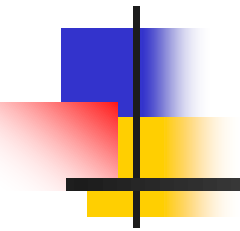


# 第7章 复制

# 第7章 复制

- 简介
- 系统模型
- 容错服务
- 复制数据上的事务
- 高可用服务的实例研究：gossip体系结构、Bayou和Coda
- 小结

# 复制的概念

- 在多个节点上保存相同数据的一个副本
  - 数据库，文件系统，缓存系统……
- 一个拥有数据拷贝的节点称为副本管理器

# 为什么需要复制？

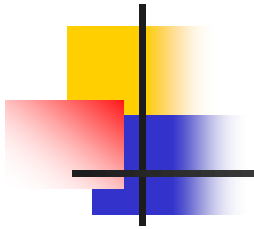
- 容错：如果一些副本不能被访问，另外一些仍能够被访问
  - 异常（软件，硬件和网络……）
  - 更新

# 容错（续）

- 可用性：在合理的响应时间内获得服务的次数所占比例应该接近100%
  - 服务器故障：$1-p^n$
    - 1个服务器出故障的概率是0.05
    - 2个服务器同时故障的概率是0.0025
    - 服务可用的概率是1-0.0025=0.9975
  - 网络分区和断链操作：预先复制
    - 乘飞机的用户，无线网络可能会中断（断链工作或者断链操作）

# 容错（续）

- 正确性：允许一定数量和类型的故障
  - 如果f+1个服务器，有至多f个服务器崩溃，理论上还有1个服务器能够提供服务
  - 如果至多f个服务器发生拜占庭故障，那么理论上2f+1个服务器能够通过正确的服务器以多数票击败故障服务器，提供正常服务



- 性能：将负载分散到多个副本
  - 负载均衡（系统扩展性，例如，DNS，Web.....）
  - 资源缓存（减少延迟，例如，迅雷会员.....）

# 复制的需求

- 数据在同一个域中的多个服务器之间进行资源缓存和透明复制

- 复制透明性
  - 客户不需要知道存在多个物理副本
- 一致性
  - 不同应用一致性强度有所不同

# 分布式系统的特点（CAP）

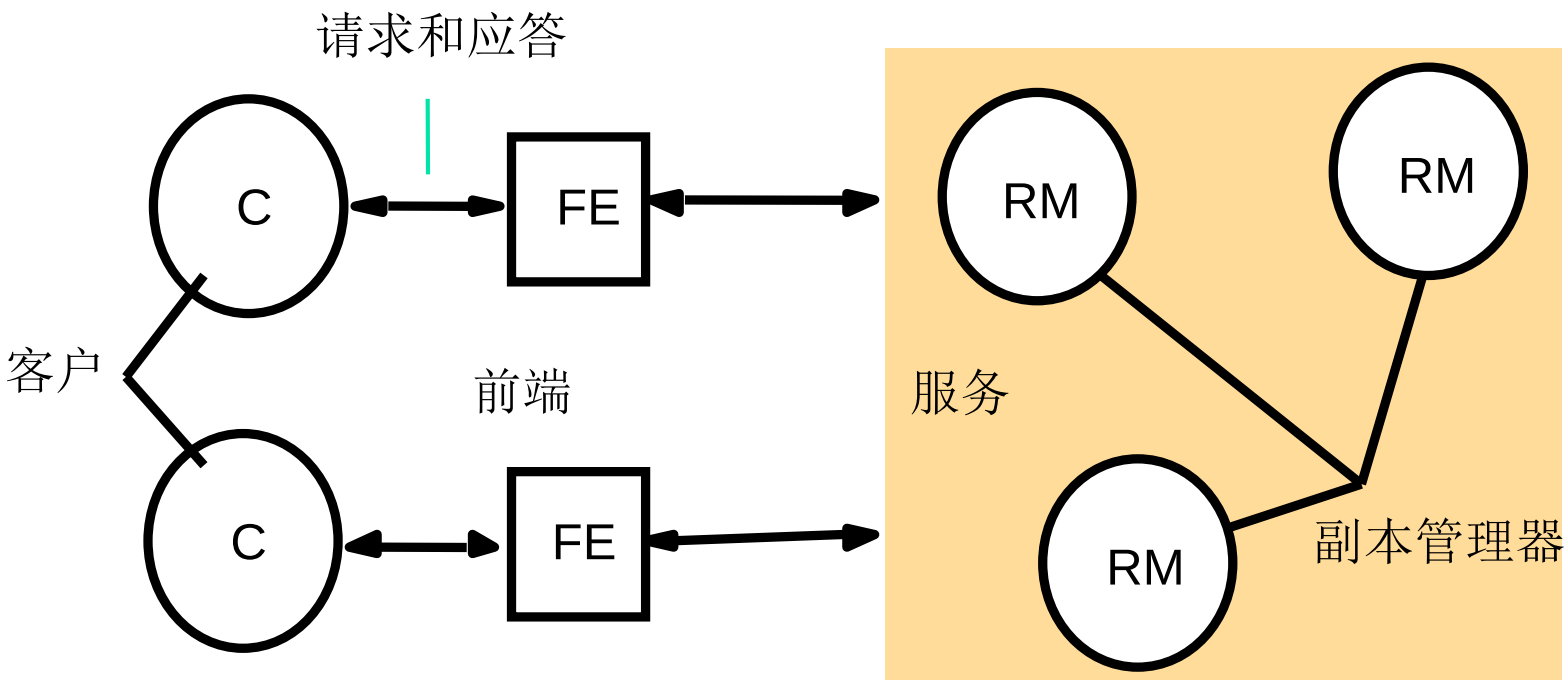
- Consistency（一致性）：所有节点在相同时间看到相同的数据（每个读操作都接收到最近的写）
- Availability（可用性）：无论节点的单个状态如何，每个请求都会获得（非错误）响应。
- Partition（分区）：断链的情况，消息被丢弃或延迟

# 第7章 复制

- 简介
- 系统模型
- 容错服务
- 复制数据上的事务
- 高可用服务的实例研究：gossip体系结构、Bayou和Coda
- 小结

# 系统模型

## 基本模型



# 系统模型

## 基本模型组件

- 前端

- 接收客户请求
- 通过消息传递与多个副本管理器进行通信
- 为什么不让客户直接进行通信？（保证复制的透明性）

- 副本管理器

  - 接收前端请求
  - 对副本执行原子性操作，其执行等价于以某种严格顺序执行（保证复制的一致性）

# 系统模型

## 副本操作

1. **请求：** 前端将请求发送至一个或多个副本管理器
   - 前端和某个副本管理器通信，此管理器再和其它副本管理器通信
   - 前端将请求组播到各个副本管理器
2. **协调**
   - 保证执行的一致性
   - 对不同请求进行排序
     - FIFO序：前端发送请求r然后r’，副本管理器端处理r然后r’
     - 因果序：请求r在r’之前（happen-before），副本管理器处理r在r’之前
     - 全序：一个副本管理器处理r在r’之前，任何副本管理器处理r在r’之前

3. **执行**：副本管理器执行请求
   - 包括试探性执行，即执行效果可以去除
4. **协定**：对于要提交的请求的执行结果达成一致
   - 副本管理器可共同决定放弃或者提交事务
5. **响应**：根据应用需要，一个或多个副本管理器响应前端
   - 高可用性：第一个到达的应答返回客户
   - 容忍拜占庭故障：大多数副本管理器的应答传送给客户

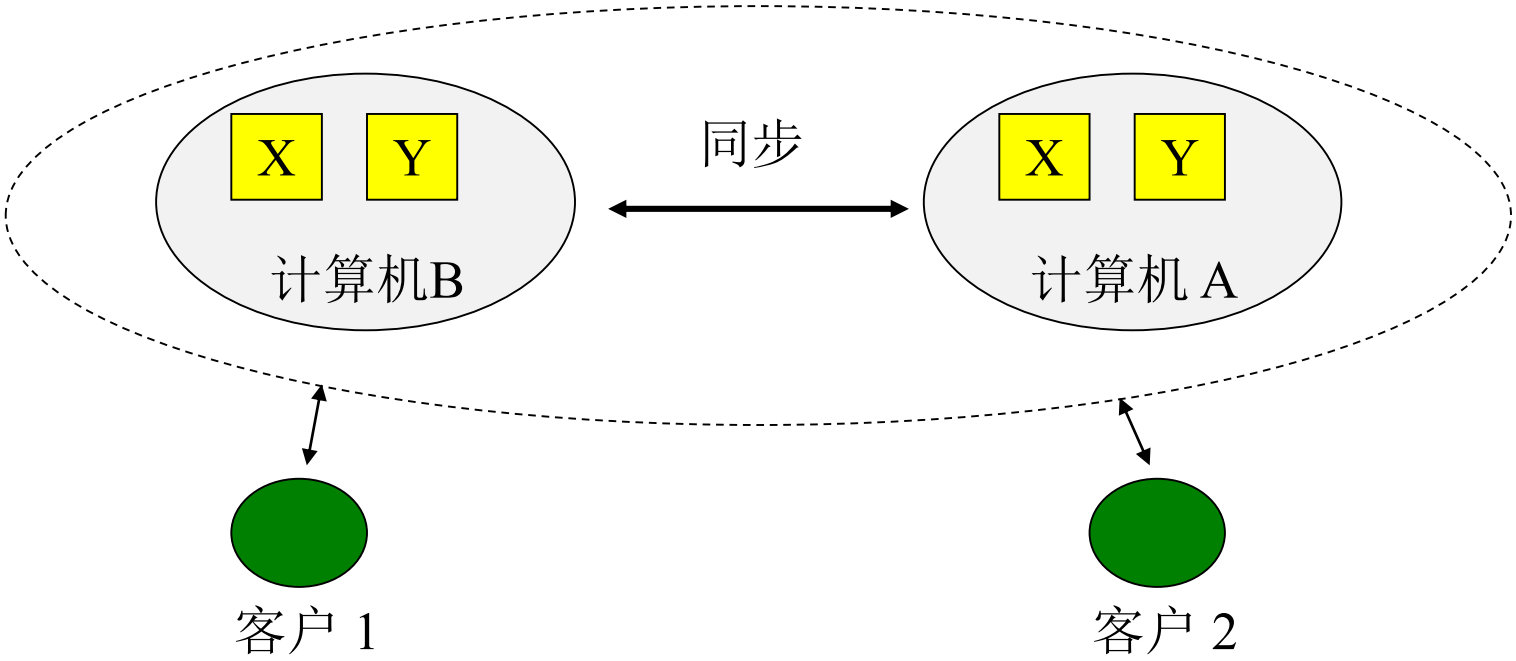
# 第7章 复制

- 简介
- 系统模型
- 容错服务
- 复制数据上的事务
- 高可用服务的实例研究：gossip体系结构、Bayou和Coda
- 小结

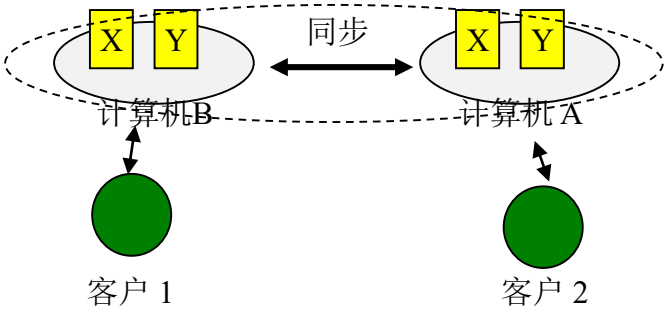
# 容错服务

- 副本之间的不一致性将导致容错能力失效

1. 银行帐户x和y的两个副本管理器位于计算机A、B上
2. 客户在本地的副本管理器上读取和更新帐户
3. 若本地副本管理器出现故障，则尝试使用另一个管理器
4. 响应完客户后，副本管理器在后台相互传播更新



# 容错服务




客户1:

$setBalance_B(x,1)$

B 出现故障…

$setBalance_A(y,2)$

切换到A…

客户2:

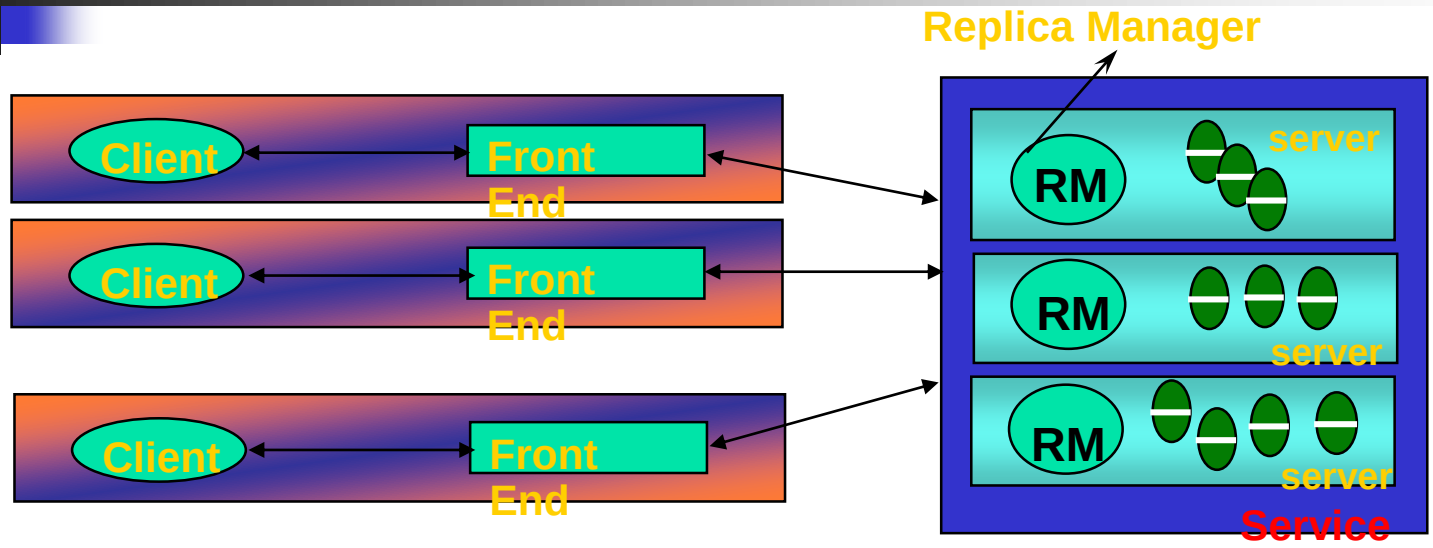
$getBalance_A(y)$->2

$getBalance_A(x)$->0

B出现故障，x更新未发送…

由于B在把帐户x的更新传送至A前出现故障，所以产生了不一致现象

# 容错服务




- 对所有的对象副本进行一致性更新
  - 可线性化
  - 顺序一致性

# 可线性化

- 目标：
  - 所有操作都像在单个副本上操作（尽管事实上是多个副本）
  - 每一个读操作都返回最新的值（强一致性，C in CAP theorem）

# 可线性化（概念）

- 一个被复制的共享对象服务，如果对于任何执行，存在某一个由全体客户操作的交错序列，满足以下两个准则，则该服务被认为是可线性化：
  - 操作的交错执行序列符合对象单个副本所遵循的规约
  - 操作的交错执行序列和实际运行中的次序实时一致

