一致性

赵来平
天津大学软件学院

一致性

- 这是一个在不同语境下含义完全不同的词！
- ▶️ ACID 中的一致性：
  一次事务会把数据库从一个“一致”状态转换到另一个“一致”状态。这里的“一致” = 满足应用程序特定的业务规则。
  例如：“每一门有学生选课的课程，必须至少有一名任课教师。”

一致性

- 这是一个在不同语境下含义完全不同的词！
- ▶️ ACID 中的一致性：
  一次事务会把数据库从一个“一致”状态转换到另一个“一致”状态。这里的“一致” = 满足应用程序特定的业务规则。
  例如：“每一门有学生选课的课程，必须至少有一名任课教师。”
- **一次事务会把数据库从一个一致状态，变到另一个一致状态。**
- **意思是：**
  1. **在事务开始之前**，数据库是“合法的”（满足所有不变式）。
  2. **事务执行过程中**，内部可能临时出现“奇怪”的中间状态（比如先删老师后插课程），但这些中间状态**对外是不可见的。**
  3. **事务提交之后**，数据库必须重新回到一个“合法的状态”（所有不变式再次被满足）。
  4. 如果事务会破坏这些不变式（比如插入了“有学生选课但没有老师的课程”），
     👉 数据库应该 **拒绝提交这个事务（回滚）**，从而保持一致性。

一致性

- 这是一个在不同语境下含义完全不同的词！
- ▶️ ACID 中的一致性：
  一次事务会把数据库从一个“一致”状态转换到另一个“一致”状态。这里的“一致” = 满足应用程序特定的业务规则。
  例如：“每一门有学生选课的课程，必须至少有一名任课教师。”
- **一次事务会把数据库从一个一致状态，变到另一个一致状态。**
- **意思是：**
  1. **在事务开始之前**，数据库是“合法的”（满足所有不变式）。
  2. **事务执行过程中**，内部可能临时出现“奇怪”的中间状态（比如先删老师后插课程），但这些中间状态**对外是不可见的。**
  3. **事务提交之后**，数据库必须重新回到一个“合法的状态”（所有不变式再次被满足）。
  4. 如果事务会破坏这些不变式（比如插入了“有学生选课但没有老师的课程”），
     👉 数据库应该 **拒绝提交这个事务（回滚）**，从而保持一致性。

事务结束后，数据库里不能出现违反业务规则的脏数据。

一致性

- **假设你做一个事务，想做两件事：**
  1. 给某门课插入一个学生的选课记录
  2. 但这门课还没有老师，也没有在同一个事务里给它安排老师
- **那会发生什么？**
- **如果数据库配置了这个约束（比如通过外键、触发器或应用检查）**
- **事务尝试提交时，数据库会发现：**
  “咦，这门课有人选了，但没有老师，这违反了不变式”
- **于是数据库会：**
  ❌ **拒绝提交（回滚整个事务）**
  ✅ **保证提交之后的状态仍然是“一致”的**

一致性

- 这是一个在不同语境下含义完全不同的词！
- ▶️ ACID 中的一致性
- ▶️ 写后读一致性（Read-after-write consistency）
  - 同一个客户端在自己写入（update）之后，再去读（read）同一份数据时，至少要看到自己刚刚写进去的最新结果，而不是旧数据。

写后读一致性

示例：

1. 你发了一条朋友圈（写入一条记录）
2. 立刻打开“我的朋友圈”列表（读取你自己的数据）

- 满足写后读一致性 ⇒
  你一定能在自己的列表里看到那条刚发的动态。
- 有些“最终一致性”的系统，如果没有额外保证，就可能出现：
- 你已经发成功了（服务器其实存上了）
- 但因为复制 / 缓存延迟，你打开自己的列表时
  还是看不到刚发的那条
  👉 这就是典型的“写后读不一致”现象。

分布式事务

- **回忆一下 ACID 事务中的**原子性（atomicity）**概念：**
  - 一个事务要么**提交（commit）**，要么**中止/回滚（abort）**
  - 如果它提交了，它的更新是**持久的（durable）**
  - 如果它中止了，它就**没有任何可见的副作用**
  - ACID 中的“一致性”（保持不变式）是依赖于**原子性**来实现的
- **如果一个事务要在多个节点上更新数据，这就意味着：**
  - 要么**所有节点都提交**，要么**所有节点都中止**
  - **只要有任何一个节点崩溃，所有节点都必须中止**
- **确保这一点，就是所谓的原子提交问题（atomic commitment problem）。**

在多个参与者（多个数据库/分片/服务）都参与的一笔分布式事务里，
如何保证“原子提交”——要么大家都提交，要么大家都回滚？

两阶段提交协议

Phase 1：准备阶段（Prepare / Vote）

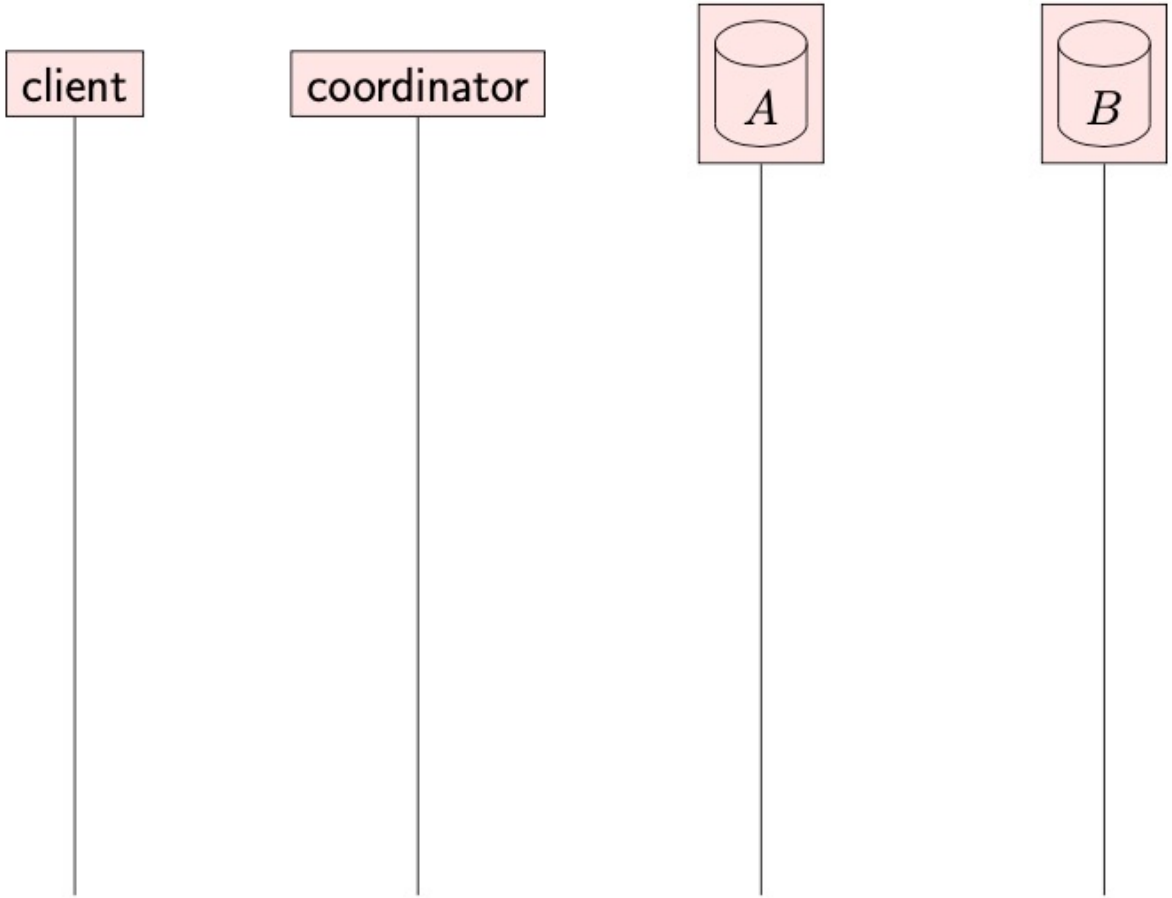
1. **协调者发送：PREPARE(T) 给所有参与者**
2. **参与者收到后：**
    1. 本地执行到“可提交”点（写 WAL/redo、锁住相关数据等）
    2. 能提交则回 VOTE_COMMIT，不能则回 VOTE_ABORT

- **这个阶段的核心：参与者把自己“锁死在一个决定点附近”，并把必要的日志落盘，保证之后即便崩溃恢复也能继续。**

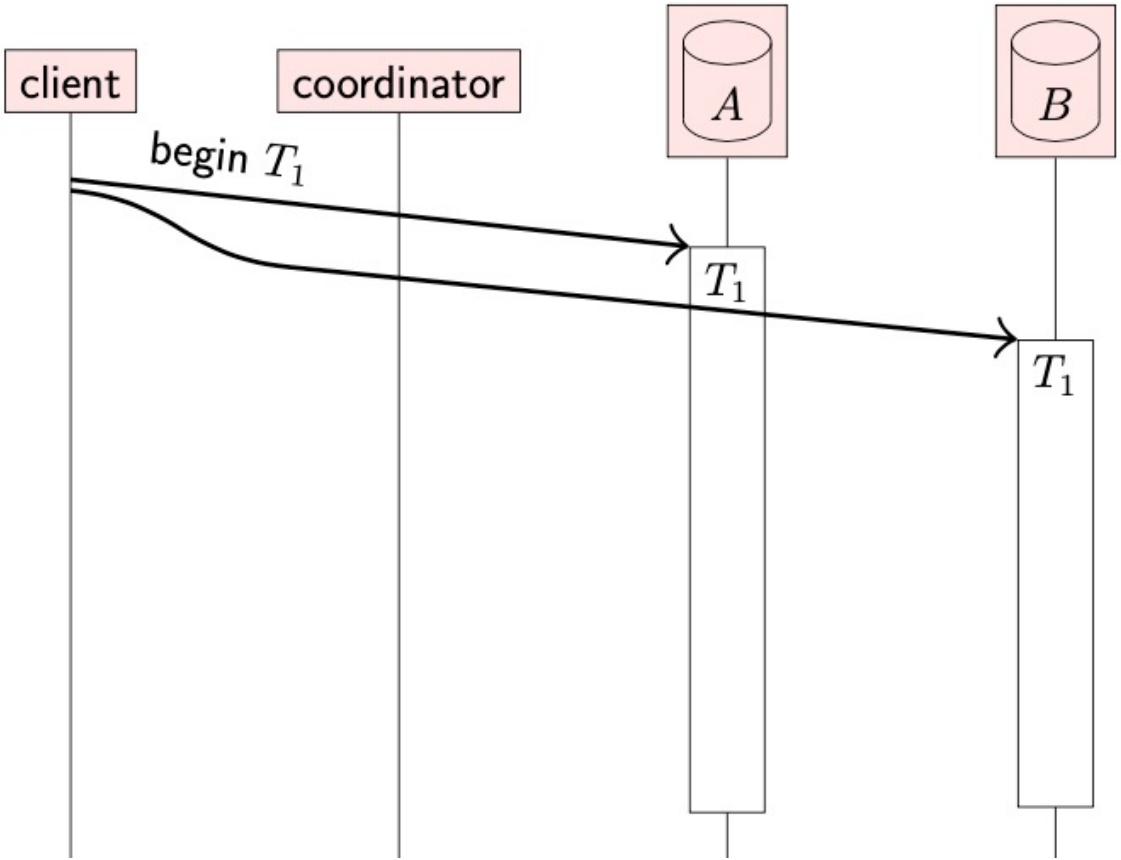
Phase 2：提交阶段（Commit / Abort）

- **如果协调者收齐所有 VOTE_COMMIT：写下“决定提交”的日志，然后广播 COMMIT(T)**
- **否则（任何一个 VOTE_ABORT / 超时 / 失败）：写下“决定回滚”，广播 ABORT(T)**
- **参与者收到 COMMIT/ABORT 后执行落地动作并释放锁，回复 ACK（可选）。**

