



第四章

数据复制与一致性

第四章、数据复制与一致性

01

4.1 复制架构

02

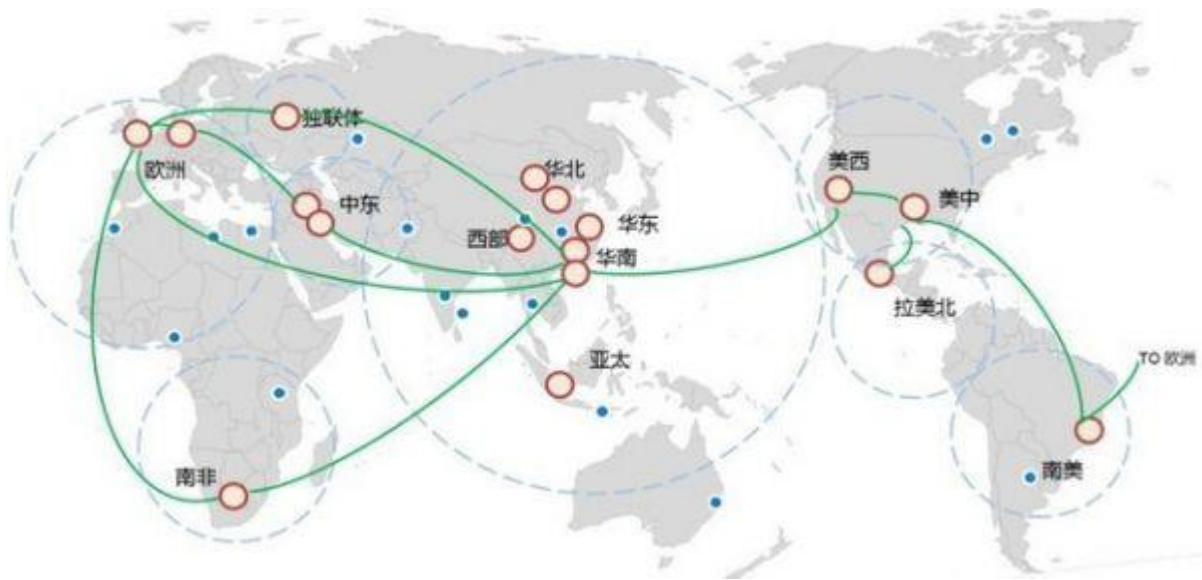
4.2 复制策略

03

4.3 副本一致性



数据复制的意义

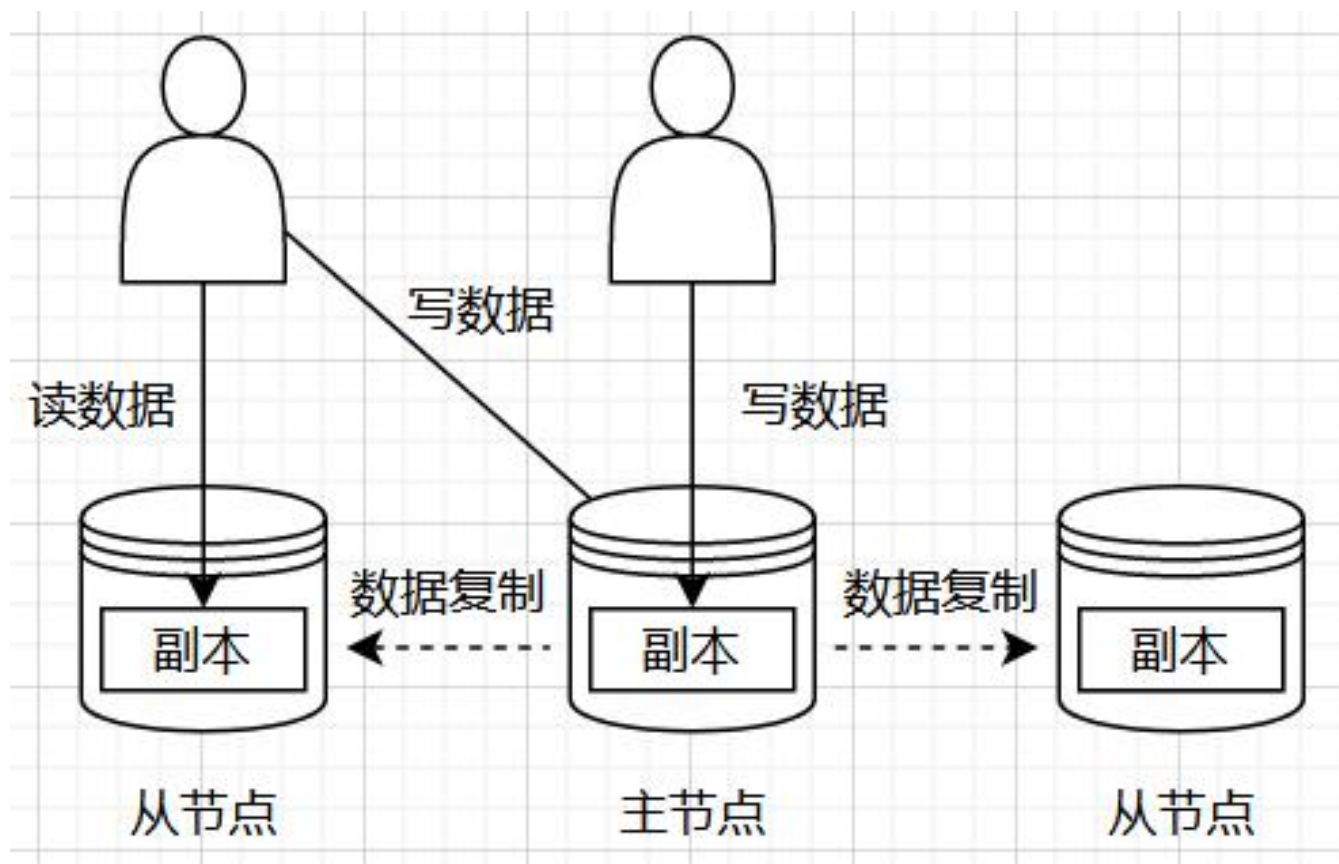


华为数据中心已经形成全球8个100ms圈和高速骨干，覆盖全球8个区域，180个国家，用户在全球任何地点100ms内可以接入到最近的数据中心。

- **高可用：**通过在多个节点上存储数据的副本，可以在某个节点发生故障时，通过其他节点提供服务，从而提高系统的总体可用性。
- **降低延迟：**在靠近用户的地理位置部署数据副本，可以显著降低数据访问延迟，提供更快的服务响应。
- **扩展性：**数据复制使得系统能够处理更多的并发请求，通过增加副本数量来水平扩展系统，提高处理能力。