# 可线性化（翻译）

- 系统应该造成单个副本的错觉
- 无论哪个客户，读都会返回最近写的结果
- 无论哪个客户，所有后续读都应返回相同的结果，直到下一次写

- “最近”&“所有后续”
  - 由时间决定

可线性化（**Linearizability**）≠可串行化（**Serializability**）

**CAP** **ACID**

# 所有顺序由时间决定

- 无重叠（基于时间的明确次序）

a.write(x)

a.read()

a.read()

- 有重叠（基于时间的不明确次序）

a.write(x)

a.read() -> 0

a.read() -> x

a.read() -> x

如果a.read()→0则
不支持可线性化

# 容错服务

客户1:

$setBalance_B(x,1)$

$setBalance_A(y,2)$

客户2:

$getBalance_A(y)$->2

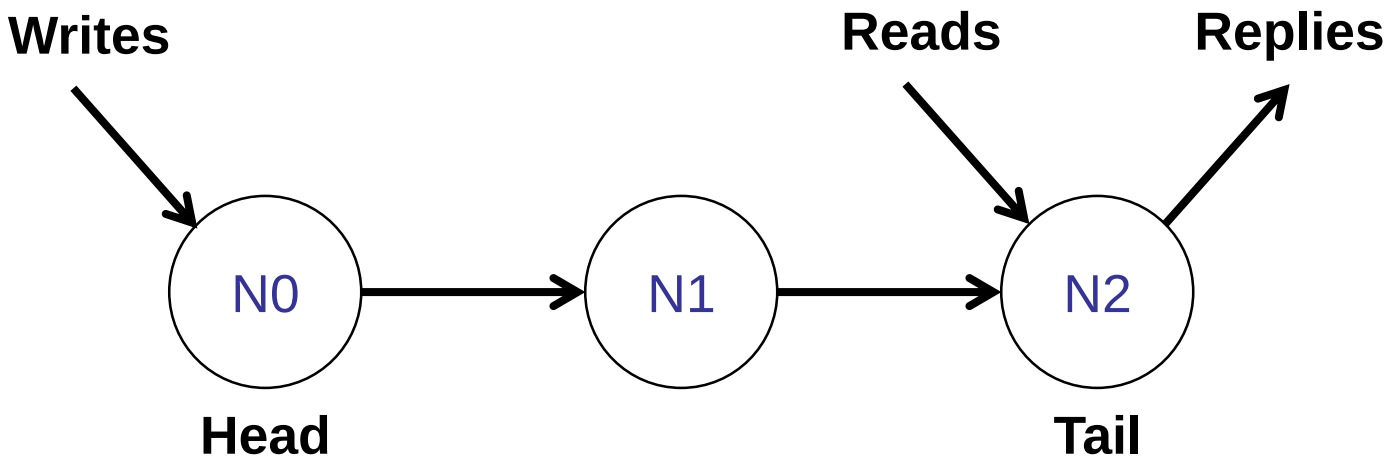
$getBalance_A(x)$->**0**

**全局实时性：如果$setBalance_B(x,1)$，则 $getBalance_A(x)$→1**

规约：转账资金不会消失，存款或者取款会影响余额
如果账户y的更新发生在账户x更新以后，
那么观察到y的更新，x的更新也应该被观察到

# 容错服务

链式复制（支持可线性化的技术）



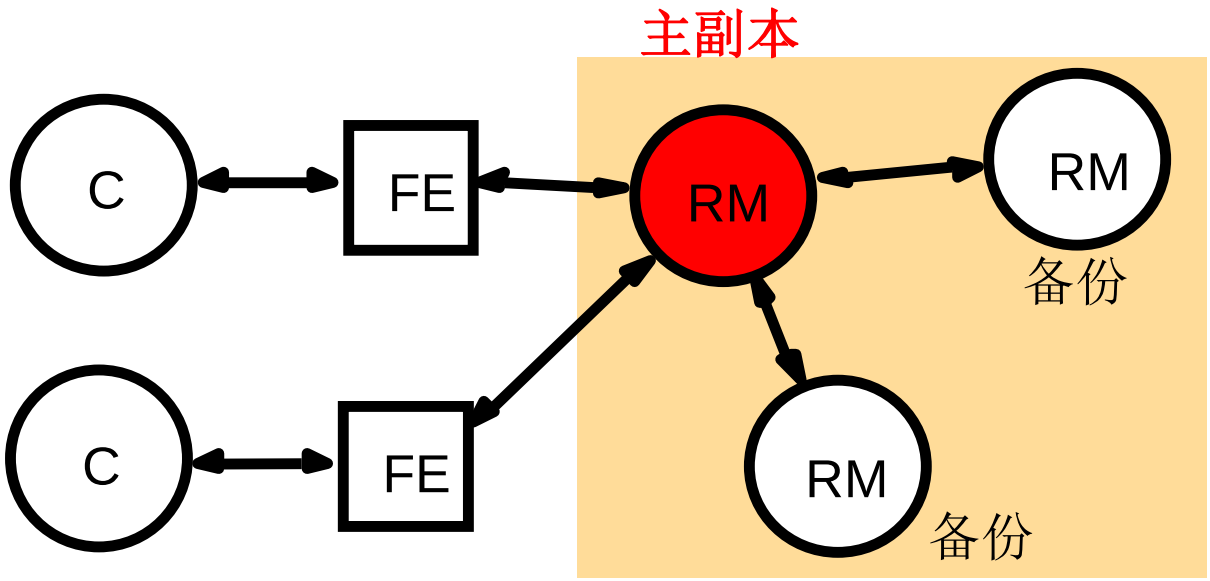
# 容错服务

## 被动(主备份)复制（支持可线性化的技术）

- 一个主副本管理器＋多个次副本管理器

  - 若主副本管理器出现故障，则某个备份副本管理器将提升为主副本管理器。

- 模型



# 容错服务

## 被动复制时的事件次序

1. 请求：前端将请求发送给主副本管理器
2. 协调：主副本管理器按接收次序对请求排序
3. 执行：主副本管理器执行请求并存储响应
4. 协定
   - 若请求为更新操作，则主副本管理器向每个备份副本管理器发送更新后的状态、响应和唯一标识符。
   - 备份副本管理器返回确认。
5. 响应
   - 主副本管理器将响应发送给前端
   - 前端将响应发送给客户

# 顺序一致性（概念）

- 可线性化对实时性要求过高，更多的时候要求顺序一致性。
- 一个被复制的共享对象服务被称为顺序一致性，满足以下两个准则：
  - 操作的交错执行序列符合对象单个正确副本所遵循的规约
  - 操作在交错执行中的次序和每个客户程序中执行的次序一致。

# 顺序一致性（翻译）

- 系统应该造成单个副本的错觉
- 无论哪个客户，读都会返回最近写的结果。
- 无论哪个客户，所有后续读都应返回相同的结果，直到下一次写。

- “最近”&“所有后续”
  - 同客户的操作：由时间决定（程序的次序）
  - 跨客户的操作：不由时间决定
    - 可以重新排序
    - 只需要保留每个客户中程序的次序

# 可线性化和顺序一致性

- 相同点
  - 两者都希望造成单个副本的错觉
    - 从外部观察者来看，系统好像只有单个副本
- 不同点
  - 可线性化关心时间
  - 顺序一致性关心程序的次序

# 可线性化和顺序一致性

- 可线性化
  - 所有客户之间的交错操作，几乎已经根据时间确定了。

- 顺序一致性
  - 系统可以自由选择如何交错来自不同客户的操作，只要保留每个客户的顺序即可

# 可线性化和顺序一致性（1）

| 客户1 | 客户2 |
| --- | --- |
| $setBalance_B(x,1)$ | |
| | $getBalance_A(y)$->2 |
| | $getBalance_A(x)$ ->0 |
| $setBalance_A(y,2)$ | |

不满足可线性化（不满足实时性）

不满足顺序一致性

# 可线性化和顺序一致性（2）

| 客户1 | 客户2 |
| --- | --- |
| $setBalance_B(x,1)$ | |
| | $getBalance_A(y)$->0 |
| | $getBalance_A(x)$ ->0 |
| $setBalance_A(y,2)$ | |

存在一种顺序满足顺序一致性

$getBalance_A(y)$->0，$getBalance_A(x)$ ->0， $setBalance_B(x,1)$， $setBalance_A(y,2)$

不满足可线性化（不满足实时性）

# 线性化能力和顺序一致性（3）

| 客户1 | 客户2 |
| --- | --- |
| $setBalance_B(x,1)$ | |
| | $getBalance_A(y)$->0 |
| | $getBalance_A(x)$ ->1 |
| $setBalance_A(y,2)$ | |

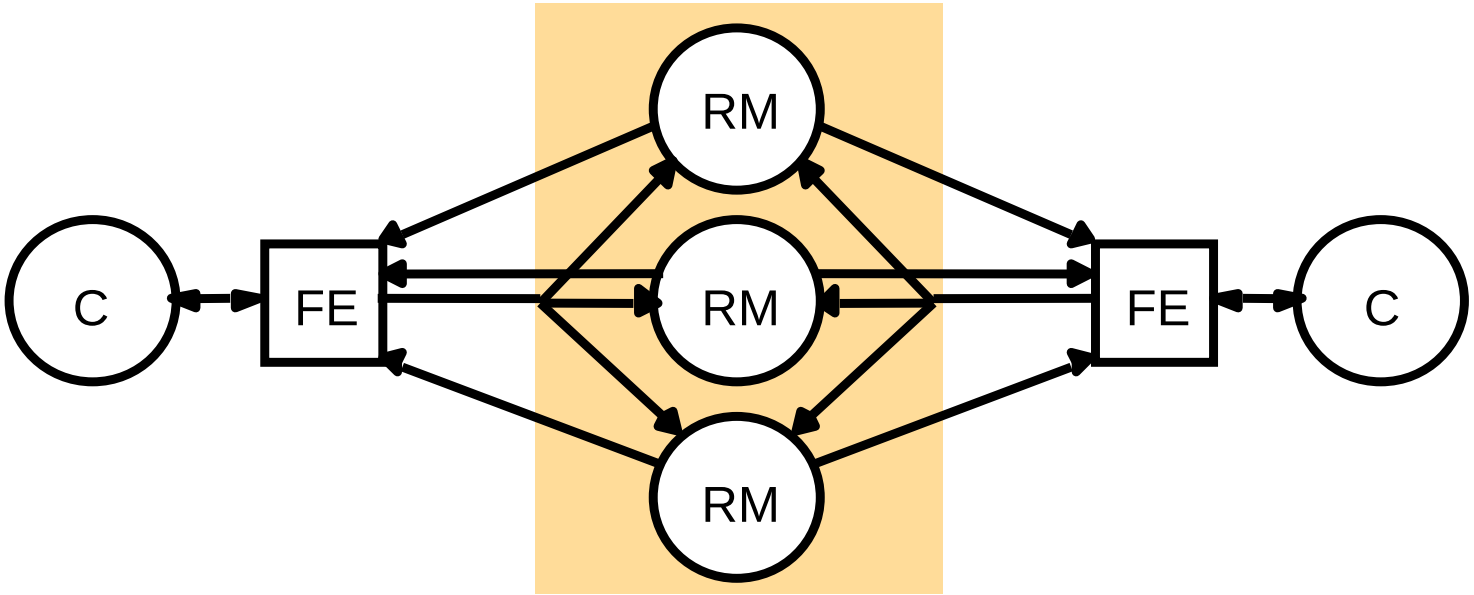
满足顺序一致性

满足线性化能力

# 容错服务

主动复制（支持顺序一致性的技术）

- 副本管理器地位对等，前端组播消息至副本管理器组
- 模型



# 容错服务

## 主动复制时的事件次序

1. 请求：前端使用全序、可靠的组播原语将请求组播到副本管理器组
2. 协调：组通信系统以同样的次序(全序)将请求传递到每个副本管理器
3. 执行：每个副本管理器以相同的方式执行请求
4. 协定：鉴于组播的传递语义，不需要该阶段
5. 响应
   - 每个副本管理器将响应发送给前端
   - 前端将响应发送给客户

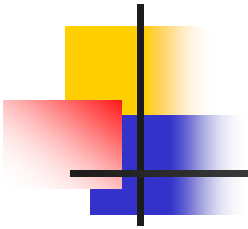
# 可线性化和顺序一致性

- 可线性化
  - 操作的次序由时间决定
  - 链式复制可以提供可线性化
  - 被动复制（主备份）可以提供可线性化
- 顺序一致性
  - 每个客户保留程序的次序
  - 主动复制可以提供顺序一致性

# 两种更弱的一致性

- 可线性化和顺序一致性
  - 单个副本的错觉
- 更弱的一致性
  - 甚至不关心单个副本的错觉

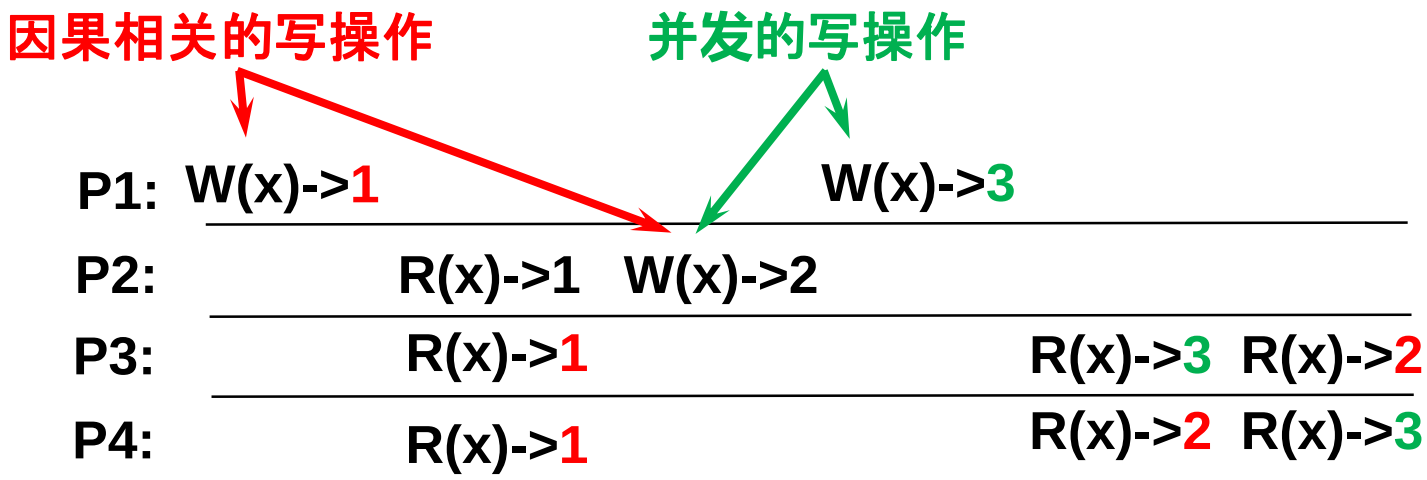
- 因果一致性
  - 正确排序因果相关的写操作。
- 最终一致性
  - 只要所有的副本最终都收敛到同一个副本



# 因果一致性

- 所有进程必须以相同的次序看到因果相关的写操作
- 不同进程可能会以不同的次序看到并发的写操作
  - 弱于顺序一致性
- 如何定义两次写操作之间的“因果关系”？
  - 一个客户读了另一个客户写的数据；然后这个客户又写了数据。

