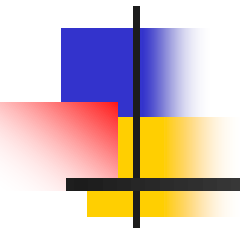
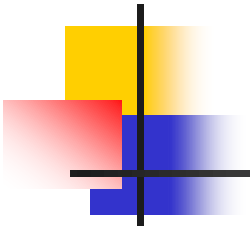


第6章 分布式事务

第6章 分布式事务

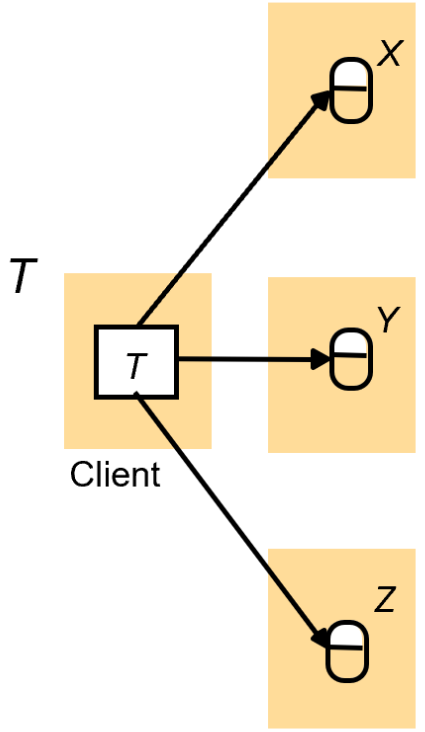
- 简介
- 原子提交协议
  - 单阶段提交协议
  - 两阶段提交协议
- 分布式事务的并发控制
- 分布式死锁
- 小结



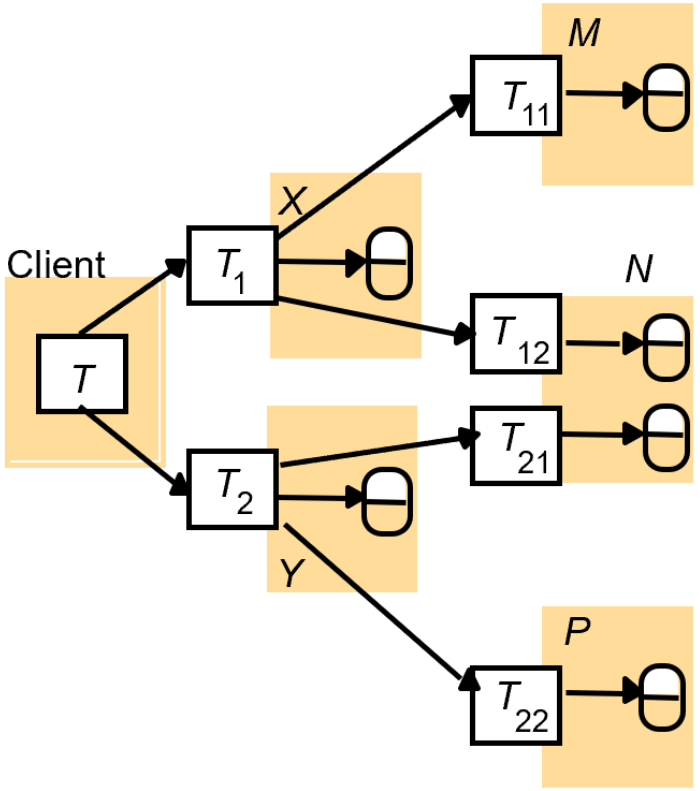
- 分布式事务
  - 访问由多个服务器管理的对象（事务在多个节点上更新数据）的平面事务或嵌套事务

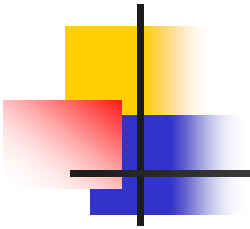
平面事务与嵌套事务

- 在平面事务中，客户给多个服务器发送请求。
- 事务T是一个平面事务，调用了服务器X、Y和Z上的操作对象
- 一个平面客户事务完成一个请求后才发起下一个请求。因此，每个事务顺序访问服务器上的对象
- 当服务器使用加锁机制时，事务一次只等待一个对象