复制架构

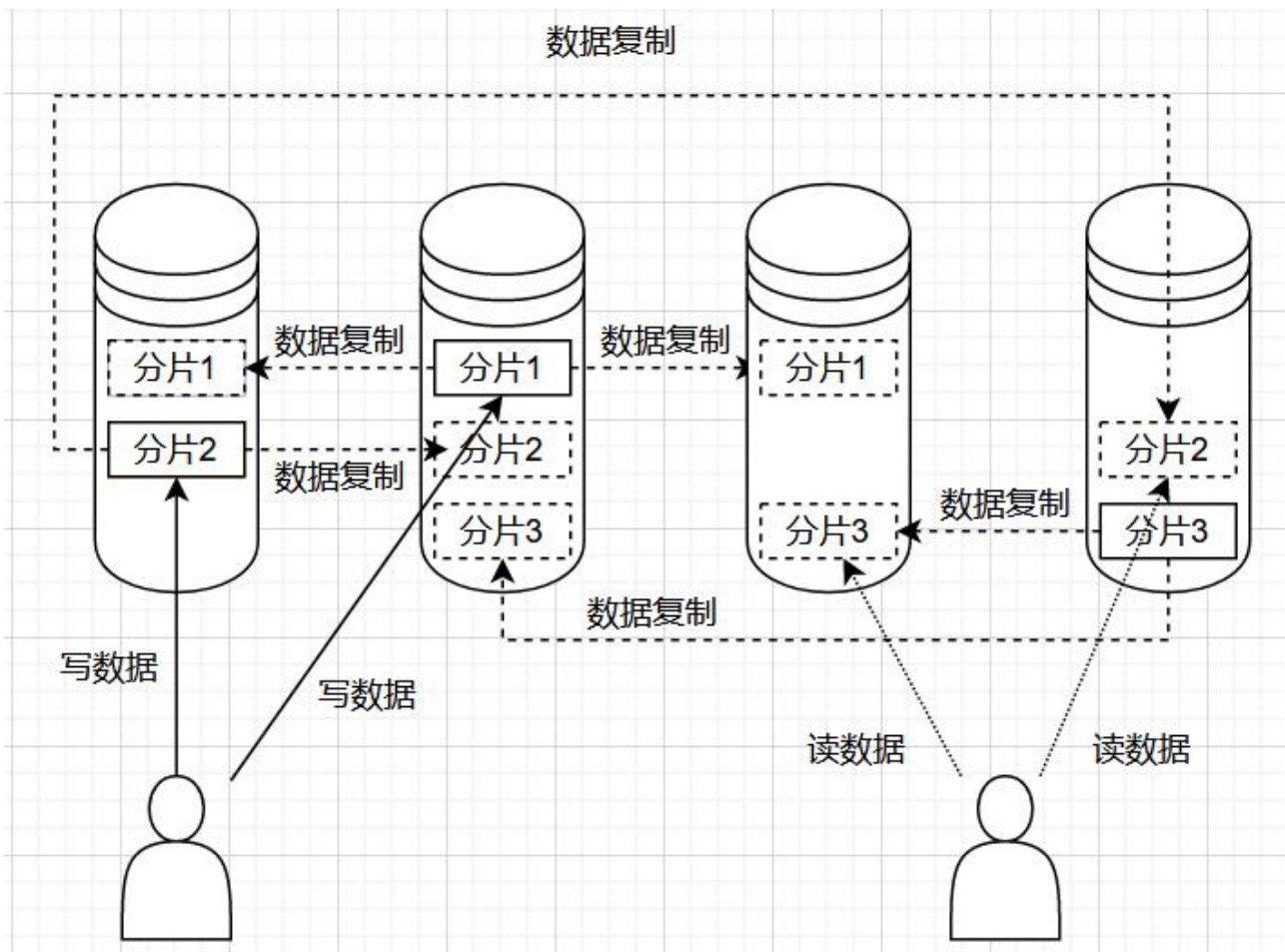
主从复制



- 在主从复制架构中，一个节点被指定为主节点，负责处理所有的读写请求，其他节点作为从节点，主要处理只读请求并接收来自主节点的数据更新。
- 从节点无法执行写操作，但会同步主节点的数据更新，以保证数据的一致性。
- 当主节点负载过重时，部分非实时性的读请求可以转发至从节点处理，以此减轻主节点的负担并实现负载均衡。

复制架构

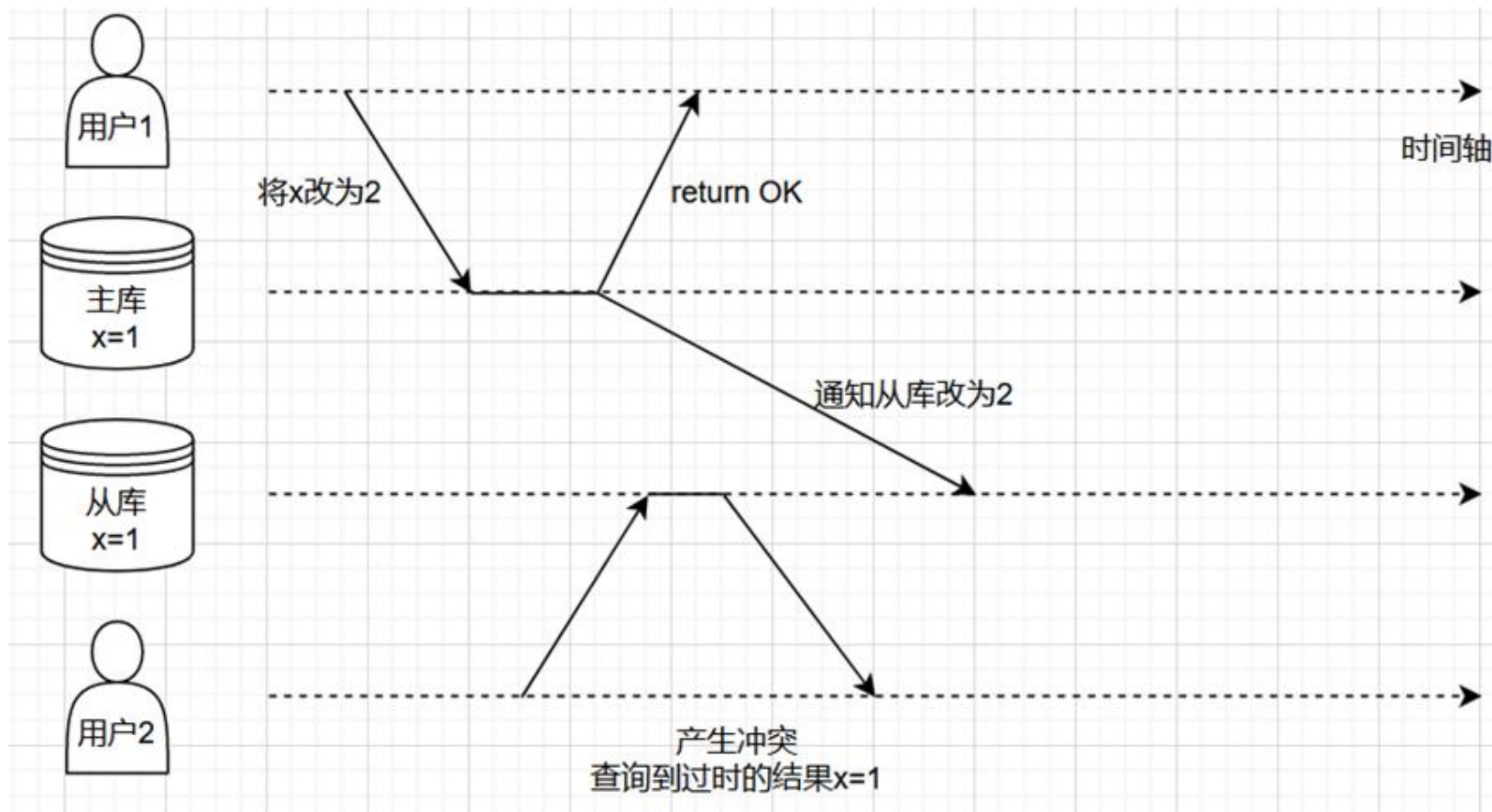
分片



- 当单个节点无法存储全部数据或面临高查询压力时，数据需要被拆分成多个**分片**（也称为分区）。
- 这些分片的副本可以部署在同一区域或跨区域的不同节点上，以提高数据的可用性和访问速度。
- 在这些节点中，指定一个节点作为主分片，负责处理所有写入操作，确保数据更新的中心化管理。

