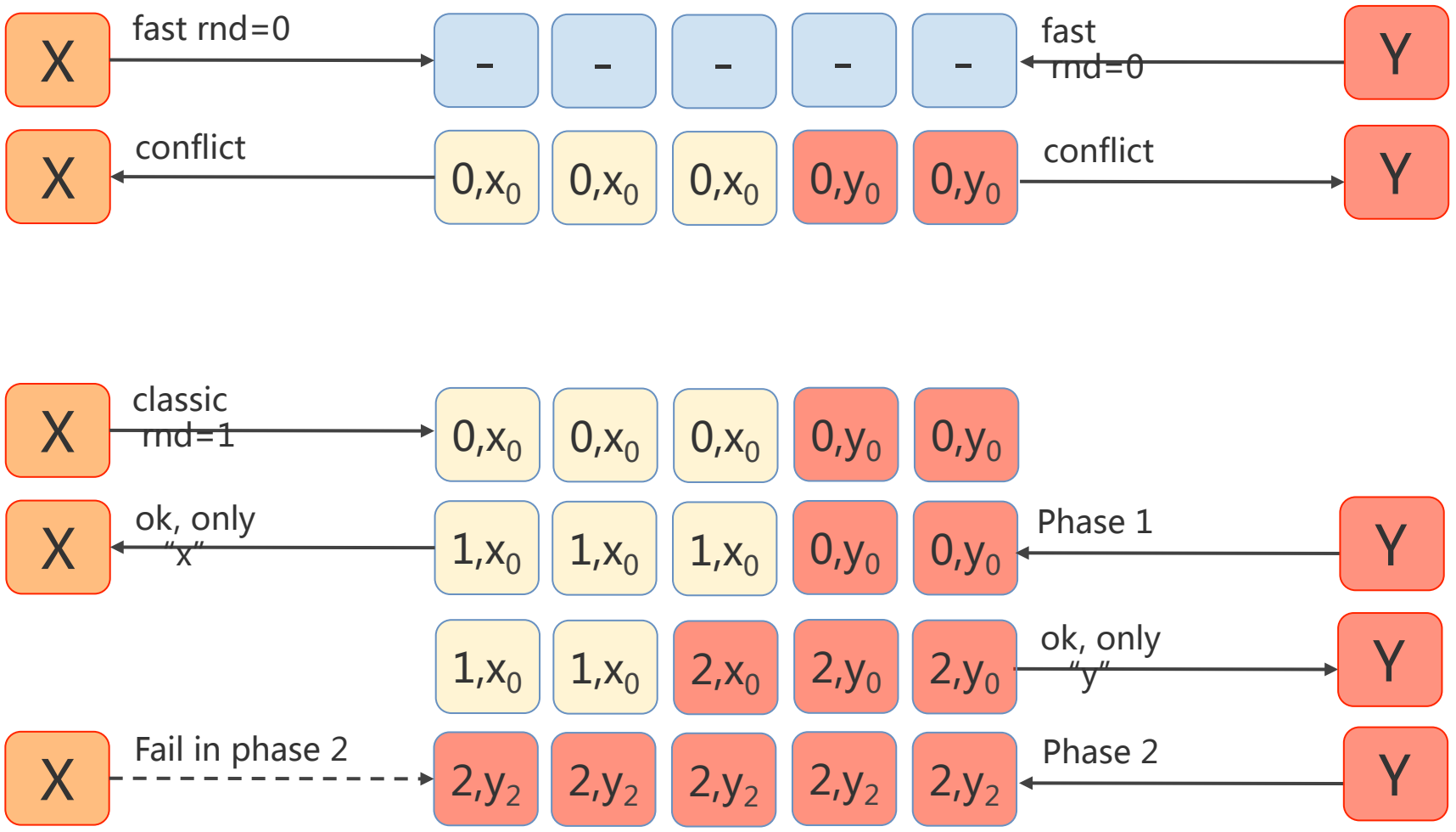
Acceptor只在v是nil的时候才接受

当遇到多个Fast并发冲突:
产生新的rnd>0执行Classic Paxos来解决冲突

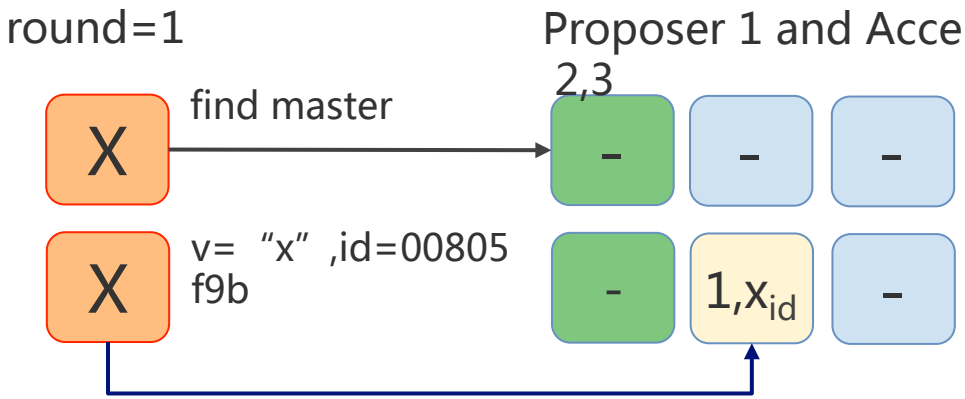
Fast Paxos 4/5 Y 冲突



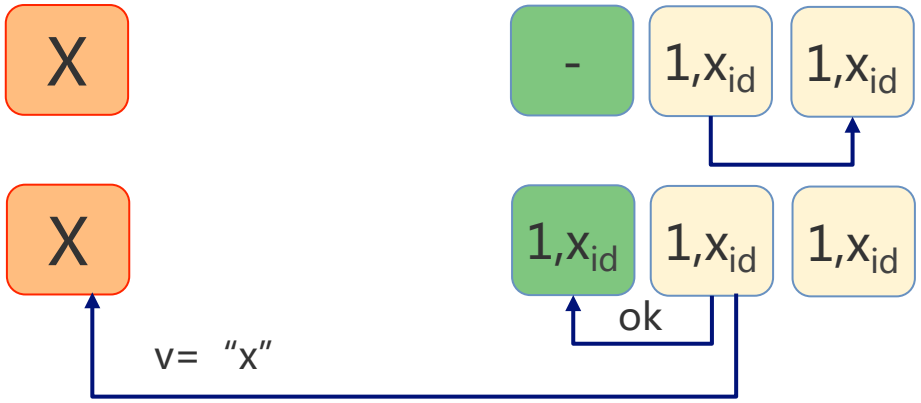
Fast Paxos 4/5 XY 冲突



Paxos改进

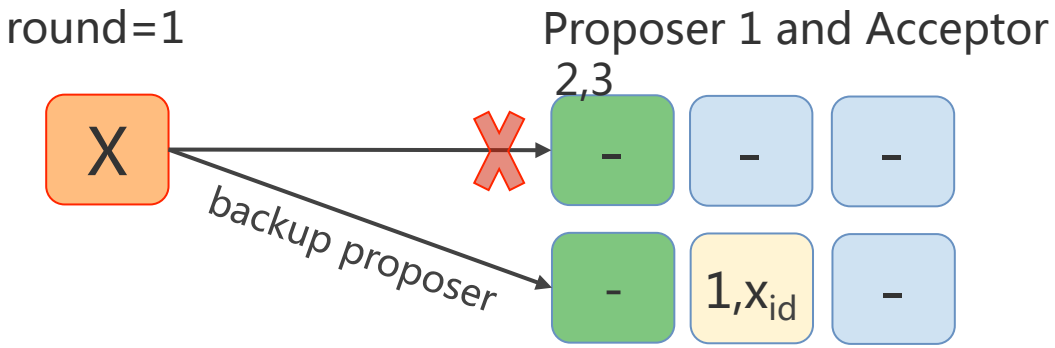


选择一个Server1作为Acceptor+Proposer，Server1负责找到Master，将v发给它。
此时Master作为Learner，它将v值发送给Acceptor3。

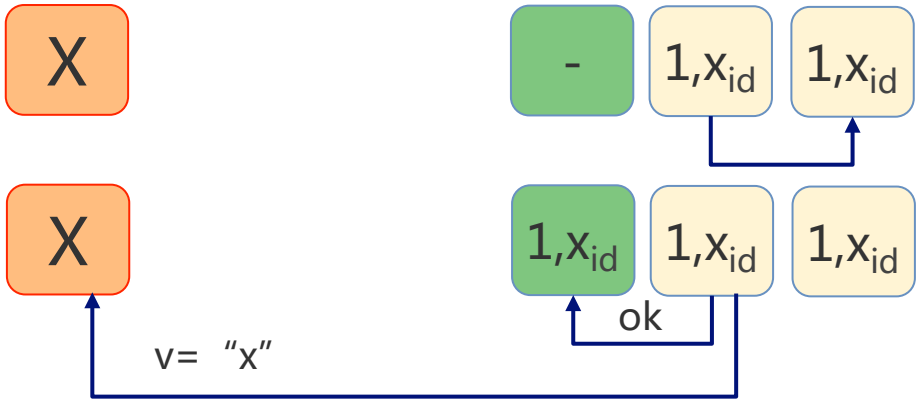


Learner 发送v值给Acceptor3后，将响应发送给Proposer，并将结果返回给Client X。

Paxos改进



如果Proposer不响应，Client X切换到备用的Proposer。Proposer负责找到Master，将v发给它。Master检查v的id值，如果id值已经被更新过，则执行去重复操作。