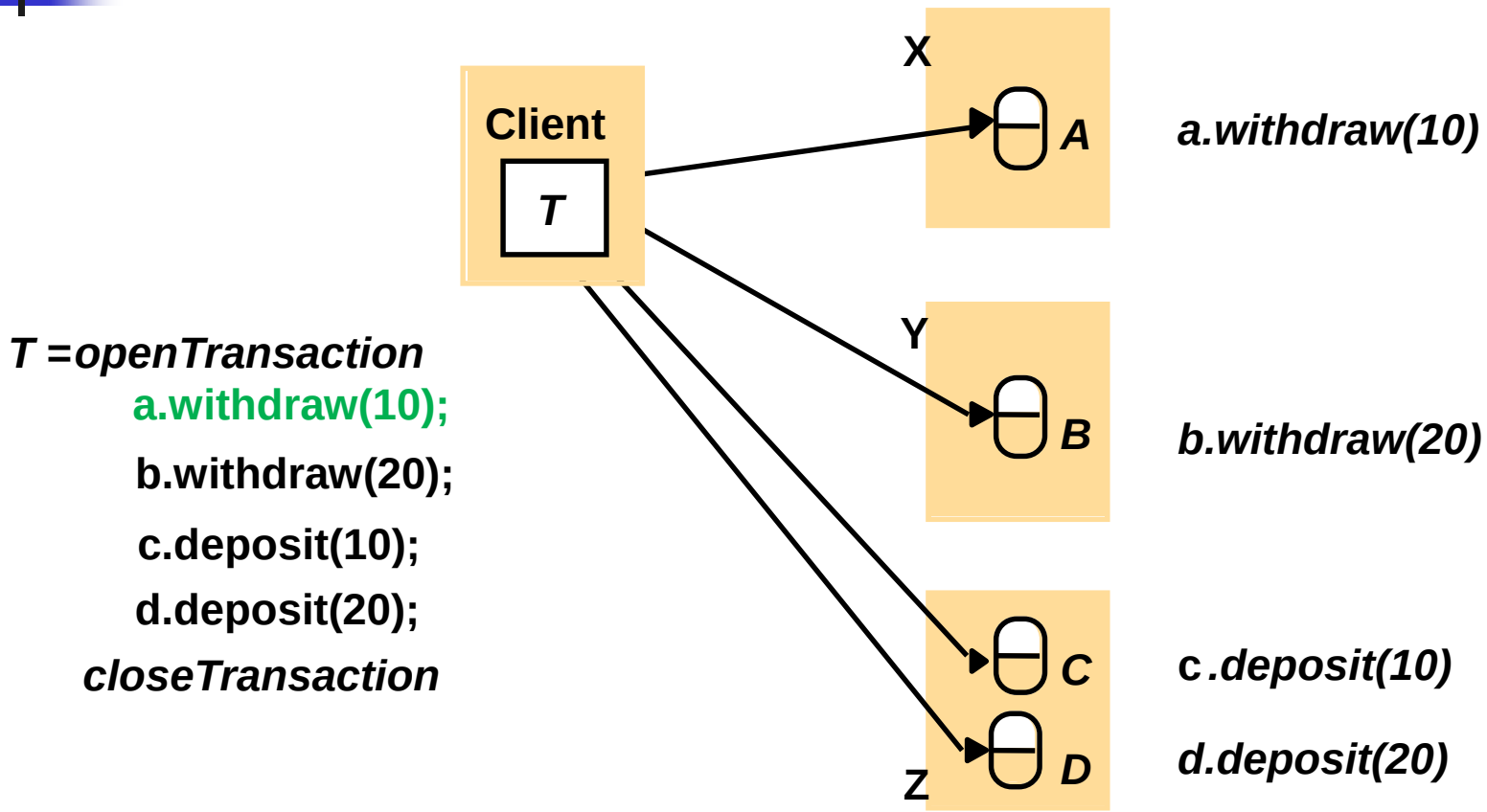


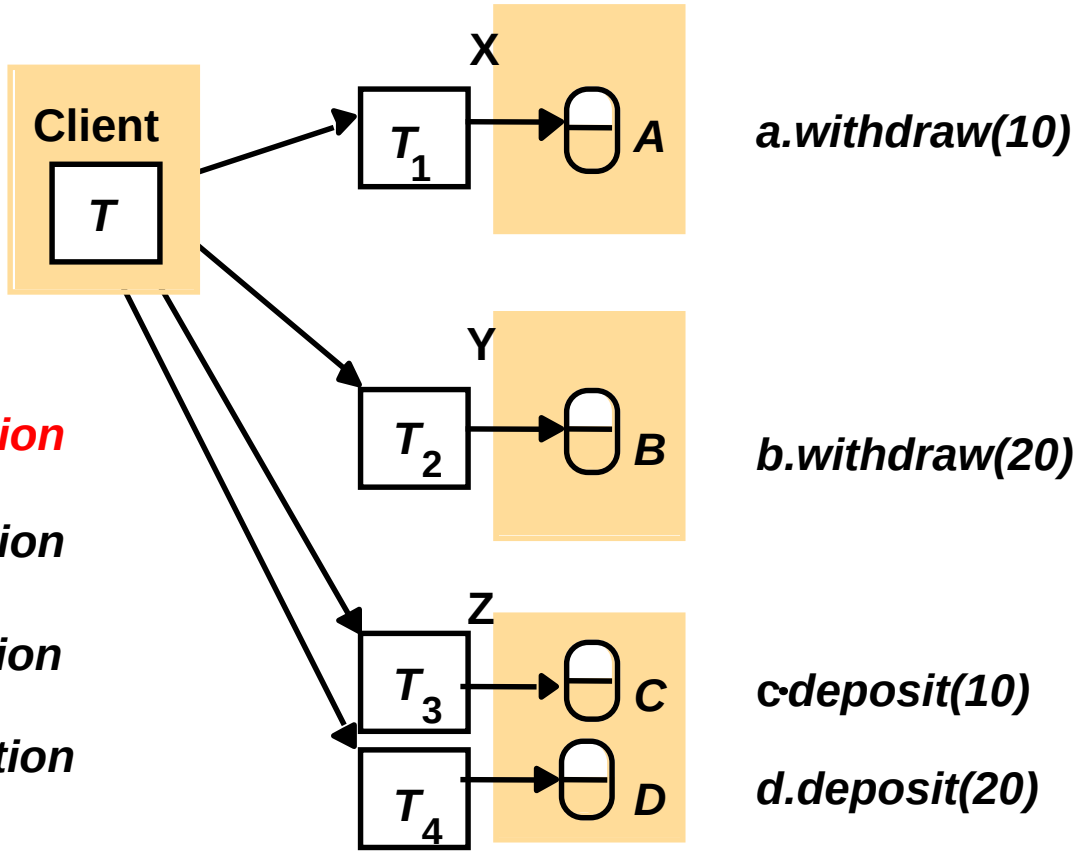
- 在嵌套事务中，顶层事务可以创建子事务，子事务可以进一步以任意深度嵌套子事务
- 同一层的子事务可以并发执行





- 客户从A账户转账$10到C账户，然后从B账户转账$20到D账户。
- 账户A和B分别在服务器X和Y上，而账户C和D在服务器Z上。
- 如果将该事务组成4个嵌套事务，那么4个请求可以并行运行，从而整体性能优于4个操作被顺序调用的简单事务。


Client
T
X
A
Y
B
C
D
Z
a.withdraw(10)
b.withdraw(20)
c .deposit(10)
d.deposit(20)
T =openTransaction
a.withdraw(10);
b.withdraw(20);
c.deposit(10);
d.deposit(20);
closeTransaction



T =*openTransaction*

openSubTransaction
a.withdraw(10);

openSubTransaction
b.withdraw(20);

openSubTransaction
c.deposit(10);

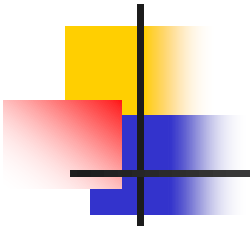
openSubTransaction
d.deposit(20);

closeTransaction

分布式事务的协调者

执行分布式事务请求的服务器需要相互通信，以确保事务提交时能够协调

- 协调者
  - 负责事务的开始、提交和放弃
- 参与者
  - 处理本地操作的服务器进程
  - 配合协调者共同执行提交协议

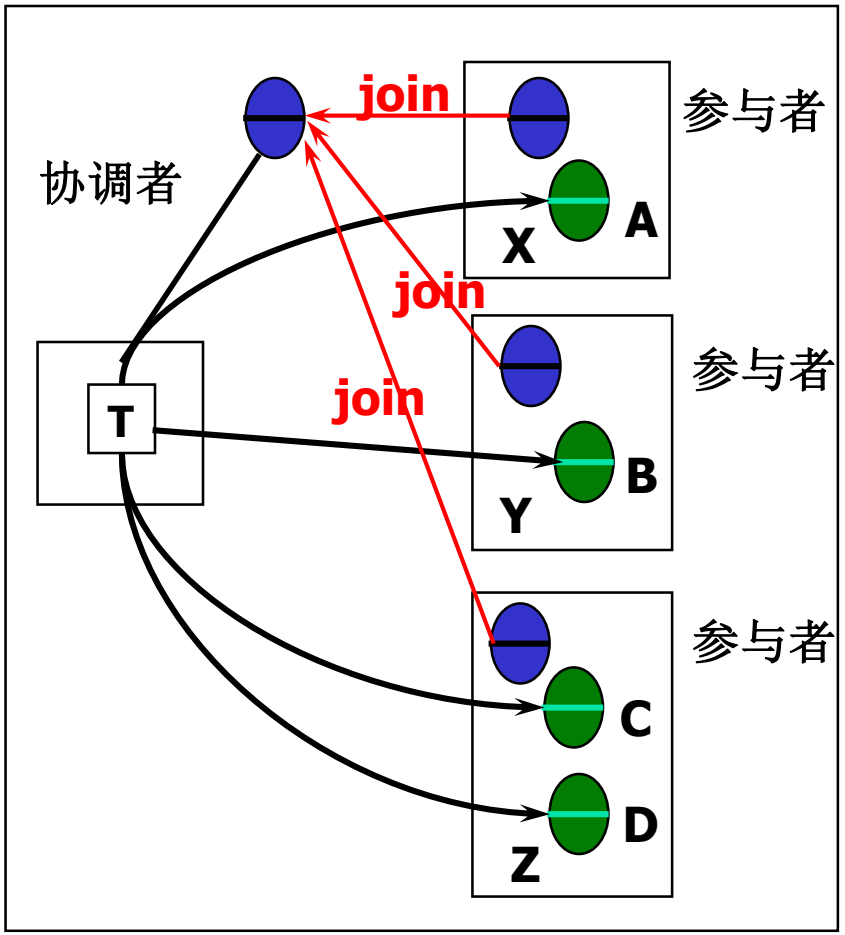


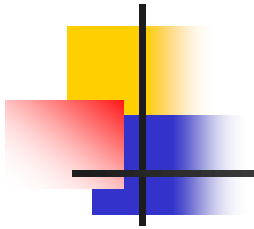
- 客户在启动一个事务时，向任意一台服务器上的协调者发出一个openTransaction请求
- 协调者处理完openTransaction请求后，将事务标识符（TID）返回客户
  - 创建该事务的服务器标识符（IP地址）
  - 一个对该服务器来说唯一的数字