复制架构

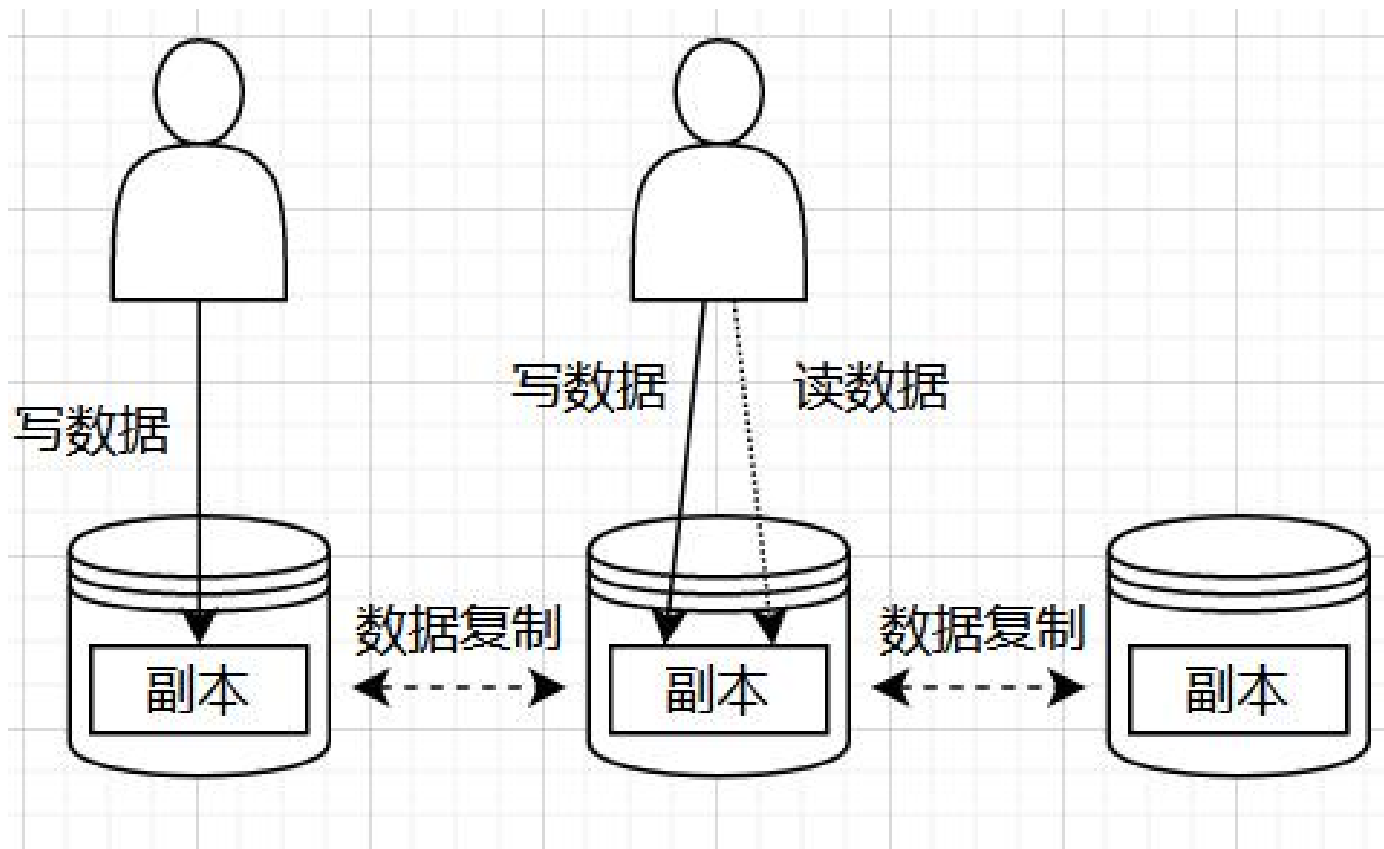
主从复制可能的冲突



- 假设有一个主从复制架构，包括一个主节点和一个从节点，主节点和从节点中均储存 $x=1$ 。
- 在某一时刻，用户1向主节点提交更改请求，将 x 的值由1改为2。主节点收到请求后将 x 的值修改为2，并向用户1返回OK，表示修改成功，随后将 $x=2$ 同步给从节点。
- 在主节点修改完 x 的值之后的某一时间，用户2向从节点查询 x 的值。
- 由于主节点向从节点同步的时延较大，那么有可能在主节点同步 $x=2$ 的消息到达从节点前，用户2已经查询到了 x 的值，此时 x 的值为过时的结果 $x=1$ ，读到了过时的数据。