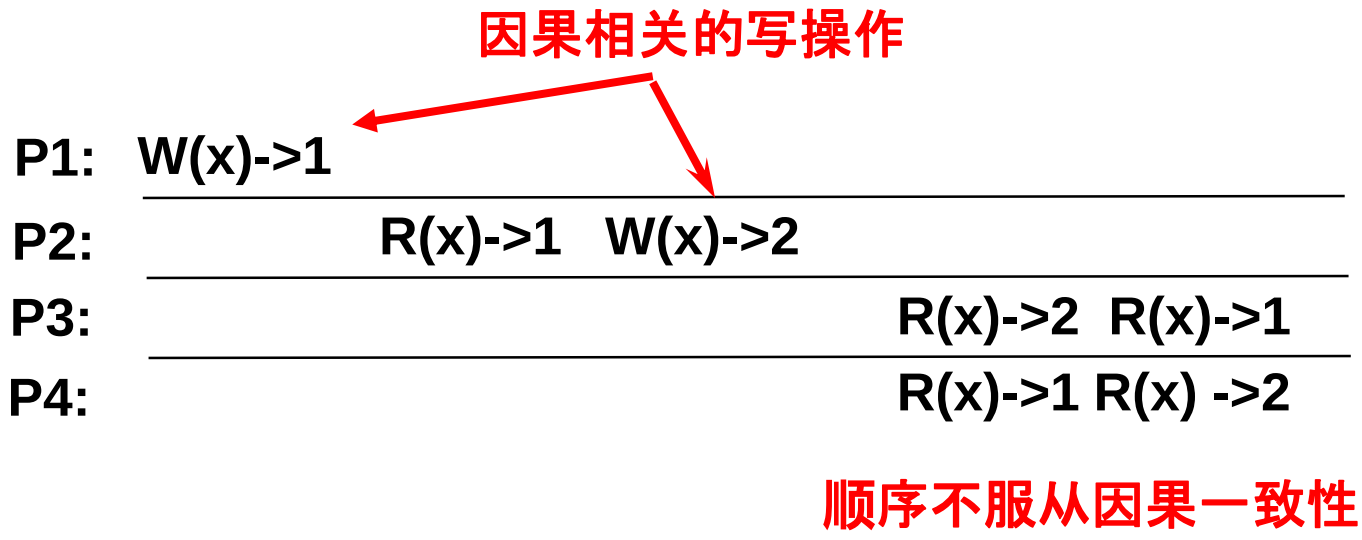
# 案例1




顺序服从因果一致性

# 案例2

- 是否是因果一致性？



# 案例3

- 是否是因果一致性？

| | | | | |
|---|---|---|---|---|
| P1: | W(x)->1 | | | |
| P2: | | W(x)->2 | | |
| P3: | | | R(x)->2 | R(x)->1 |
| P4: | | | R(x)->1 | R(x)->2 |

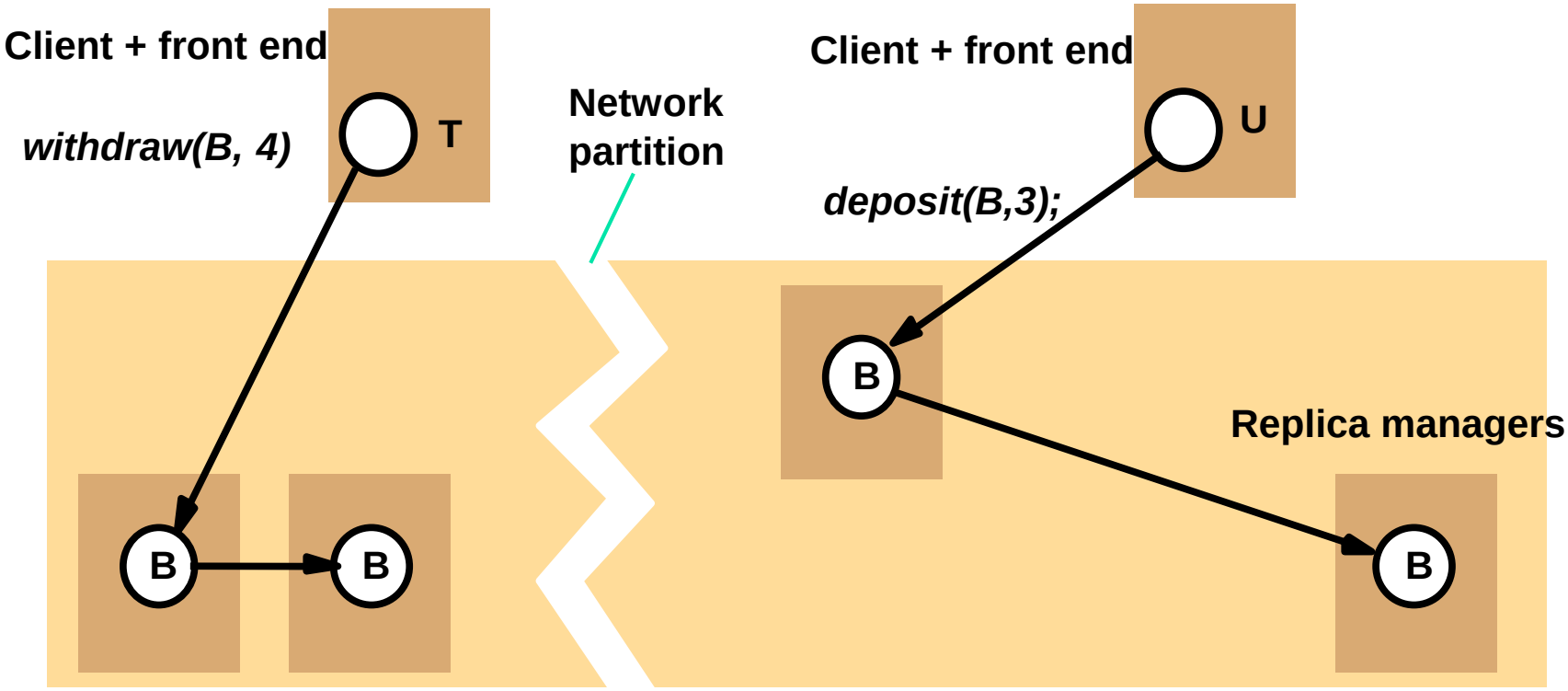
顺序服从因果一致性

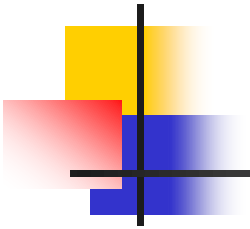
# 最终一致性

## 网络分区

- 网络分区将一个副本管理器组分为两个或更多的子组
- 一个子组的成员可相互通信，不同子组的成员不能通信

- 如果没有发生分区，相互冲突的两个事务之一将放弃
- 当分区存在时，冲突事务已经被允许在不同分区中提交





# 困境

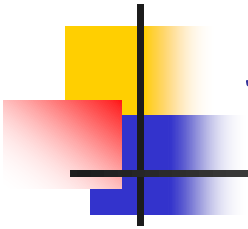
在存在网络分区的情况下：

- 为了保持副本的一致，系统需要阻塞
  - 从外部观察，系统似乎不可用
- 如果仍为不同分区的请求提供服务，那么副本将出现差异
  - 系统可用，但没有一致性

# CAP原理

- Consistency（一致性）
  - 读最新的写
- Availability（可用性）
  - 以合理的延迟响应
- Partition（分区）
  - 网络分区的情况

- 在P的情况下，只能选C或者A，即2选1



# 最终一致性

- 副本管理器只根据它们局部的状态处理操作
- 如果没有更多的更新，最终所有的副本管理器会处于相同的状态（并不保证多长时间可以达到这种状态）

# 第7章 复制

- 简介
- 系统模型
- 容错服务
- 复制数据上的事务
- 高可用服务的实例研究：gossip体系结构、Bayou和Coda
- 小结

# 复制的正确性

- 在没有复制的系统中，事务以某种次序逐一执行
  - 通过确保事务的交错操作串行等价实现
- 如何在复制中实现类似的目标？
- 单拷贝可串行化：客户在复制对象上执行的事务的效果应该与它们在一个对象集上逐一执行相同（即每个对象 1 个副本）
  - 相当于结合串行等价+复制透明性/一致性

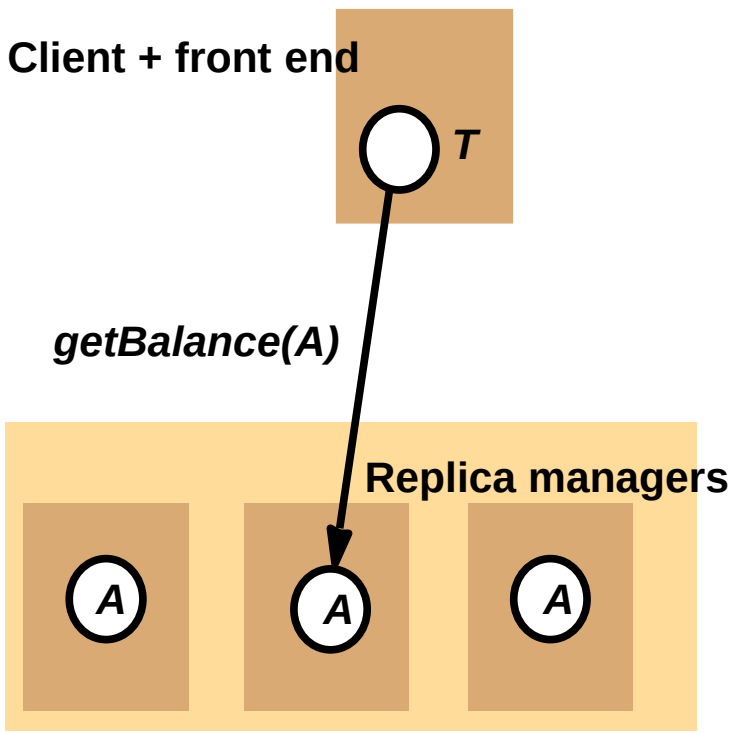
# 复制数据上的事务

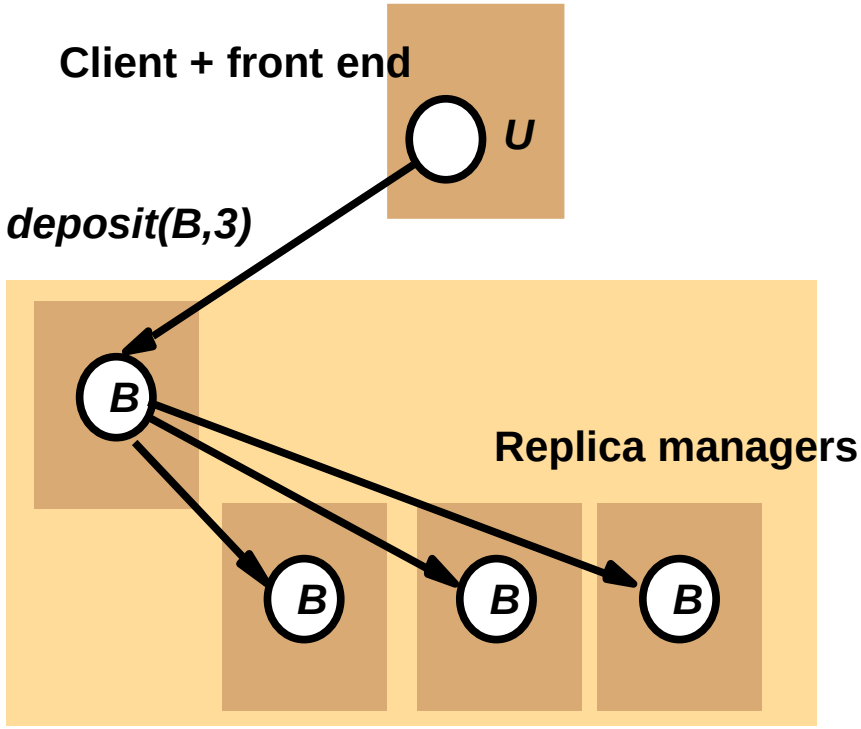
- 对客户而言，有复制对象的事务看上去应该和没有复制对象的事务一样
- 作用于复制对象的事务应该和它们在一个对象集上的一次执行具有一样的效果，这种性质叫做单拷贝串行化
- 单拷贝串行化可以通过“读一个/写所有”实现复制方案