- 协调者接口提供了一个额外的方法join，用于将一个新的参与者加入当前事务

  join(TID, reference to participant)

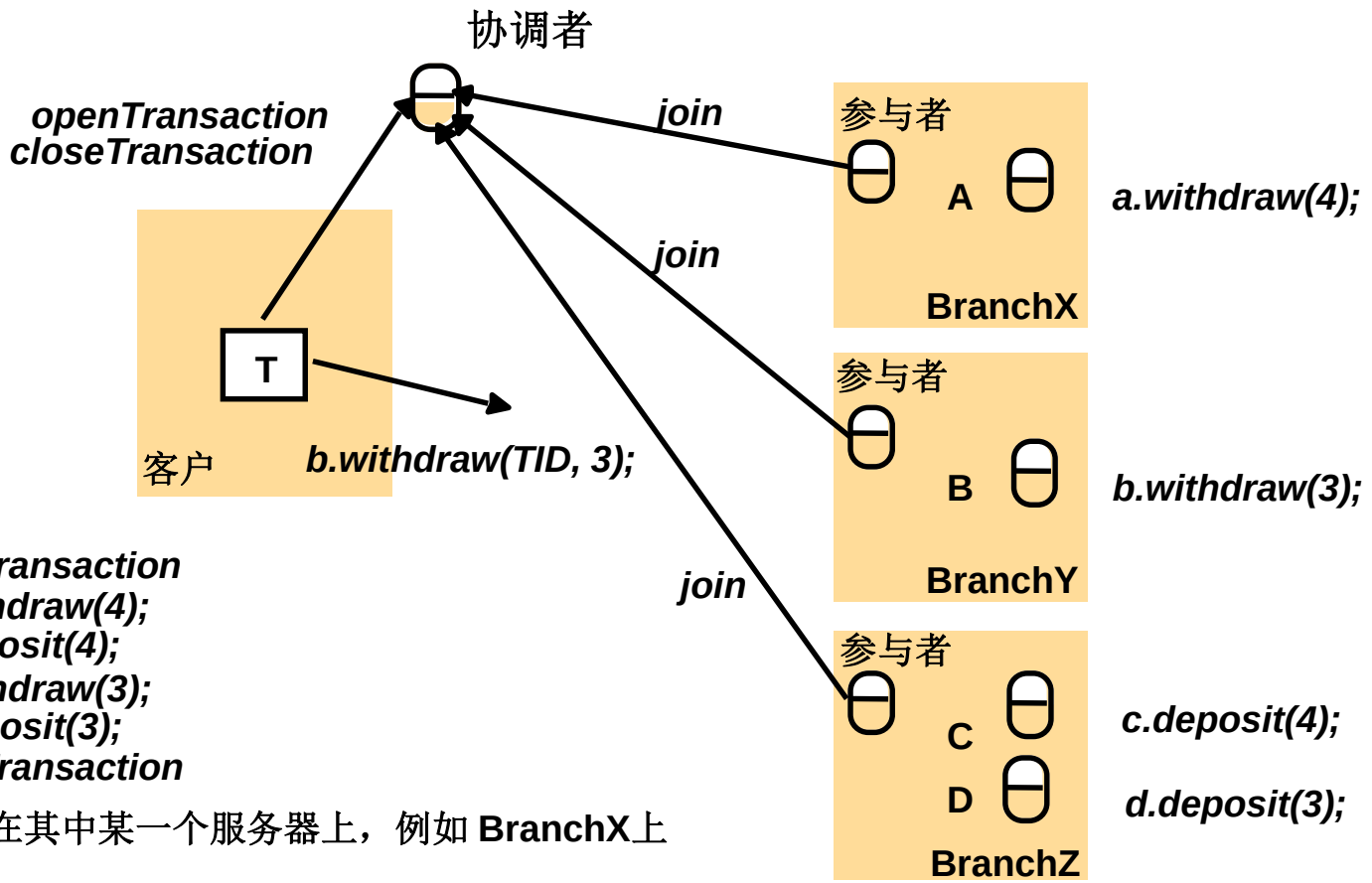
- 协调者知道所有参与者，每个参与者也知道协调者





- 一个分布式银行事务
  - （平面）银行事务涉及服务器BranchX, BranchY和BranchZ上的账户A、B、C和D。
  - 该客户事务T从账户A转账$4到账户C，然后从账户B转账$3到账户D。

- T的openTransaction和closeTransaction被发送到协调者
- 每个服务器上都有一个参与者，通过协调者的join方法加入该事务
  - 当客户调用事务中的方法b.withdraw(TID, 3)时，接收该调用的对象（服务器BranchY的B对象）将通知参与者对象自己属于事务T
  - 如果之前没有通知协调者，参与者对象调用join操作来通知协调者
  - 客户调用closeTransaction时，协调者就拥有了所有参与者的引用




```
T =openTransaction
    a.withdraw(4);
    c.deposit(4);
    b.withdraw(3);
    d.deposit(3);
  closeTransaction
```

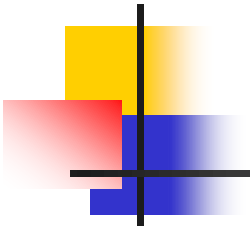
注：协调者在其中某一个服务器上，例如 BranchX上

第6章 分布式事务

- 简介
- 原子提交协议
  - 单阶段提交协议（1970）
  - 两阶段提交协议（1978）
- 分布式事务的并发控制
- 分布式死锁
- 小结

回顾：ACID

属性	含义	数据库系统的实现
Atomicity 原子性	事务中的操作要么全部执行，要么完全不执行	预写式日志（Write-ahead logging, undo），两阶段提交（分布式事务）
Consistency 一致性	数据库系统必须保证事务的执行使数据库从一个一致性状态转移到另一个一致性状态（满足完整性约束）	实现对A、I、D三个属性的支持
Isolation 隔离性	多个事务并发执行，对每个事务来说，不会感知系统中有其它事务在同时执行	两阶段加锁、乐观并发控制
Durability 持久性	一个事务在提交之后，该事务对数据库的改变是持久的	预写式日志（Write-ahead logging, redo）、存储管理



- 当一个分布式事务结束时，事务的原子特性要求所有参与该事务的服务器必须全部提交或全部放弃该事务。
- “原子提交协议”，允许服务器之间相互通信，以便共同决定提交或放弃。

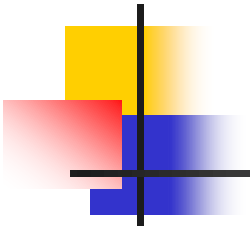
原子提交协议—问题

- 如果一个事务在多个服务器上更新数据，这说明：
  - 要么所有的节点必须提交，要么所有的节点必须放弃
  - 如果任意的节点崩溃，则所有的节点必须放弃
- 确保以上两点就是原子提交问题
- 是否与共识很相似？

共识	原子提交
一个或者多个节点提出一些值	所有节点都需要投票提交或者放弃
可以简单的选取其中一个被提出的值	如果所有节点投票提交，事务会提交；如果任意一个节点（≥1）投票放弃，事务会放弃
可以容忍少数节点崩溃，只要法定节点数有效	只要任意一个参与者崩溃，事务必须放弃

单阶段原子提交协议

- 单阶段原子提交协议是一种简单的原子提交协议
- 协调者向事务所有的参与者发送Commit或者Abort请求
- 协调者不断地发送请求直至所有参与者都确认



- 单阶段原子提交协议是不充分的
- 不允许参与者单方面决定放弃事务
- 可能有一些并发控制问题会阻止参与者提交，协调者可能不知道这种变化
  - 为了解除死锁需要放弃事务
  - 乐观并发控制验证失败导致放弃事务
  - 参与者可能崩溃，需要放弃事务