复制架构

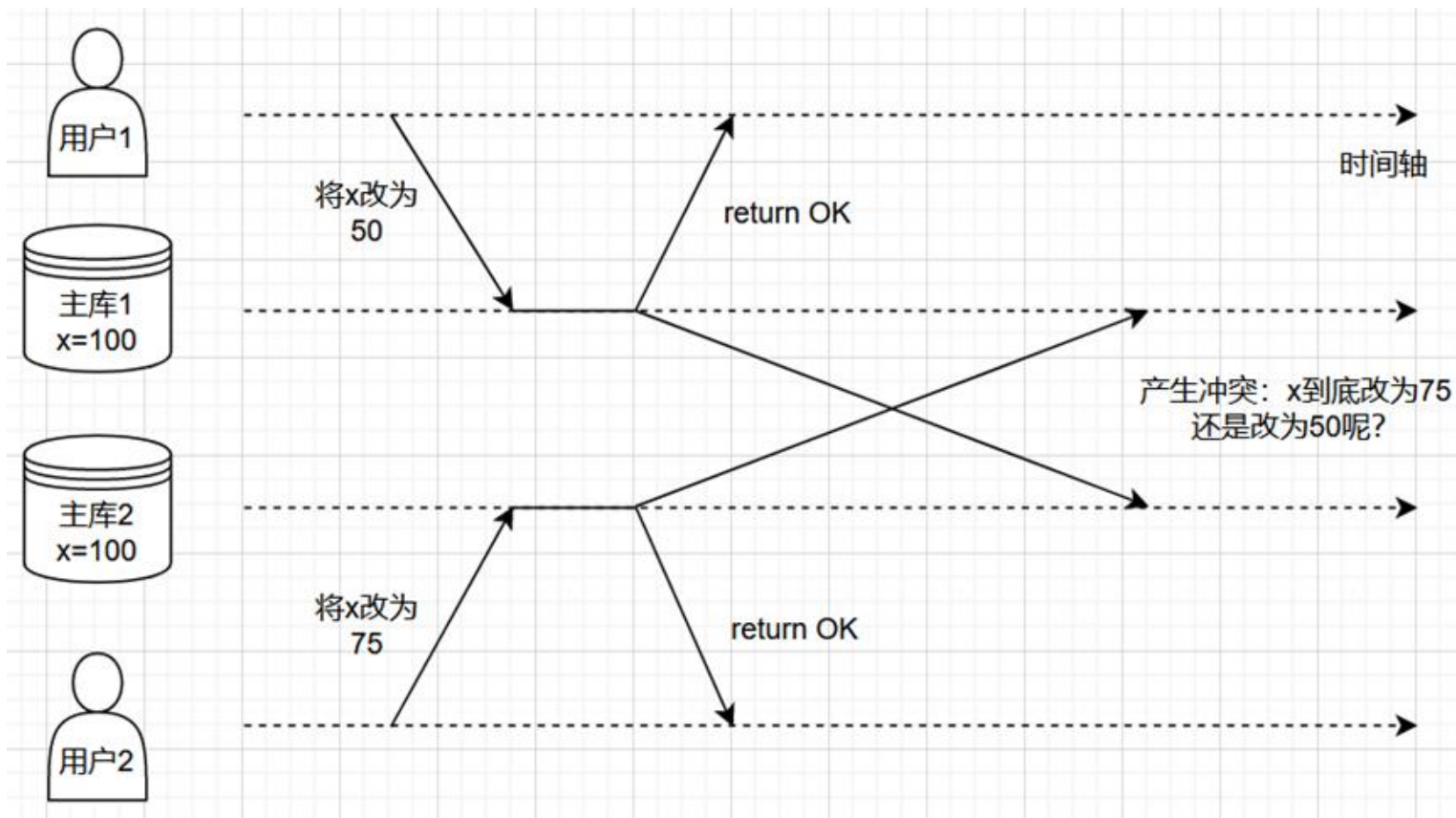
多主复制



- 多主复制架构中，每个节点都是主节点，能够独立处理读写请求，提高数据库的高可用性。
- 在此架构下，所有主节点可以同时数据进行修改，增强了系统的写入能力和灵活性。
- 如果不同的主节点同时对相同的数据项进行修改，会引起写写冲突。

复制架构

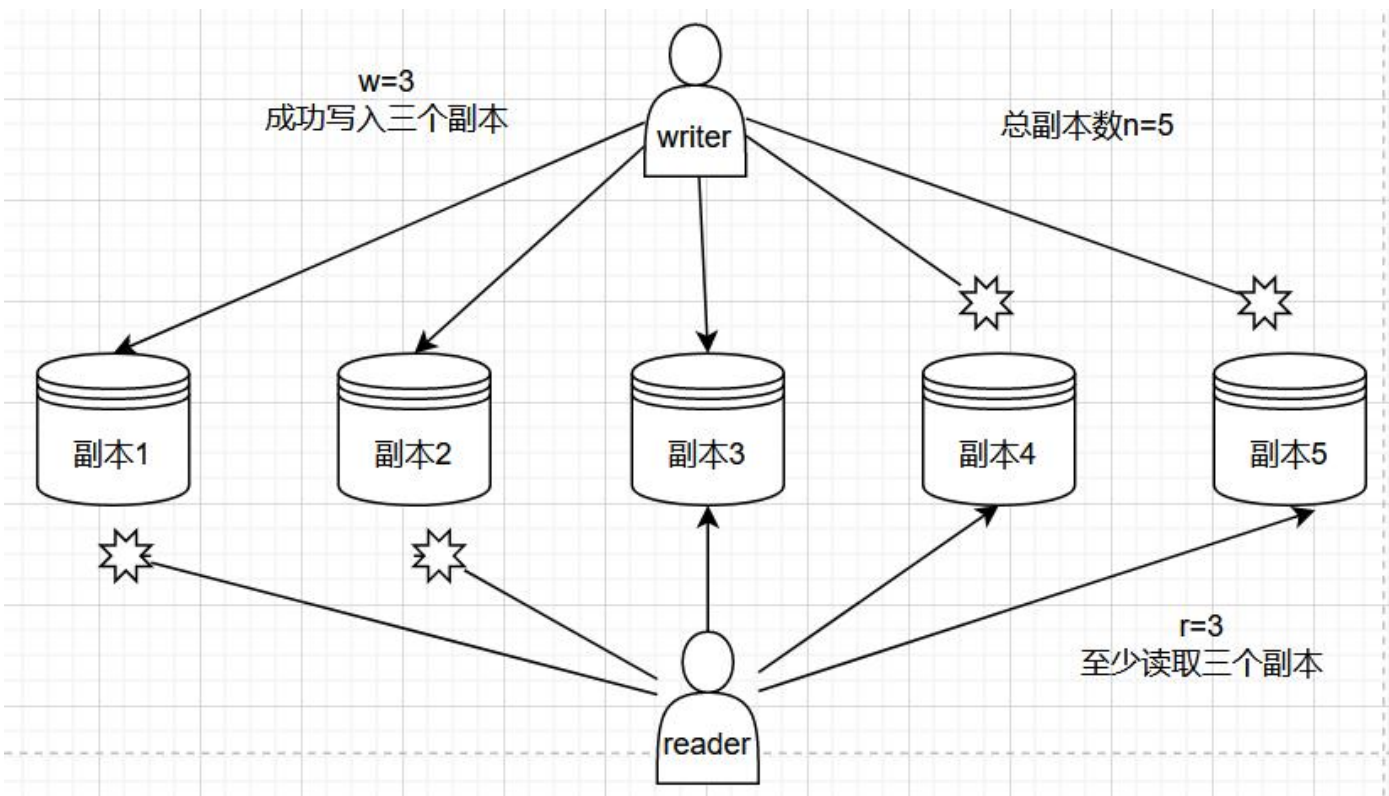
多主复制可能的写写冲突



- 假设有两个主库，均储存 $x=100$ 。在某一时刻，用户1向主库1提交更改请求，将 x 的值由100改为50。
- 主库收到请求后将 x 的值修改为50，并向用户1返回OK，表示修改成功。
- 几乎在同一时刻，用户2向主库2提交更改请求，将 x 的值由100改为75。主库2收到请求后将 x 的值修改为75，并向用户1返回OK，表示修改成功。
- 在主库1和主库2修改 x 值之后，就分别向对方同步信息，此时就产生了冲突，两个主库的 x 值产生了不一致。