Learner发送v值给Acceptor3，Acceptor3也执行去重复操作，Learner将响应发送给Proposer，并将结果返回给Client X。

基于 Fast Paxos的MDCC协议

MDCC协议是由UC Berkeley大学提出的Fast Commit协议，用在维护大规模分布式数据中心部署的数据库的一致性上。
MDCC协议是一个新的commit 协议和事务编程模型，有效地达到了跨数据中心部署的数据库强一致性。

MDCC主要包括两个模块：
PLANET 事务编程模型
MDCC 提交协议

PLANET 事务编程模型

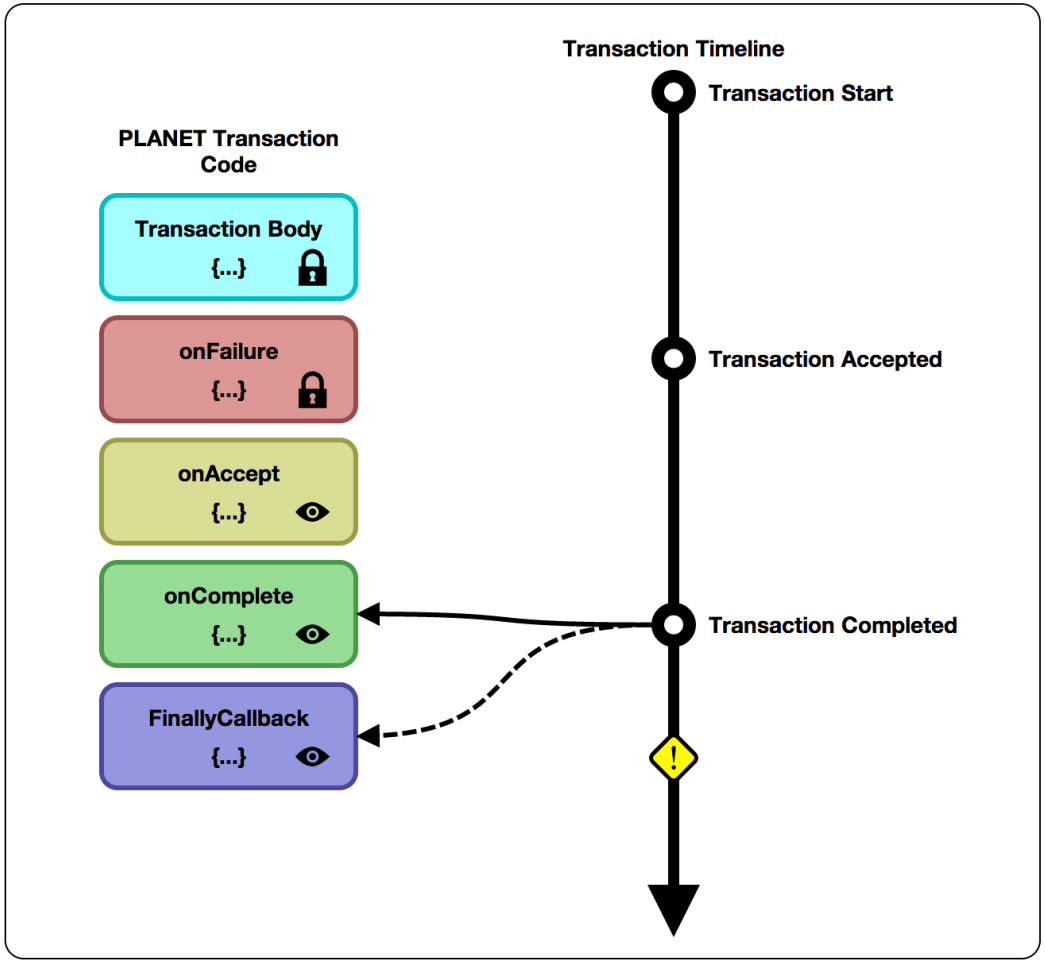
PLANET帮助开发人员发现事务状态的细节，支持回调，实现一个提交近似模型，优化事务。
为开发人员提供了处理不可预期网络延时的灵活性。
在处理事务时，编程模型强制指定了一个SLO超时，为每个事务考虑可接受的响应时间。
此编程模型保证了在特定的时间内，将事务执行的结果返回给应用。当执行返回给应用时，事务将处于以下三种状态之一：*onFailure,onAccept,onComplete*。
事务将在最近的状态上执行相关代码。