两阶段提交协议

- 设计动机

  允许任意一个参与者自行放弃他自己的那部分事务

- 基本思想
  - 一个协调者、多个参与者共同完成一个事务提交
  - 分2个阶段完成事务提交
  - 在第一阶段，协调者询问所有的参与者是否准备好提交
  - 在第二阶段，协调者通知所有参与者提交（或放弃）事务

两阶段提交协议中的基本操作

canCommit?(trans)->Yes/No

协调者使用该操作询问参与者是否能否提交事务，参与者将回复它的投票结果

doCommit(trans)/doAbort(trans)

协调者使用该操作告诉参与者提交/放弃它那部分事务

haveCommitted (trans, participant)

参与者使用该操作告知协调者它已经提交了事务

getDecision(trans)->Yes/No

当参与者投yes票后一段时间内未收到应答时，参与者使用该操作向协调者询问事务的投票表决结果。该操作用于从服务器崩溃或消息延迟中恢复。

两阶段提交协议

阶段1 (投票阶段):

1) 协调者向分布式事务的所有参与者发送canCommit?请求。

2) 当参与者收到canCommit?请求后，它将向协调者回复它的投票（Yes或No）。在投Yes票之前，它将在持久性存储中保存所有对象，准备提交；如果投No票，参与者立即放弃。

阶段2 (根据投票结果完成事务):

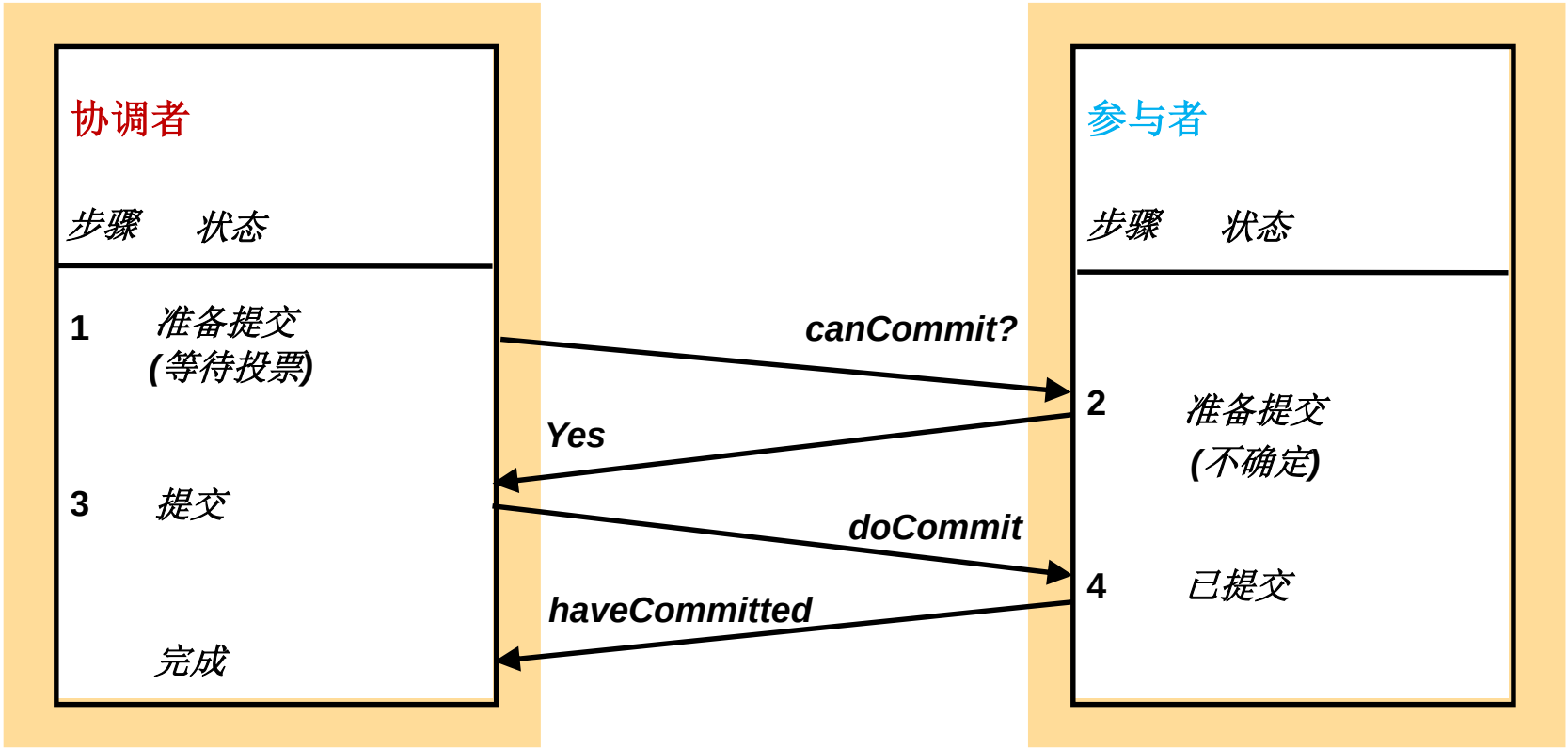
3) 协调者收集所有的投票（包括它自己的投票）

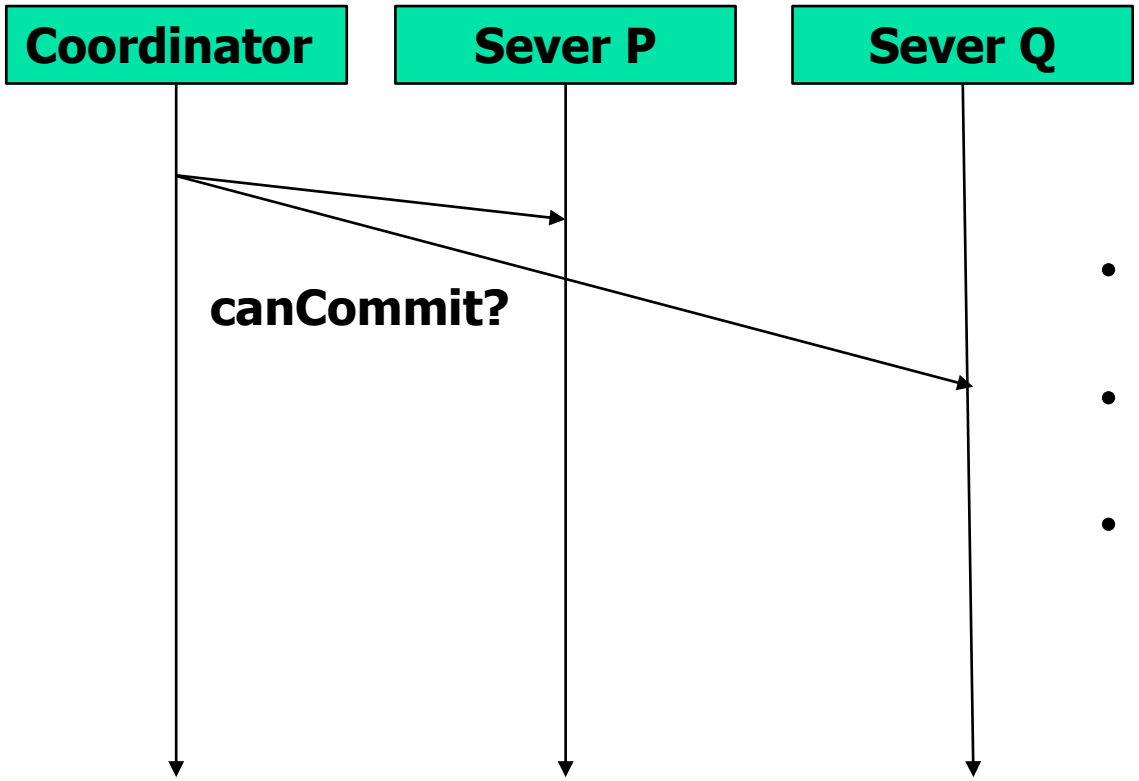
(a) 若不存在故障且所有的投票结果均是Yes时，则协调者决定提交事务并向所有参与者发送doCommit请求。