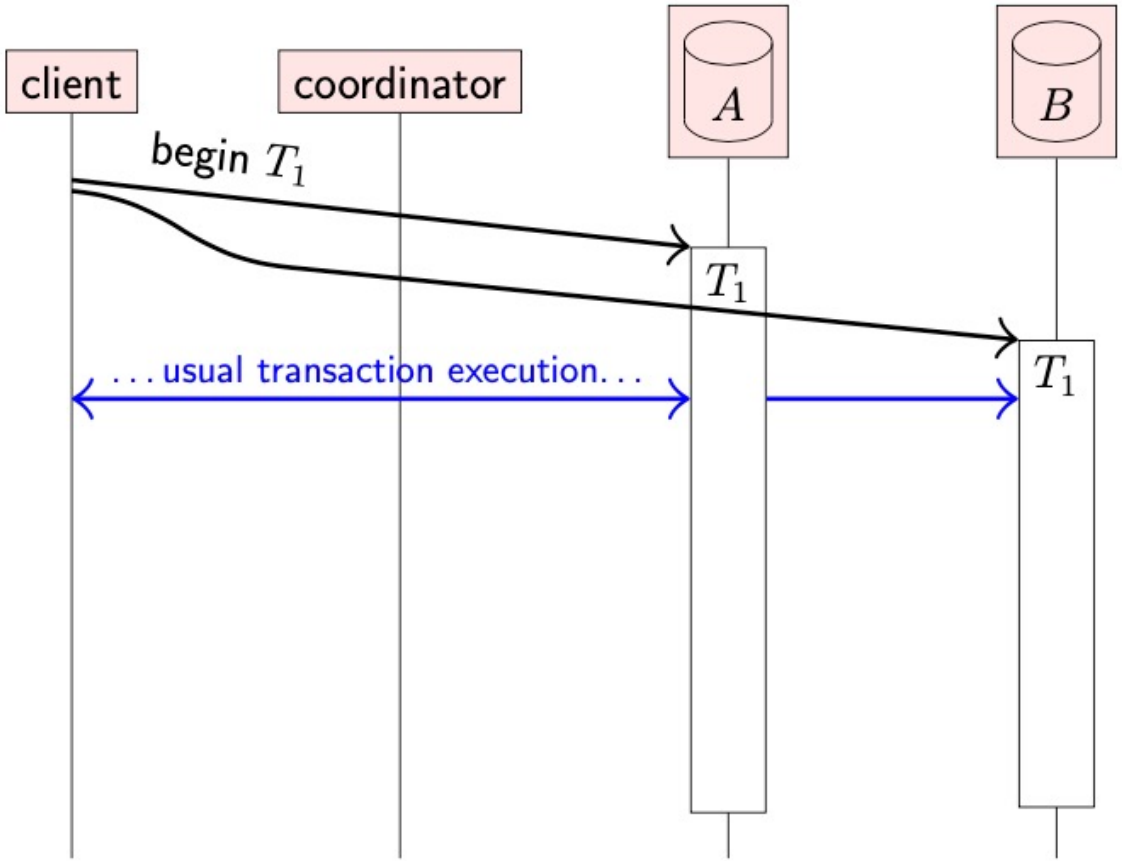
两阶段提交协议



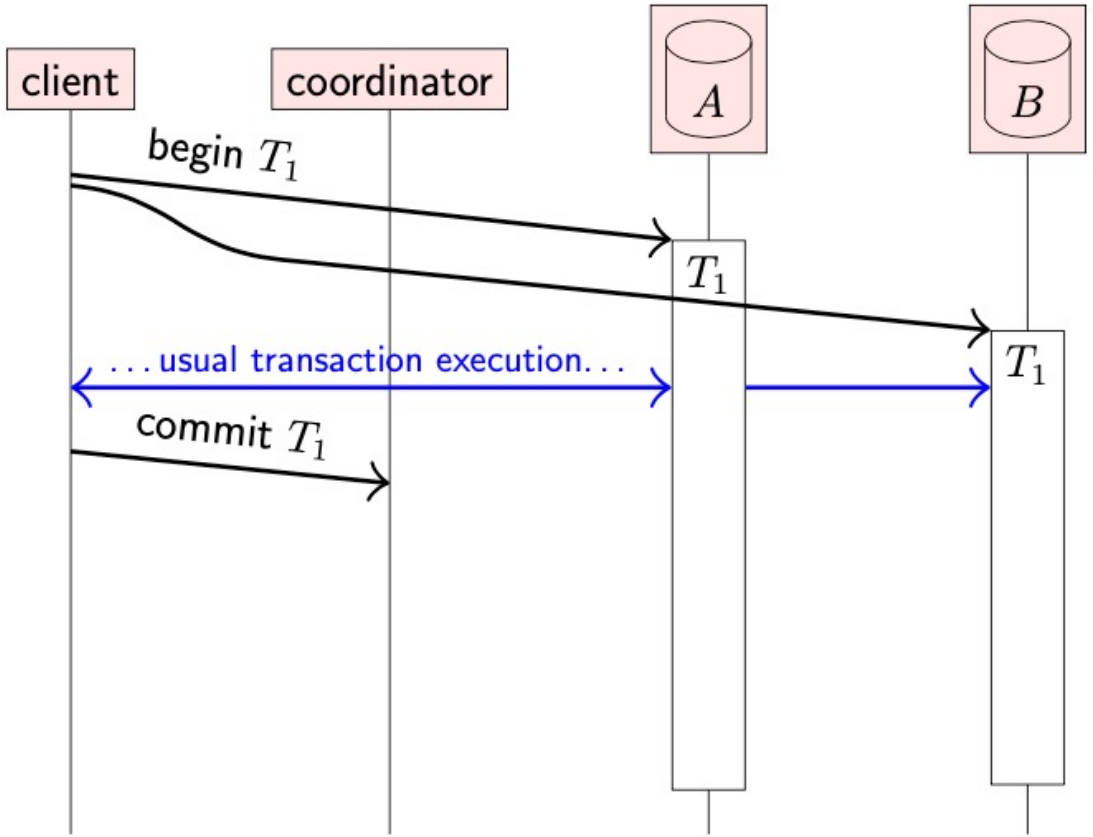
两阶段提交协议



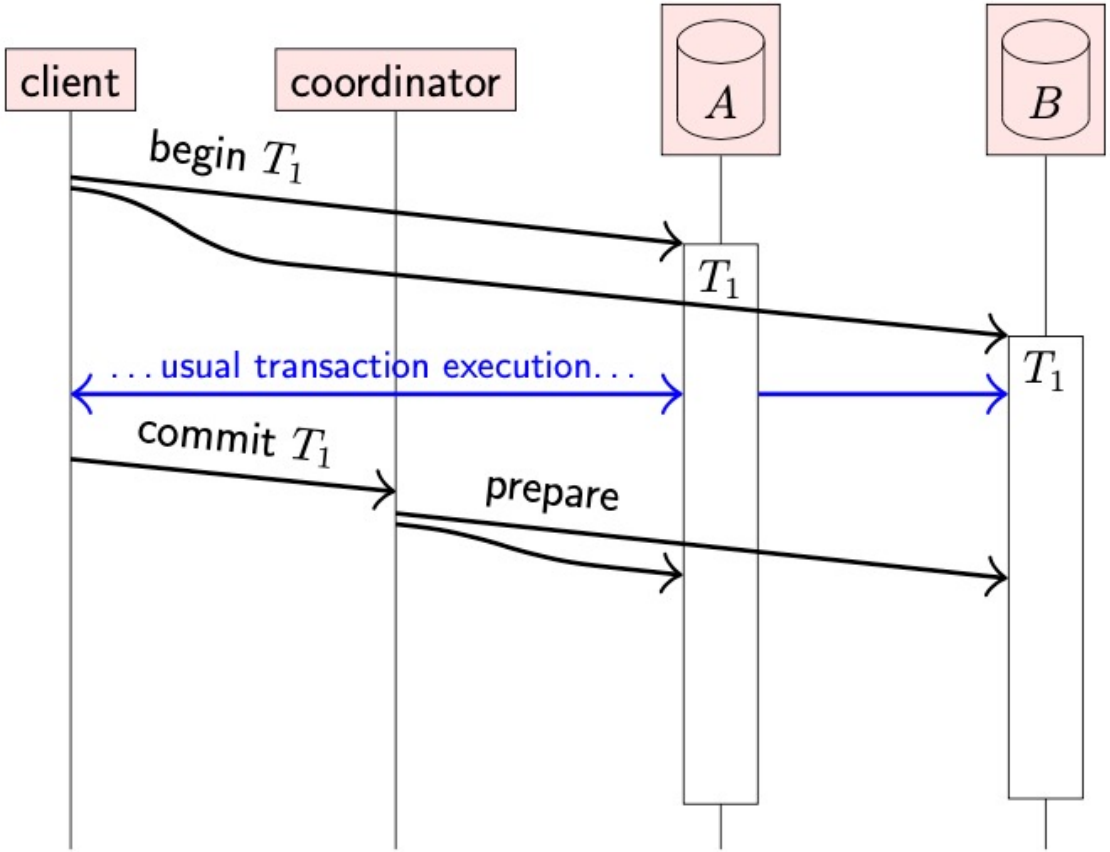
两阶段提交协议


client
coordinator
A
B
begin T_1
T_1
T_1
. . . usual transaction execution. . .

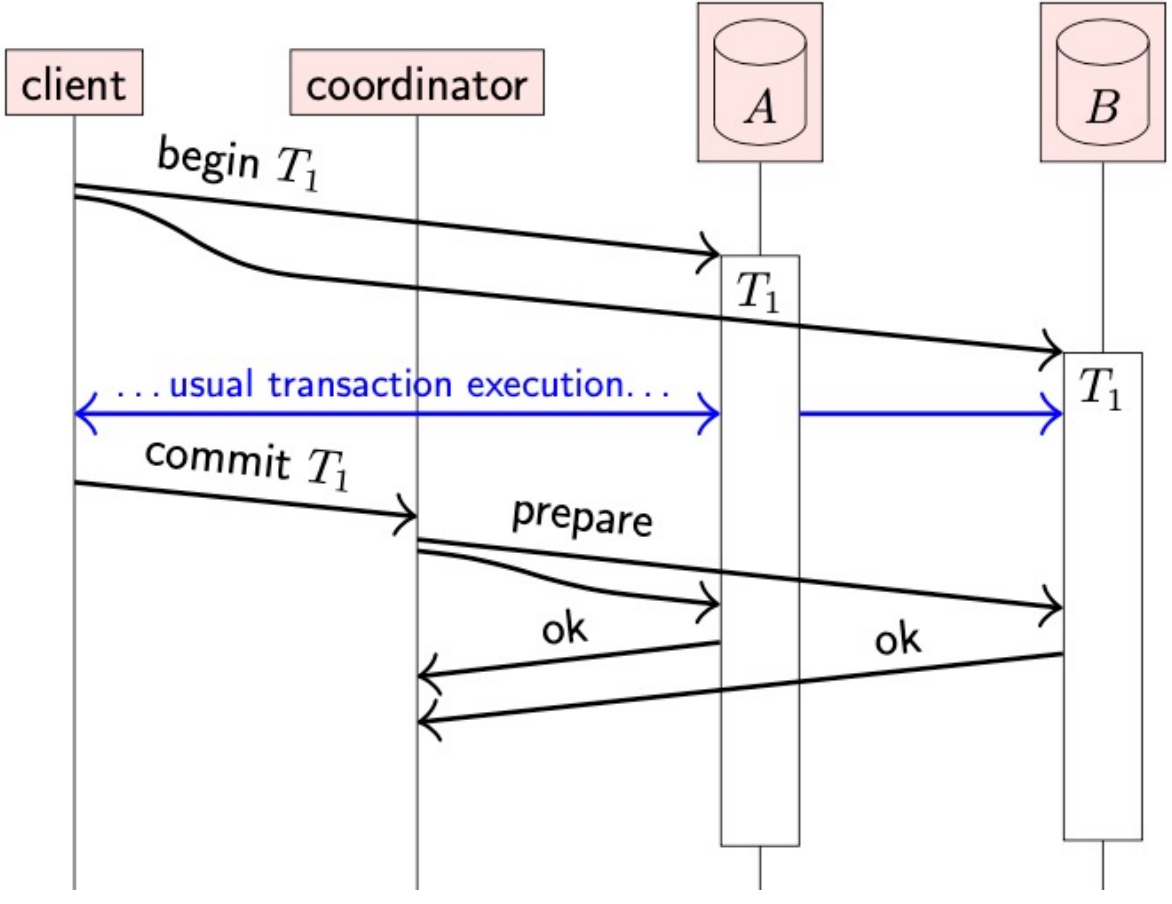
两阶段提交协议


client
coordinator
A
B
begin T_1
T_1
T_1
. . . usual transaction execution. . .
commit T_1

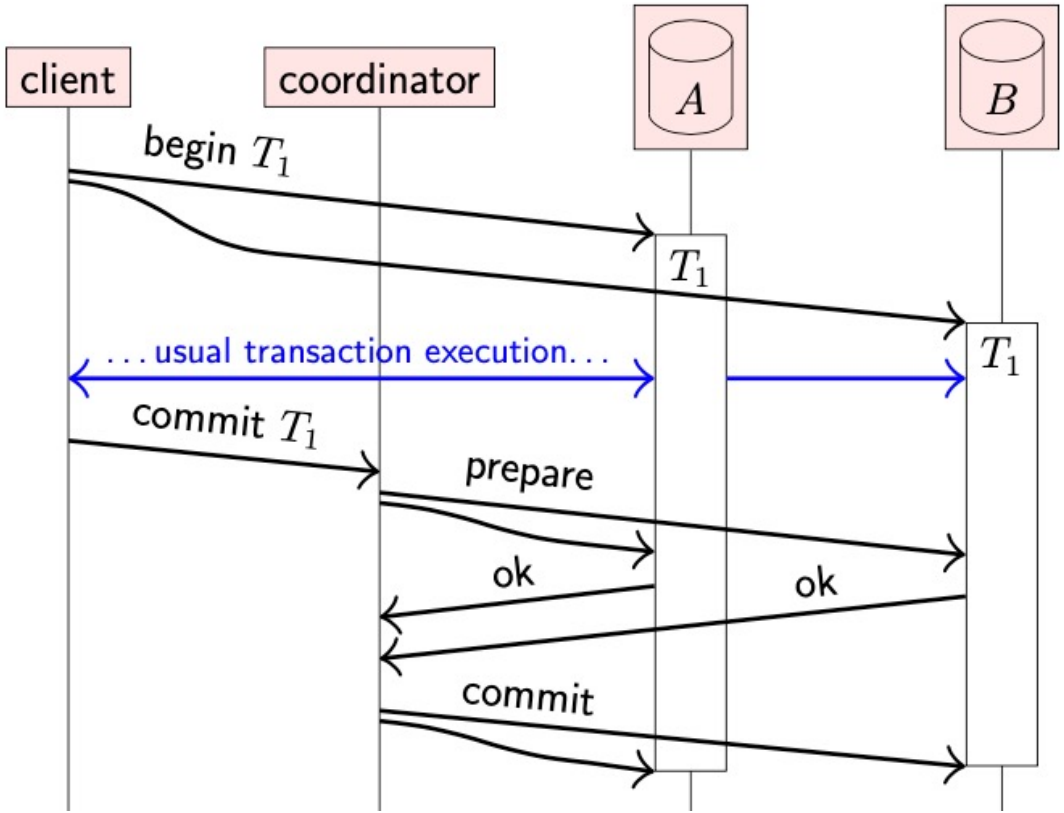
两阶段提交协议


client
coordinator
A
B
begin T_1
T_1
T_1
. . . usual transaction execution. . .
commit T_1
prepare

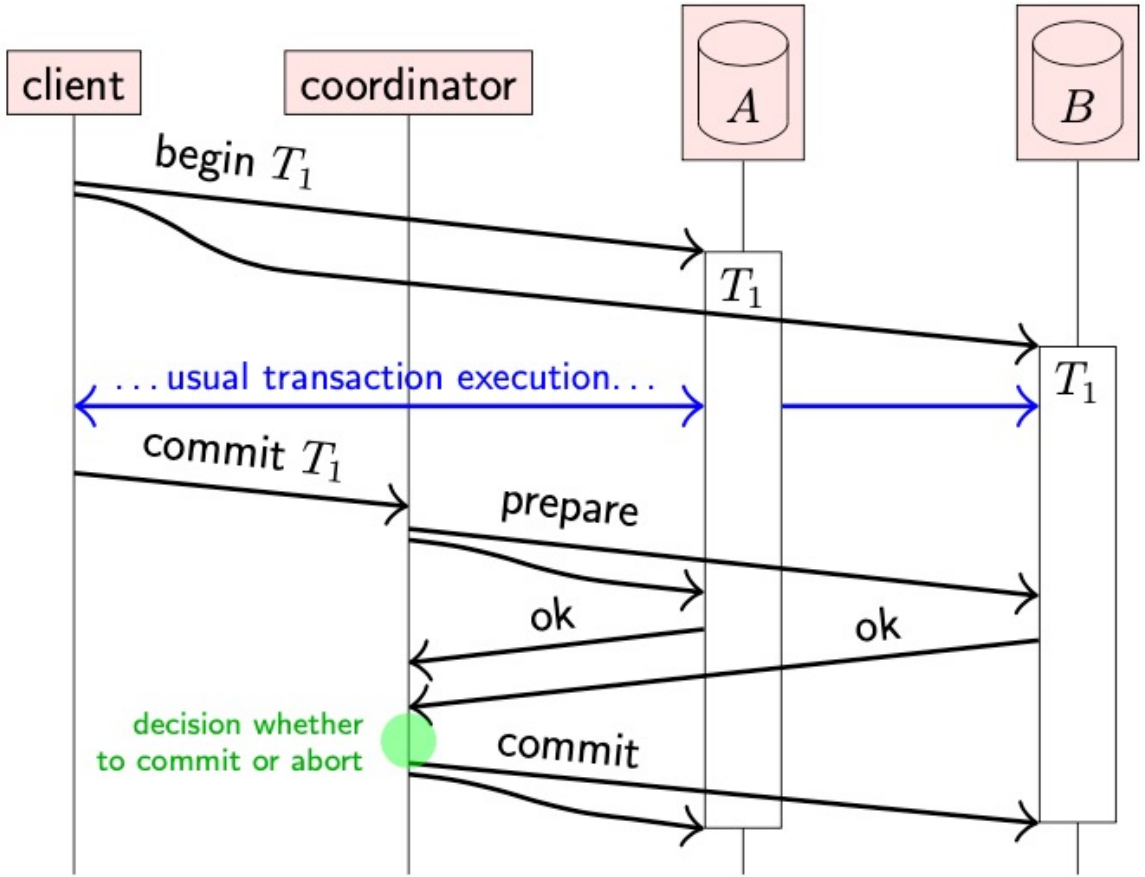
两阶段提交协议


client
coordinator
A
B
begin T_1
T_1
T_1
. . . usual transaction execution. . .
commit T_1
prepare
ok
ok

两阶段提交协议


client
coordinator
A
B
begin T_1
T_1
T_1
. . . usual transaction execution. . .
commit T_1
prepare
ok
ok
commit