提交近似模型
PLANET使用了提交近似(*commit likelihood*)计算，让开发者能够预测提交的值。在MDCC中，PLANET有统计模型，能够利用本地统计来计算初始化的提交近似值，在事务处理的过程中，持续提交近似值。另外，启用推测提交(*speculative commits*)，PLANET也使用接纳控制(*admission control*)，高效地使用系统资源。

PLANET 交互示例

此交互示例详细介绍了事务处于不同阶段的应用场景和超时设置，左半部分为不同的事务阶段，右半部分为事务的时间周期，箭头显示了哪个阶段被执行，和具体执行时间。

进一步了解请访问
http://planet.cs.berkeley.edu/



MDCC 协议

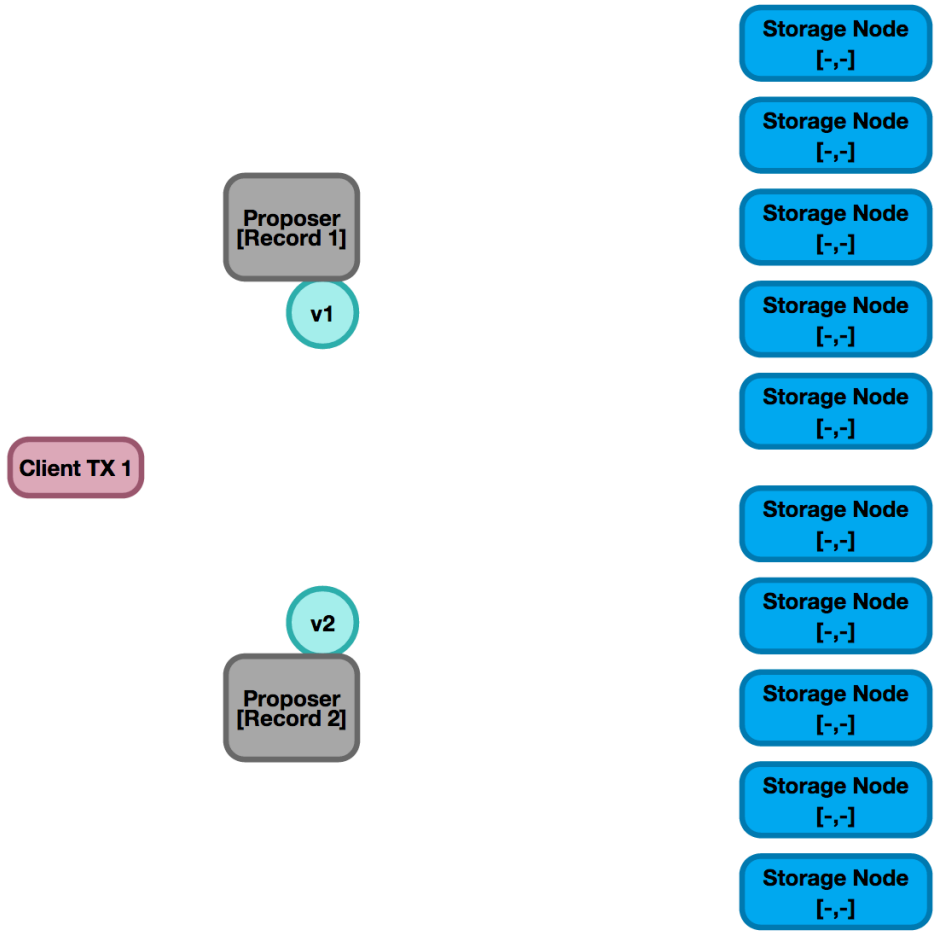
MDCC 协议基于Paxos 算法族，整合了classic-, multi-, fast-,generalized-paxos多个 paxos 变种版本，在不同的应用场景下使用不同的变种协议，能够在一个round trip时间提交事务。

选项(options)：
MDCC 让acceptor对更新选项达成一致，一个更新选项可以看做是一个承诺(在未来的某个时间点完成更新)，当acceptor对事务中的所有更新选项达成一致时，事务被提交。
Paxos保障更新选项不丢失，因此事务是持久性的。最终发送异步提交消息来执行更新，并且更新是可见的。
MDCC 主要使用Generalized-Paxos的round周期来达到一次round-trip。
即使以前阻塞的options已经提交或放弃，当options提交时，acceptor也只接受更新options。
因此为了接受一个option，acceptor必须考虑阻塞option的所有提交和放弃的可能性。这保障了如果一个 acceptor接受了更新option，那么提交将不会失败。