(b)否则，协调者决定放弃事务，并向所有投Yes票的参与者发送doAbort请求。

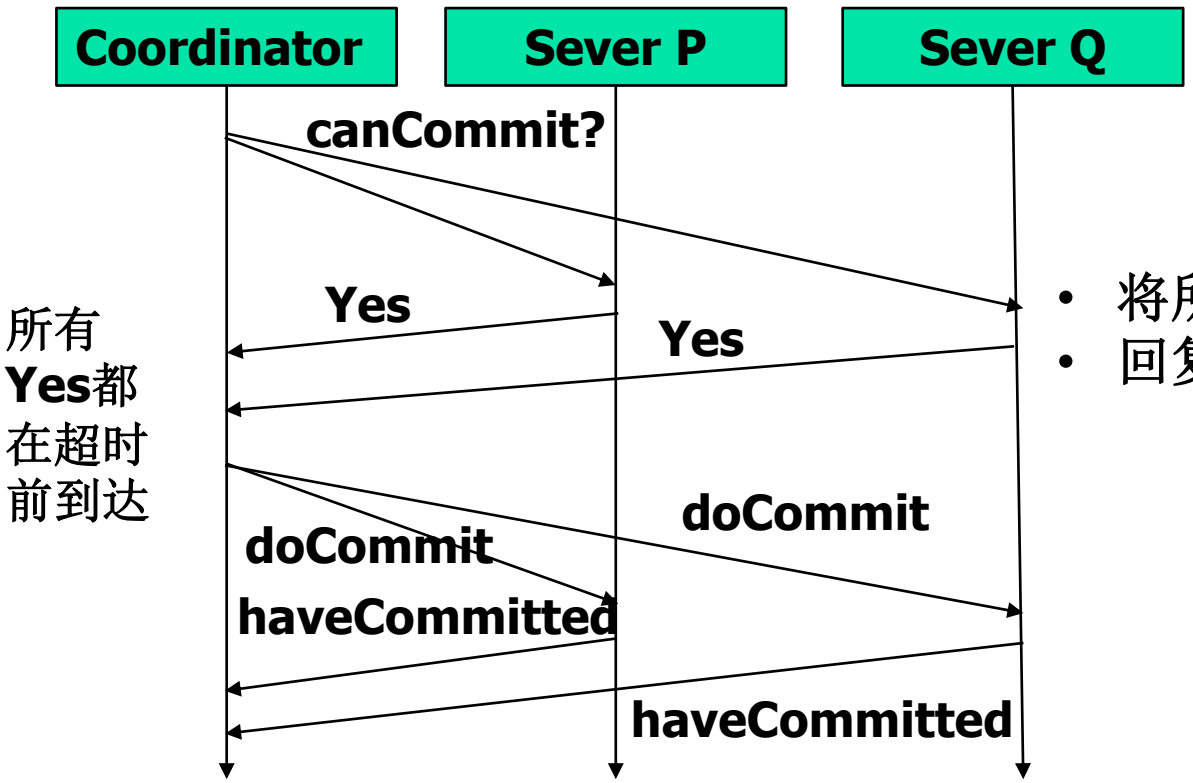
4) 投Yes票的参与者等待协调者发送的doCommit或者doAbort请求。一旦参与者收到任何一种请求消息，它根据该请求放弃或者提交事务。如果请求是提交事务，那么它还要向协调者发送一个haveCommitted来确认事务已经提交。

两阶段提交协议的通信

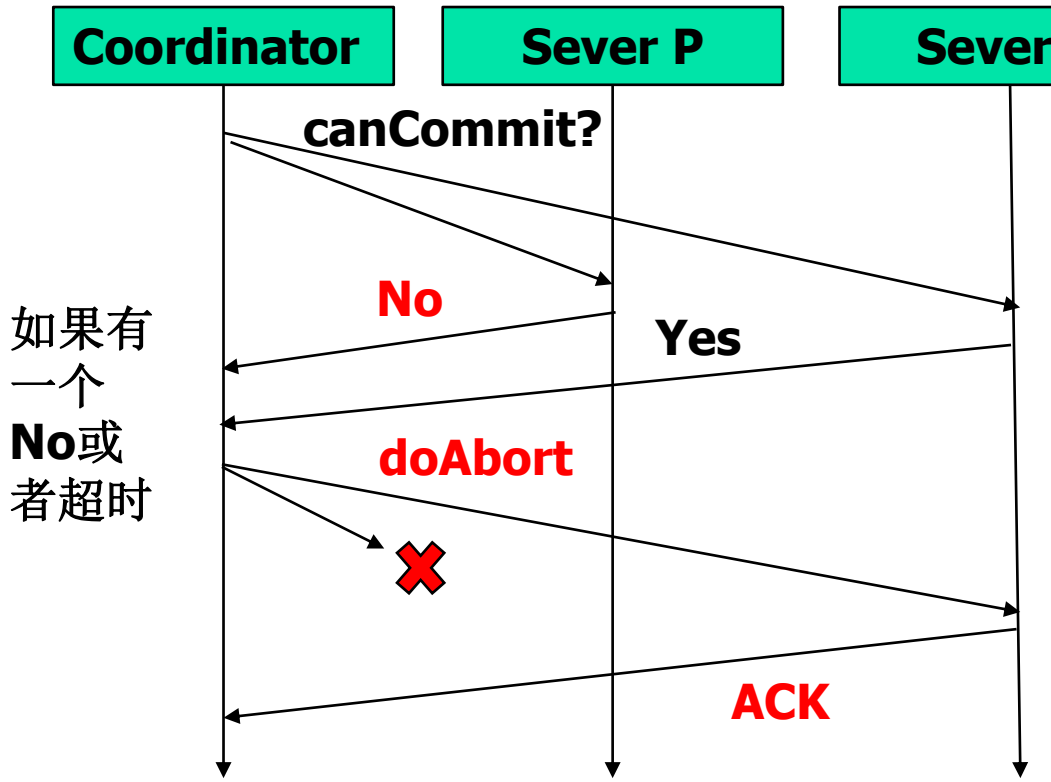

协调者
步骤 状态
1 准备提交
(等待投票)
3 提交
完成
canCommit?
Yes
doCommit
haveCommitted
参与者
步骤 状态
2 准备提交
(不确定)
4 已提交



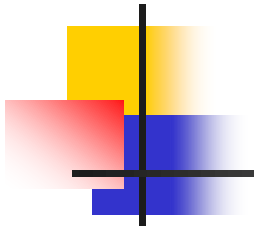
- 协调者依赖超时避免不必要的延迟
- 对于每一条消息，协调者都开始计时
- 超时被认为失效



- 将所有更新写入稳定存储
- 回复**Yes**或者**No**



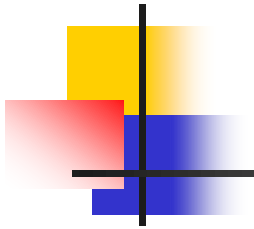
- 将所有更新写入稳定存储
- 回复**Yes**或者**No**



- 如果没有失效，两阶段原子提交可以顺利完成
- 几种失效的情况可能发生
  - 参与者崩溃
    - 一个或多个参与者
  - 协调者崩溃
  - 消息丢失
    - 任意消息