两阶段提交协议


client
coordinator
A
B
begin T_1
T_1
T_1
. . . usual transaction execution. . .
commit T_1
prepare
ok
ok
decision whether
to commit or abort
commit

两阶段提交中的协调者

- **如果协调者崩溃了怎么办？（What if the coordinator crashes?）**
  - 协调者会把自己的**决策写入磁盘**。
  - 当它恢复时，会**从磁盘读取决策**，然后把这个决策发送给各个副本；
    如果在崩溃前还没有做出决策，则**中止（abort）事务**。
  - 问题在于：如果协调者在 **发送 prepare 之后、但在广播最终决策之前** 崩溃，
    那么其他节点就**不知道它到底决定了什么**。
  - 对于那些参与该事务的副本来说，在它们对 *prepare* 请求已经回复 “ok” 之后：
    👉 它们既**不能自行提交（commit）**，也**不能自行中止（abort）**，
    否则就有可能**破坏原子性（atomicity）**。
  - 因此，**整个算法会被阻塞**，直到协调者恢复为止。

2PC 的关键性质

- **原子性**（Atomicity）：要么都提交，要么都回滚（在故障恢复正确实现前提下）。
- **一致性**（Consistency）：协议本身不保证业务一致性，但能保证各参与者对“提交/回滚决定”一致。
- **阻塞性**（Blocking）：2PC 最大的问题：协调者在 Phase 1 后崩溃时，参与者可能会“卡住”等决定。

具备容错能力的两阶段提交协议

1. 投票阶段（Vote Phase）

- **每个参与副本在收到 Prepare(T) 后：**
  - 本地检查：**这个事务 T 能不能在我这里提交？**
    - 约束是否满足？
    - 锁冲突吗？
    - 资源够不够？
  - 然后通过 **全序广播（total order broadcast）** 发送：
    - (Vote, T, replicaId, ok)
    - ok = true 表示“我同意提交”，ok = false 表示“我不同意”。
- **如果怀疑某个副本 replicaId 崩溃：**
  - 其他节点会代它对所有相关事务广播：
    - (Vote, T, replicaId, false)
  - 等价于：**崩溃视为“反对票”**，保证最终不会卡在“等它的投票”这一步。

具备容错能力的两阶段提交协议

2. 决策规则（Decision Rules）

- 在每个节点上，随着全序广播源源不断送来 (Vote, ...) 消息，它会不断更新 commitVotes[T]，并依据如下规则做出决定：
  1. **只要有一个 false ⇒ 全员中止（abort）**
     - 一旦有任何副本对事务 T 投了 false，说明存在约束不满足 / 冲突 / 崩溃等问题。
     - 为保持原子性：**所有副本都必须中止。**
     - 节点：
       - 本地调用 performAbort(T)
       - 广播 (Decision, T, abort)
  2. **所有人都 true 且投票齐了 ⇒ 提交（commit）**
     - 当 replicas[T] 中的每个参与者都已经投票，且它们的投票结果全是 true：
     - 节点：
       - 本地调用 performCommit(T)
       - 广播 (Decision, T, commit)
  3. **决策传播（Decision Broadcast）**
     - 即使有些节点“自己还没算完”，也会接收到别人广播的 (Decision, ...)。
     - 只要还没 decided[T]，就会：
       - 直接采用该决策（commit / abort）
       - 执行 performCommit 或 performAbort
     - 确保所有节点最后的**决策一致。**

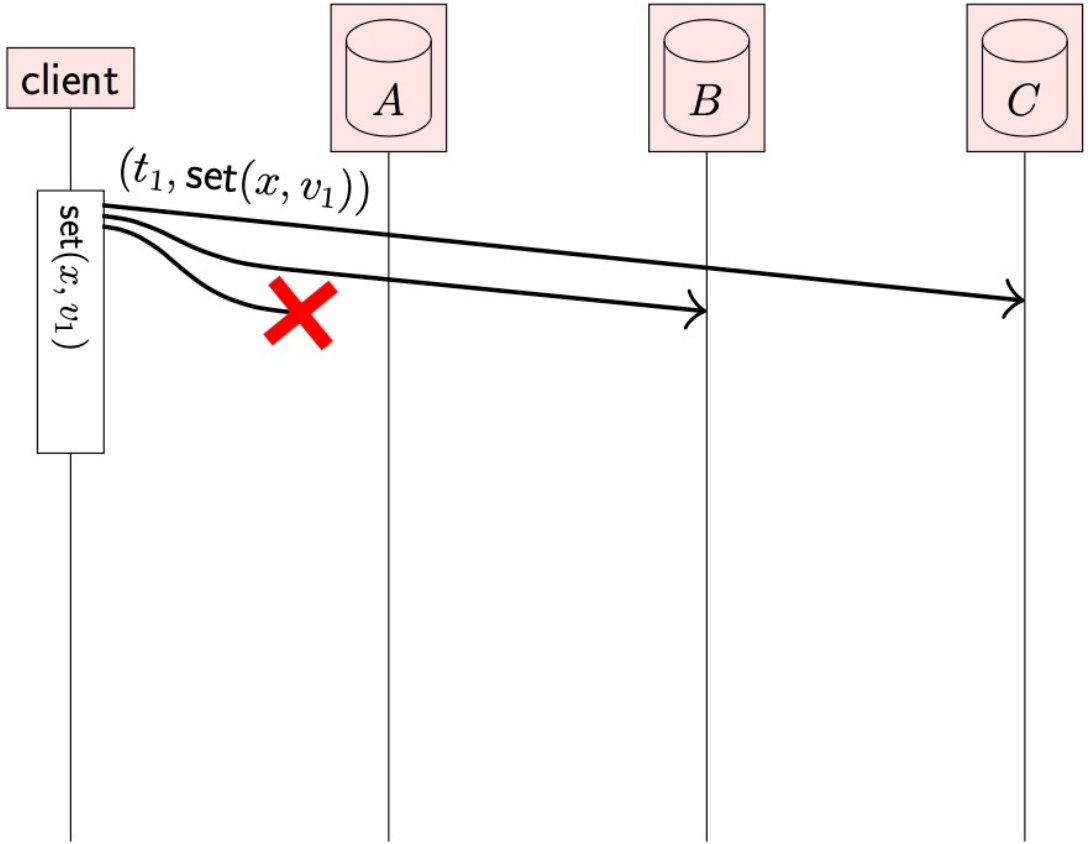
与经典 2PC 的对比

- **经典 2PC：**
  - **单个协调者** 收集所有副本的投票；
  - 协调者挂了 ⇒ 可能长时间阻塞（大家都在等它发最终决策）。
- **容错 2PC（基于全序广播 + Crash 投反对票）：**
  - 投票信息 & 最终决策通过 **total order broadcast** 在所有节点间传播；
  - 节点被怀疑崩溃时，有人会代它发 (Vote, …, false)；
  - 一旦出现 false，所有人都会在同一个全序下尽快做出 **abort 决策**；
  - 避免“永远等不到某个崩溃节点的投票”导致的长期阻塞。

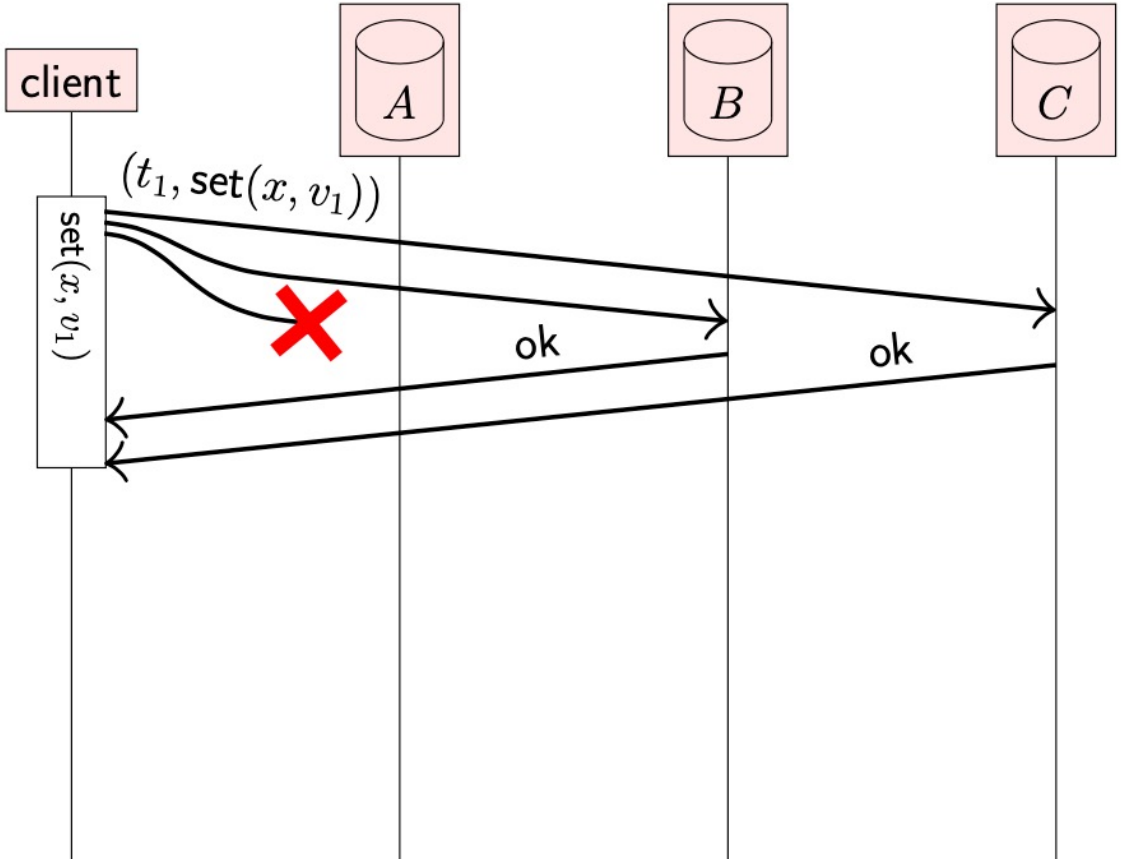
线性一致性

- **多个节点并发访问同一组副本数据，在这种情况下，我们如何定义“一致性”？**
- **最强的一种选择：线性一致性（linearizability）**
  - 非正式地说：**每个操作在它开始之后、结束之前的某个时刻，原子地生效。**
  - 所有操作的行为，好像都是在**同一份数据副本**上执行的（尽管实际上存在多个副本）。
  - 结果：每个操作看到的都是**“最新”的值**，也就是所谓的“**强一致性**”。
  - 线性一致性不仅出现在分布式系统中，在**共享内存并发**中也是一个概念（多核 CPU 上的内存默认并不是线性一致的！）。
- **注意：线性一致性（linearizability）≠ 可串行化（serializability）。**