可交换更新(commutative updates)：
MDCC 使用可交换更新进一步优化事务，在并发更新时避免冲突。然而，对于可交换操作，数据库中的一些属性有一些域完整性限制，很难在一个大规模分布式系统中做到。
MDCC 使用一个新的分界协议(demarcation)减少全局协调的数量，从而保证完整性限制。

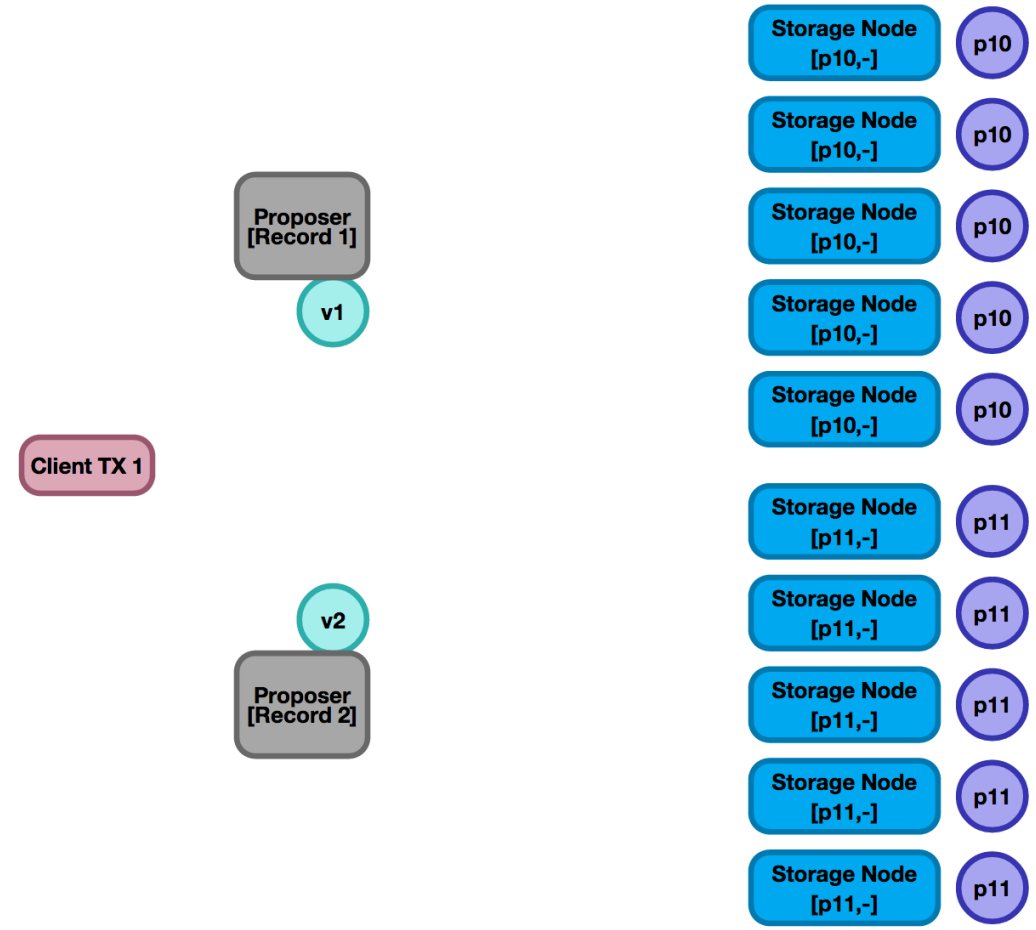
MDCC 协议算法流程

client发起事务，将”v1”值发送到
proposer1，将”v2”值发送到proposer2



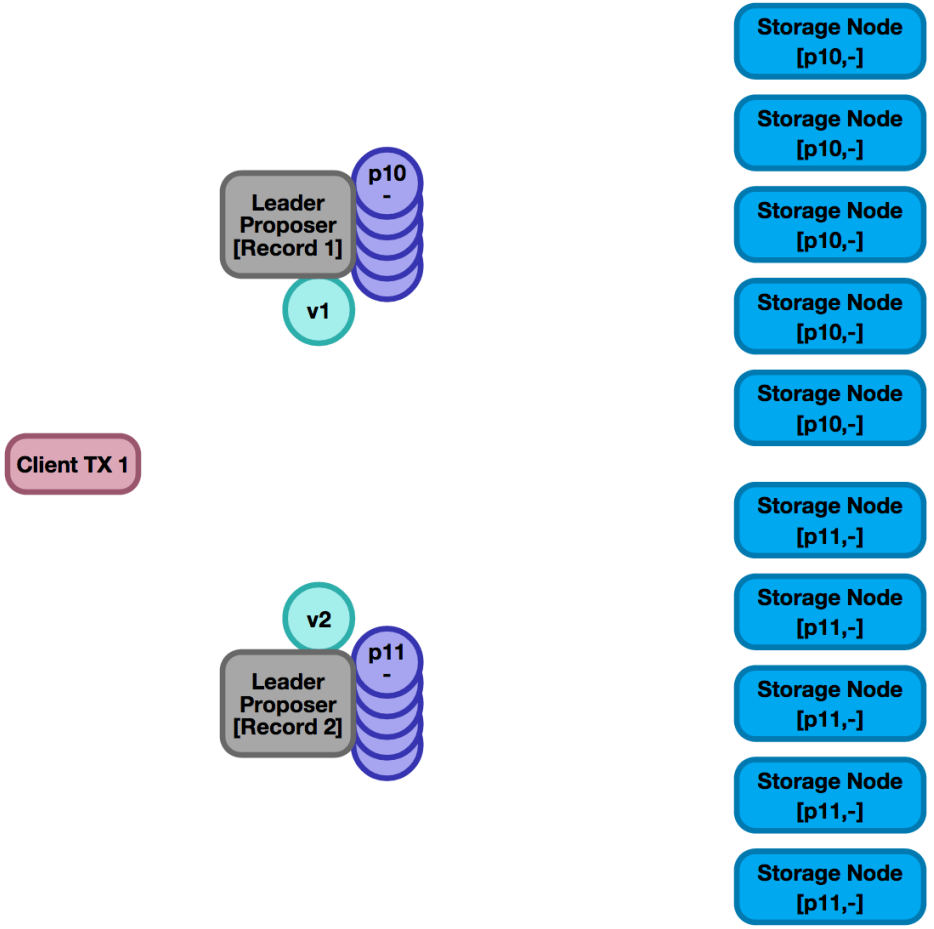
MDCC 协议算法流程

Classic Paxos:Phase1a
Proposers现在不是leaders，所以它们通过发送phase1a消息到Storage Node，尝试变成leader。