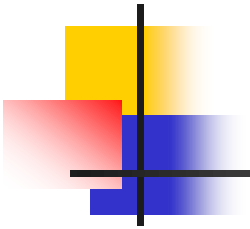
**Read** 请求可以由单个副本管理器执行

**Write** 请求必须由组中所有副本管理器执行

# 复制数据上的事务





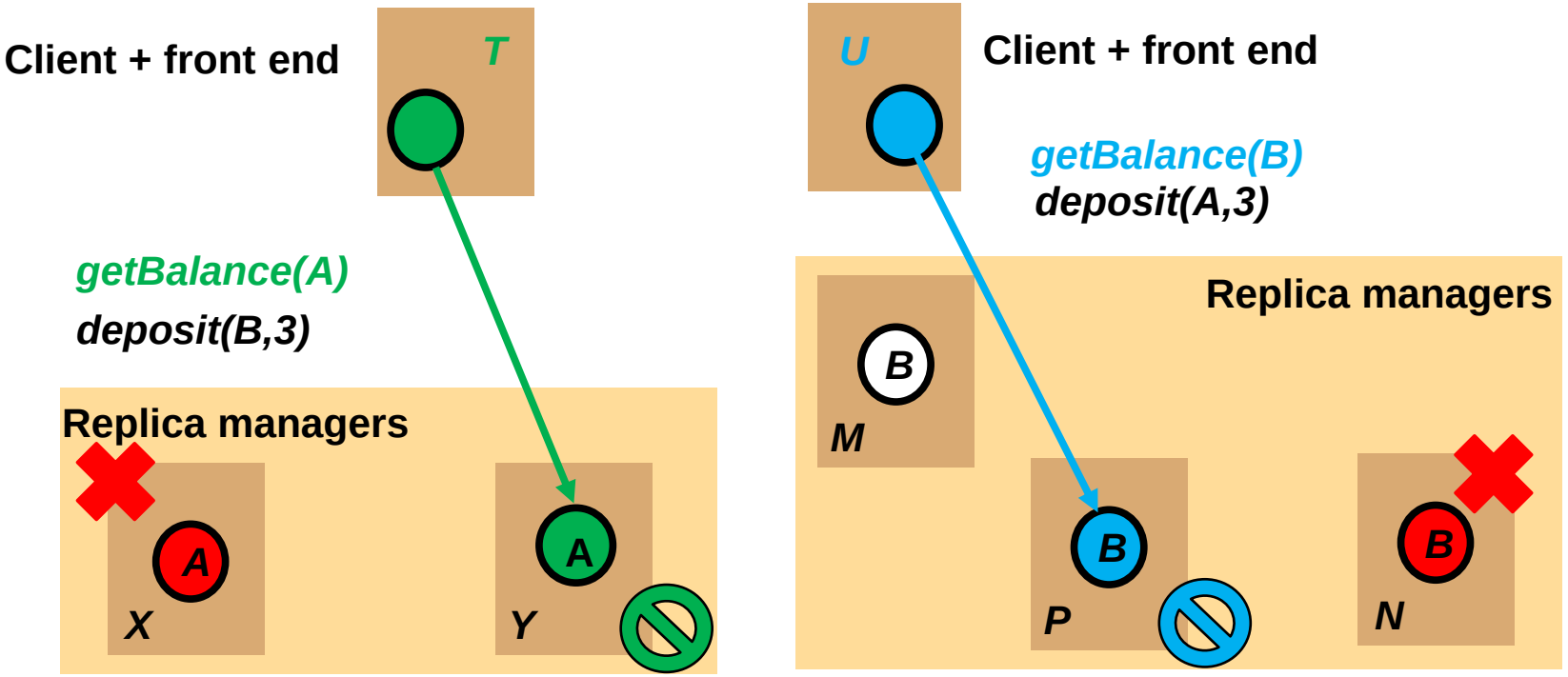
# 复制方案

- 服务器崩溃
  - 可用拷贝复制
- 网络分区
  - 法定数共识
  - 虚拟分区

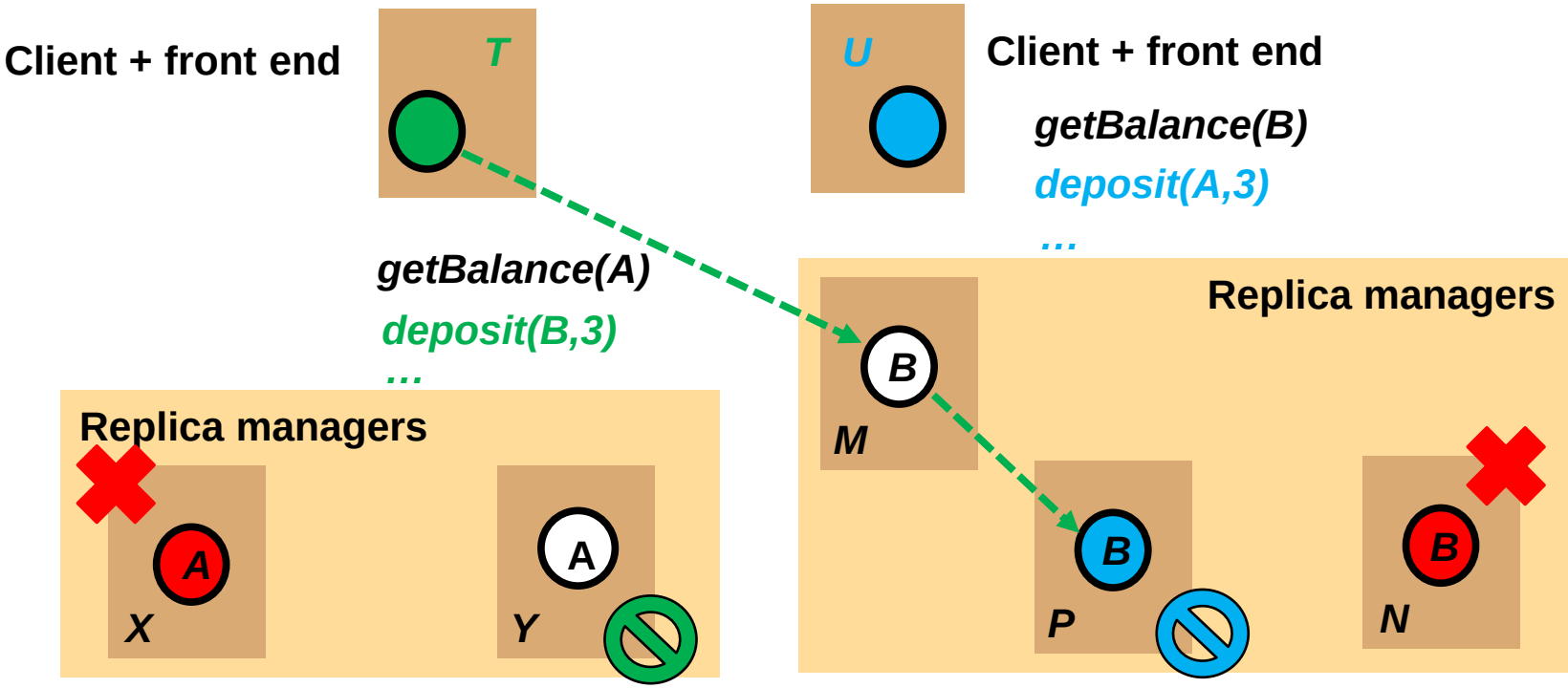
# 可用拷贝复制

- 读一个/写所有在副本管理器崩溃或者通讯故障时不是一个现实的方案（无法写所有）
- 可用副本复制方案允许某些副本管理器暂时不可用
  - 读请求可由任何可用的副本管理器执行，而写请求必须由所有可用副本管理器执行
  - 只要可用的副本管理器集没有变化（前提），本地的并发控制和读一个/写所有复制一样可获得单拷贝串行化，但如果执行过程中副本管理器故障或恢复，需要额外验证

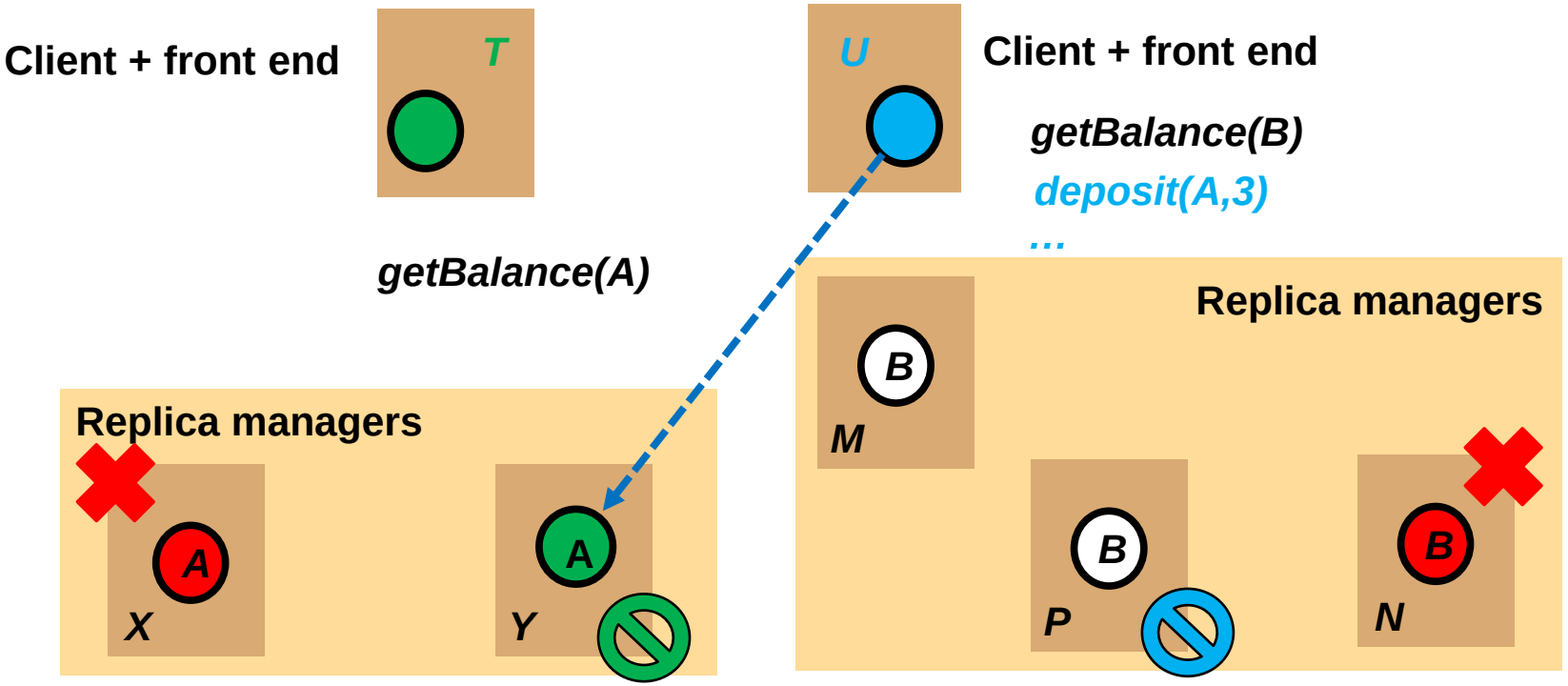
# 副本管理器集在事务执行中无变化
# 读一个（T对A加锁，U对B加锁）

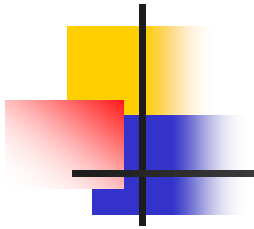


# 副本管理器集在事务执行中无变化写所有（T）



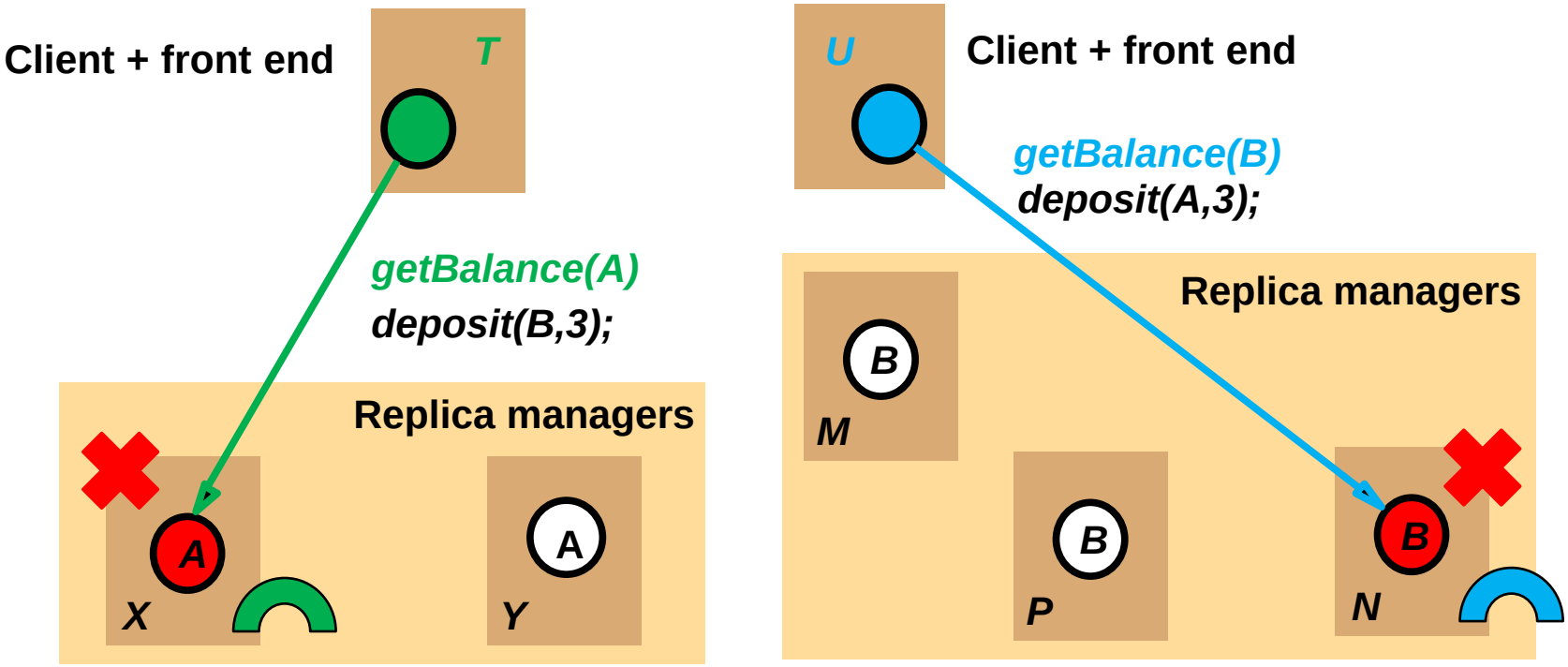
# 副本管理器集在事务执行中无变化
# 写所有 （U）





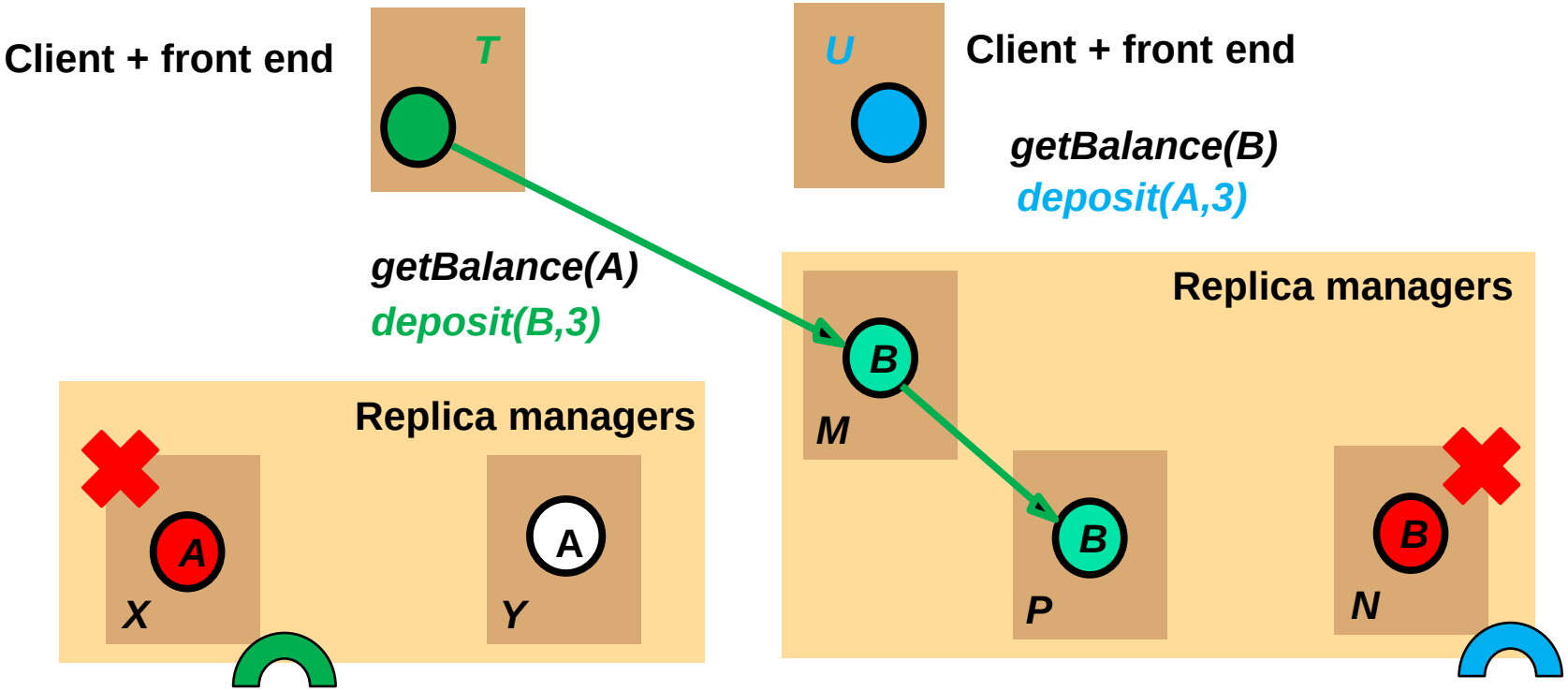
| T | U |
| --- | --- |
| *getBalance(A)　T锁住A* | |
| | *getBalance(B) U锁住B* |
| *deposit(B,3)　等待U在B上的锁* | |
| ... | *deposit(A,3) 等待T在A上的锁* |
| | … |