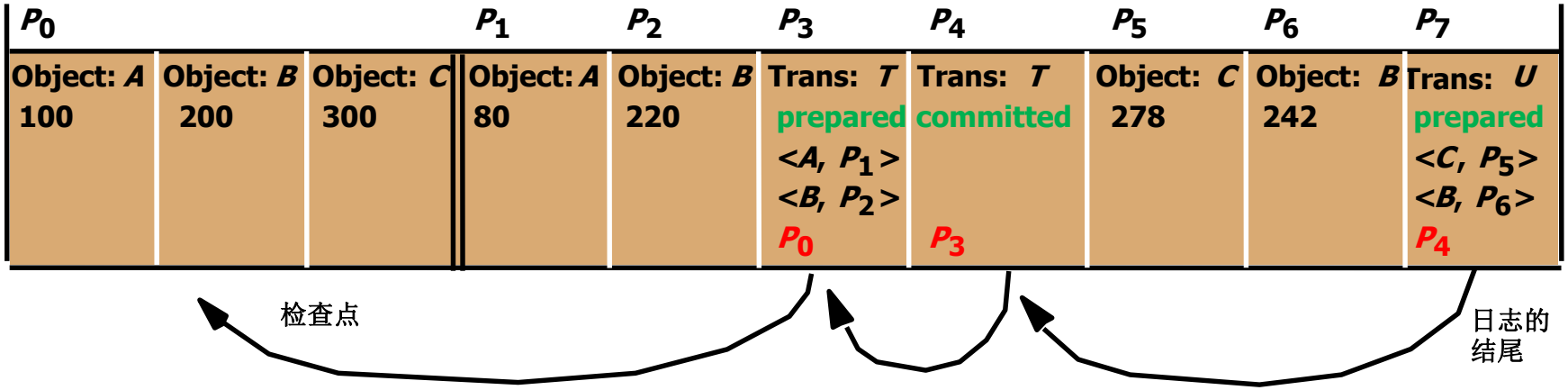
参与者与协调者崩溃

- 事务恢复就是保证服务器上对象的持久性并保证服务提供故障原子性
  - 持久性：要求对象被保存在持久性存储中并一直可用
  - 故障原子性：要求即使在服务器出现故障时，事务的更新作用也是原子的
- 事务的恢复过程实际上就是根据持久存储中最后提交的对象版本来恢复服务器中对象值



- 意图列表
  - 用来记录该服务器上的所有活动事务，每个事务的意图列表都记录了该事务修改的对象的值和引用列表<ObjectID, Pi>
    - 事务提交时，用来确定所受影响的对象，然后事务将用对象的临时版本替换成对象的提交版本
    - 当事务放弃时，用来删除该事务形成的对象的所有临时版本

- 日志
  - 恢复文件包含该服务器执行的所有事务的历史，历史由对象值、事务状态和意图列表组成



每个事务状态记录都包含一个指针，指向恢复文件中前一个事务状态记录的位置

两阶段提交协议的日志

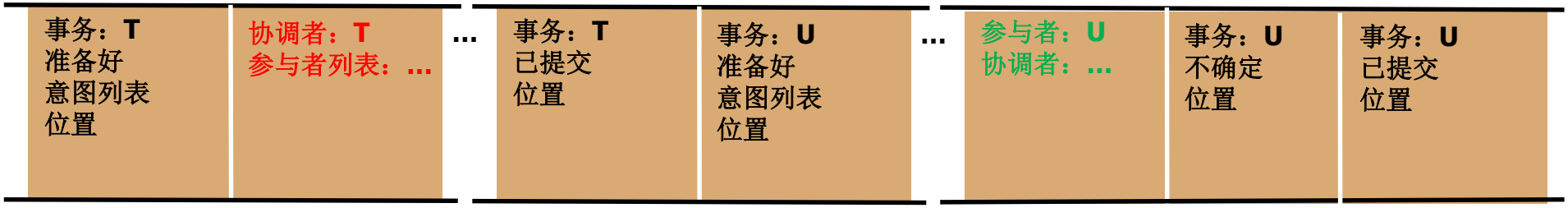
- 两个新的事务状态

  “完成（done）”+“不确定（uncertain）”

- 两种记录类型

  协调者：事务标识符，参与者列表

  参与者：事务标识符，协调者



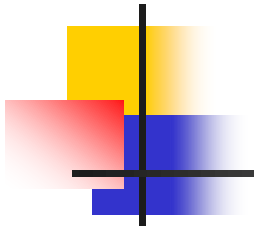
与两阶段提交协议相关的日志记录

两阶段提交协议的恢复

角色	状态	动作
协调者	准备好	由于服务器发生故障时尚未做出任何决定，因此向参与者列表中的所有服务器发送abortTransaction命令，并在恢复文件中记录一个已放弃记录。在已放弃状态下的操作也是这样。如果目前还没有参与者列表，那么参与者将由于超时最终放弃事务
协调者	已提交	在服务器故障发生时已经做出决定要提交事务。因此向参与者列表中所有参与者发送doCommit命令，继续执行两阶段提交协议的第4步
参与者	已提交	参与者向协调者发送haveCommitted消息。这允许协调者在下一个检查点处丢弃该事务的信息
参与者	不确定	参与者在获得决议之前发生故障，那么它在协调者通知它前不能确定事务的状态。因此参与者向协调者发送getDecision请求来询问事务状态。当它获得恢复后再提交或放弃事务
参与者	准备好	参与者尚未投票，它可以单方面放弃事务
协调者	完成	不需要任何操作

消息丢失

- 处理canCommit？丢失
  - 超时后协调者可能会决定放弃
- 处理Yes/No丢失
  - 协调者在超时后放弃事务（悲观！）。它必须向那些投票者宣布 doAbort。
- 处理 doCommit 丢失
  - 参与者可以等待超时，发送 getDecision 请求（重试直到收到答复）- 在投票“Yes”之后但在接收 doCommit/doAbort 之前无法放弃！



- 两阶段提交协议的性能—N个参与者
  - 无服务器崩溃、通信异常等情况下：
    N个canCommit?消息+N个应答+N个doCommit
    没有haveCommitted仍能正确运行，不计算在内
  - 最坏情况下，可能出现任意多次服务器和通信异常
  - 可能造成参与者长时间处于“不确定”状态

两阶段提交协议的问题

- **同步阻塞**

  在投票Yes与doCommit之间，所有参与者处于阻塞状态，无法进行其他任何操作。更甚者，若协调者在发起提议后崩溃，那么投Yes票的参与者阻塞至协调者恢复后发送决议。

- **单点故障**

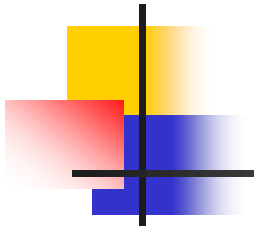
  协调者存在性能瓶颈及单点失效问题

- **数据不一致**

  当协调者发送doCommit后，若发生了局部网络异常或者协调者在尚未完全发送doCommit前自身发生了崩溃，导致只有部分参与者接收到doCommit。那么，接收到doCommit的参与者就会进行提交事务，而未收到doCommit的参与者不提交事务，进而形成了数据不一致性。

三阶段提交

- 第一阶段
  - 协调者向分布式事务所有参与者发送canCommit请求
  - 参与者回复投票Yes或者No
- 第二阶段
  - 协调者收集所有投票并做出决定。如果决定是No，那么它放弃事务并通知所有投Yes票的参与者；如果决定是Yes，它向所有的参与者发送preCommit请求。
  - 每个投Yes票的参与者都等待preCommit或者doAbort请求。收到preCommit请求后会回复确认，收到doAbort请求后会放弃事务
- 第三阶段
  - 协调者收集确认消息。一旦收集到所有的确认，它就提交事务并且向参与者发送doCommit请求。每个参与者等待doCommit请求到达后提交事务。