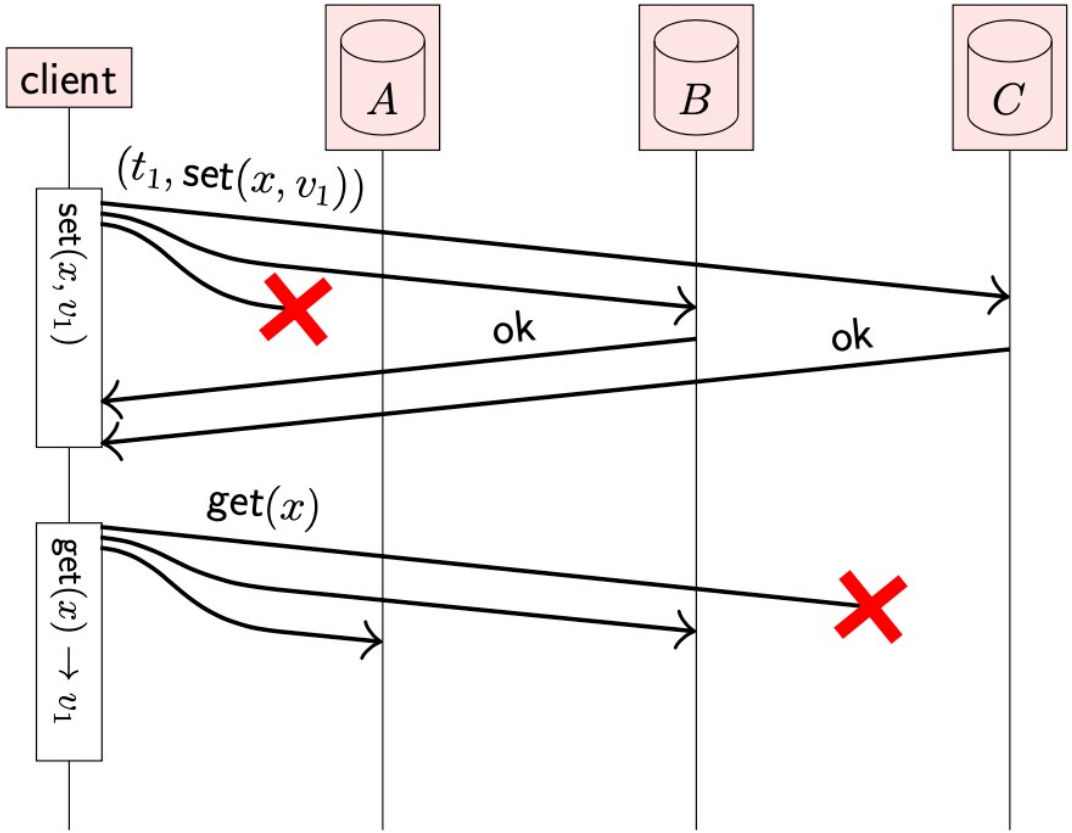
再来看：写后读一致性



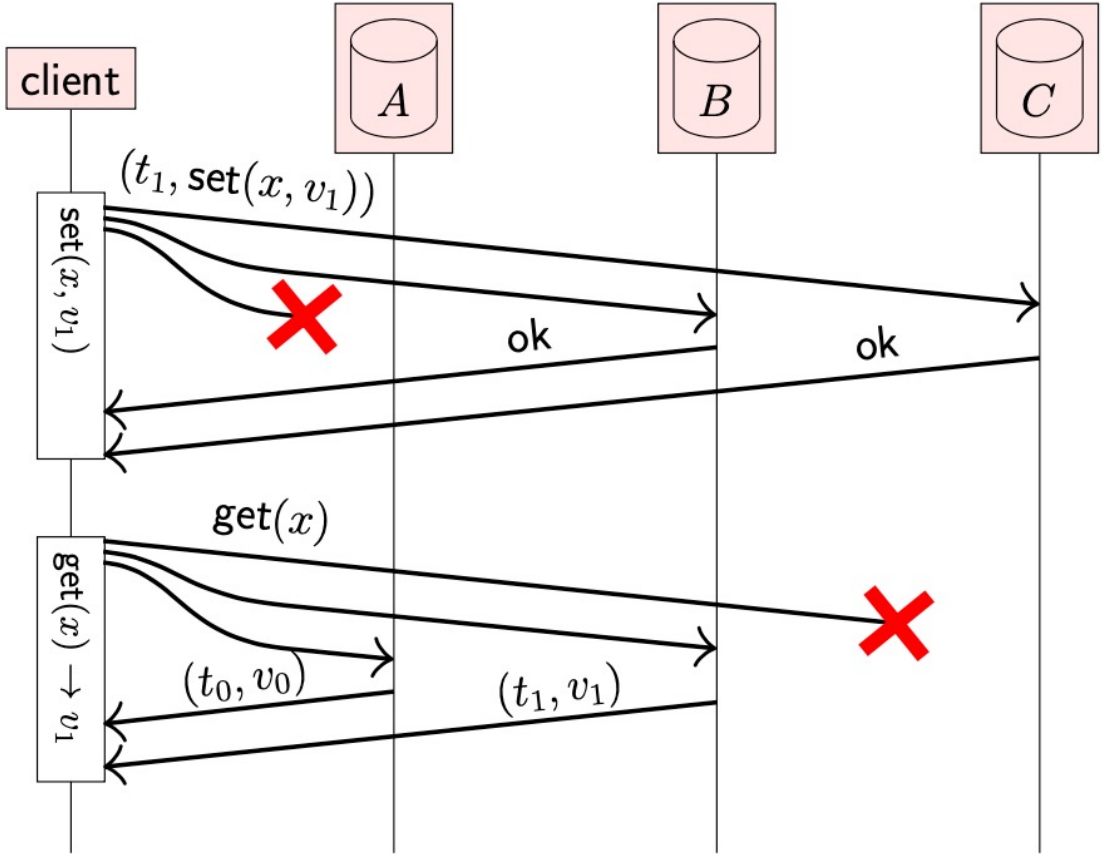
再来看：写后读一致性



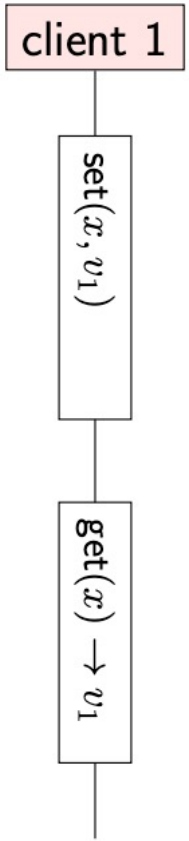
再来看：写后读一致性



再来看：写后读一致性

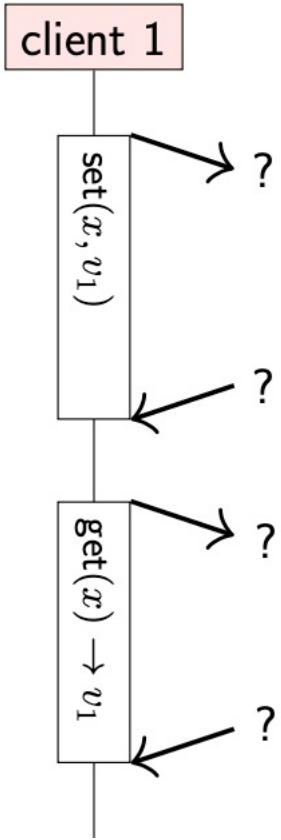

client
A
B
C
$(t_1, \text{set}(x, v_1))$
set(x, v_1)
ok
ok
get(x)
get$(x) \to v_1$
(t_0, v_0)
(t_1, v_1)

从客户端的视角来看



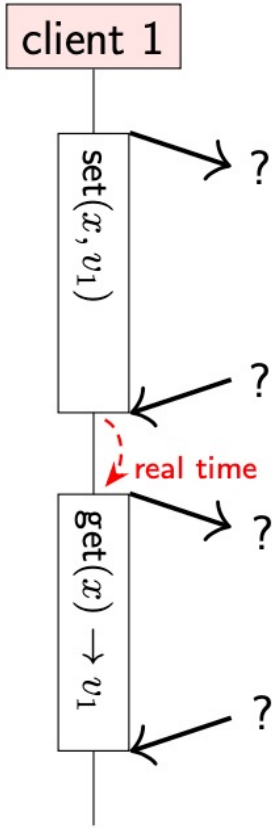
- 关注客户端能观察到的行为：操作何时返回，以及返回了什么？

从客户端的视角来看



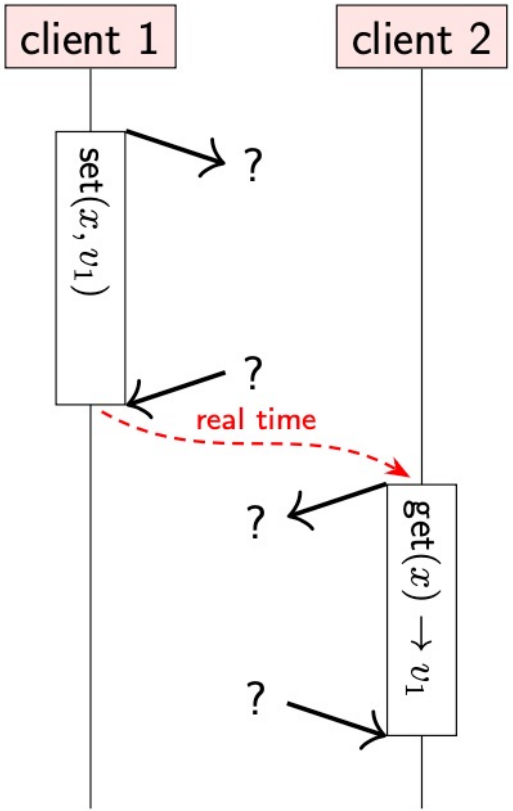
- 关注客户端能观察到的行为：操作何时返回，以及返回了什么？
- 忽略复制系统在内部是如何实现的。

从客户端的视角来看



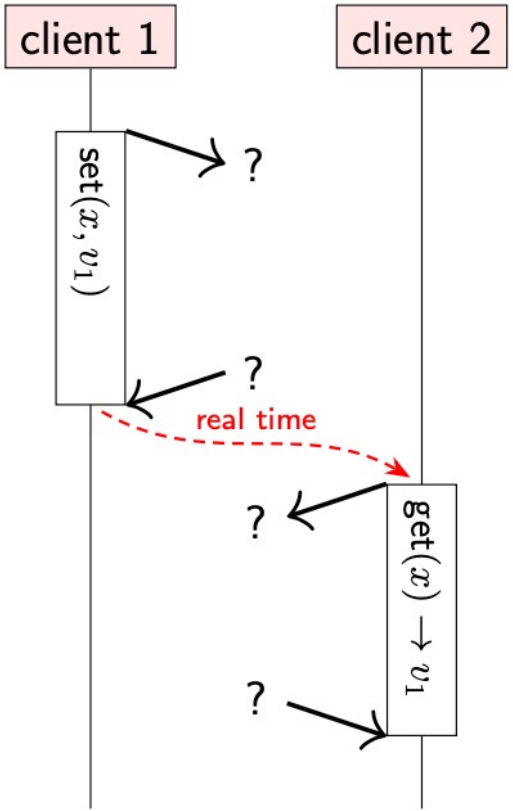
- 关注客户端能观察到的行为：操作何时返回，以及返回了什么？
- 忽略复制系统在内部是如何实现的
- 操作 A 是否在操作 B 开始之前就已经完成？

从客户端的视角来看



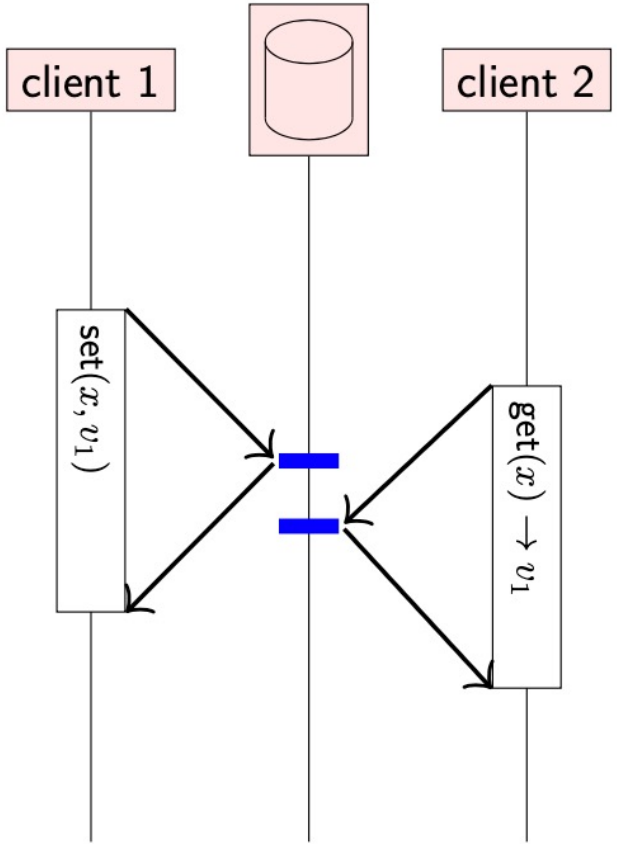
- 关注客户端能观察到的行为：操作何时返回，以及返回了什么？
- 忽略复制系统在内部是如何实现的
- 操作 A 是否在操作 B 开始之前就已经完成？
- 即使这两个操作发生在不同的节点上，也要这样判断吗？

从客户端的视角来看



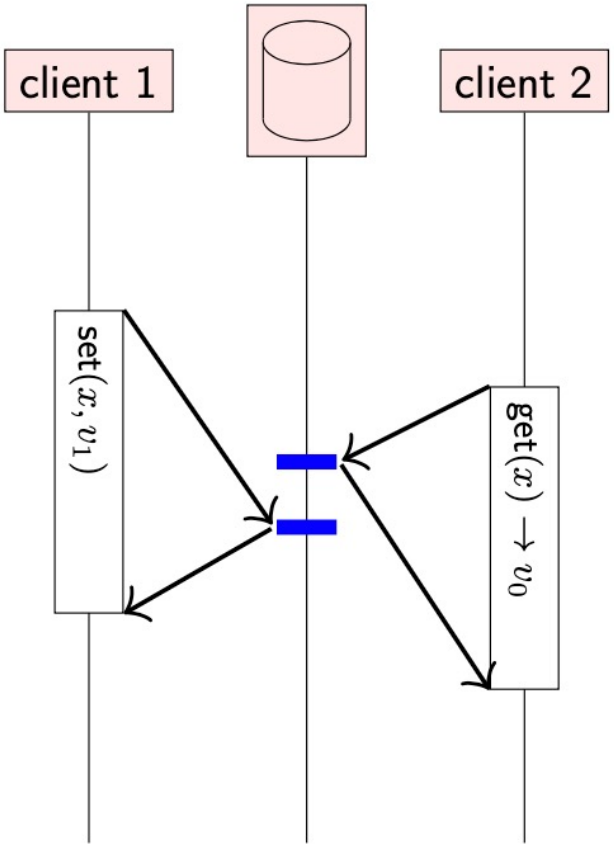
- 关注客户端能观察到的行为：操作何时返回，以及返回了什么？
- 忽略复制系统在内部是如何实现的
- 操作 A 是否在操作 B 开始之前就已经完成？
- 即使这两个操作发生在不同的节点上，也要这样判断吗？
- 我们希望客户端 2 能读到客户端 1 写入的值，即使这两个客户端之间从未直接通信过！

在时间上互相重叠的操作



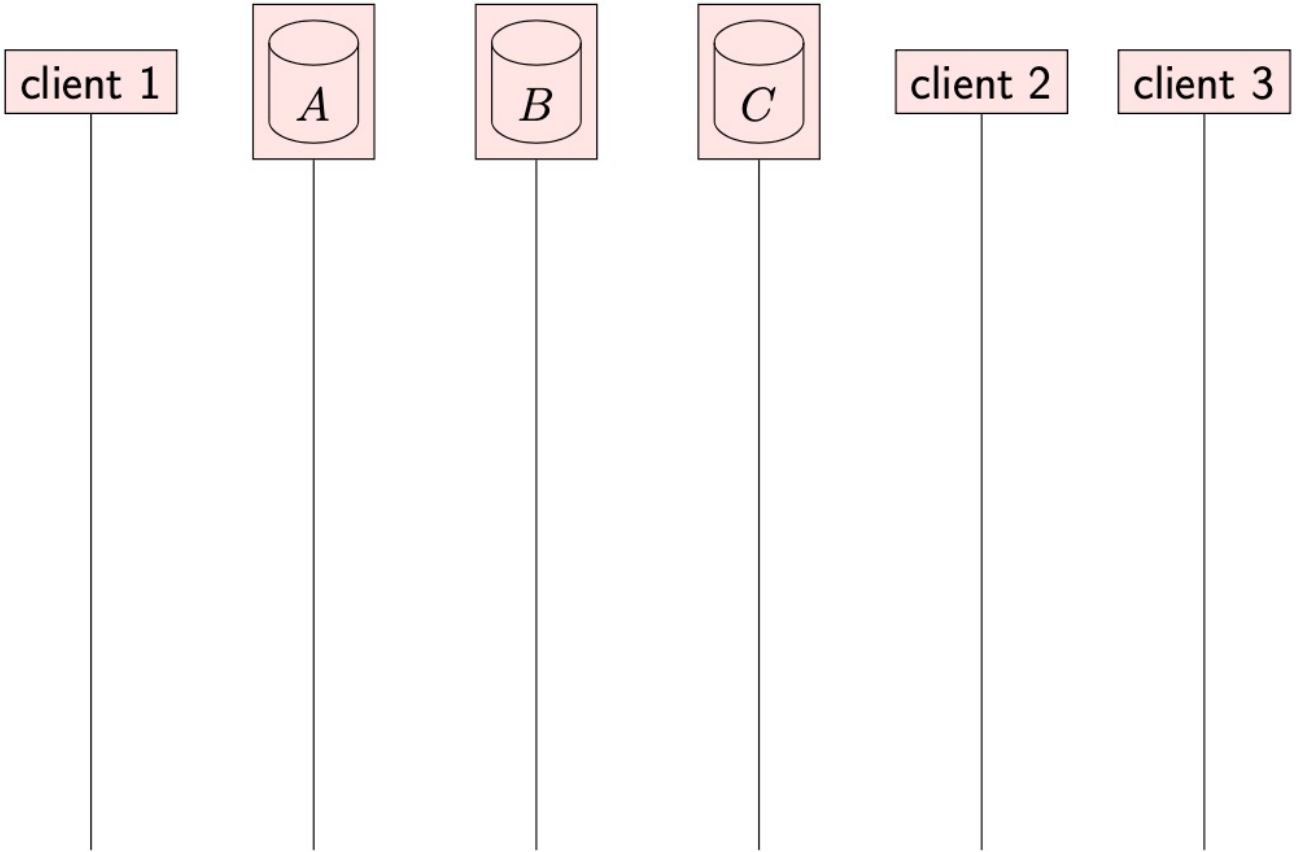
- 客户端 2 的 get 操作 和客户端 1 的 set 操作 在时间上是重叠的。
- 有可能是 set 操作先 生效 吗？