# 副本管理器集在事务执行中变化
## X在T读后崩溃，N在U读后崩溃




T在getBalance(A)和U在getBalance(B)后，X和N崩溃

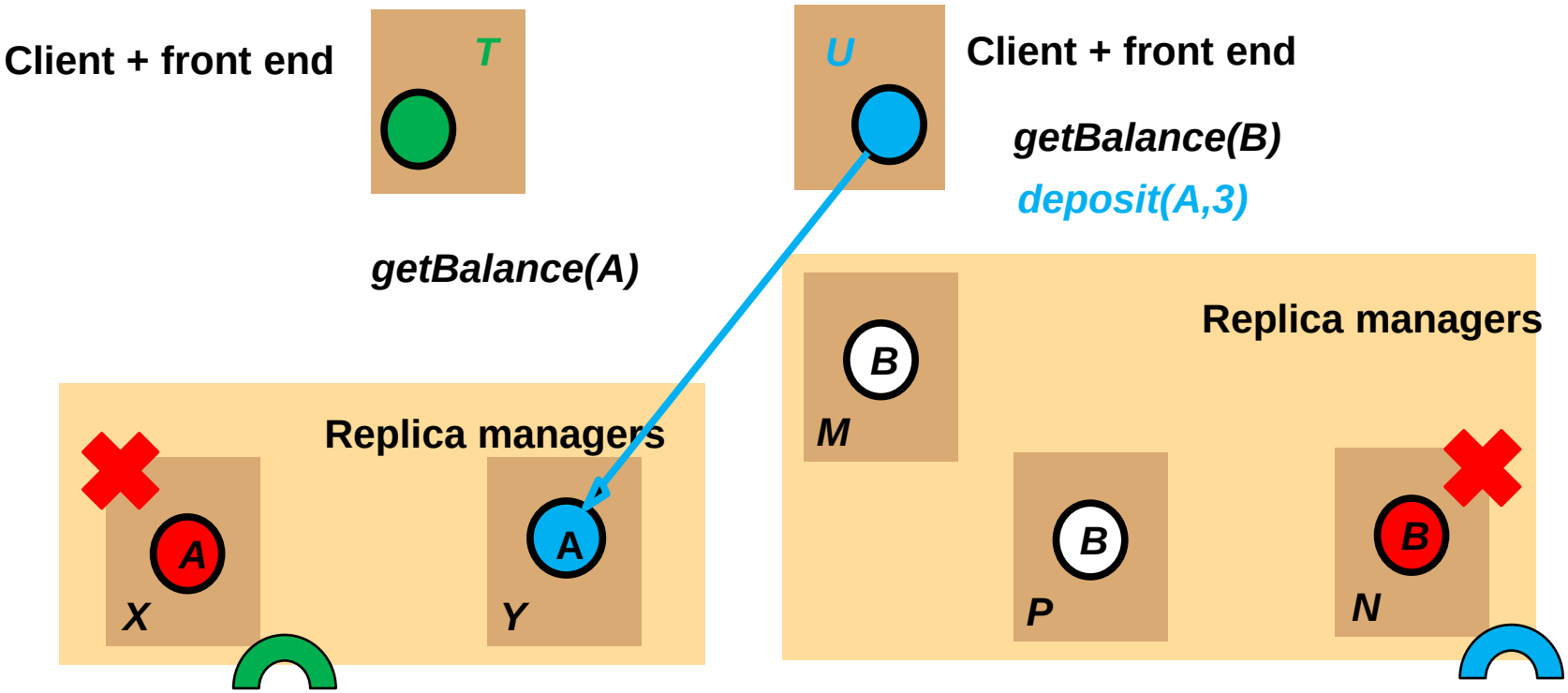
# 副本管理器集在事务执行中变化
## N失效，T可以继续在M和P上更新B




T的deposit(B,3)会在M和P上执行

# 副本管理器集在事务执行中变化

## X失效，U可以继续在Y上更新A




U的deposit(A,3)会在Y上执行

# 不足以保证单副本串行化

## T在U之前执行，或T在U后执行

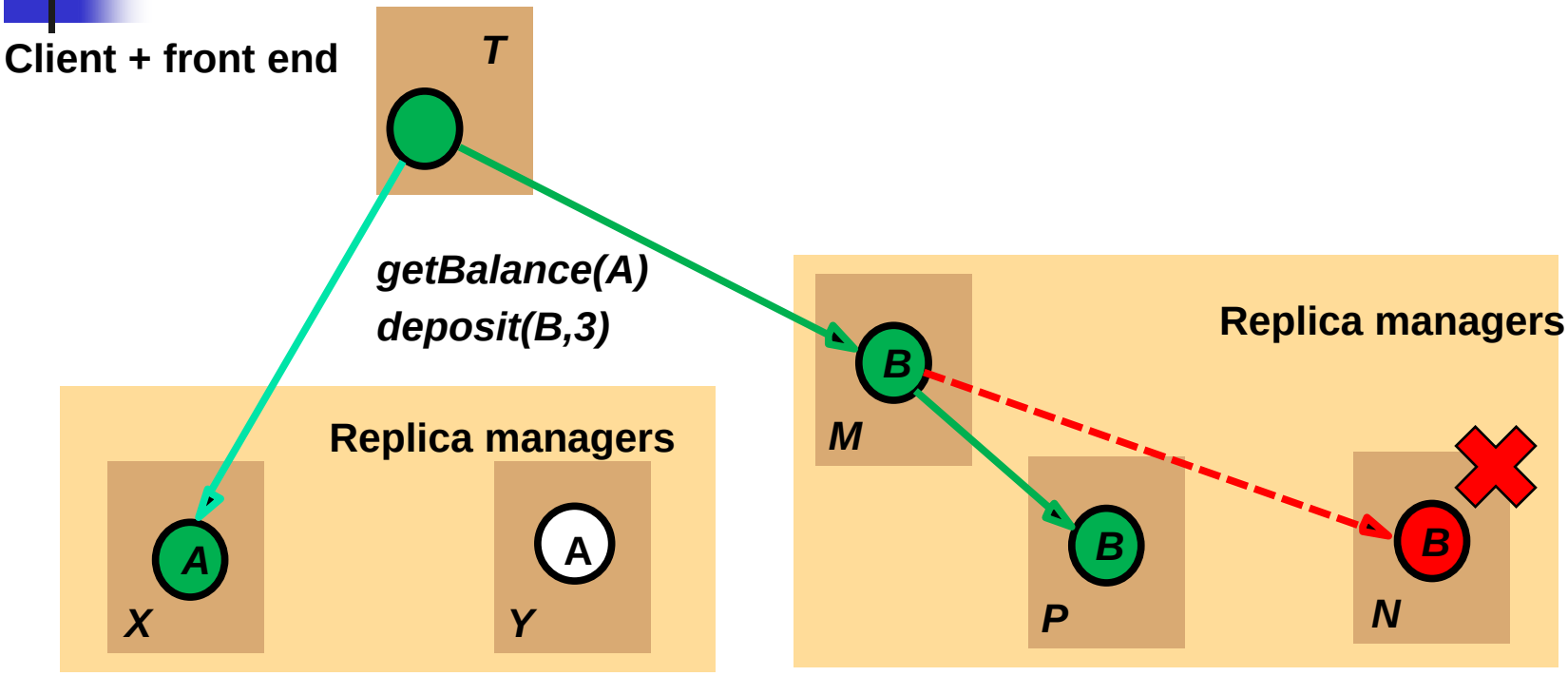
## 需要额外的并发控制

| T | U |
|---|---|
| *getBalance(A) T锁住A [X]* | |
| X崩溃（X上的锁失效） | *getBalance(B) U锁住B [N]* |
| *deposit(B,3) T锁住B* | N崩溃（N上的锁失效） |
| *closeTransaction 对B解锁* | *deposit(A,3) U锁住A* |
| | *closeTransaction 对A解锁* |

**对象A：先T后U**
**对象B：先U后T**

# 本地验证

## 如果T提交，从T的角度看（T是一个不可分割的整体）

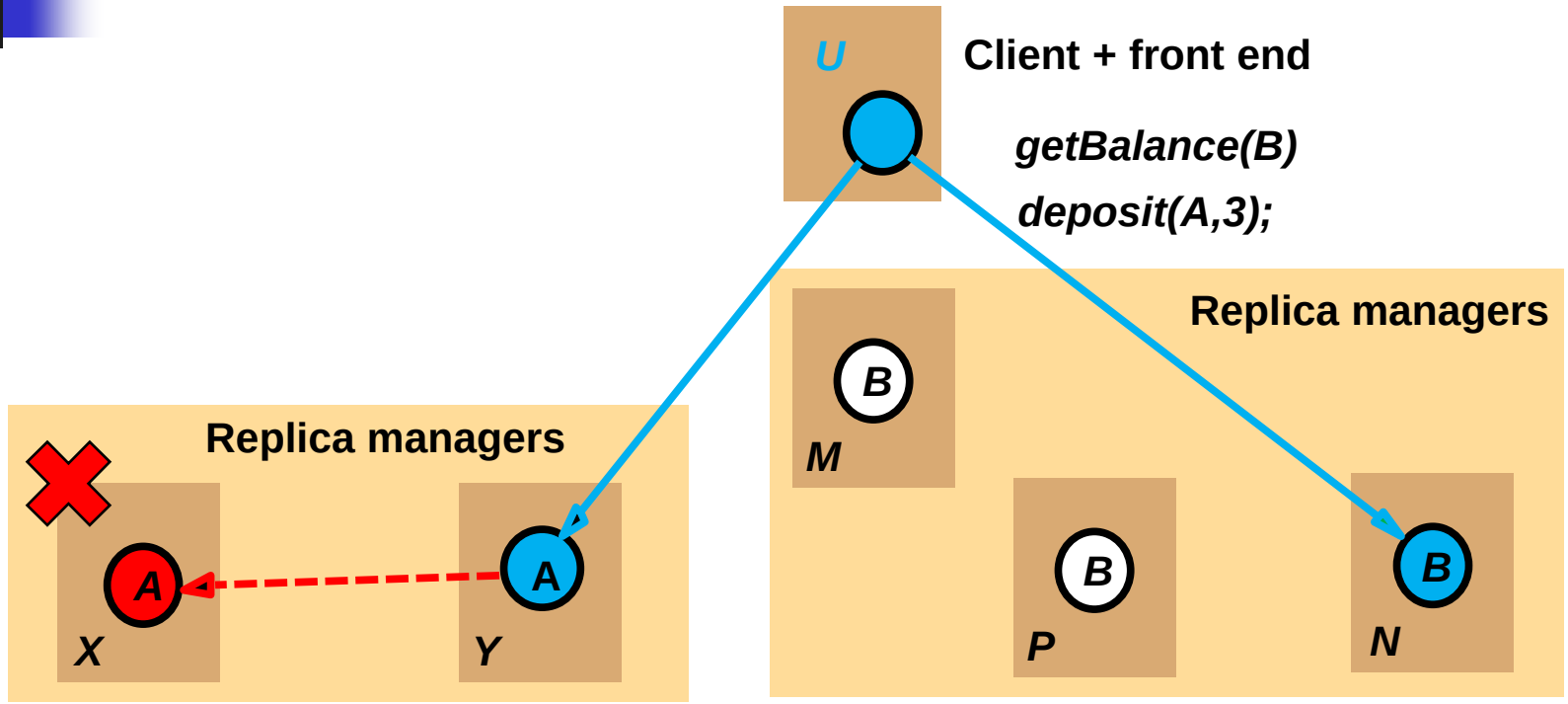



T对X进行read操作 → X故障在T完成后

T尝试对N进行write操作出了故障 → N故障在T之前

N出故障 → T在X上读对象A；T在M和P上写对象B → T提交 → X出故障

# 如果U提交，从U的角度看（U是一个不可分割的整体）




U对N进行read操作 → N故障在U完成后

U尝试对X进行write操作出了故障 → X故障在U之前

X出故障 → U在N上读对象B；U在Y上写对象A → U提交 → N出故障

# 本地验证

- N故障→T在X上读对象A；T在M和P上写对象B→T提交→ X故障
- X故障→U在N上读对象B；U在Y上写对象A→U提交→ N故障

  两个不相容的序列同时发生

- 本地验证：确保任何故障或恢复不会在事务执行中发生
- 事务提交前检查已访问的副本管理器集是否变化（故障和恢复）
  - T可以提交：T写N故障，T验证N未恢复→副本管理器集无变化

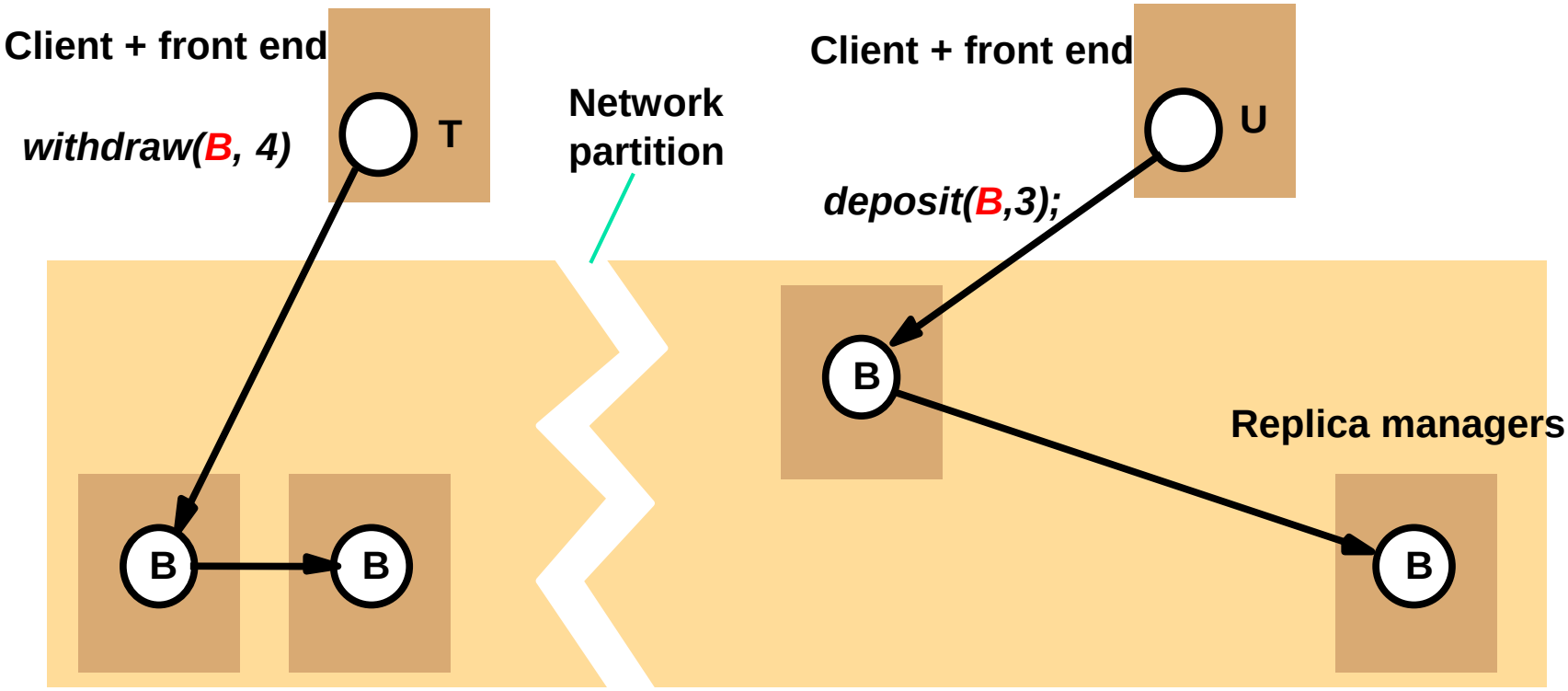
    N故障→T验证
  - U验证失败： U读N可用，U验证前N已故障→副本管理器集变化