复制架构

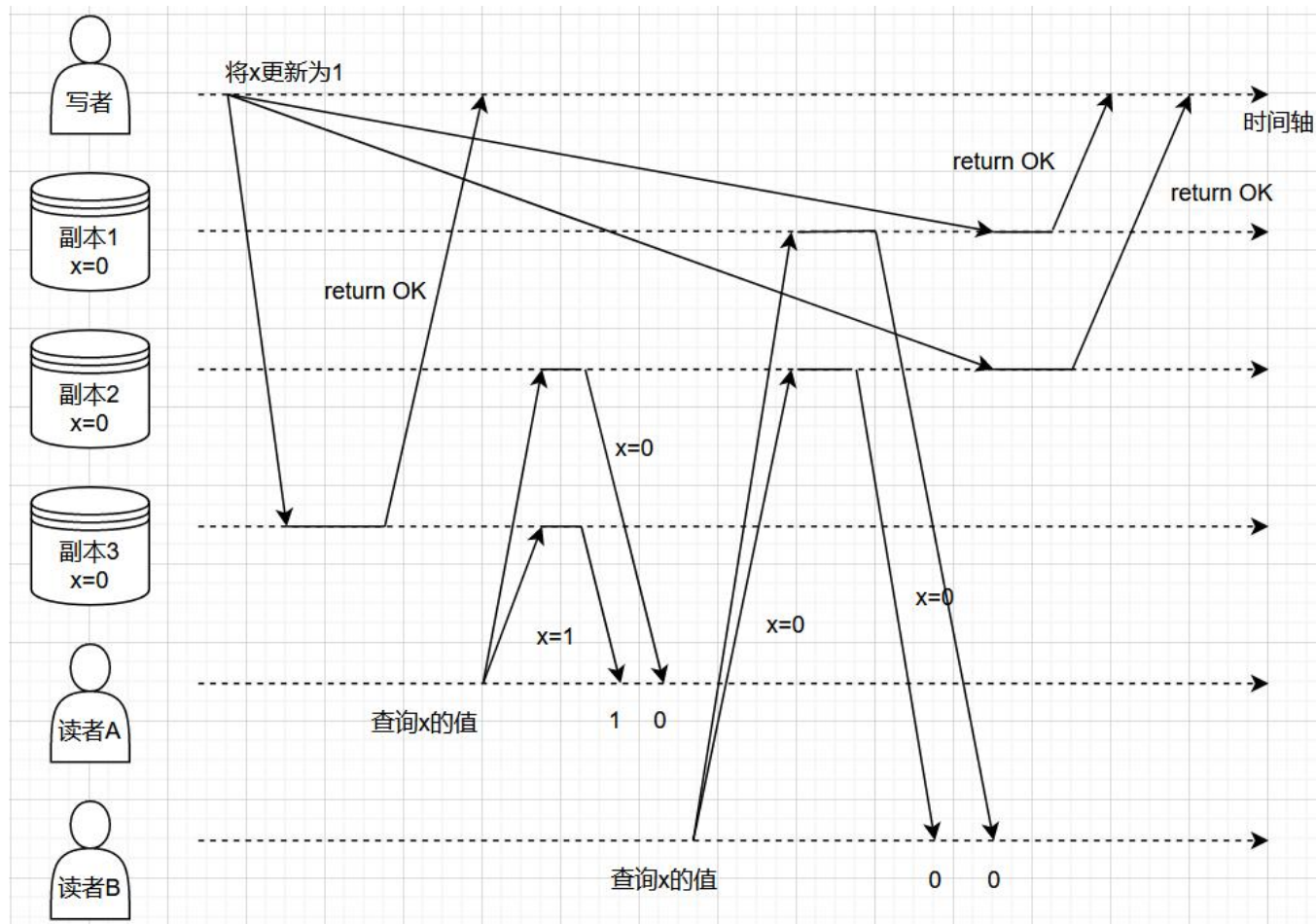
无主复制



- 无主复制架构中，集群没有固定的主节点，每个节点都可以接收并处理用户的读写请求。
- 系统通过使用单调版本号来保证所有用户都能读取到数据的最新版本。
- 无主复制使用Quorum机制来确保数据一致性，其中读和写操作需要涵盖足够多的节点，满足条件 $w+r>n$ (w 为写入节点数， r 为读取节点数， n 为总节点数)，以保证至少一个节点提供的是最新的数据。
- Quorum机制还帮助确保即使访问到的是数据的陈旧版本，客户端也能触发集群内的数据同步，更新副本至最新状态

复制架构

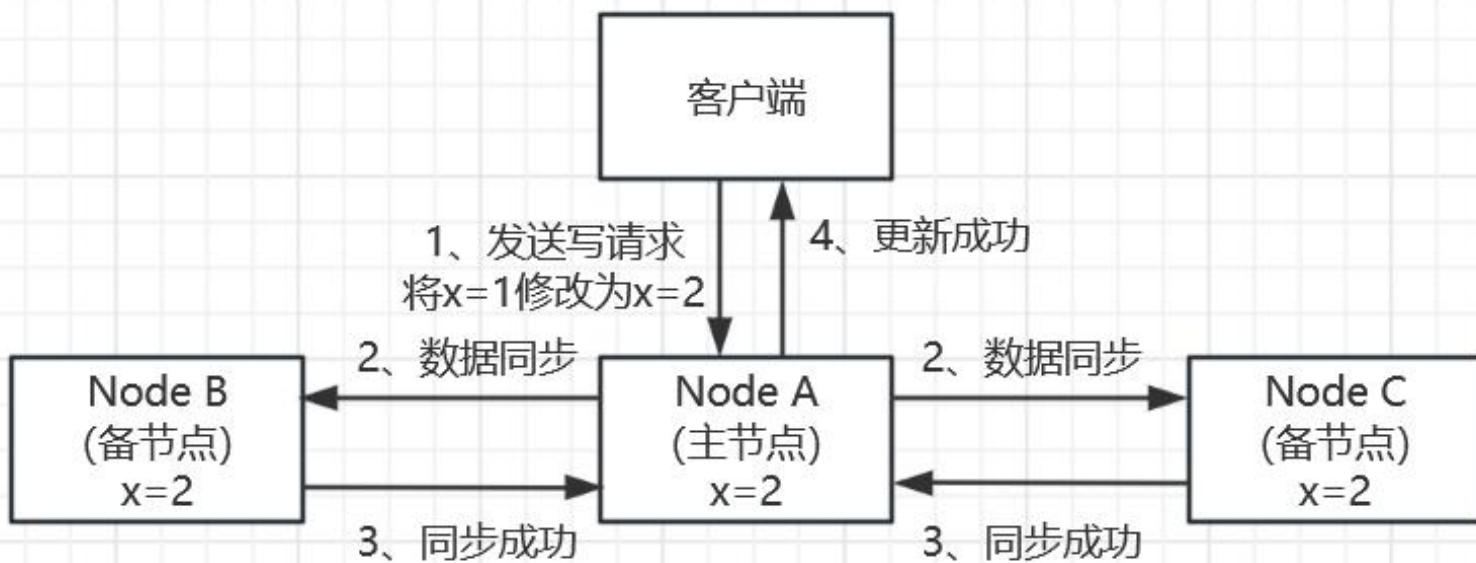
无主复制可能的冲突



- 如图所示, x 的初始值为0, 写者向所有三个副本 ($n=3, w=3$) 发送写请求将 x 更新为1。
- 与此同时, 读者A从两个节点 (副本2和副本3, $r=2$) 查询 x 的值, 然后在副本3上看到新值1。此时读者A认为 $x=1$ 。
- 稍后读者B也从副本1和副本2查询 x 的值, 由于网络延迟, $x=1$ 的消息还未传到这两个副本, 于是读者B得到了旧值 $x=0$, 三个副本中的数据不一致, 发生了冲突。