在时间上互相重叠的操作

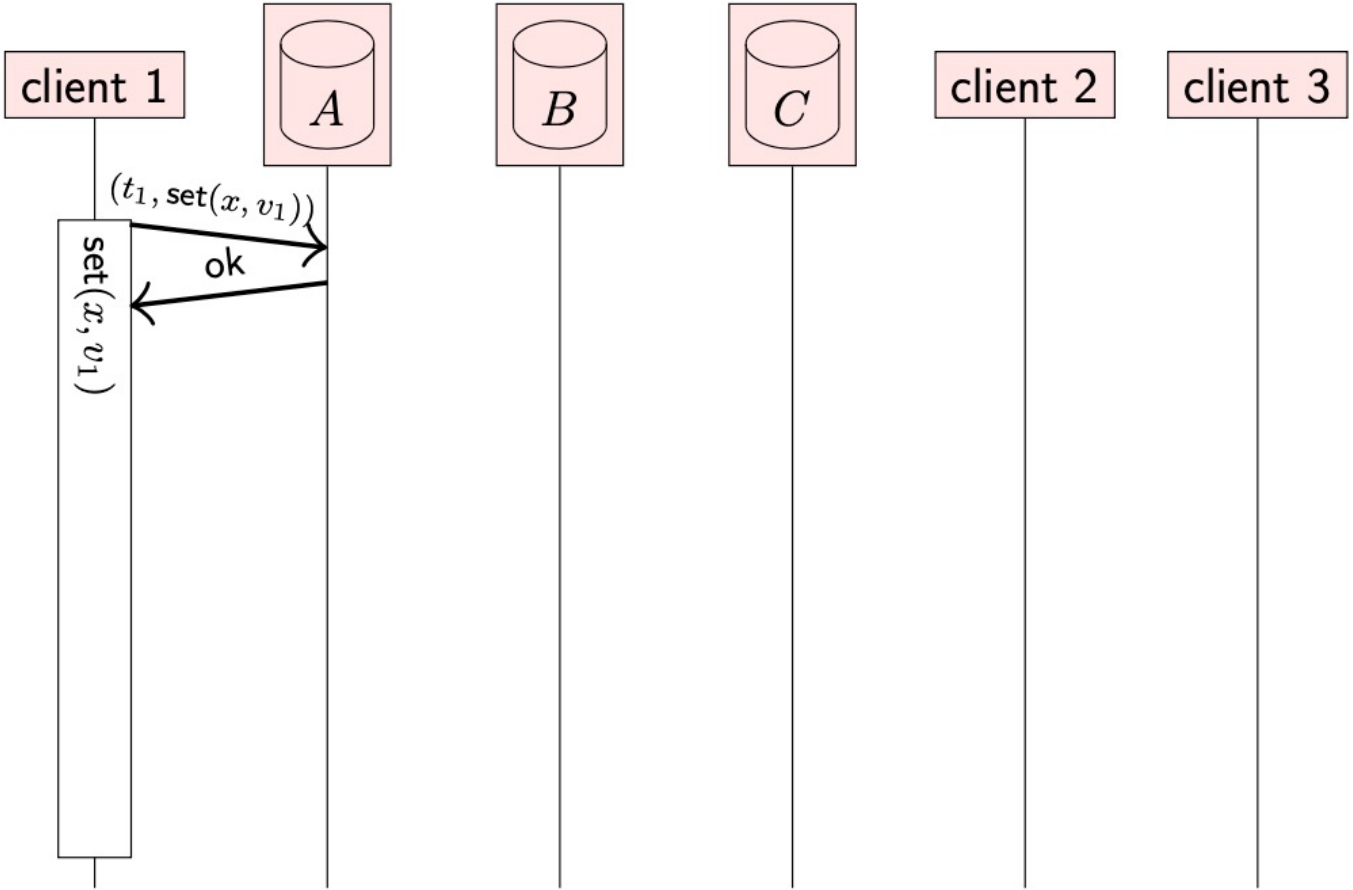


- 客户端 2 的 get 操作 和客户端 1 的 set 操作 在时间上是重叠的。
- 有可能是 set 操作先 生效 吗？
- 同样有可能是 get 操作 先被执行。
- 在这个例子中，无论哪种先发生都是可以接受的。