    T读X可用→T验证→U写X故障→U验证（X故障在T验证后U验证前）

    N故障→T验证→U验证

    N可用→ N故障→U验证

# 网络分区


Client + front end
withdraw(B, 4)
T
Network
partition
Client + front end
U
deposit(B,3);
B
B
B
B
Replica managers

# 处理网络分区

- 假设分区最后会被修复
- **乐观方法**：可能冲突的事务分别在分区中进行，等网络分区修复后，再进行修正工作。
- **悲观方法**：终止一个分区中的事务；或等待网络分区修复后再进行事务的提交。

# 法定数共识方法

- 一种阻止分区中事务产生不一致的方法：使用法定数法，法定数是一个副本管理器子组，它的大小使它具有执行操作的权利。
  - 例如，大多数成员是一个标准，因为其它子组不会拥有大多数成员
- 一个逻辑对象上的更新操作可以成功地被副本管理器组中的一个子组完成。

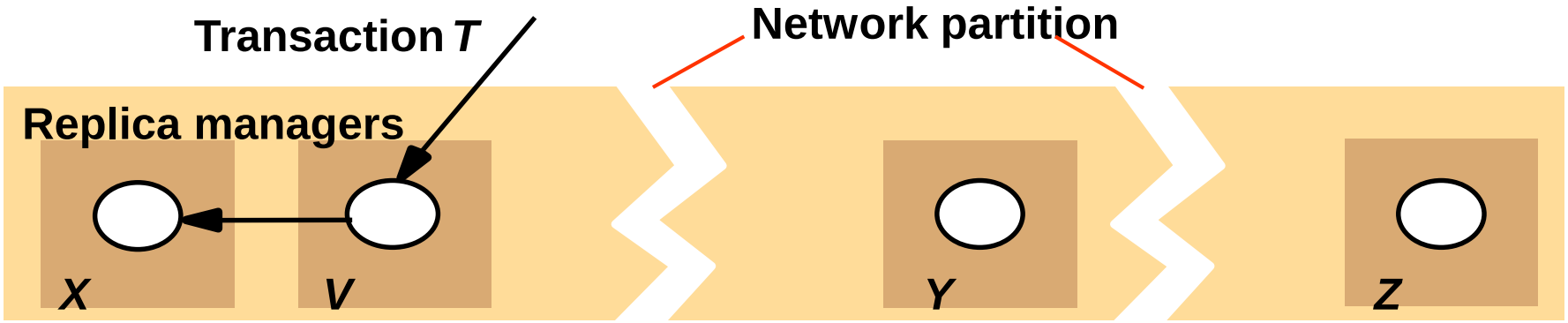
# 法定数共识方法

- 悲观的法定数共识
  - 在分区中有大多数的副本管理器才可以，当分区修复时，则将更新请求传送给其它副本管理器。
  - 只在一个分区进行。
- 乐观的法定数共识
  - 在任何分区都可进行写操作，没有大多数的要求。
  - 任何写操作，如果违背了单拷贝串行化性质，那么该事务将被终止。

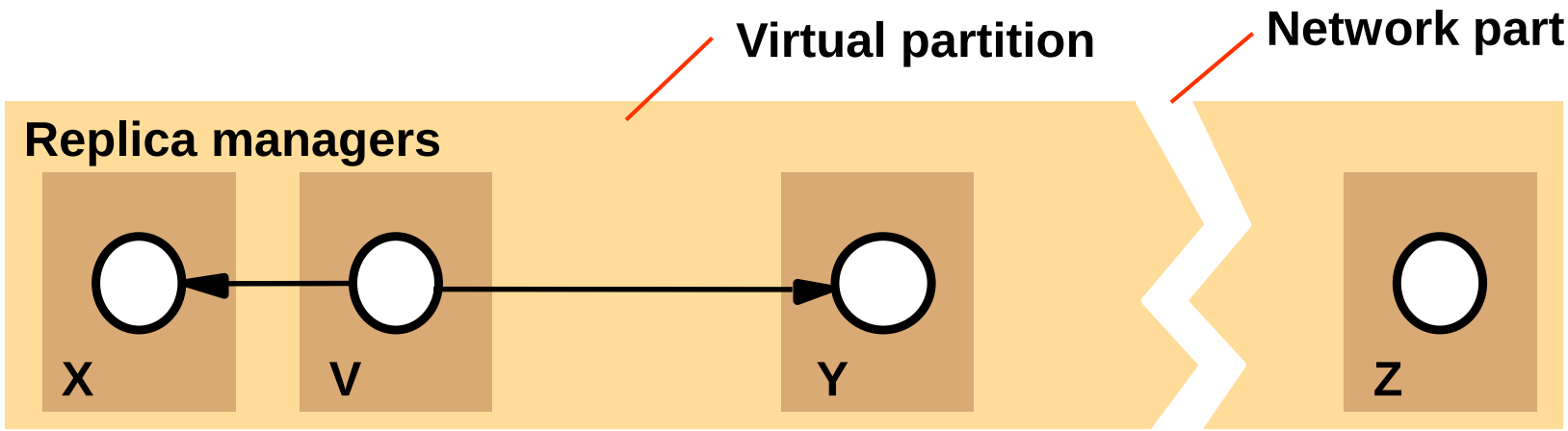
# 虚拟分区算法

- 结合法定数共识和可用拷贝两种算法
  - 虚拟分区＝一组具有读法定数和写法定数的副本管理器
    - w>N/2（一半以上的副本管理器）
    - r+w>N（读集和写集重合，可以读到最新的写）
  - 如果可以形成虚拟分区，则使用可用拷贝算法提高读取性能
  - 如果虚拟分区在事务期间发生变化，则事务被放弃

# 虚拟分区算法



# 虚拟分区算法




当一个副本管理器不能和先前链接的副本管理器相连时，它不断尝试，直到可以创建一个新的虚拟分区为止。

例如，**V**不断尝试连接**Y**和**Z**，直到其中一个或两个回应为止。副本**V**、**X**和**Y**组成一个虚拟分区。

# 第7章 复制

- 简介
- 系统模型
- 容错服务
- 复制数据上的事务
- 高可用服务的实例研究：gossip体系结构、Bayou和Coda
- 小结

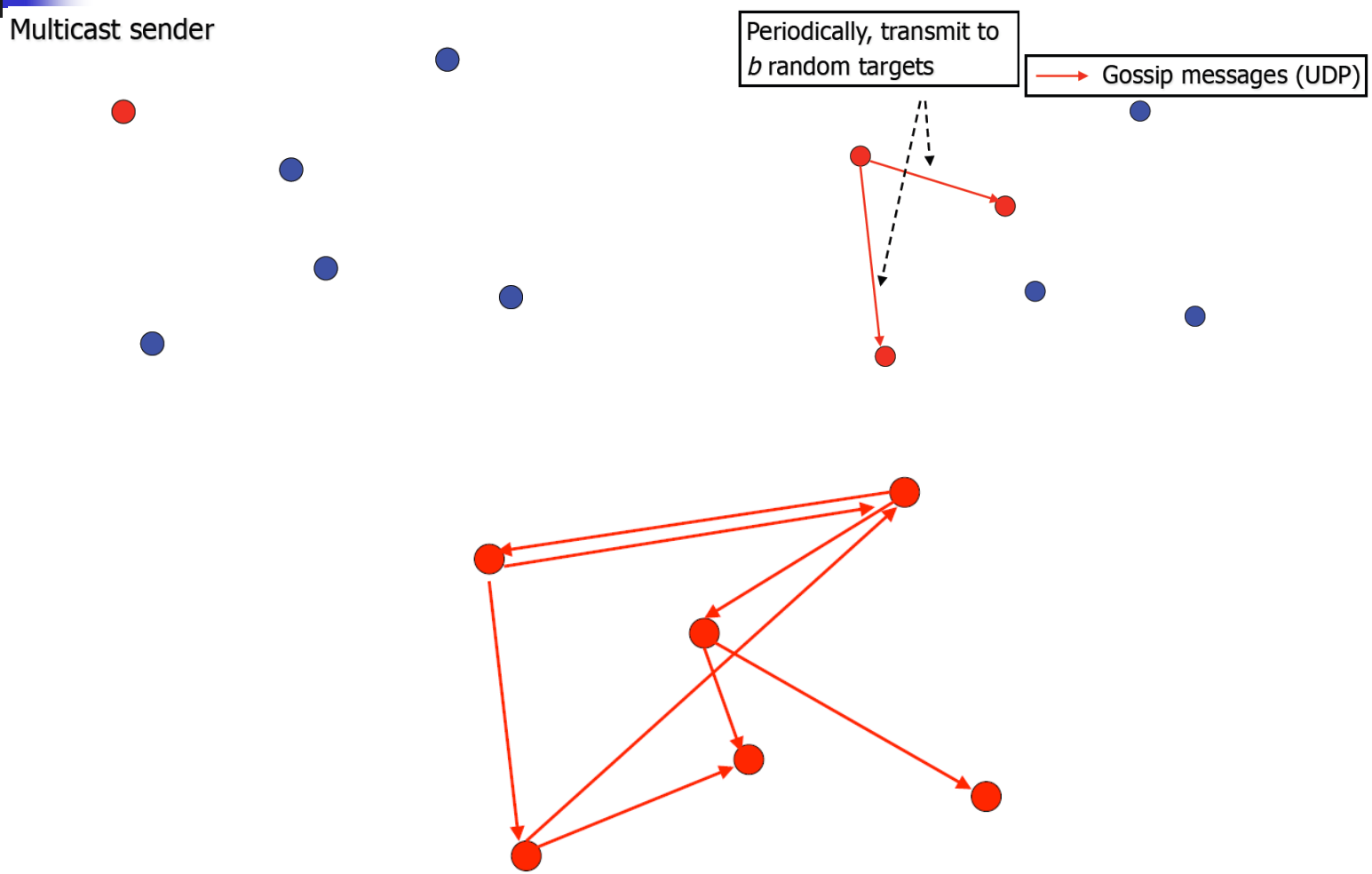
# 高可用服务的实例研究

- 高可用性
  - 系统通过使用最小的与客户连接的副本管理器集合，提供一个可接受级别的服务
  - 当副本管理器协调它们的动作时，应该尽量减少客户的等待时间
  - 采用较弱程度的一致性，提高共享数据的可用性
    - 实例：gossip、Bayou和Coda

# Gossip协议

- Gossip算法又被称为反熵（Anti-Entropy），熵是物理学上的一个概念，代表杂乱无章，而反熵就是在杂乱无章中寻求一致。
- Gossip的特点：在一个有界网络中，每个节点都随机地与其他节点通信，经过一番杂乱无章的通信，最终所有节点的状态都会达成一致。每个节点可能知道所有其他节点，也可能仅知道几个邻居节点，只要这些节点可以通过网络连通，最终他们的状态都是一致的。

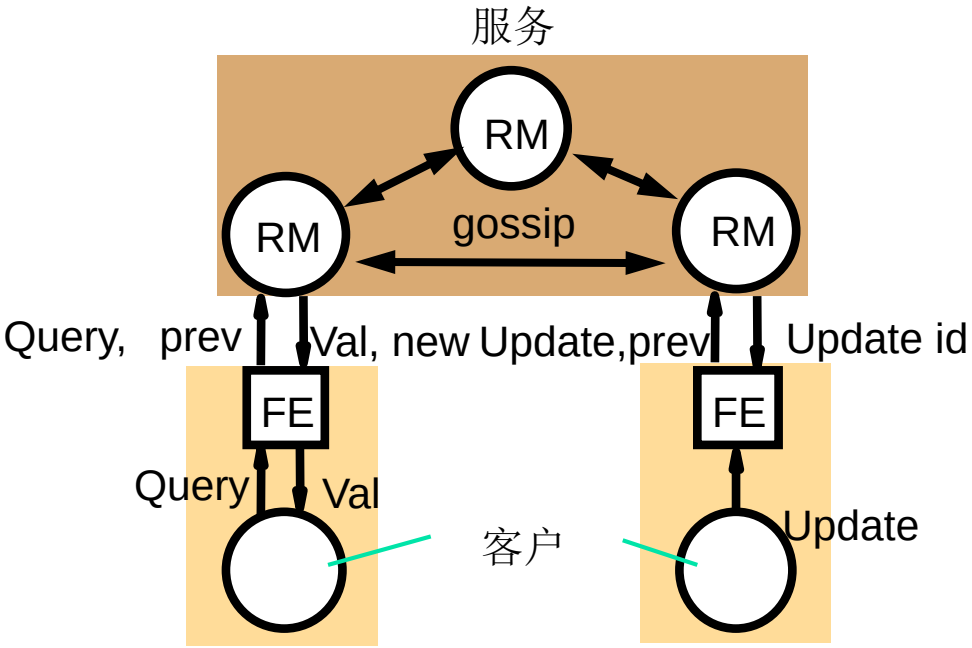
# Gossip协议


Multicast sender
Periodically, transmit to
b random targets
Gossip messages (UDP)