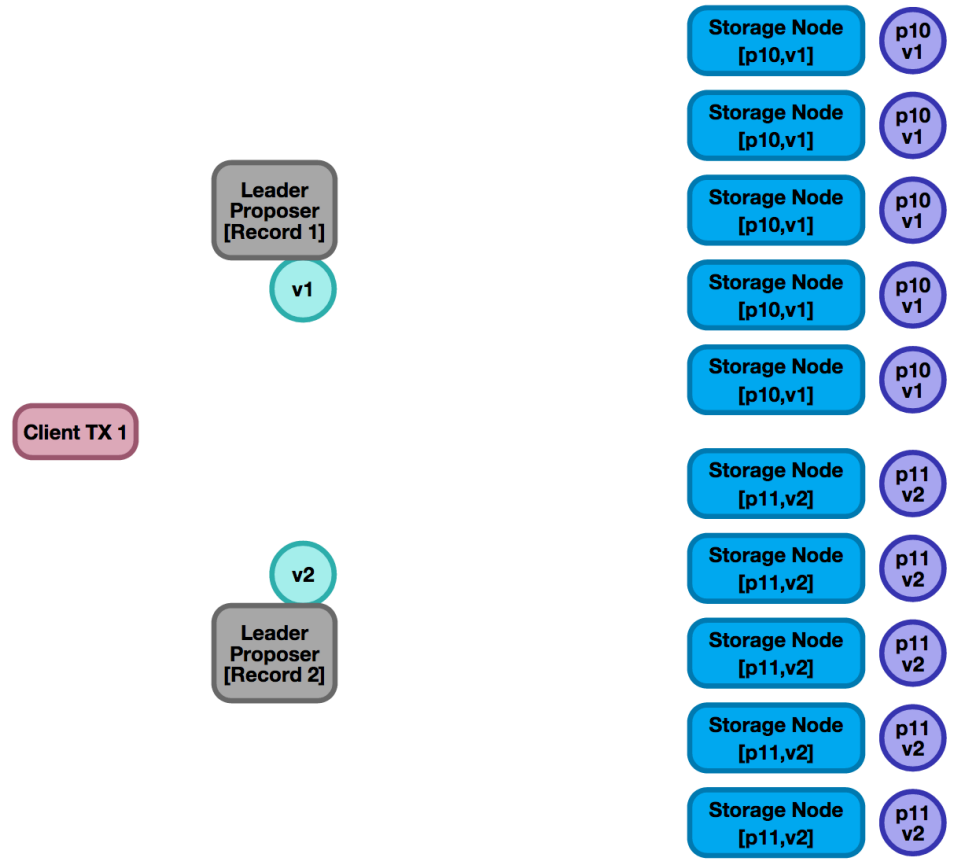
MDCC 协议算法流程

Classic Paxos:Phase1b
Storage Node接受phase1a消息，发送回一个phase1b消息到Proposers，消息带有最大的proposer号码，但没有分配v值。



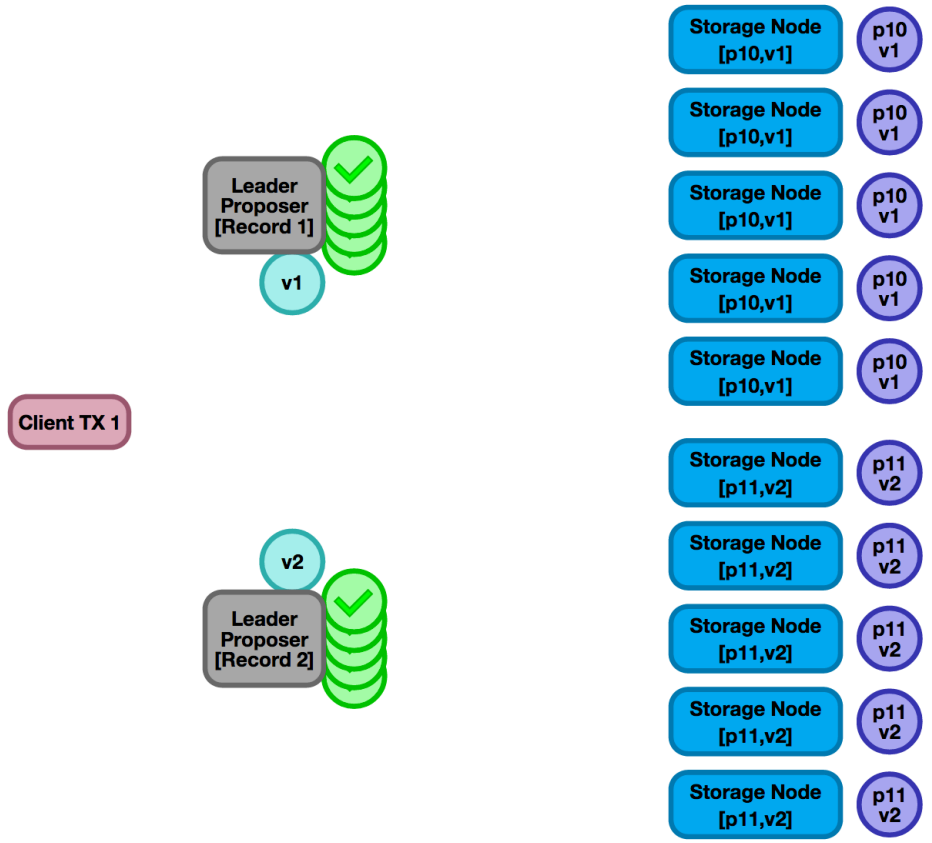
MDCC 协议算法流程

Classic Paxos:Phase2a
Proposers决定多数派接受它们的proposer号码，因此它们是leaders。从Node发出的值并没有分配，因此Proposers可以写入任意的值，proposers发送带有“v1”和“v2”值的phase2a消息。