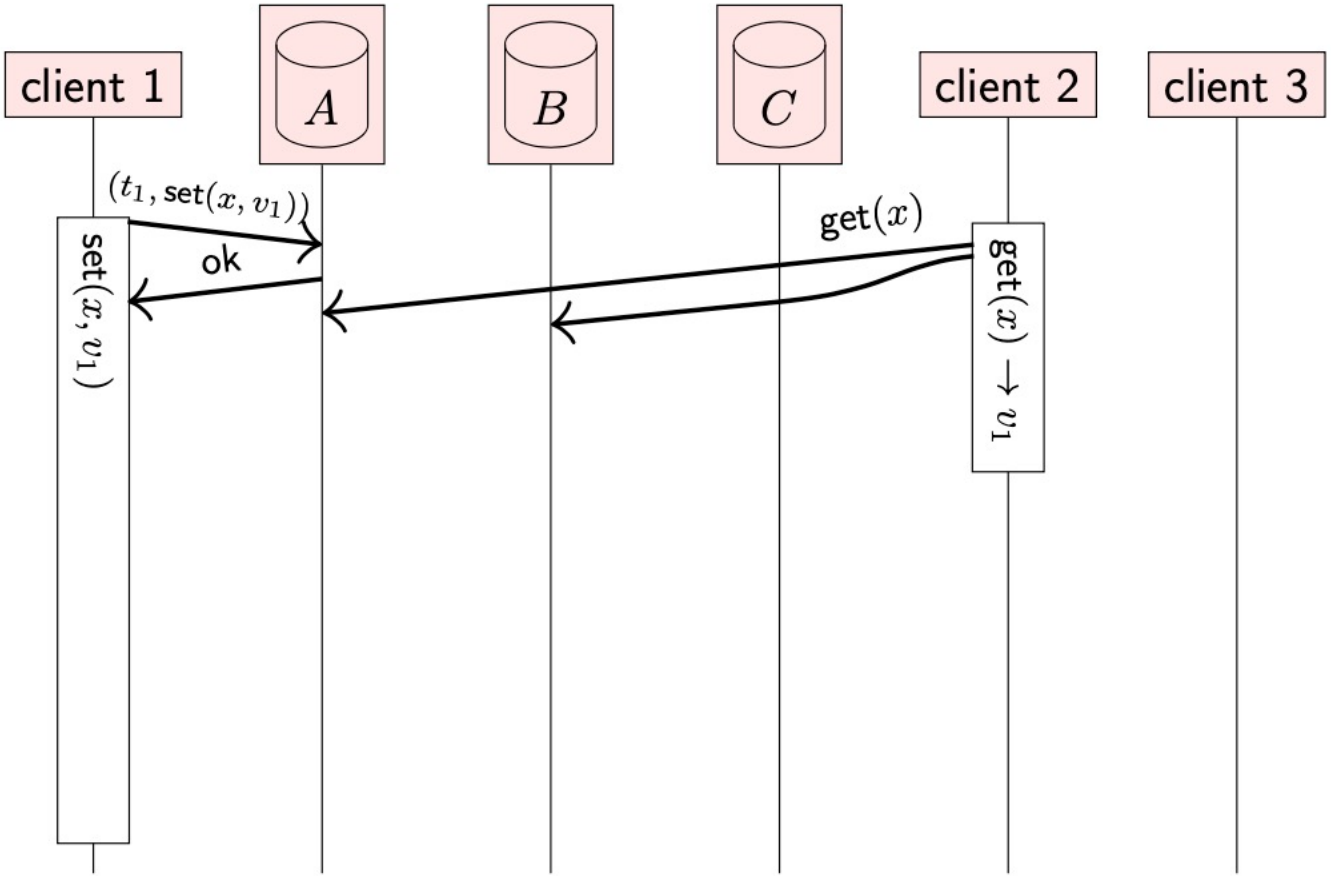
即使使用了读/写仲裁，系统仍然不是线性一致



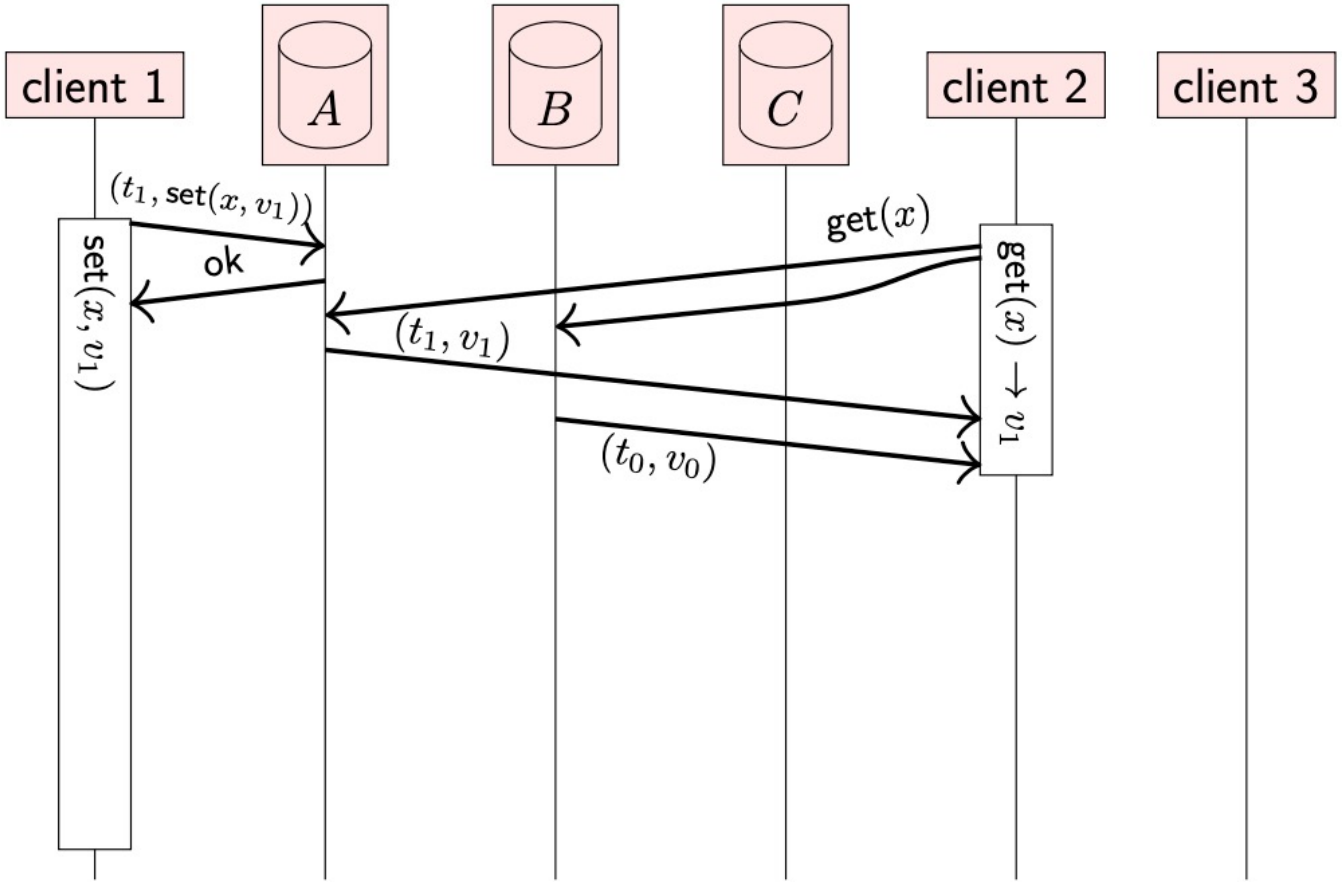
即使使用了读/写仲裁，系统仍然不是线性一致



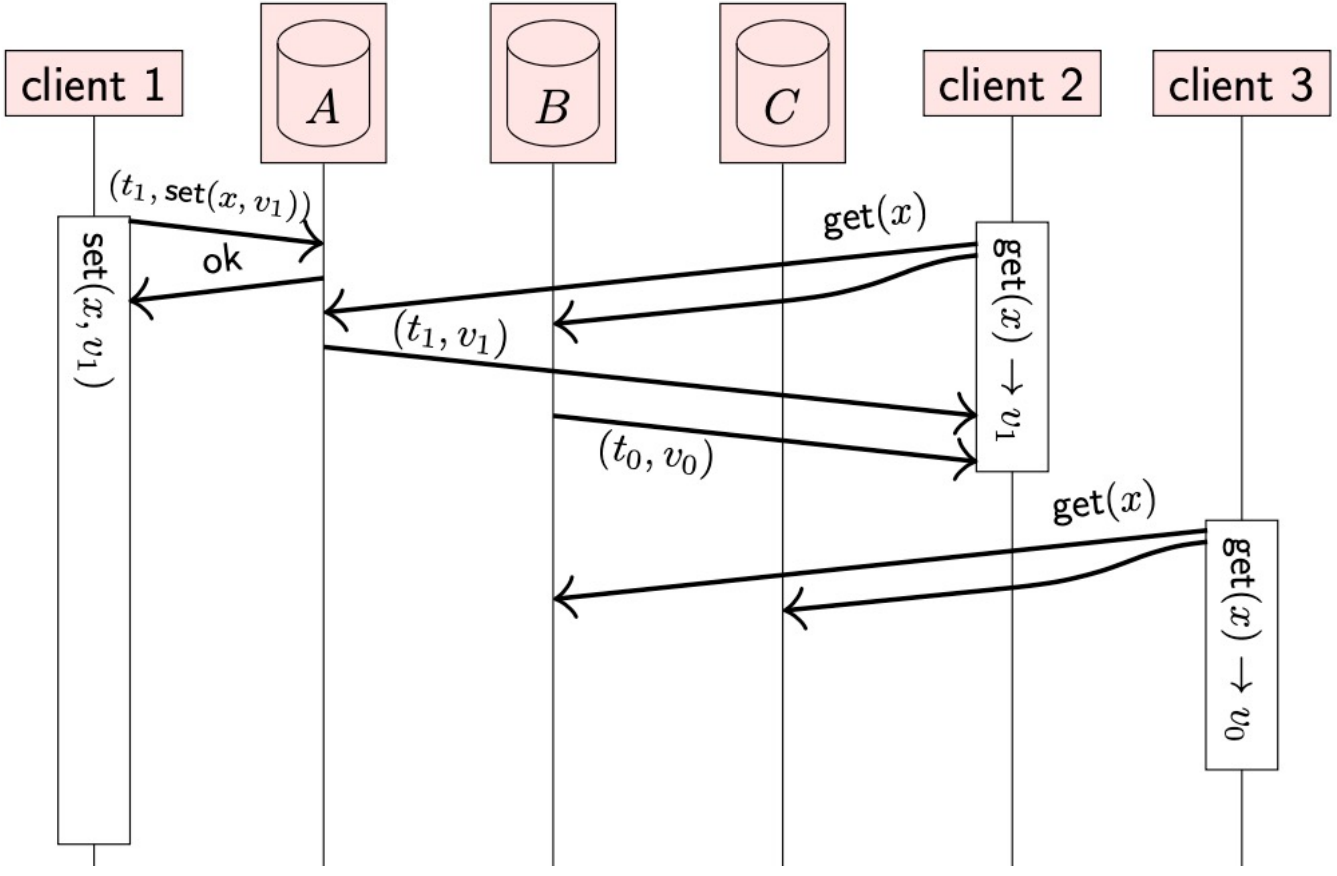
即使使用了读/写仲裁，系统仍然不是线性一致



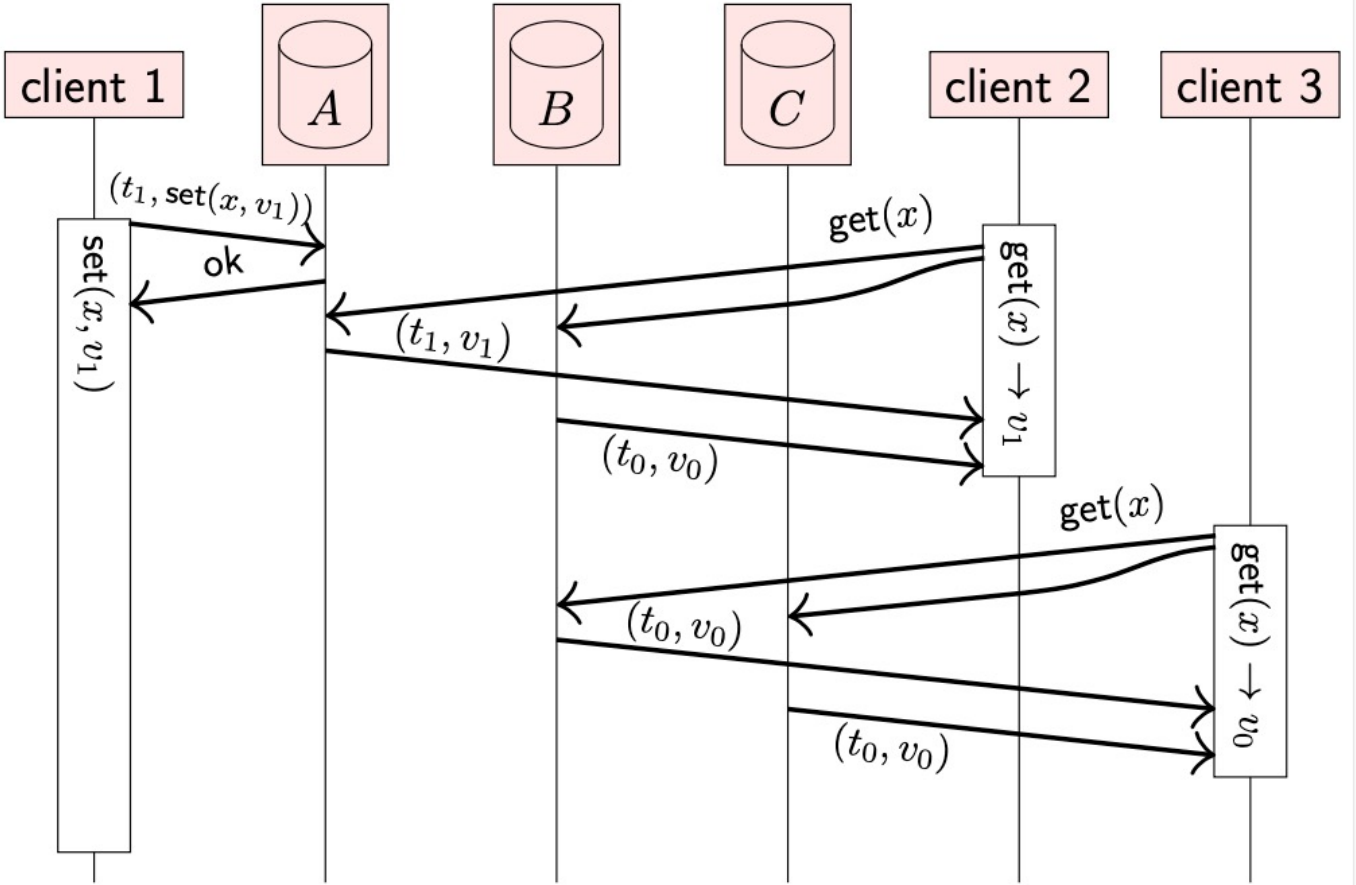
即使使用了读/写仲裁，系统仍然不是线性一致



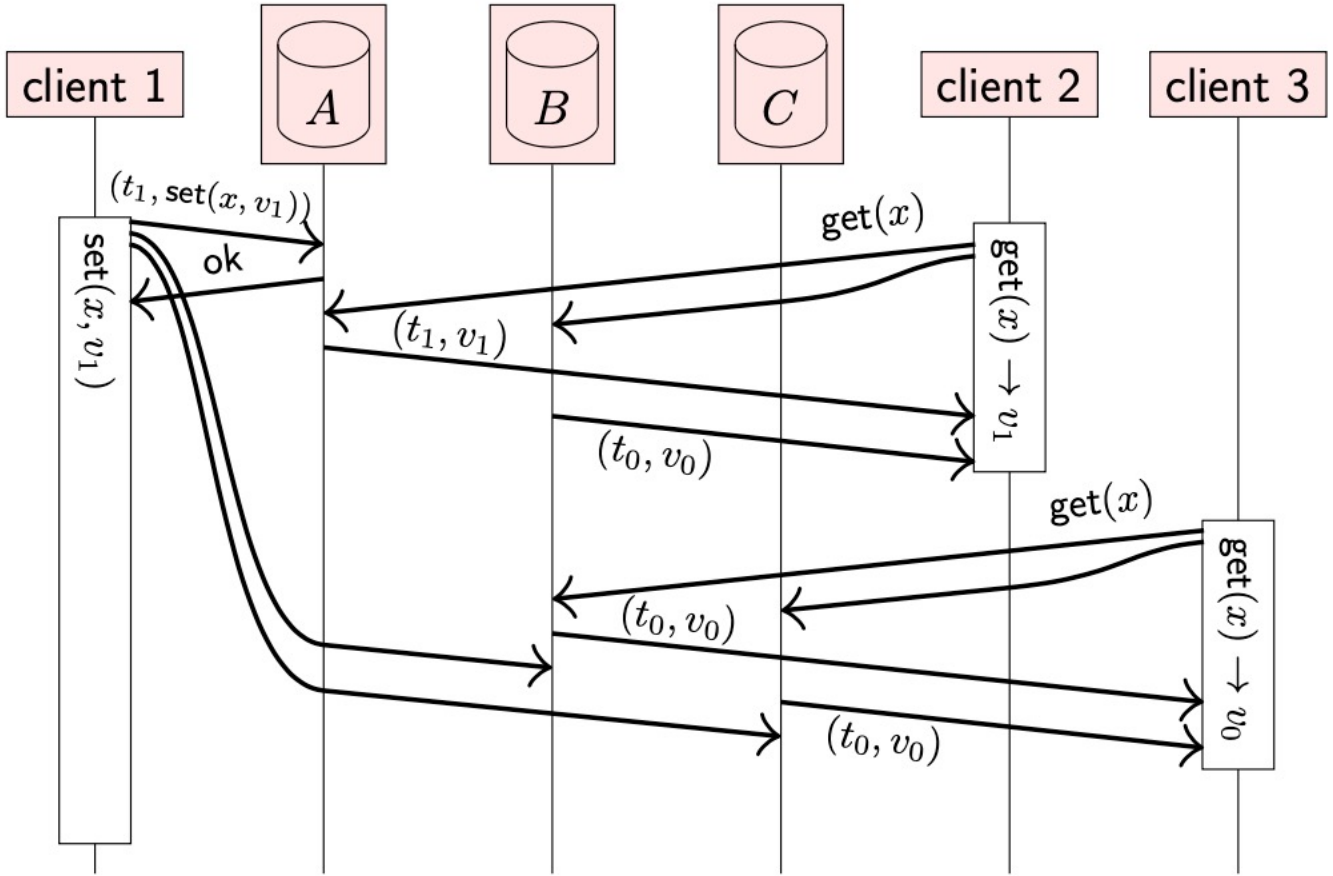
即使使用了读/写仲裁，系统仍然不是线性一致



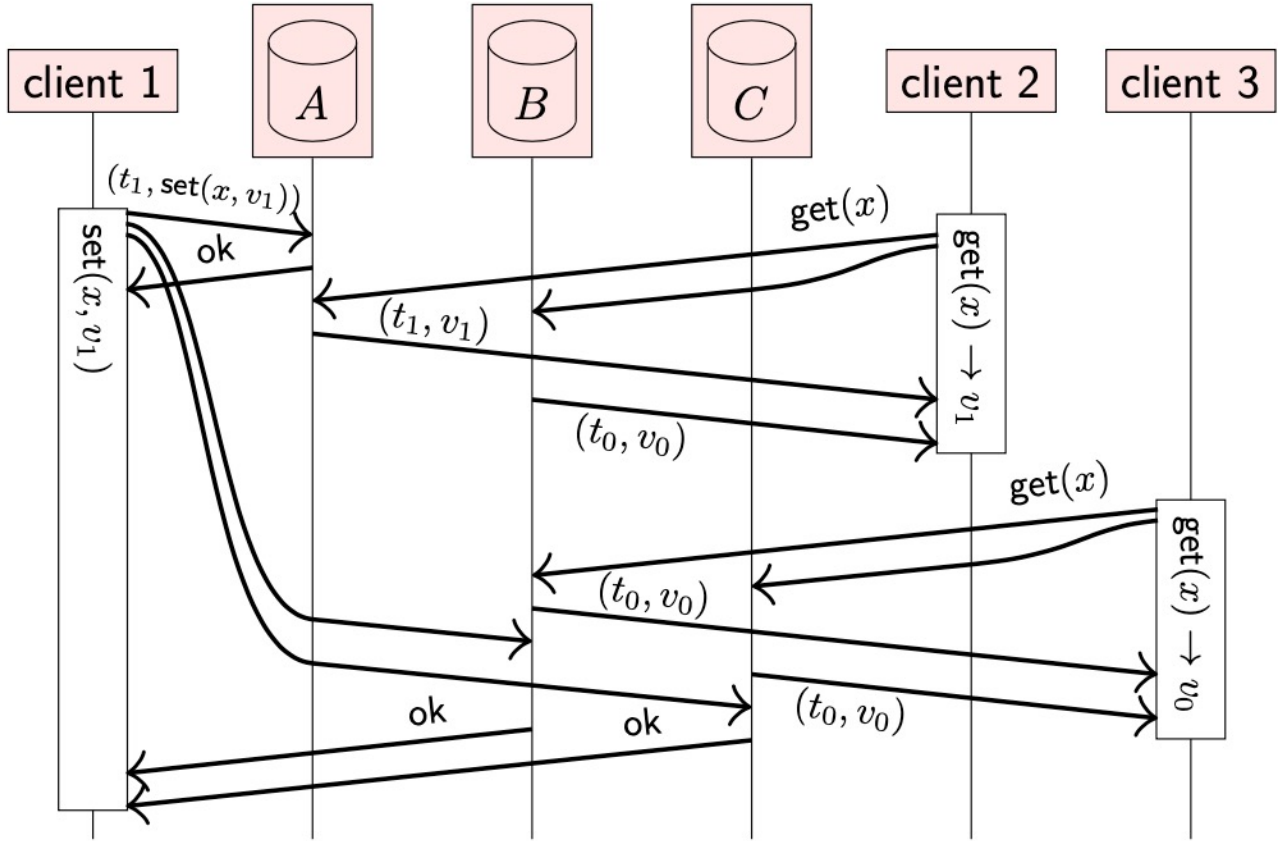
即使使用了读/写仲裁，系统仍然不是线性一致



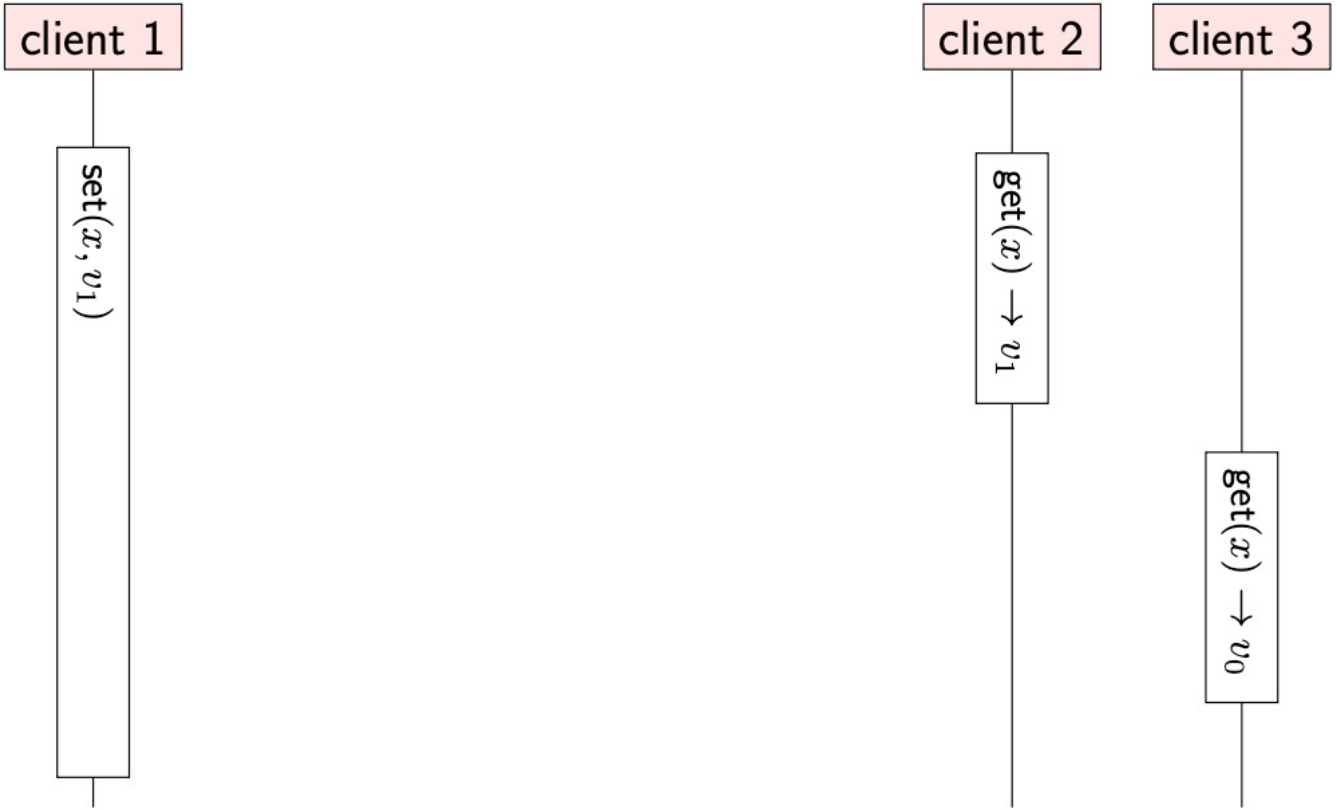
即使使用了读/写仲裁，系统仍然不是线性一致



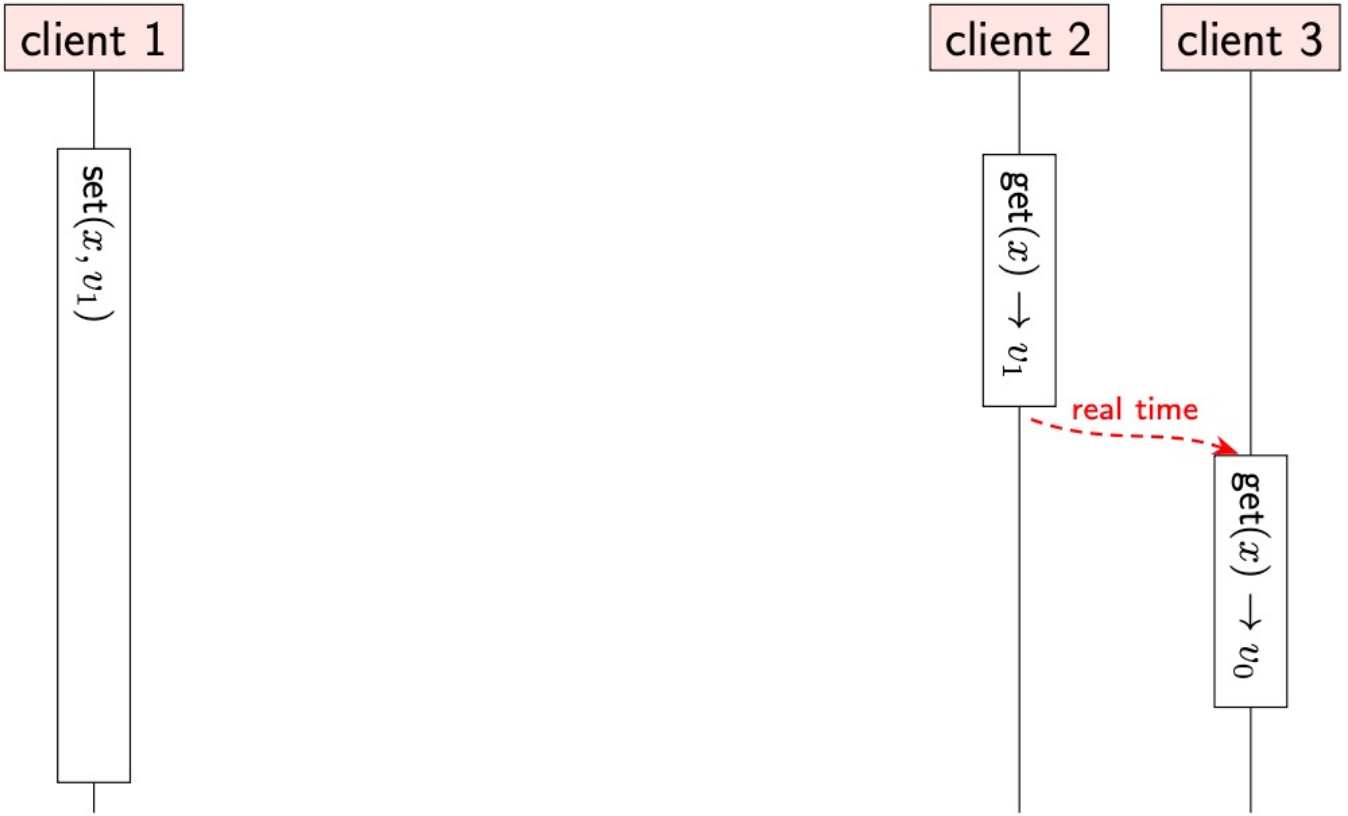
即使使用了读/写仲裁，系统仍然不是线性一致