# gossip体系结构

## 体系结构

- 前端可以选择副本管理器（可用且响应时间合理）
- 提供两种基本操作：*查询+更新*
- 副本管理器定期通过gossip消息来传递客户的更新

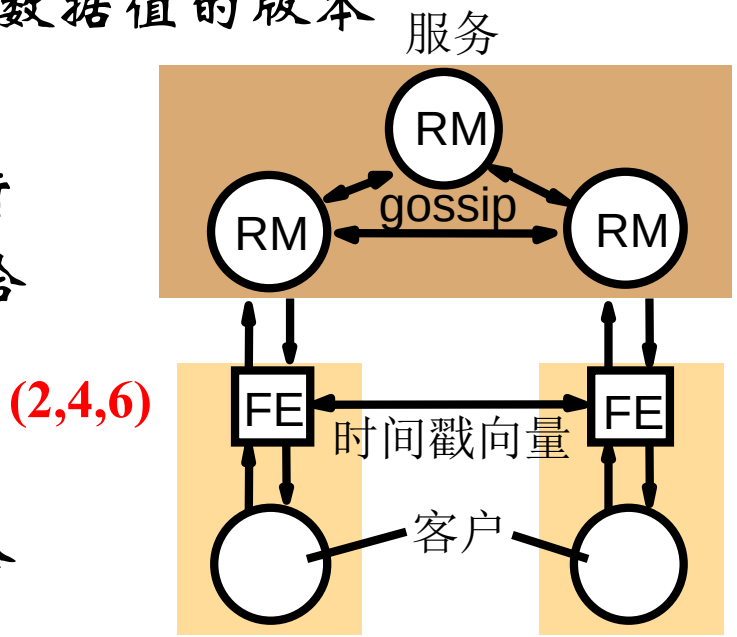


# gossip体系结构

- 前端的版本时间戳
  - 为了控制操作处理次序，每个前端维持了一个向量时间戳，用来反映前端访问的最新数据值的版本

- 将其放入每一个请求中，与更新或者查询操作的描述一起发送给副本管理器

- 前端也可以将时间戳直接发送给其他的前端

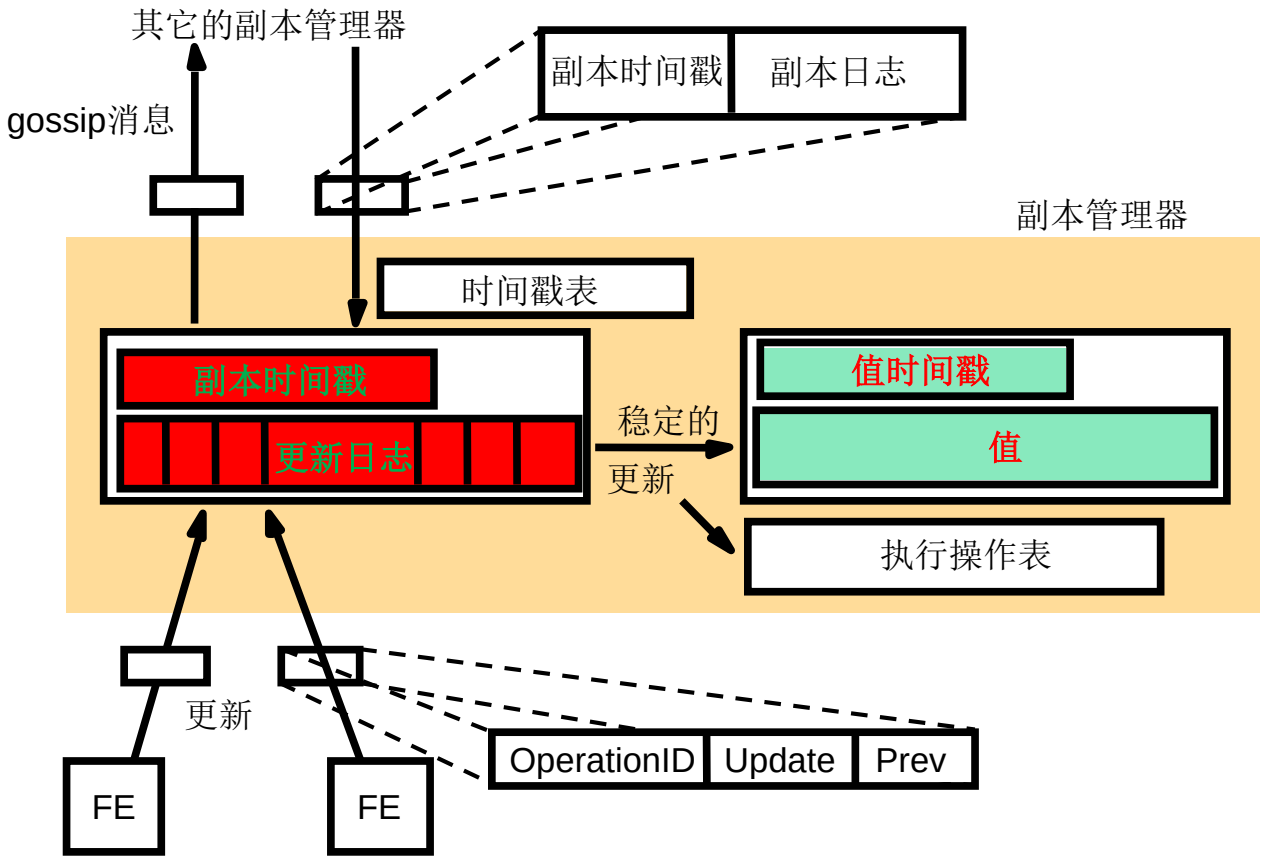


# gossip体系结构

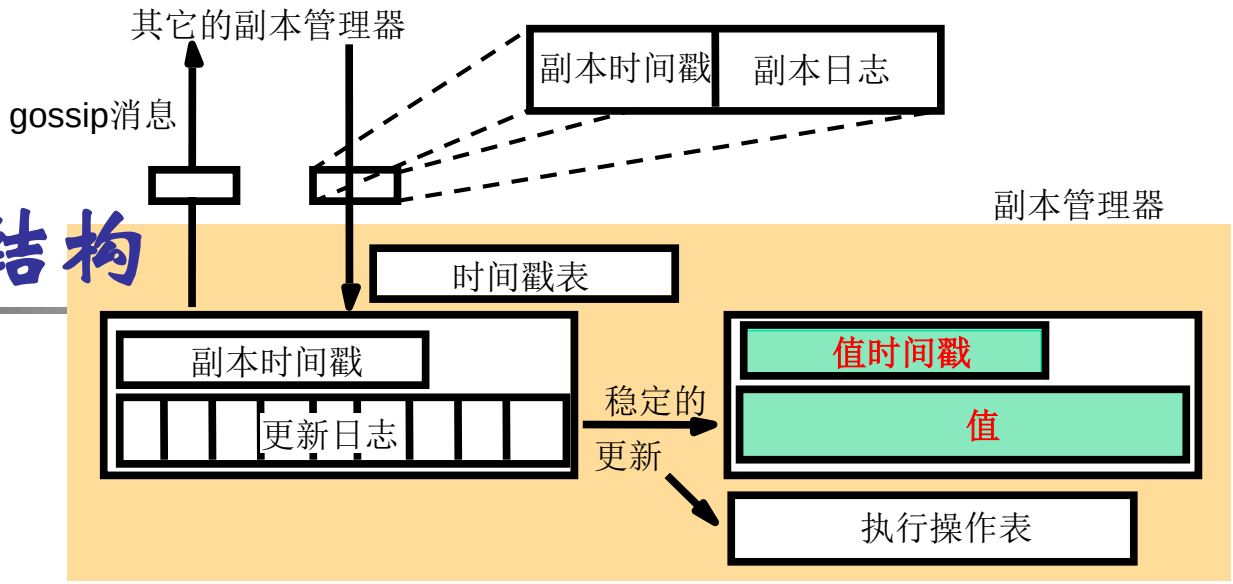
- 前端的版本时间戳(续)
  - 每个前端维护一个向量时间戳
    - 每个副本管理器有一条对应的记录
    - 前端发送更新或查询信息中包含时间戳
    - 副本管理器返回的时间戳与前端时间戳合并，用于记录已经被客户观察到的复制数据版本
  - 向量时间戳的作用
    - 反映前端访问的最新数据值

# gossip体系结构

## 副本管理器状态



# gossip体系结构



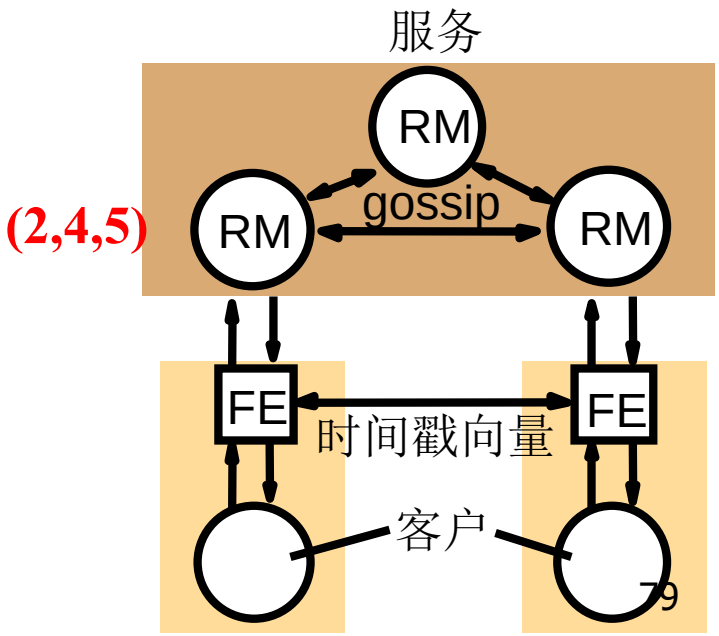

## 副本管理器状态

1. 值：副本管理器维护的应用状态的值
   - 始于特定初始值，状态完全是施加更新结果
2. 值的时间戳：反映在值中更新的向量时间戳
   - 包含了每个副本管理器的条目，每当值更新时被更新
3. 更新日志：记录更新操作（为什么要记录？）
   - 更新操作不稳定（排序保证），不稳定的更新需要保留而不处理
   - 更新是稳定的，但还未收到更新被其他所有副本管理器收到的确认
4. 副本的时间戳：已经被副本服务器接收到的更新（即已经放在管理器日志中的更新）

5. 已执行操作表：记录已经执行的更新的唯一标识符，防止重复执行
6. 时间戳表：时间戳来自gossip消息，确定何时一个更新已经应用于所有的副本管理器

(2,4,5) 代表：

- 从管理器0的前端接收2个更新
- 从管理器1接收到前4个更新
- 从管理器2接收到前5个更新



# gossip体系结构

系统的两个保证

- 随着时间的推移，每个用户总能获得一致服务
  - 副本管理器提供的数据能反映迄今为止客户已经观测到的更新
- 副本之间松弛的一致性
  - 两个客户可能会观察到不同的副本
  - 客户可能观察到过时数据
  - 所有副本管理器最终将收到所有更新

# gossip体系结构

**查询和更新操作流程**

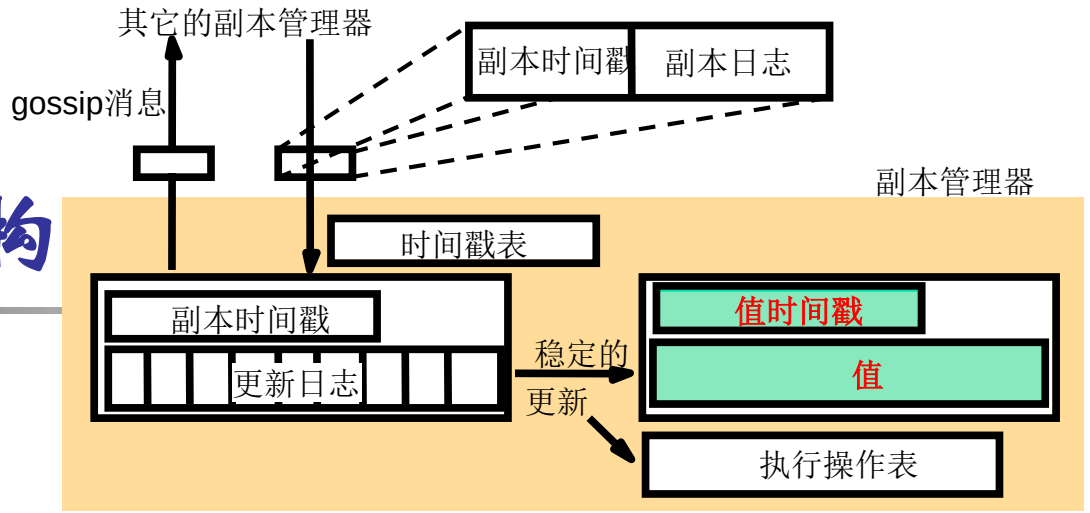
1. 请求：前端将请求发送至副本管理器。
   - 查询：前端阻塞
   - 更新：默认前端立即返回，为提高可靠性，客户可以被阻塞到已经传给f+1个副本管理器后继续执行
2. 对更新的响应：副本管理器只要一收到更新就立即回答
3. 协调：收到请求的副本管理器并不处理操作，直到它能根据所要求的次序约束处理请求为止。这可能涉及接收其他副本管理器以gossip形式发送的更新

# gossip体系结构

## 查询和更新操作流程(续)

4. 执行：副本管理器执行请求
5. 对查询的响应：副本管理器对查询请求作出应答
6. 协定：副本管理器通过交换gossip消息进行相互更新，gossip消息的交换是偶尔的
   - 在收集到若干更新以后；或者发现丢失一个发送到其他副本管理器的更新，而在处理新请求时又需要这个更新时，才会交换gossip消息

# gossip体系结构



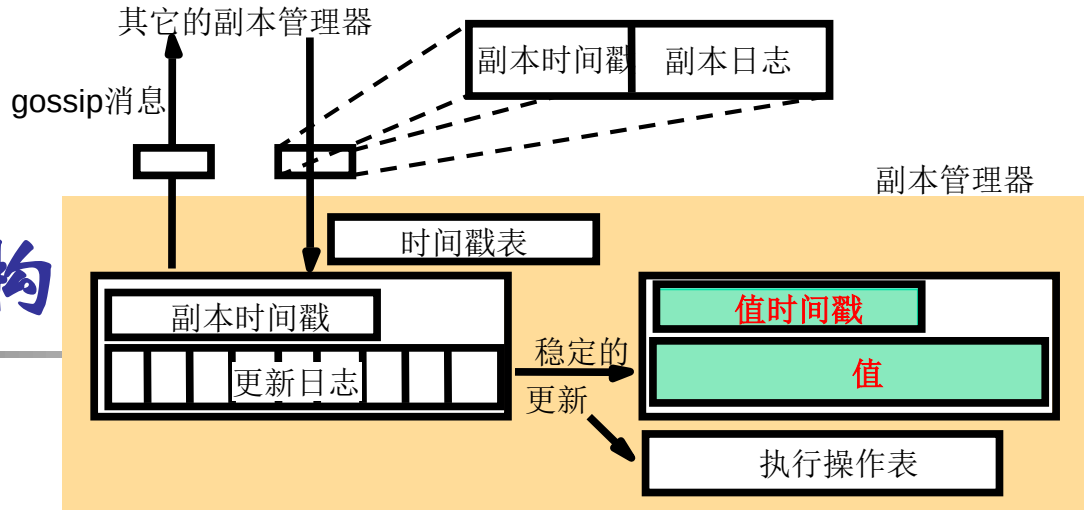

## 查询操作

- 副本管理器收到查询
  - q.pre（前端值的最新值）≤ valueTS（副本管理器的值时间戳）
    - 副本管理器返回消息中的时间戳为valueTS
  - 否则
    - 副本管理器将消息保存到保留队列，直到满足条件

      如：***q.pre(2,4,6), valueTS(2,5,5)*** 等待来自2的更新
- 前端收到查询响应

  - 合并时间戳：frontEndTS:=merge(frontEndTS,new) ***(2,5,6)***，前端没有看到1上的更新

# gossip体系结构




## 按因果次序处理更新

- 前端发送更新请求：(u.op（更新类型和参数），
  u.prev（前端的时间戳），u.id（唯一标识符））
- 副本管理器i接收请求
  - 丢弃：操作已经处理过
  - 否则，将更新记录日志
    - ts =u.prev, ts[i]=ts[i]+1（记录副本管理器i从前端收到的更新个数）
    - logRecord= <i, ts, u.op, u.prev, u.id>
  - 副本管理器将ts返回给前端
    - frontEndTS=merge(frontEndTS, ts)

# gossip体系结构

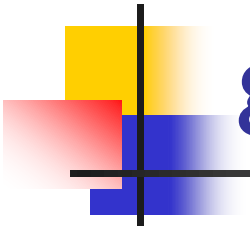
## 按因果次序处理更新(续)

- 更新请求u的稳定性条件

  u.prev ≤ valueTS

- 副本管理器的更新操作，对于一个更新记录r，当稳定条件满足时，副本管理器将更新值、值的时间戳和已执行操作表

  - value（值） := apply(value, r.u.op)
  - valueTS （值的时间戳）:= merge(valueTS, r.ts)
  - executed （已执行表）:= executed∪{r.u.id}



# gossip体系结构

- **gossip消息**
  - 副本管理器可以发送包含一个或多个更新消息的gossip消息，以便使其他副本管理器的状态更新为最新的
  - gossip消息m包含两项：日志m.log和副本时间戳m.ts

# gossip体系结构

- 副本管理器收到gossip消息后执行的操作
  - 日志合并
    - 若r.ts ≤replicsTS（接收者的副本时间戳），则丢弃
    - 否则，将记录加入到日志，合并时间戳

      replicaTS := merge(replicaTS, m.ts)
  - 执行任何以前没有执行并已经稳定的更新
    - 根据向量时间戳的偏序“≤”对已稳定的更新进行排序
  - 若c是创建记录r的副本管理器，c能够丢弃r的要求是
    - 所有的副本管理器i

      tableTS[i][c]（接收者的副本时间表）≥ r.ts[c]