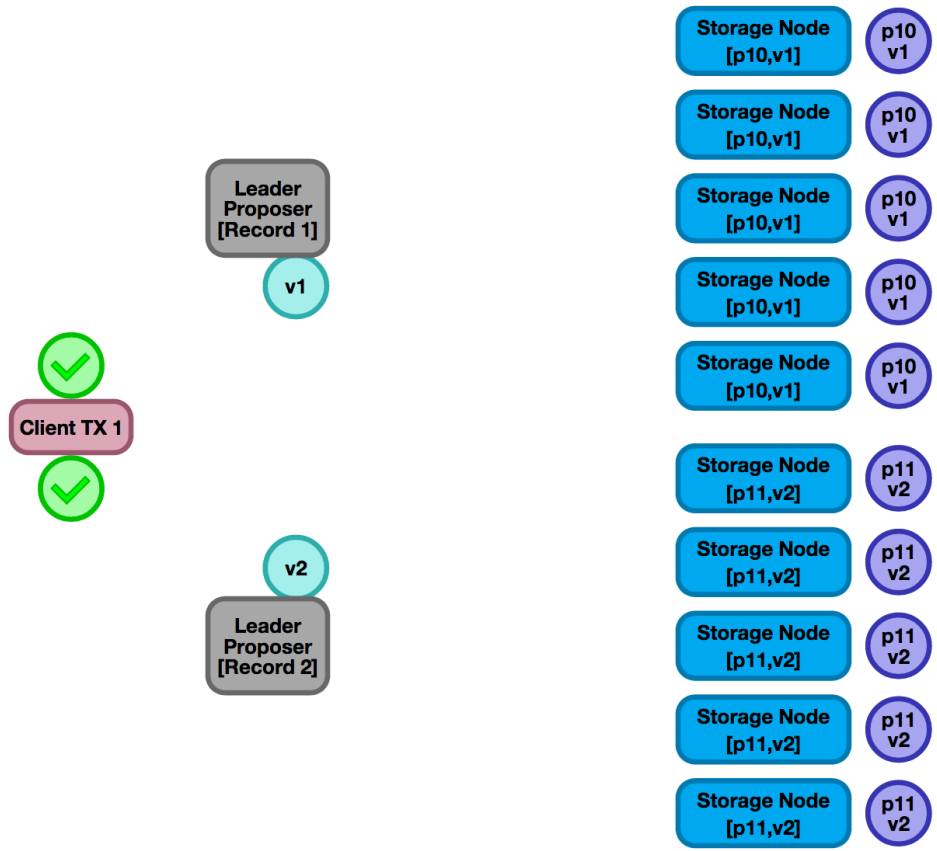
MDCC 协议算法流程

Classic Paxos:Phase2b
因为更高的proposal号码还没有看到，Storage Node接受新值，Nodes发回了Phase2b消息确认已经接受。



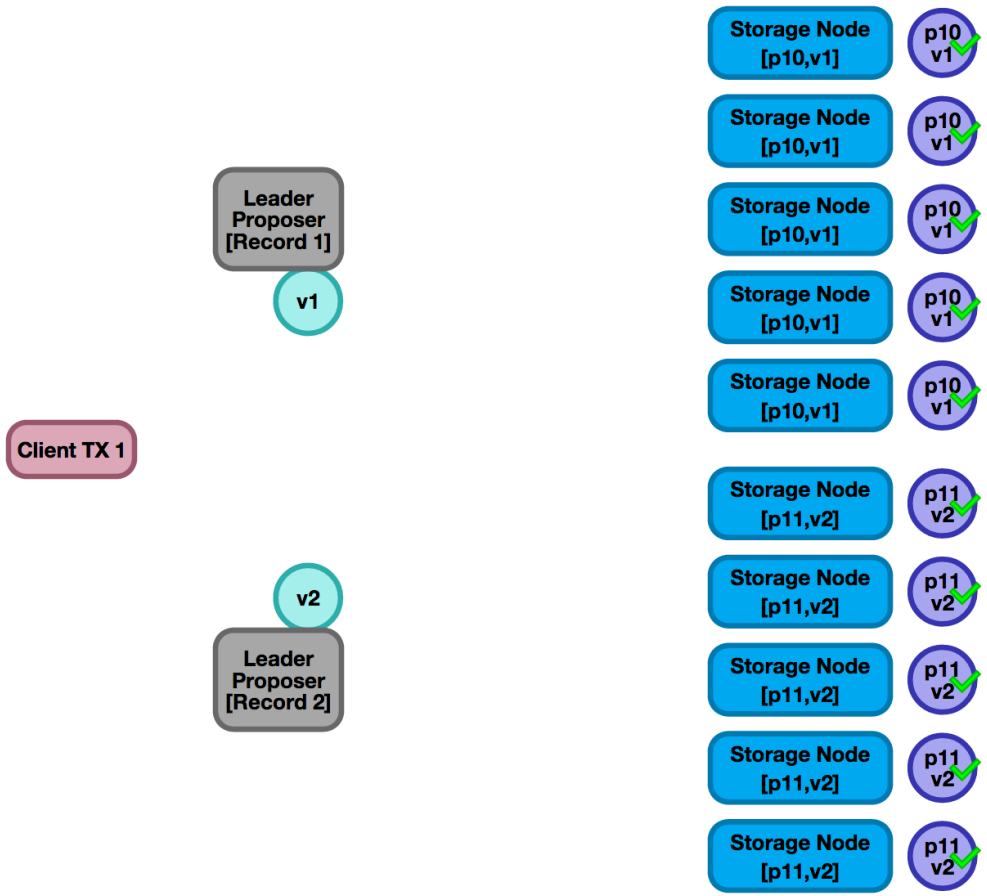
MDCC 协议算法流程

Proposers 转发接受消息
Proposers 确定多数派已经接受了新值，因此它们转发这个接受状态给client。在这一点上，事务中所有记录都接受了更新option，事务被永久提交。