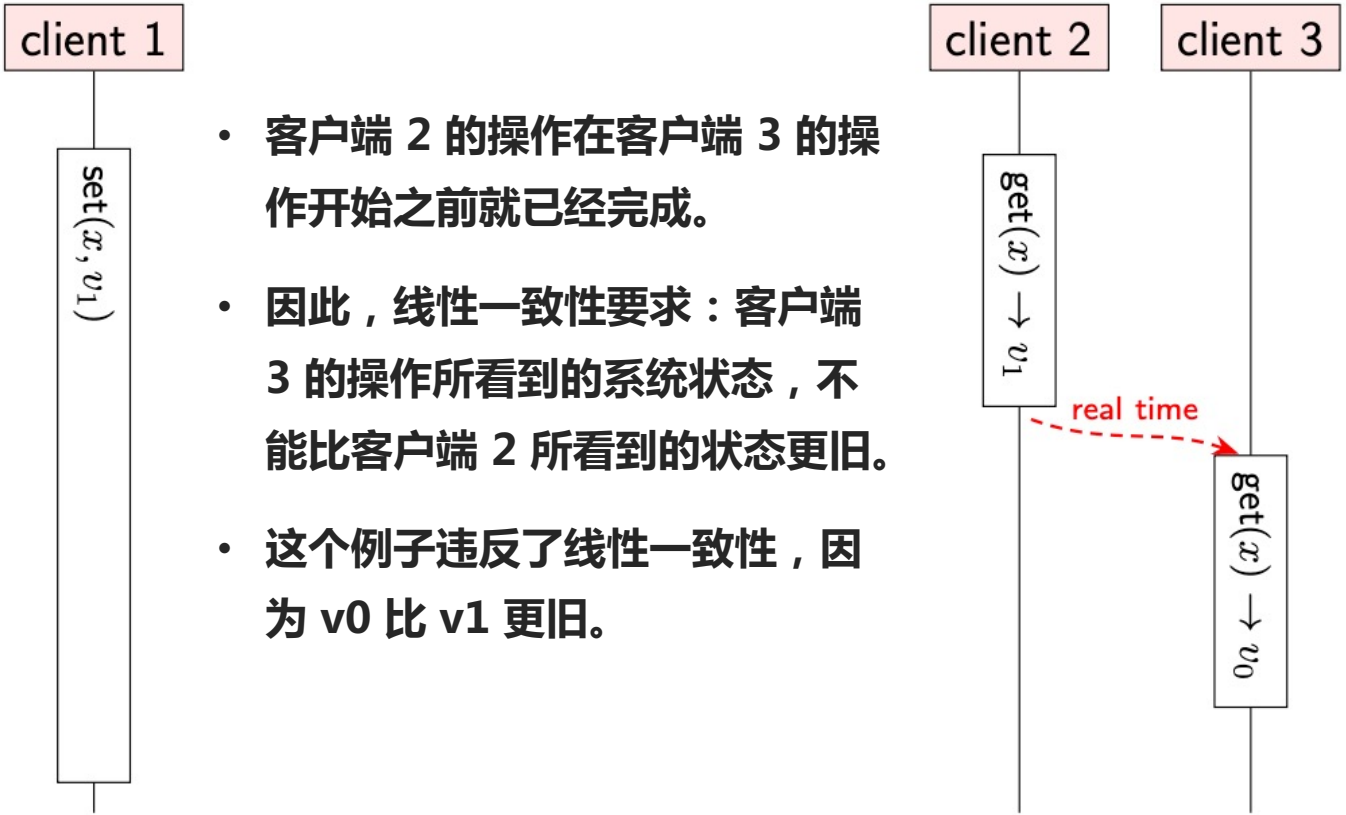
即使使用了读/写仲裁，系统仍然不是线性一致



即使使用了读/写仲裁，系统仍然不是线性一致



即使使用了读/写仲裁，系统仍然不是线性一致

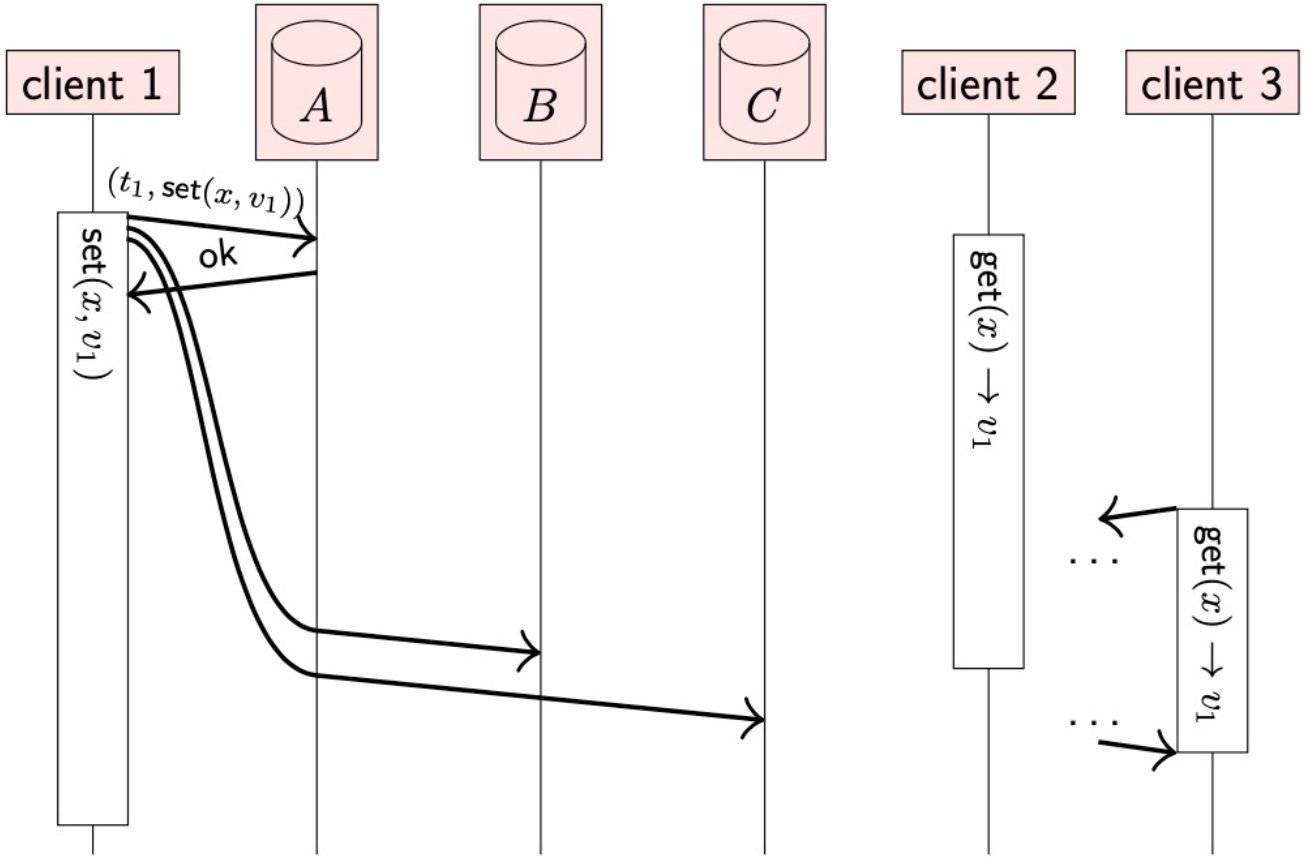


- 客户端 2 的操作在客户端 3 的操作开始之前就已经完成。
- 因此，线性一致性要求：客户端 3 的操作所看到的系统状态，不能比客户端 2 所看到的状态更旧。
- 这个例子违反了线性一致性，因为 v0 比 v1 更旧。

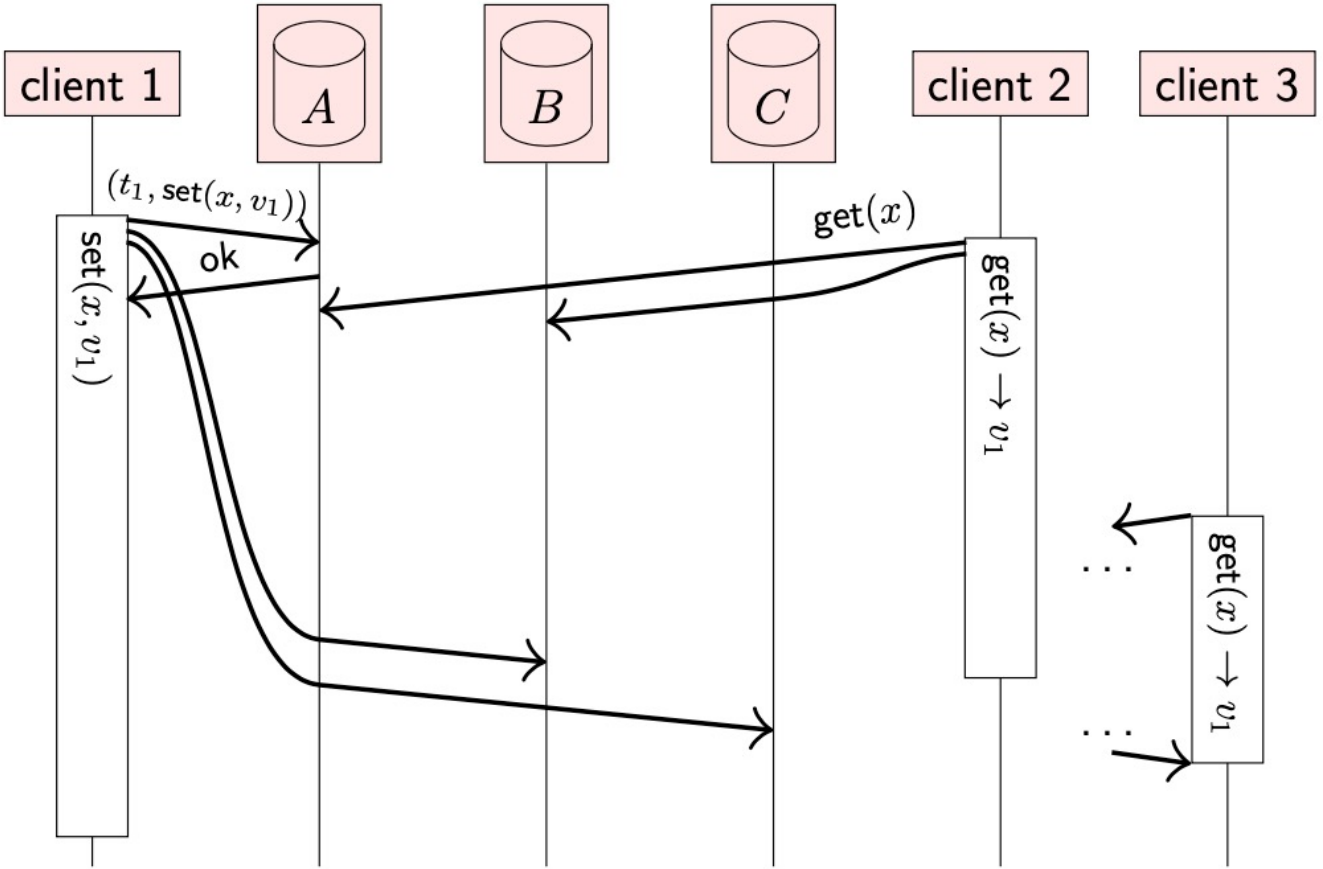
ABD（Attiya–Bar-Noy–Dolev）协议

- **在异步消息传递 + 可能崩溃（crash-stop）的系统里，面向一组副本的读写操作如何实现一个线性一致性？**
  - 在有 N 个副本存同一个变量 x 时，面对：
    - 网络延迟任意、消息可能乱序/重复
    - 最多 f 个副本崩溃
  - ABD保证：
    - write(v) 完成后，之后任何 read() 都不会读到比它更旧的值
    - 并发读写时，也能给每个操作找到一个“瞬时生效点”