复制策略

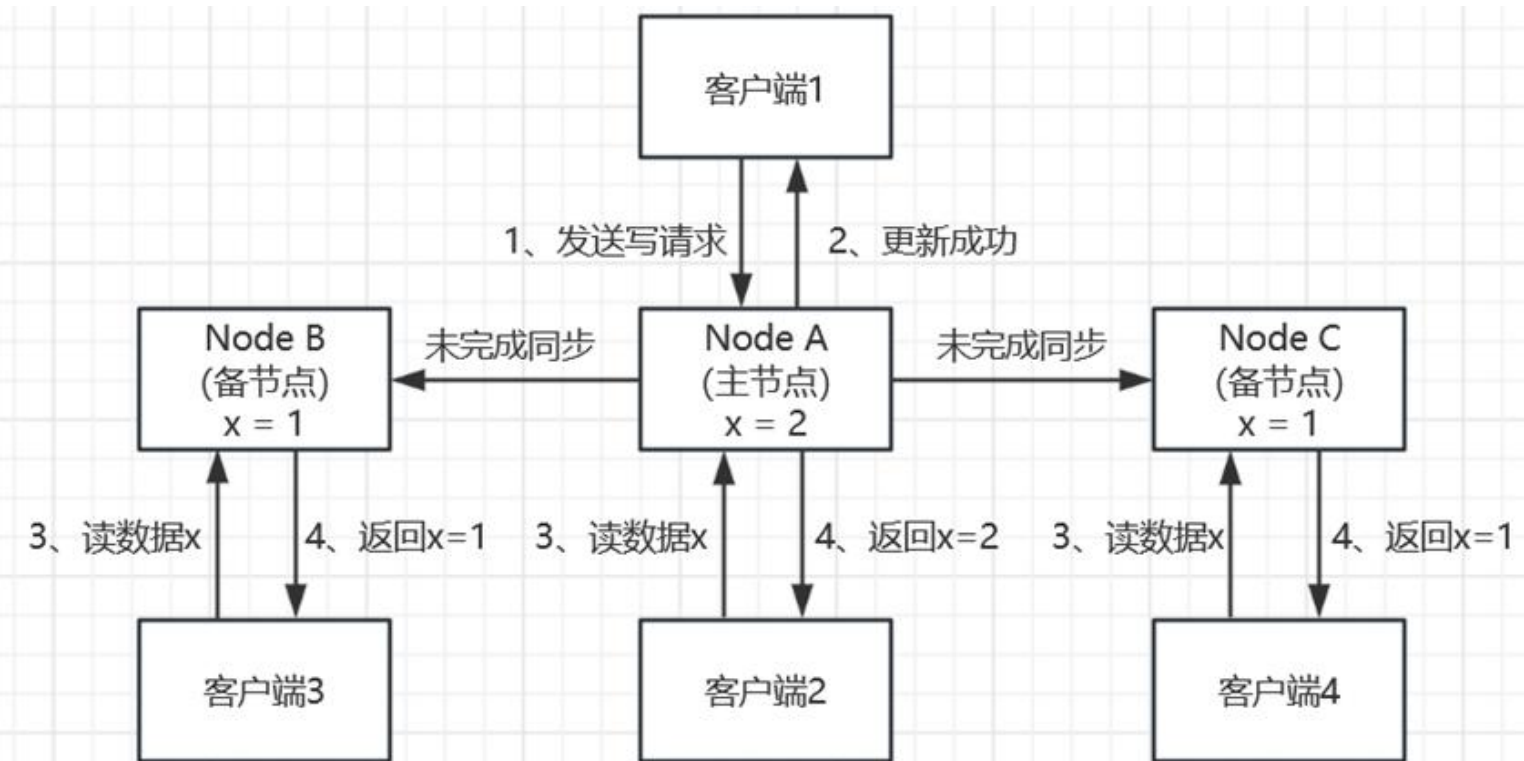
同步复制



- 同步复制要求在所有副本上完成数据更新操作后，才能确认操作成功，以确保所有副本冗余备份存储。
- 每个写操作都必须在所有副本上同步完成，这样才能继续后续的写操作或确认成功。
- 由于需要等待所有副本同步完成，同步复制通常会导致较高的操作延迟。

复制策略

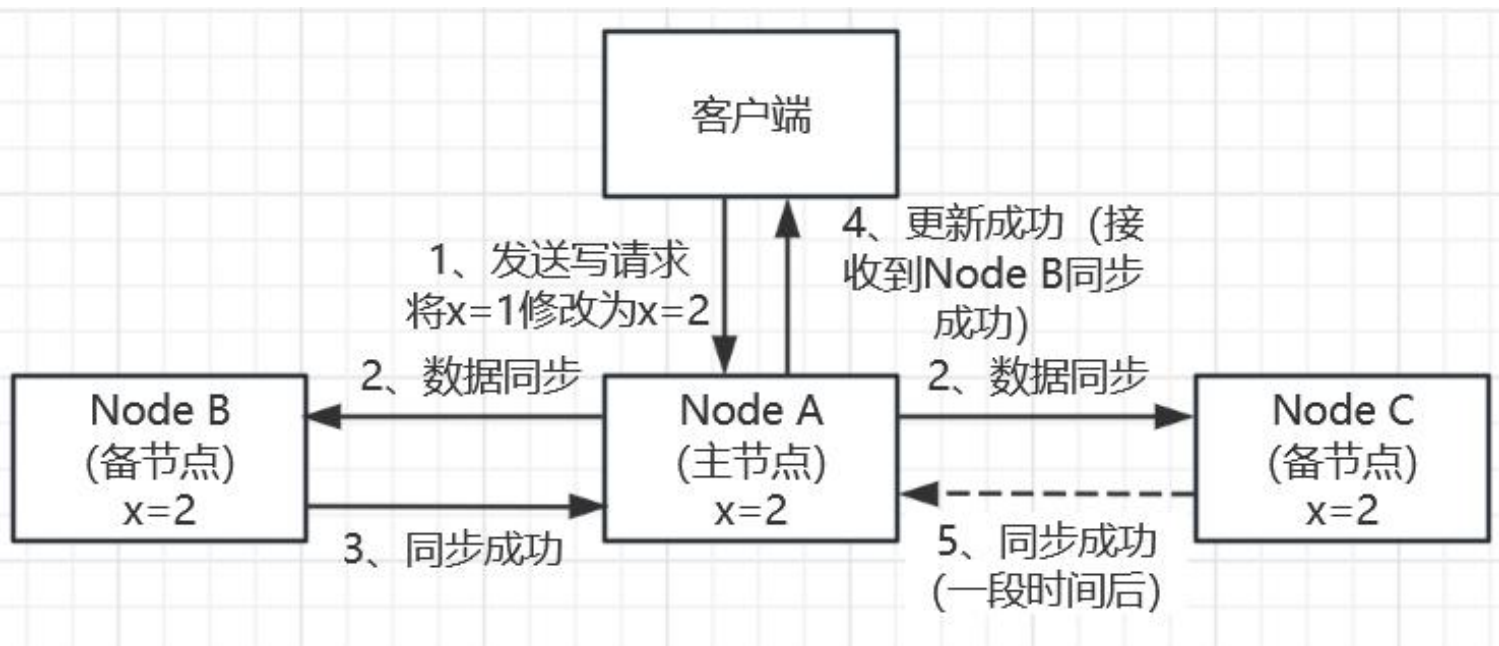
异步复制



- 异步复制允许数据在写入主节点后立即返回操作成功，无需等待所有副本的同步完成。
- 副本节点将在之后异步地同步主节点上的数据更新。
- 由于异步复制避免了复制延迟对写操作的影响，因此能够提供更低的写入延迟。