MDCC 协议算法流程

可见异步提交
Client 事务异步发送消息到nodes
，通知它们提交，同时使更新可见。



MDCC 分界协议

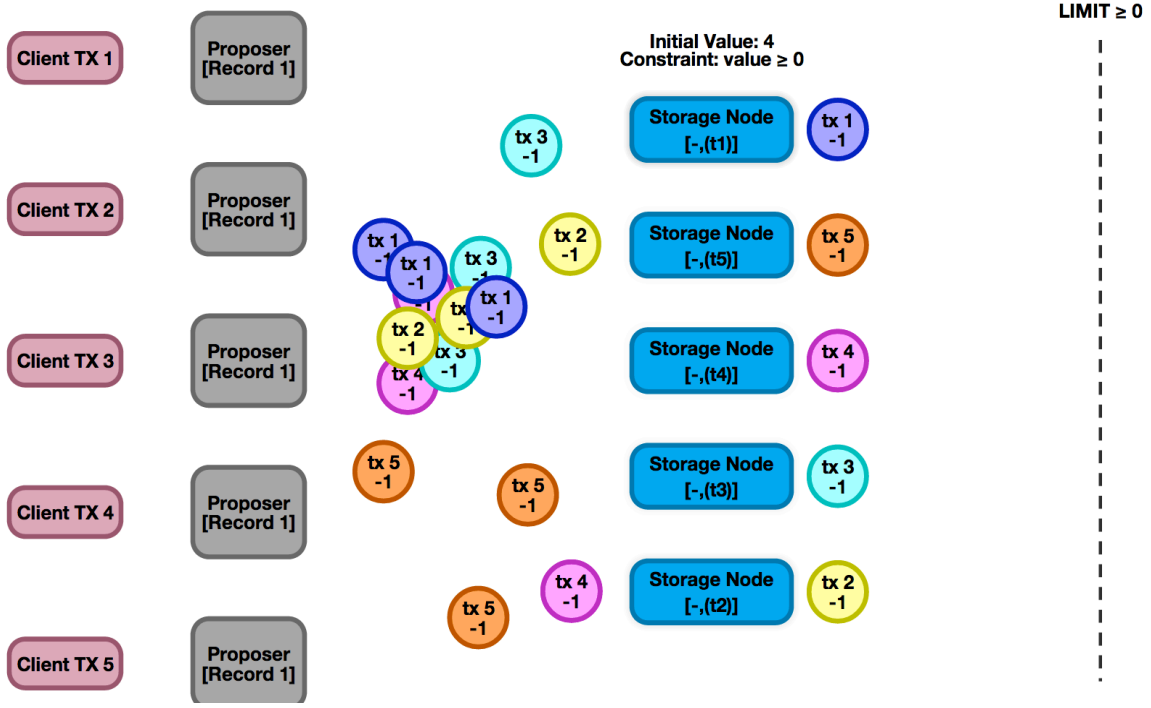
一个属性作为一个限制条件 attribute1 >=0，有一个初始值V。假设有 N=5的Storage Node，法定人数Q=4。如果Storage Node允许递减更新，直到本地值为0，极有可能是太多更新事务提交，这样就违反了完整性限制。
因为一个事务仅需要 $\frac{Q}{N} = \frac{4}{5}$ Storage Node 获得响应，最差的情况是，只有Q=4消息被接受。最低限制如下：

$$lowerLimit \geq \frac{N-Q}{N} \cdot V = \frac{V}{5}$$

最低限制保证了完整性限制，维持了足够低的限制获得更好的并发性。当Storage Node达到最低限制时，将开始拒绝新的更新。

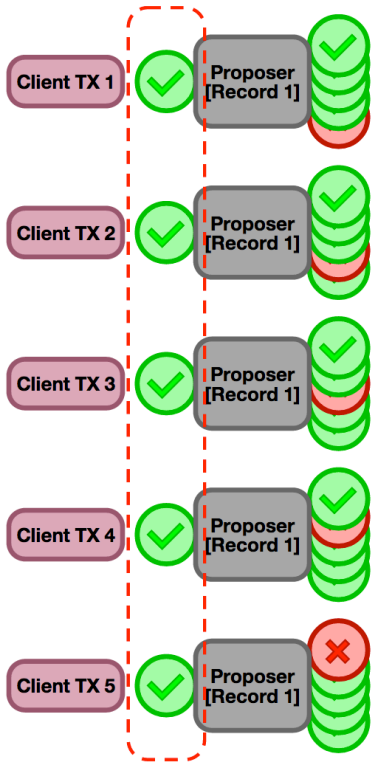
MDCC 分界协议

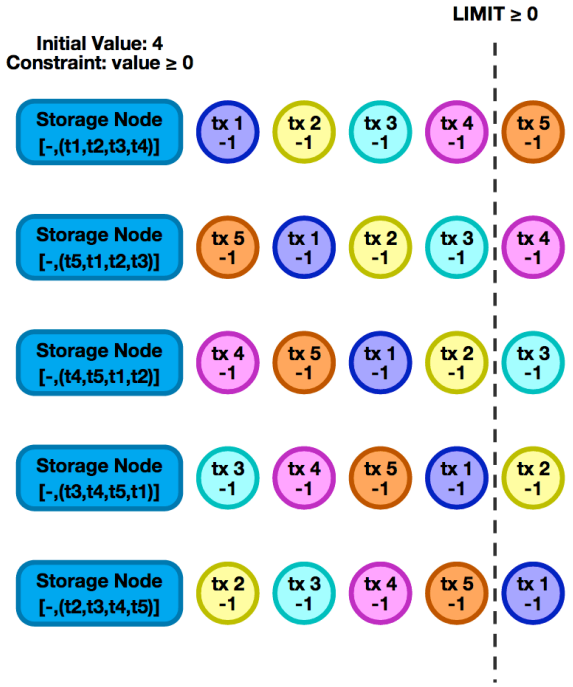
快速propose w/ 可交换更新：
MDCC 使用分界协议修改了每个Storage Node的限制，这个限制更加保守，但是保证了完整性限制。因此，每个Storage Node都接受前3个事务，拒绝后2个事务。