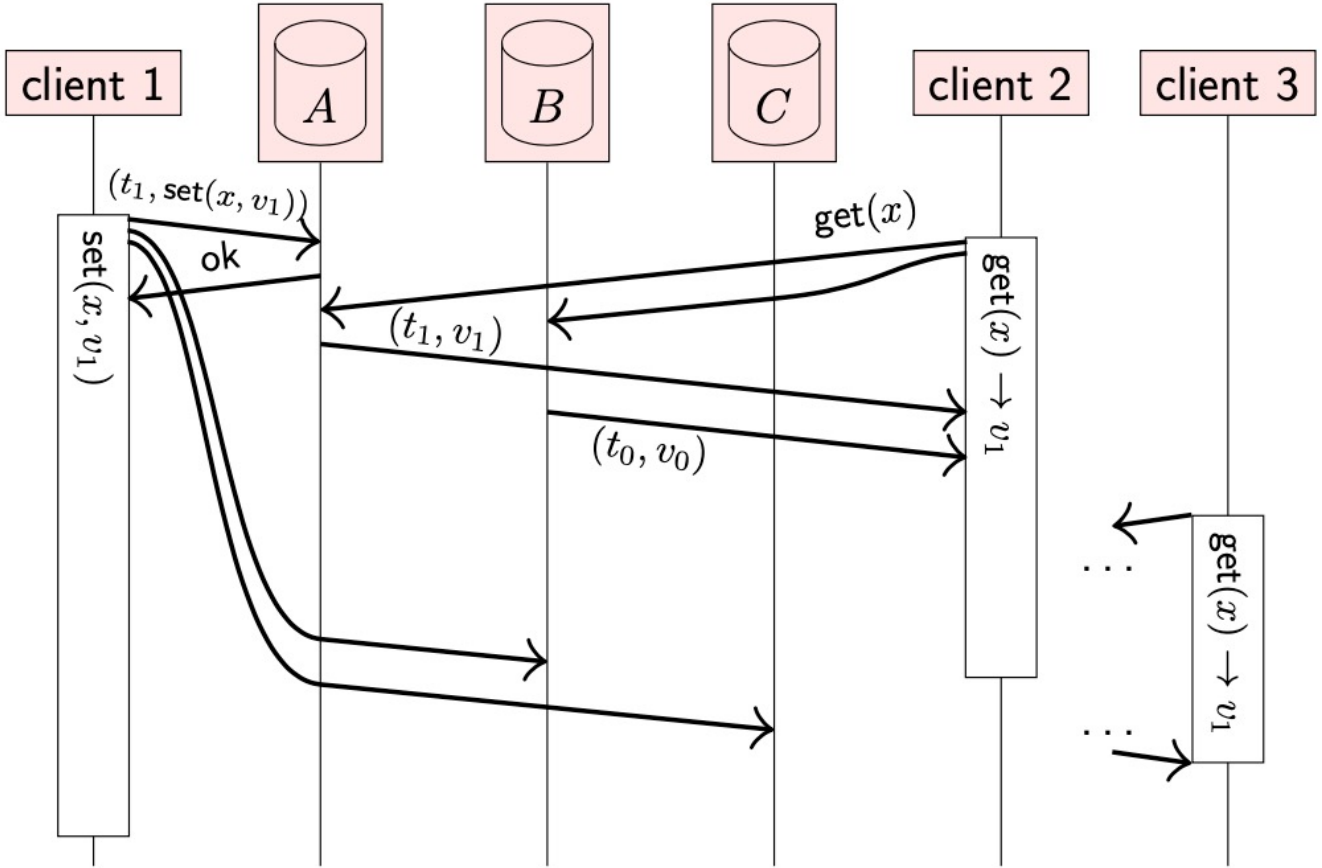
ABD：让仲裁读/写实现线性一致性



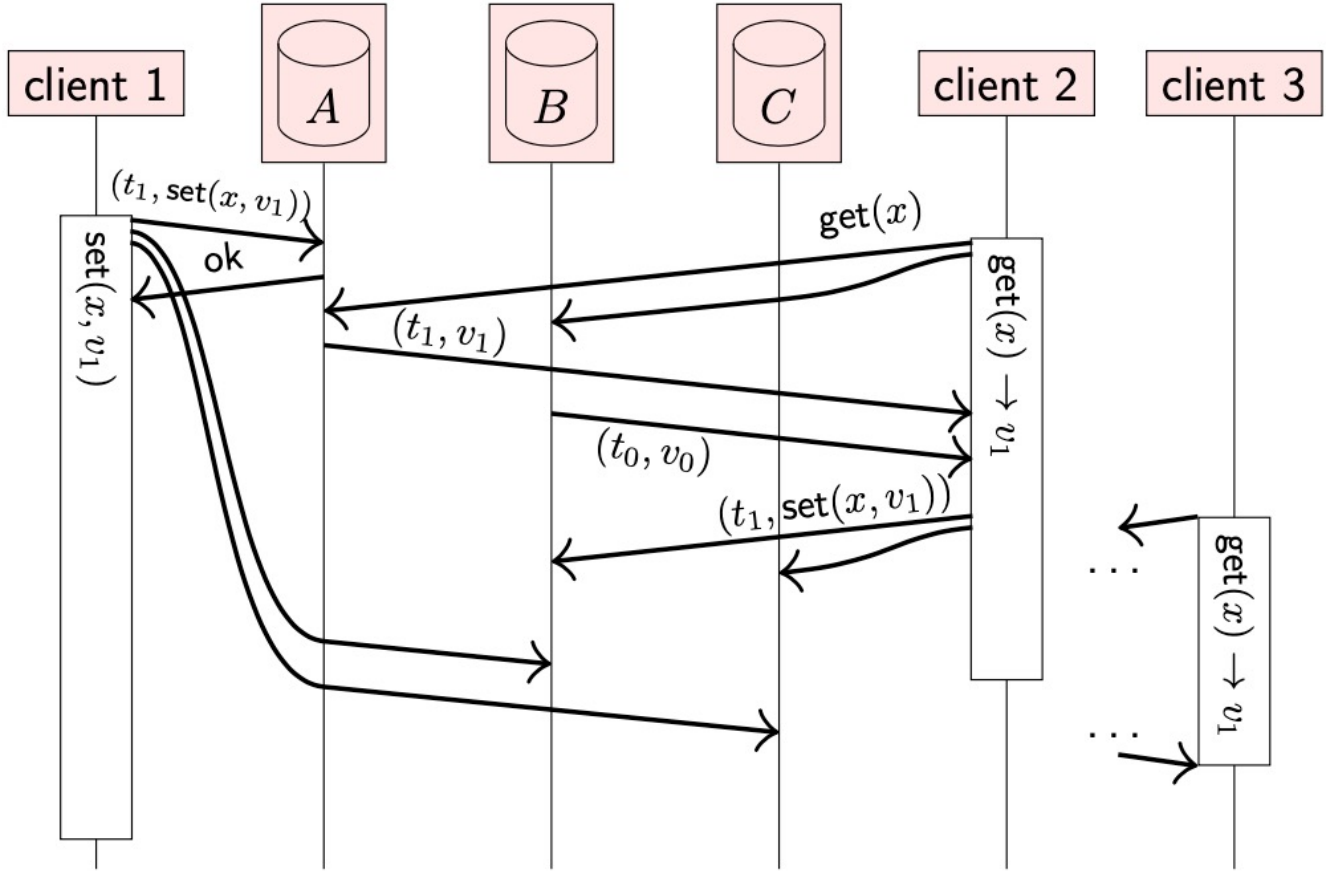
ABD：让仲裁读/写实现线性一致性



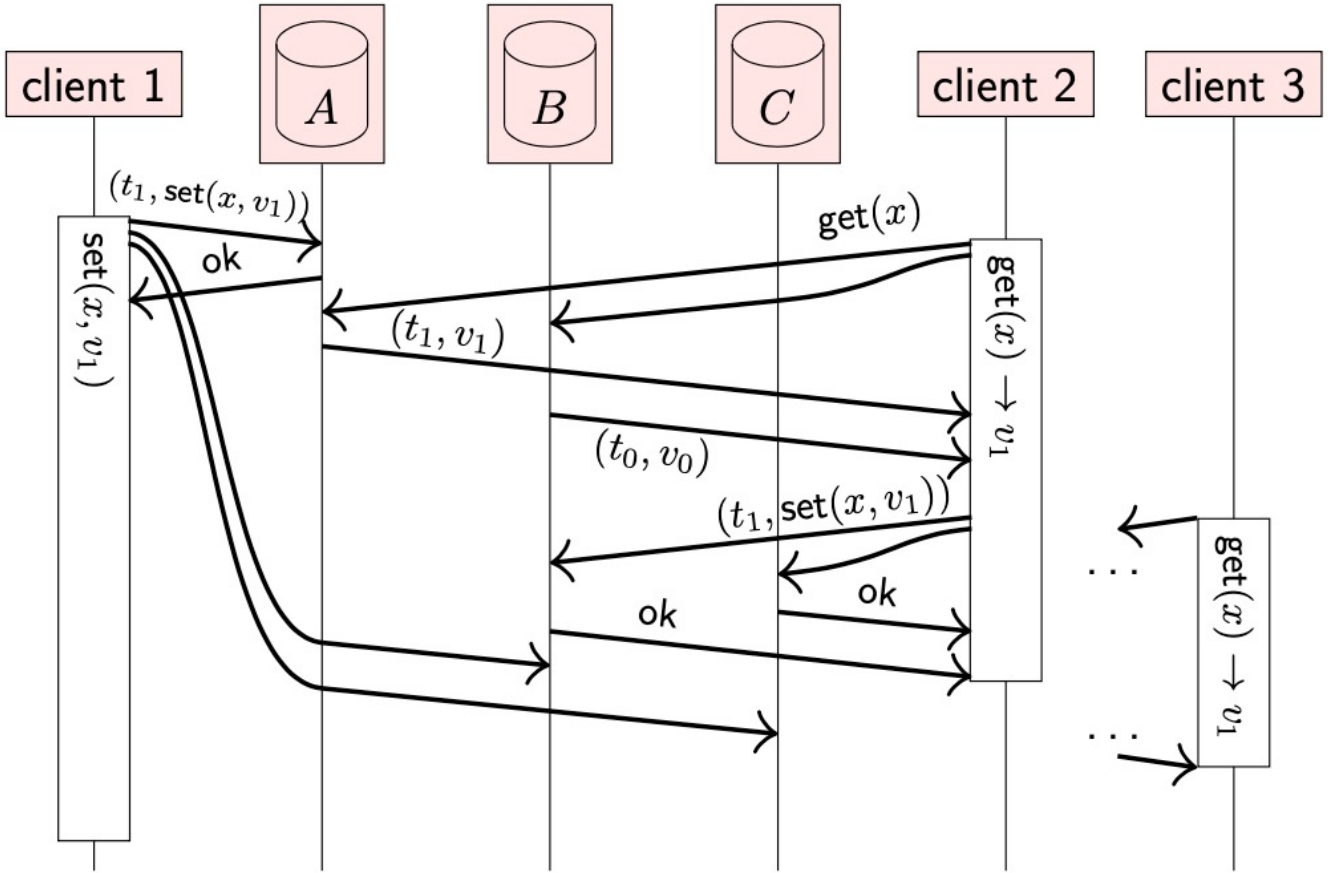
ABD：让仲裁读/写实现线性一致性



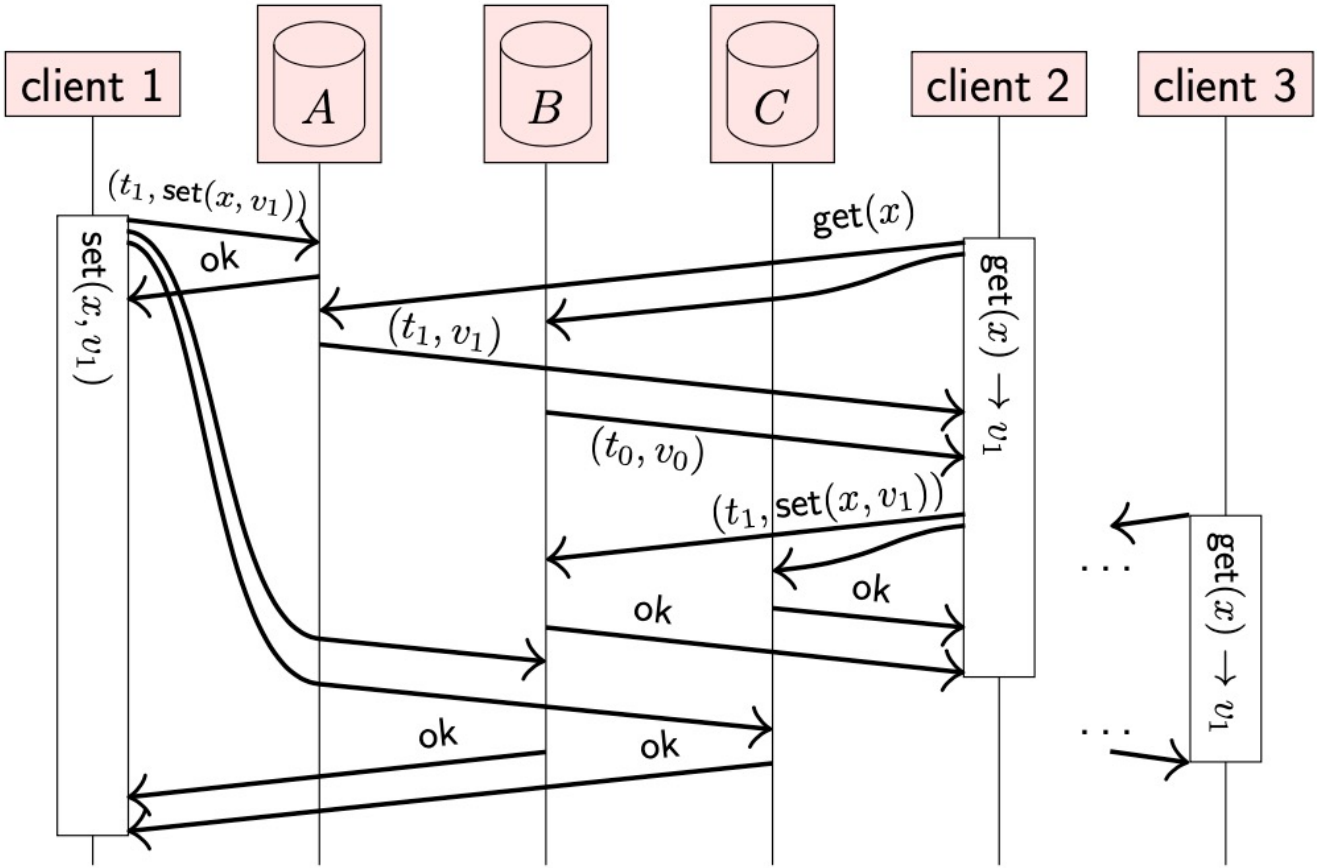
ABD：让仲裁读/写实现线性一致性



ABD：让仲裁读/写实现线性一致性



ABD：让仲裁读/写实现线性一致性



性能与局限

- 读：2 轮（Query + Write-back）
- 能容忍：f < N/2 崩溃
- 提供的是原子寄存器读/写；如果你要 CAS/事务，通常要用更强的共识/原子广播

不同类型操作之间的线性一致性保证

- **这保证了 get（仲裁读） 和 set（对仲裁集合的盲写） 的线性一致性。**
  - 当某个操作完成时，它读到 / 写入的值已经被存储在**一个仲裁数量的副本**上。
  - 之后的每一次仲裁操作，都一定能看到这个值。
  - 当然，多次并发写入之间**可能会互相覆盖**。
- **那么，原子性的 compare-and-swap 操作 怎么办？**
  - CAS(x, oldValue, newValue)：仅当当前 x 的值等于 oldValue 时，才把 x 设为 newValue。
  - 在**分布式系统**里，能实现线性一致的 compare-and-swap 吗？
  - 答案是：**可以**——再次依靠**全序广播（total order broadcast）**来实现！

线性一致的 CAS 操作

- **系统里有多个副本（replicas），每个副本都有一份 localState（键到值的映射）。**
- **客户端发起 get(x) 或 CAS(x, old, new) 请求时，不是直接在某个副本本地执行，而是：**
  - **把操作广播出去**：total order broadcast(op)
  - **等待该操作被交付（delivered）**
  - **在交付回调里执行**：所有副本按同一顺序执行同一操作，更新同一份状态机。

线性一致的 CAS 操作

on request to perform get(x) **do**
    total order broadcast (get, x) and wait for delivery
end on

on request to perform CAS(x, old, new) **do**
    total order broadcast (CAS, x, old, new) and wait for delivery
end on

on delivering (get, x) by total order broadcast **do**
    return $localState[x]$ as result of operation get(x)
end on

on delivering (CAS, x, old, new) by total order broadcast **do**
    $success$:= false
    if $localState[x] = old$ **then**
        $localState[x] := new$; $success$:= true
    end if
    return $success$ as result of operation CAS(x, old, new)
end on

线性一致的 CAS 操作

- **全序广播保证：**
  1. 所有副本对所有操作的交付顺序完全一致（同一个全序）。
  2. 每个操作只交付一次（在常见定义里还要求可靠交付）。
- **于是每个副本都按同样顺序执行同样的状态转移，localState 会保持一致。**

- PPT部分内容来自于剑桥大学和UIUC cst.cam.ac.uk/teaching/2526/ConcDisSys/