复制策略

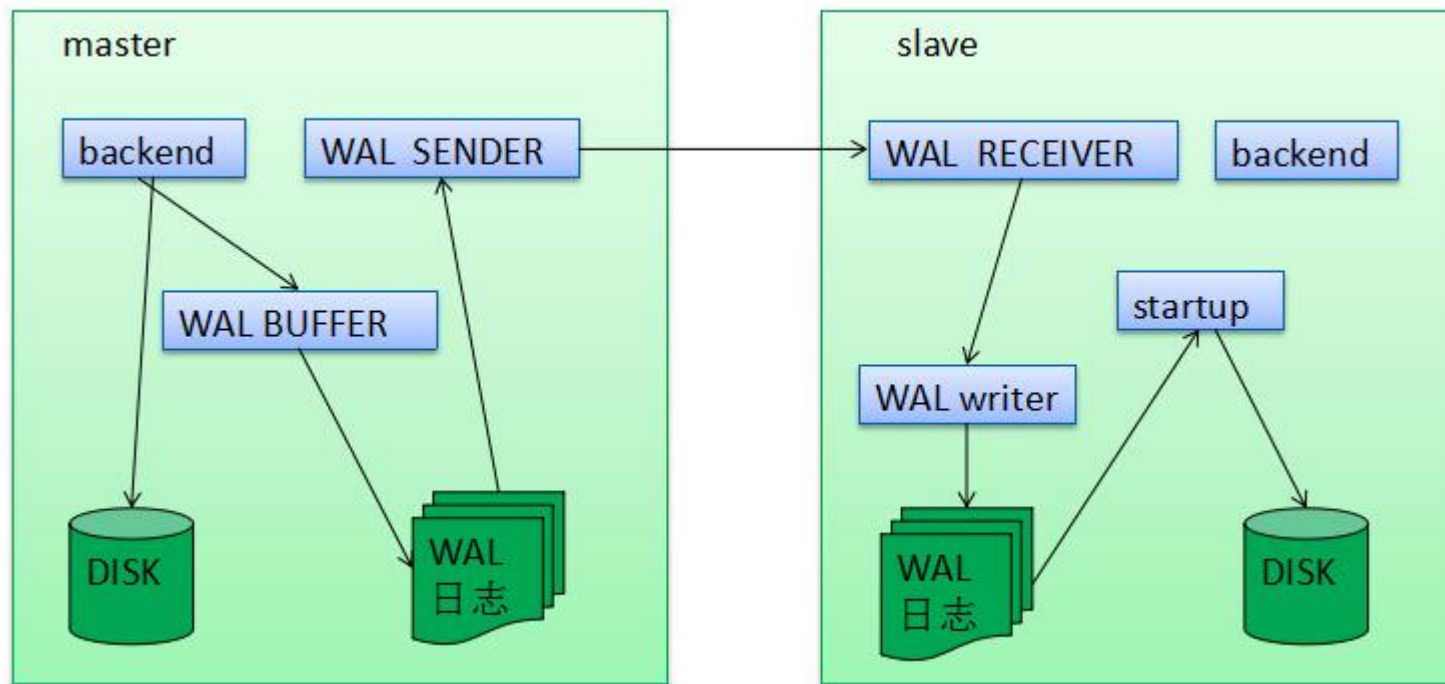
半异步复制



- 半异步复制在主节点执行写入操作后，至少需要等待一个备份节点确认数据复制成功才返回操作成功。
- 此复制策略由于不需要所有备份节点的确认，相对减少了写入延迟。
- 半异步复制适用于需要数据备份保障和较低写入延迟的中间场景。

复制策略

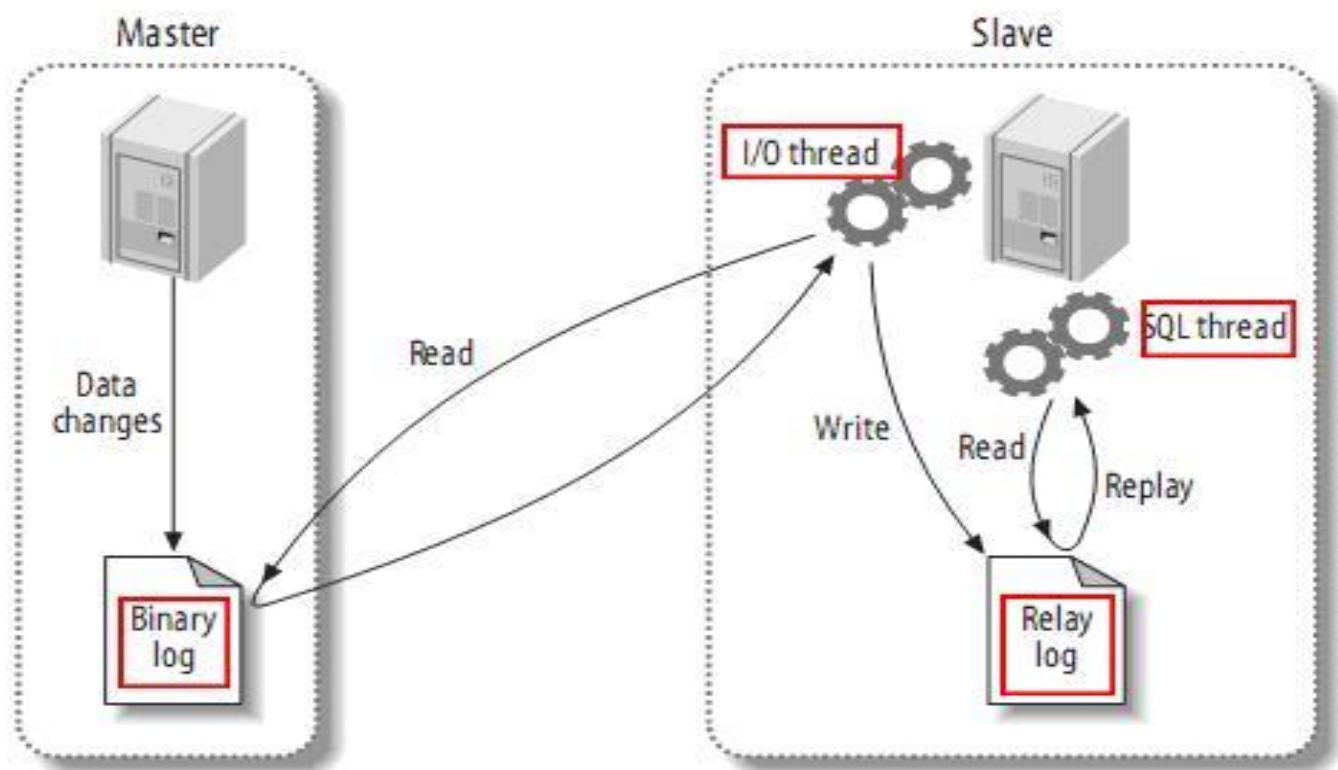
物理复制



- 物理复制将源节点的二进制数据直接复制到目标节点上，从而创建与源节点数据内容完全一致的副本。
- 在PostgreSQL中，使用WAL来实现主从复制，这种方式实质上是物理复制。
- WAL日志记录包括表空间号、数据页号、偏移量、修改数据长度及具体修改的数据。
- 修改的数据首先被记录到WAL日志中，然后这些日志被发送到目标节点。
- 目标节点接收到WAL日志后进行处理，构建出与源节点相同的数据副本，保证数据的完整性和一致性