# gossip体系结构

- **更新传播**
  - gossip体系结构未规定具体的更新传播策略
  - 如何选择合适的gossip消息的发送频率？
    - 分钟、小时或天？——*由具体应用需求决定*
  - 如何选择一个副本管理器并与之交换gossip消息（合作者）？
    - 随机策略：使用加权概率来选择
      例如，临近的优于远距离的
    - 确定策略：使用副本管理器状态的函数来选择
      例如，更新中位于最后的那个副本管理器
    - 拓扑策略：将副本管理器安排为一个固定图
      例如，网格、环、树

# gossip体系结构

- **强制的和即时的更新操作（排序保证）**

  强制更新和即时更新需要特殊处理，强制更新是全序加因果序，保证更新的强制次序的基本方法是在与更新相关的时间戳后加入一个唯一的序号，并以这个序号的次序来处理它们。

  高可用服务环境中，通常指派一个主副本管理器，决定排序

# gossip体系结构

- 目标：保证服务的高可用性
  - 即使客户落到一个分区中，只要至少有一个副本管理器正常工作，客户就能获得服务
- 存在问题：
  - 不适合接近实时的更新复制
    - 例如客户参加一个实时会议并更新一个共享文档
  - 可扩展性问题
    - 随着副本管理器数量的增长，需要传递的gossip消息数量和使用的时间戳大小也会增长

# Bayou系统和操作变换方法

- **Bayou系统简介**
  - 与gossisp体系结构和基于时间戳的反熵协议类似
  - 提供的一致性保证弱于顺序一致性（最终一致性）
  - 能够进行冲突检测和冲突解决

    - 操作变换：一个或多个相冲突的操作被取消或改变以解决冲突的过程。

    - 例子

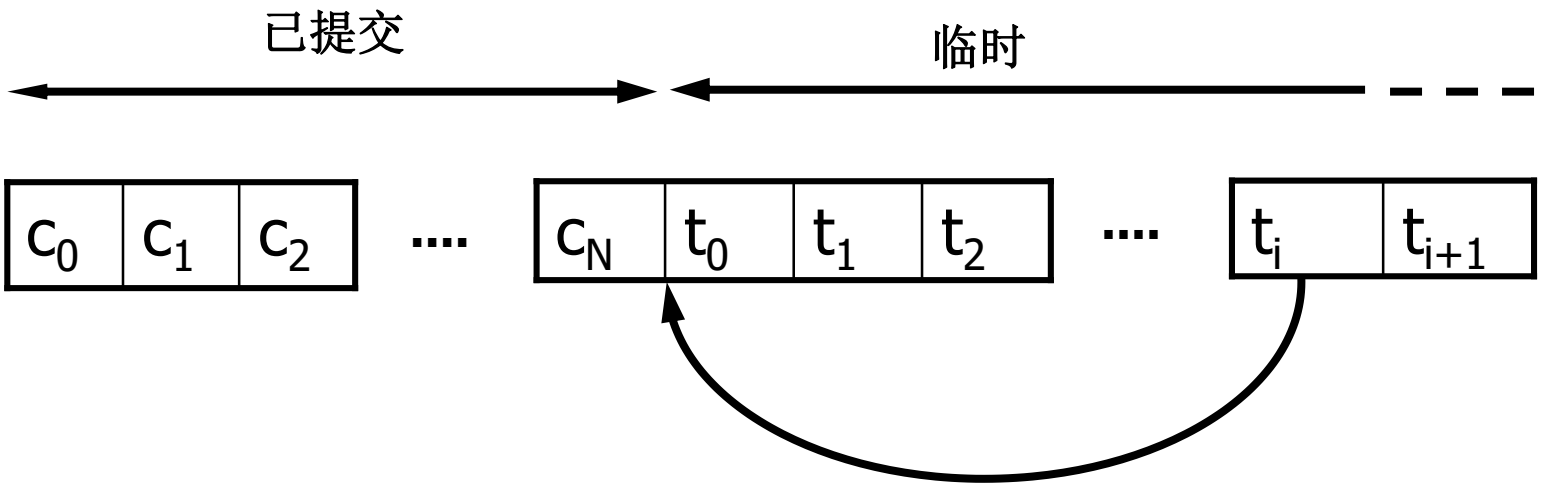
      行政主管和秘书同时预约，其中行政主管为离线更新——行政主管上线后，Bayou系统检测到冲突，然后批准行政主管的预约而取消秘书的预约。

# Bayou系统和操作变换方法

- 临时的更新和提交的更新
  - 临时的更新：更新首次应用于数据库时，被标记为临时的
  - 提交的更新：Bayou将临时的更新以规范次序放置，并添加提交标识
- 数据库副本状态
  - 提交的更新序列＋临时的更新序列
- 更新重排序
  - 新更新到达
  - 某个临时更新被修改为提交的更新

# Bayou系统和操作变换方法

## 提交更新和临时更新示例




临时更新$t_i$成为下一个提交更新，

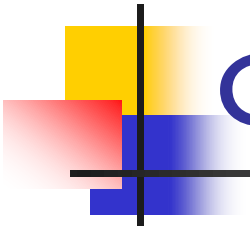
会被插入到最新提交更新$c_N$之后

# Bayou系统和操作变换方法

- 依赖检查和合并过程
  - 依赖检查
    - 一个更新执行时是否会产生冲突
    - 例子：写－写冲突、读－写冲突检测
  - 合并过程
    - 改变将要执行的操作，避免冲突，并获得相似效果
    - 无法合并→系统报错
- Bayou系统讨论
  - 复制对于应用而言是不透明的（最终一致性）
  - 工作的复杂度高（程序员工作和用户工作）

# Coda文件系统

- **AFS的主要缺陷**
  - AFS的缺陷是不能保证高可用性，特别是有断链操作的情况
- **Coda目标**
  - 提供一个共享的文件存储，并且在存储全部或部分不可访问时，可完全依赖本地资源继续操作
- **Coda对AFS的扩展**
  - 采用文件卷复制技术——提高吞吐率和容错性
  - 在客户计算机上缓存文件副本——断链处理



# Coda文件系统

- **Coda体系结构**
  - Venus/Vice进程
    - Venus：前端和副本管理器的混合体
    - Vice：副本管理器
  - 卷存储组(VSG)

    持有一个文件卷副本的服务器集合
  - 可用的卷存储组(AVSG)

    打开一个文件的客户能访问的VSG的某个子集

# Coda文件系统

- **Coda体系结构(续)**
  - 文件访问过程

    - 当前AVSG中的任何一个服务器提供文件服务，并缓存在客户计算机上。

    - 对每个副本管理器进行更新分布。
  - 关闭文件

    - 修改过的拷贝并行广播到AVSG中的所有服务器。

# Coda文件系统

- **Coda体系结构(续)**
  - 设计原则

    - 服务器上的拷贝比客户计算机缓存中的拷贝更可靠
  - 断链操作

    - AVSG（可用的卷存储器）为空时，客户使用缓存文件

    - 断链操作停止并且将缓存文件和服务器上的文件重新整合时，进行重新验证。最坏情况下，手工干预解决冲突

# Coda文件系统

- 复制策略
  - 乐观策略

    - 在网络分区和断链操作期间，仍然可以进行文件修改
  - 实现

    - Coda版本向量(CVV, Code Version Vector )
      - 作为时间戳附加在每个版本的文件上
      - CVV中的每个元素是一个估计值，表示服务器上文件修改次数的估计
      - 目的提供足够的关于每个文件副本的更新历史，以检测出潜在的冲突，进行手工干预和自动更新。

# Coda文件系统

- 复制策略(续)
  - 例如：CVV=[2，2，1]
    - 文件在服务器1上收到2个更新
    - 文件在服务器2上收到2个更新
    - 文件在服务器3上收到1个更新
  - 冲突检测
    - 若一个站点的CVV大于或等于所有其它站点相应的CVV，则不存在冲突→*自动更新* [2，2，1] 和 [1, 1, 1]
    - 若对于两个CVV而言，$v_1 \geq v_2$与$v_2 \geq v_1$均不成立，则存在一个冲突→*手工干预* [2, 2, 1]和[1, 1, 3]

# Coda文件系统

- 复制策略(续.)
  - 文件关闭

    - Venus进程发送更新消息（包括CVV和文件的新内容）到AVSG

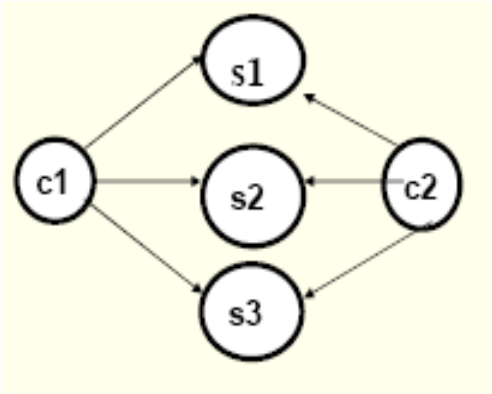
    - AVSG中的每个服务器更新文件，并返回确认。

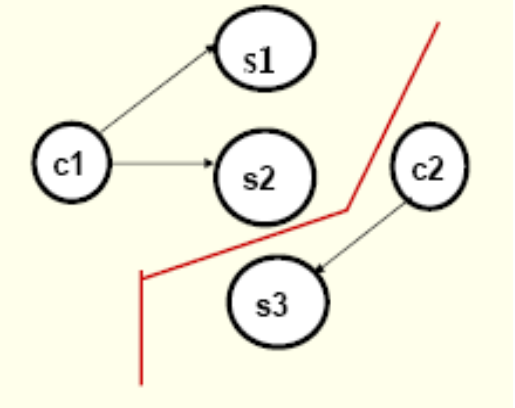
    - Venus计算新的CVV，增加相应服务器的修改记数，并分发新的CVV。

# Coda文件系统

## 构建CVV示例一

- 文件F在三个服务器$S_1$、$S_2$和$S_3$上有副本
  - VSG = {$S_1$, $S_2$, $S_3$}
  - F被$C_1$ 修改
  - 由于网络分区的原因$C_1$: {$S_1$, $S_2$}； $C_2$: {$S_3$}

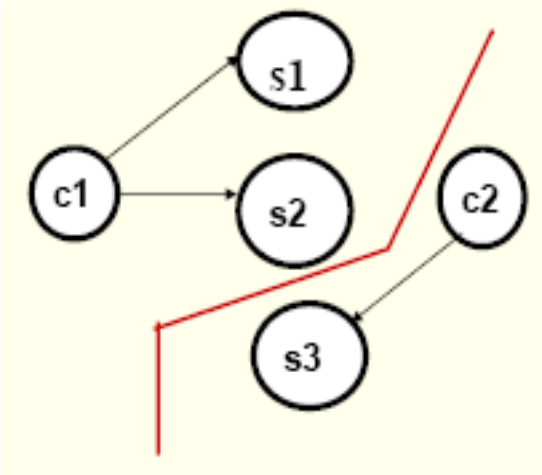


# Coda文件系统

## 构建CVV示例一(续)

- 最初，F的CVV在3个服务器上是相同的，比如[1, 1, 1]
- $C_1$修改F，然后关闭
  - $S_1$和$S_2$上的CVV更新为：[2, 2, 1]



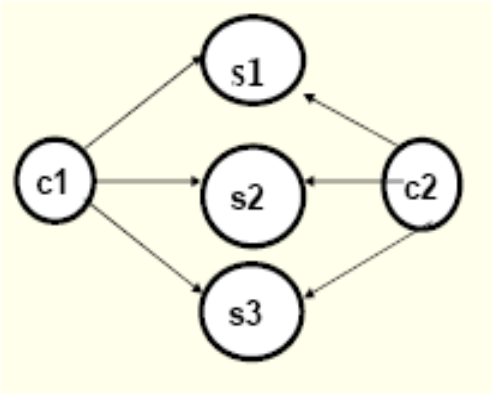
# Coda文件系统

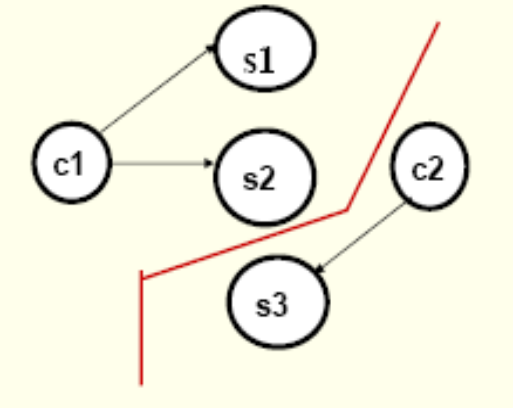
## 构建CVV示例一(续.)

- 网络故障修复后
  - $S_1$和$S_2$上的CVV为[2, 2, 1]，$S_3$上的CVV为[1, 1, 1]
  - $S_1$或$S_2$上的文件版本替代$S_3$上的文件版本

# Coda文件系统

- 构建CVV示例二
  - 文件F在三个服务器$S_1$、$S_2$和$S_3$上有副本
    - VSG = {$S_1$, $S_2$, $S_3$}
    - F在同一时间被$C_1$和$C_2$修改
    - $C_1$: {$S_1$, $S_2$}； $C_2$: {$S_3$}

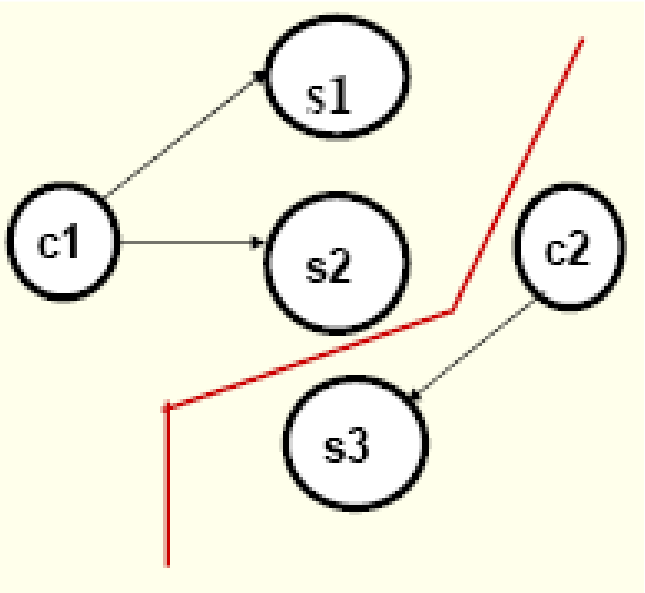


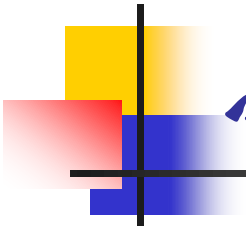



# Coda文件系统

## 构建CVV示例二(续)

- 最初，F的CVV在3个服务器上是相同的，比如[1, 1, 1]
- $C_1$修改F，然后关闭
  - $S_1$和$S_2$上的CVV更新为：[2, 2, 1]
- 同时，$C_2$上的2个进程分别修改F，然后关闭
  - $S_3$上的CVV更新为：[1, 1, 3]
- 网络故障修复后
  - $S_1$和$S_2$上的CVV为[2, 2, 1]，$S_3$上的CVV为[1, 1, 3]
  - 存在冲突，需手工干预





# 小结

- 分布式系统的复制
  - 一致性和可用性（网络分区）
- 容错服务
  - 被动（主备份）复制
  - 主动复制
- 高可用服务
  - gossip
  - bayou
  - coda