- 一旦进入第三阶段，可能会出现 2 种故障：
  - 协调者崩溃
  - 协调者和参与者之间的网络故障
- 出现了任一一种情况，最终都会导致参与者无法收到 doCommit 请求或者 doAbort 请求，针对这种情况，参与者都会在等待超时之后，继续进行事务提交

容错的两阶段提交

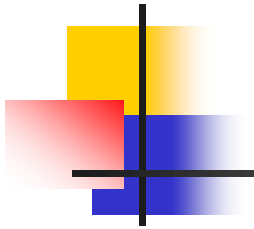
on initialization for transaction T **do**
commitVotes[T]:={}; replicas[T]:={}; decided[T]:=false
end on

on request to commit transaction T with participating nodes R **do**
for each r∈R do send (canCommit, T, R) to r
end on

on receiving (canCommit, T, R) at node *replicaId* **do**
replicas[T]:=R
yes="is transaction T able to commit on this replica"
total order broadcast (Vote, T, *replicaId*, yes) to replicas[T]
end on

on a node suspects node *replicaId* to have crashed **do**
for each trasaction T in which *replicaId* participated **do**
total order broadcast (Vote, T, *replicaId*, false) to replicas[T]
end for
end on

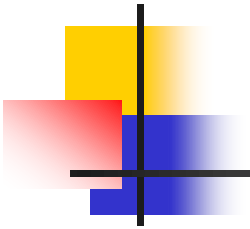
```
on delivering (Vote, T, replicaId, yes) by total order broadcast do
   if replicaId∉commitVote[T] ∧ replicaId∈replicas[T] ∧ ¬decided[T] then
      if yes=true then
         commitVotes[T]:=commitVotes[T]U{replicaId}
         if commitVotes[T]=replicas[T] then
            decided[T]:=true
            commit transaction T at this node
         end if
      else
         decided[T]:=true
         abort transaction T at this node
      end if
   end if
end on
```



- 两阶段提交协议的问题
  - 这是一个阻塞协议
  - 其他方式也是可能的，例如 3PC
  - 可扩展性和可用性问题

第6章 分布式事务

- 简介
- 原子提交协议
  - 单阶段提交协议
  - 两阶段提交协议
- 分布式事务的并发控制
- 分布式死锁
- 小结



- 分布式事务中所有服务器共同保证事务以串行等价方式执行
- 如果事务T对某一个服务器上对象的冲突访问在事务U之前，那么在所有服务器上对对象的冲突操作，事务T都在事务U之前

加锁

- 在分布式事务中，某个对象的锁总是本地持有的
- 原子提交协议进行时对象始终被锁住，其它事务不能访问这些对象
- 由于不同服务器上的锁管理器独立设置对象锁，所以对不同的事务，它们的加锁次序可能不一致

T	U
Write(A) 在服务器X上对A加锁	
	Write(B) 在服务器Y上对B加锁
Read(B) 在服务器Y上等待U	
	Read(A) 在服务器X上等待T

- 在服务器X上，事务T在事务U之前
- 在服务器Y上，事务U在事务T之前
- 这种不同的事务次序导致事务间循环依赖，从而引起分布式死锁
- 一旦检测出死锁，必须放弃某个事务解除死锁

时间戳并发控制

- 对于单服务器事务，协调者在它开始运行时分配一个唯一的时间戳。通过按访问对象的事务的时间戳次序提交对象的版本来保证串行等价
- 在分布式事务中，协调者必须保证每个事务附上全局唯一时间戳
- 时间戳是一个二元组<本地时间戳，服务器id>对。在时间戳比较中，首先比较本地时间戳，然后比较服务器id

乐观并发控制

- 每个事务在提交前必须首先进行验证
- 每个服务器验证访问自己对象的事务
- 验证在两阶段提交协议的第一阶段进行
- 由于两阶段提交需要一定的时间，每个服务器使用并行验证协议，允许多个事务同时进入验证阶段
  - 例如，向后验证除了规则2，还必须检查规则3（被验证事务的写集与较早启动的重叠事务的写集是否重叠）

第6章 分布式事务

- 简介
- 原子提交协议
  - 单阶段提交协议
  - 两阶段提交协议
- 分布式事务的并发控制
- 分布式死锁
- 小结

分布式死锁

- 在使用加锁机制进行并发控制时可能出现死锁。
  - 锁超时
    - 很难设定合适的时间间隔
  - 死锁检测
    - 通过寻找等待图中的环路实现
    - 全局等待图中的环路在局部等待图中不存在，可能出现分布式死锁