MDCC 分界协议

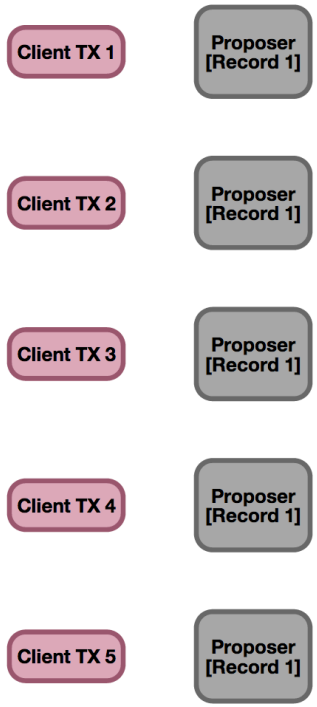
节点响应：
Proposer接受了Storage Node的响应，每个proposer接受了4个 accept 和一个reject。法定人数大小是4，每个事务都接受了足够的 accept 消息，因此可以提交。而，所有5个事务的初始值为4，完整性限制得到保障。

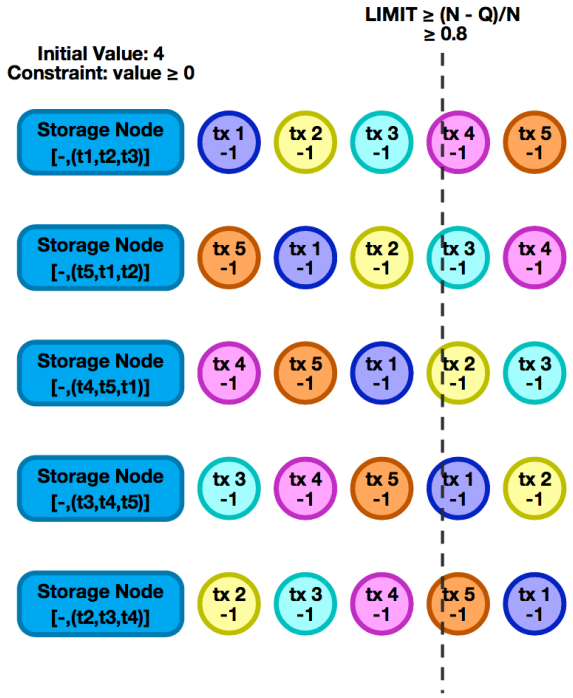




MDCC 分界协议

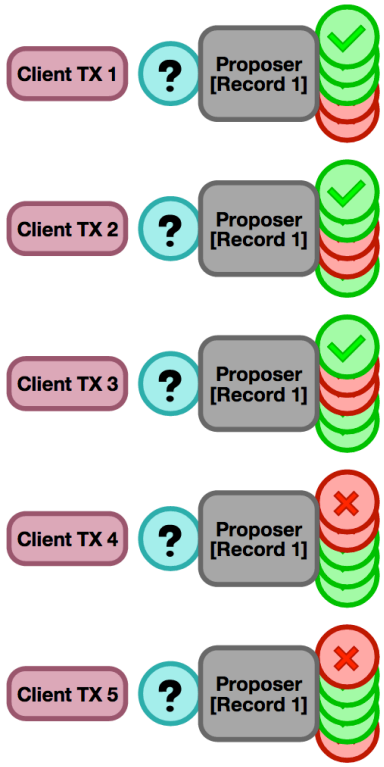
快速propose w/ 可交换更新（分界协议）：
MDCC 使用分界协议修改了每个Storage Node的限制，这个限制更加保守，但是保证了完整性限制。因此，每个Storage Node都接受前3个事务，拒绝后2个事务。

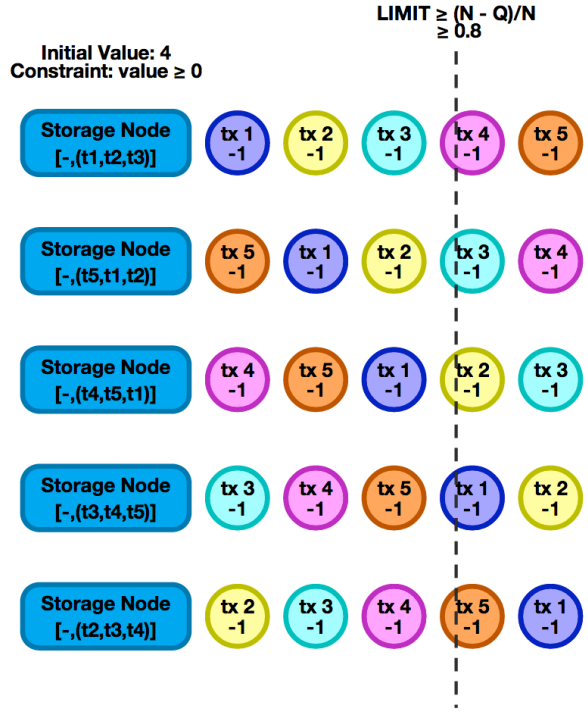




MDCC 分界协议

节点响应：
每个事务都接受了3个接收消息，2个拒绝消息。完全的法定人数4还没有达到，事务状态还未知，必须调用冲突解决方法。但是遵循了完整性限制条件。





MDCC 协议对比

	Consistency	Transactional Unit	Commit Latency	Data Loss Possible?
Amazon Dynamo	Eventual	None	1 round trip	Not possible
Yahoo PNUTS	Timeline per key	Single key	1 round trip	Possible
COPS	Causality	Multi-record	1 round trip	Possible
MySQL (async)	Serializable	Static partition	1 round trip	Possible
Google Megastore	Serializable	Static partition	2 round trips	Not possible
Google Spanner	Snapshot Isolation	Partition	2 round trips	Not possible
Walter	Parallel Snapshot Isolation	Multi-record	2 round trips	Not possible
MDCC	Read-committed without lost updates	Multi-record	1* round trip	Not possible

进一步了解 MDCC 协议，
请访问http://mdcc.cs.berkeley.edu/

相关开源项目
https://github.com/radlab/SCADS

相关论文
http://mdcc.cs.berkeley.edu/mdcc.pdf

Q&A