3个事务U、V和W的交错执行，涉及

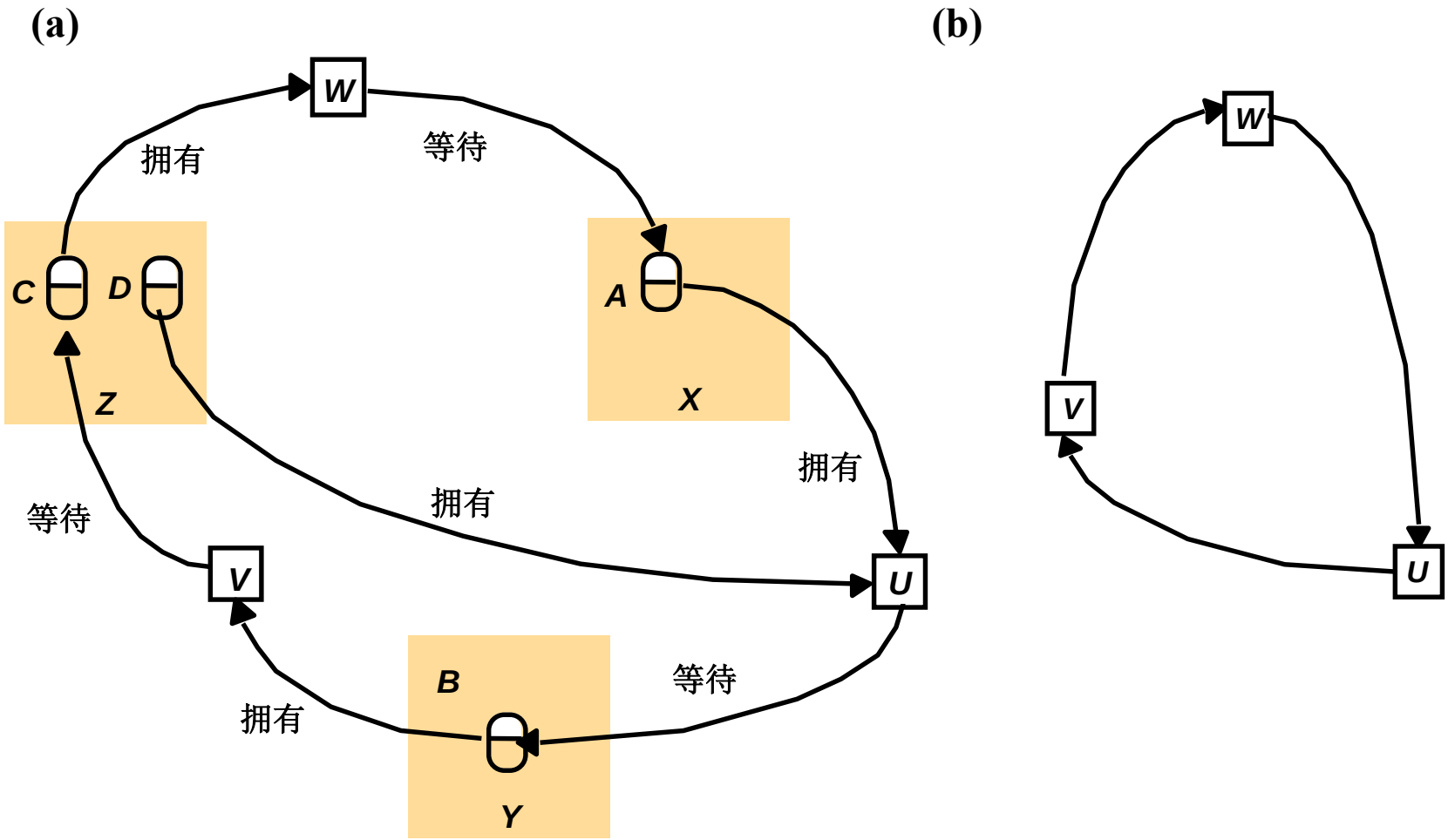
- 服务器X上的对象A
- 服务器Y上的对象B
- 服务器Z上的对象C和D

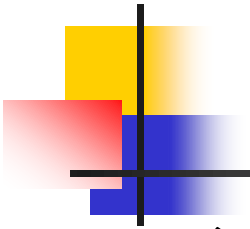
U	*V*	*W*
d.deposit(10) 在结点Z锁住D		
	b.deposit(10) 在结点Y锁住B	
a.deposit(20) 在结点X锁住A		
		c.deposit(30) 在结点Z锁住C
b.withdraw(30) 在结点Y等待		
	c.withdraw(20) 在结点Z等待	
		a.withdraw(20) 在结点X等待

分布式死锁的检测要求在分布于多个服务器的全局等待图中寻找环路

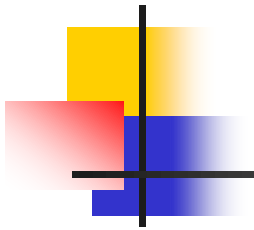
例子中各服务器的局部等待图为

- 服务器Y：U→V
- 服务器Z：V→W
- 服务器X：W→U



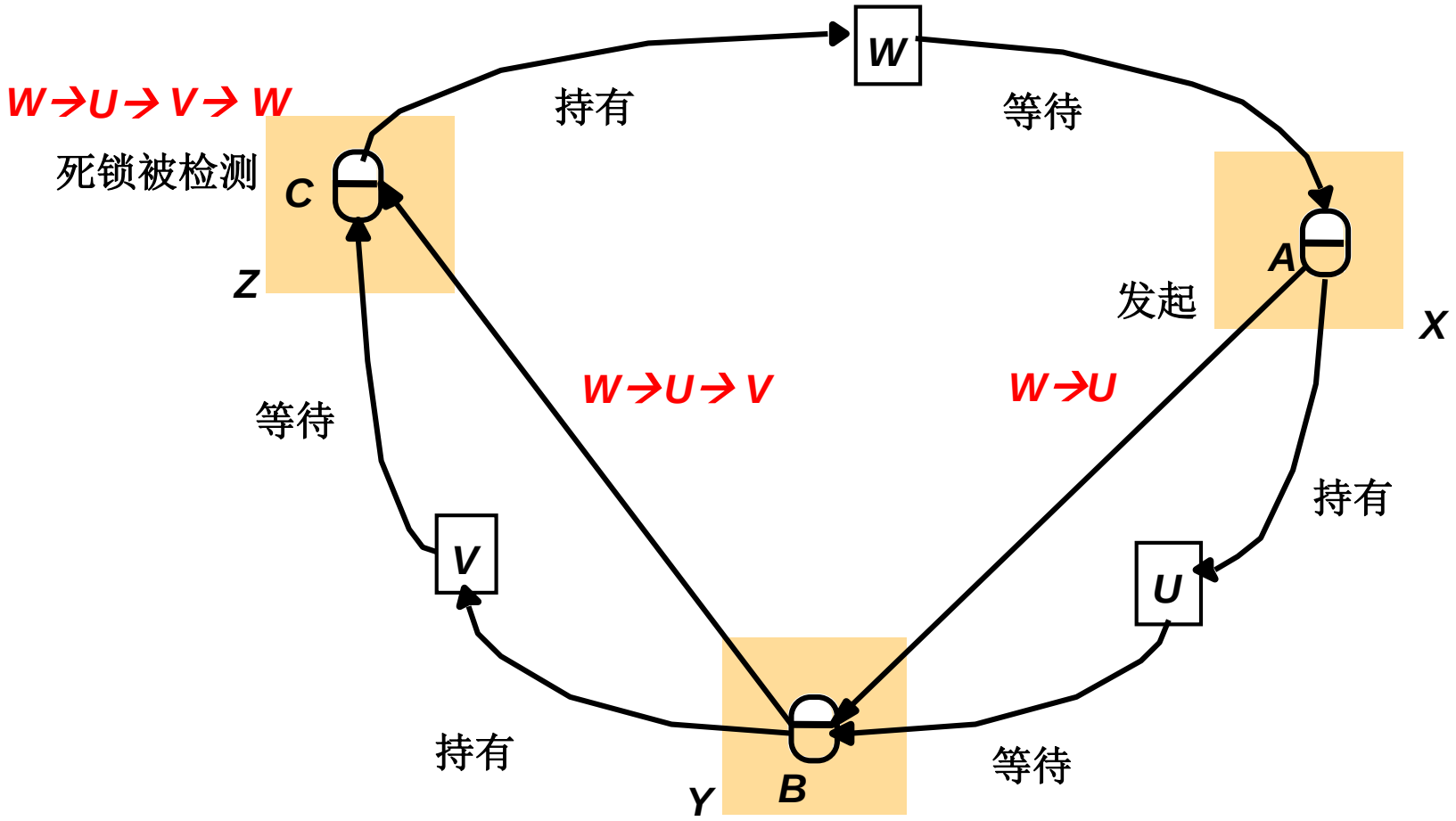


- 各服务器只有局部等待图，需要通过通信才能发现全局环路
- 集中式死锁检测
  - 其中的一个服务器担任全局死锁检测器
  - 全局死锁检测器收集局部等待图构造全局等待图
  - 一旦发现环路，通知各服务器放弃相应事务解除死锁
- 不足
  - 依赖单一服务器执行检测
  - 可靠性差、缺乏容错，没有可伸缩性
  - 收集局部等待图代价高



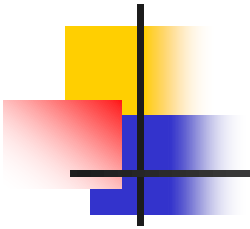
- 边追逐方法
  - 无需构建全局等待图
  - 服务器通过转发探询(Probe)消息发现环路
- 边追逐算法由下面3步组成：
  - 开始阶段：当服务器发现某个事务T开始等待事务U，而U正在等待另一个服务器上的对象时，该服务器将发送一个<T→U>的探询消息。
  - 死锁检测：接收探询消息、确定是否有死锁和是否转发探询消息
  - 死锁解除：当检测出环路后，环路中的某个事务将被放弃以打破死锁

- 服务器X发起死锁检测，向对象B的服务器Y发送探询消息<W→U>
- 服务器Y收到探询消息<W→U>后，发现对象B被事务V拥有，因此将V附加在探询消息上，产生<W→U→V>。由于V在服务器Z上等待对象C，因此该探询消息被转发到服务器Z
- 服务器Z收到探询消息<W→U→V>，并发现C被事务W拥有，那么将W附加在探询消息后形成<W→U→V→W>



第6章 分布式事务

- 简介
- 原子提交协议
  - 单阶段提交协议
  - 两阶段提交协议
- 分布式死锁
- 分布式事务的并发控制
- 小结



总结

- 分布式事务
  - ACID
- 原子提交协议
  - 单阶段提交无法很好地处理故障和放弃
  - 两阶段提交缓解了单阶段提交的问题
  - 两阶段提交有其自身的局限性：阻塞
- 分布式死锁
  - 集中式死锁检测依赖单一服务器
  - 边追逐方法通过转发探询(Probe)消息发现环路

谢谢！