复制策略

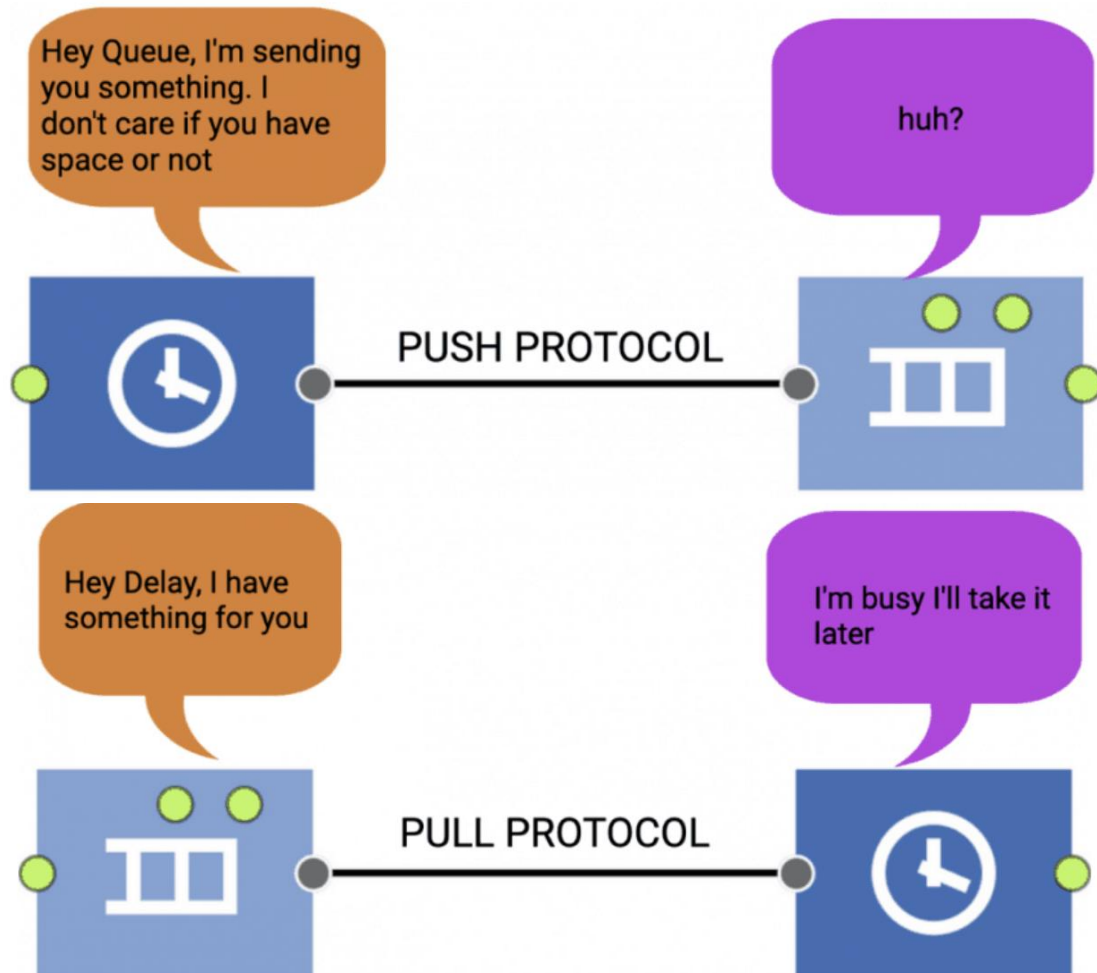
逻辑复制



- 逻辑复制不复制源节点的二进制数据，而是复制数据变更的操作语句或逻辑表示。
- 目标节点接收到源节点的数据更改操作后，执行相应的操作以保持数据的同步。
- 在MySQL中，逻辑复制通过binlog日志实现。
- 逻辑复制的binlog日志如果是statement格式，每一条会修改数据的SQL都会记录在binlog中。

复制策略

推模式和拉模式



推模式:

- 更新从主副本主动传播到其他副本，副本被动接收。
- 用于需要强一致性的场景，保证数据在所有副本间一致。
- 适用于从副本频繁读取更新的情况，确保读取到的是最新数据。

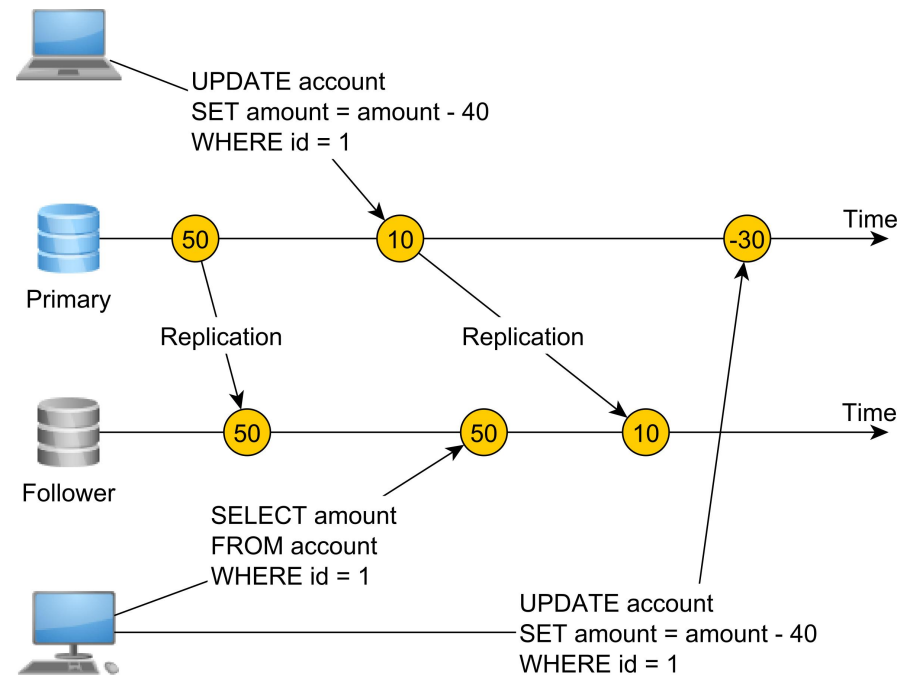
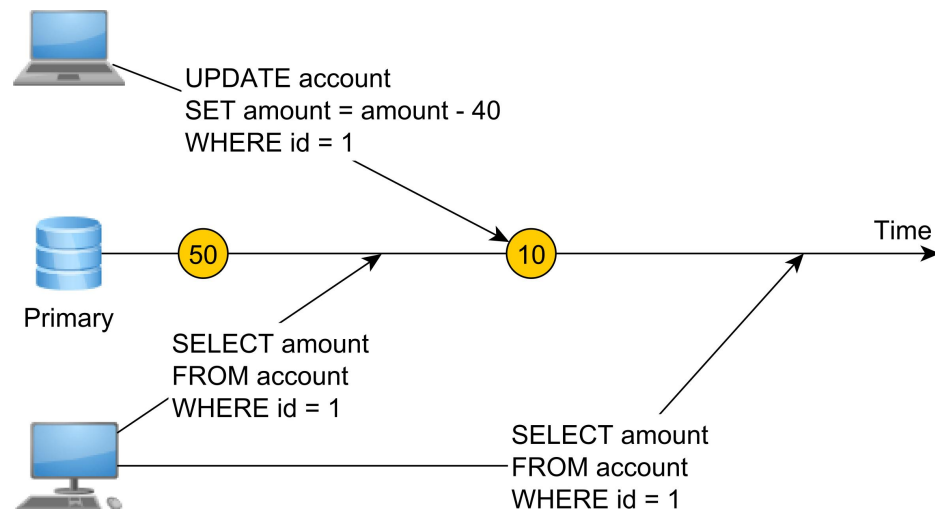
拉模式:

- 更新由从副本主动从主副本拉取，主副本被动提供数据。
- 适用于需要较高效率和减少数据传输的场景，从副本根据需要更新自己的数据。
- 常用于数据更新频率较低但需要保持较新状态的情况，例如客户端缓存，以确保缓存的数据尽可能是最新的。

副本一致性的问题

- **数据更新传播延迟：**在复制架构中，当一个节点更新数据后，这些数据需要传播到其他副本节点。如果更新传播有延迟，其他节点在同步之前可能会读到了旧数据，从而导致数据不一致。
- **多主复制中的写写冲突：**在多主复制架构中，如果不同的主节点同时对相同的数据进行修改，可能会产生写写冲突。这种冲突需要通过额外的机制来解决，否则会导致副本之间的数据不一致。
- **副本数量和读写策略：**在分布式系统中，读写请求可以分散到不同的副本处理，而副本之间的一致性策略（如同步复制或异步复制）直接影响数据的一致性。异步复制虽然可以提高性能，但在某些节点可能会读到过时的数据。
- **网络 and 系统故障：**网络问题或节点故障也会影响一致性。例如，如果主节点在将更新同步到从节点之前发生故障，那么系统可能会临时出现数据不一致的情况。

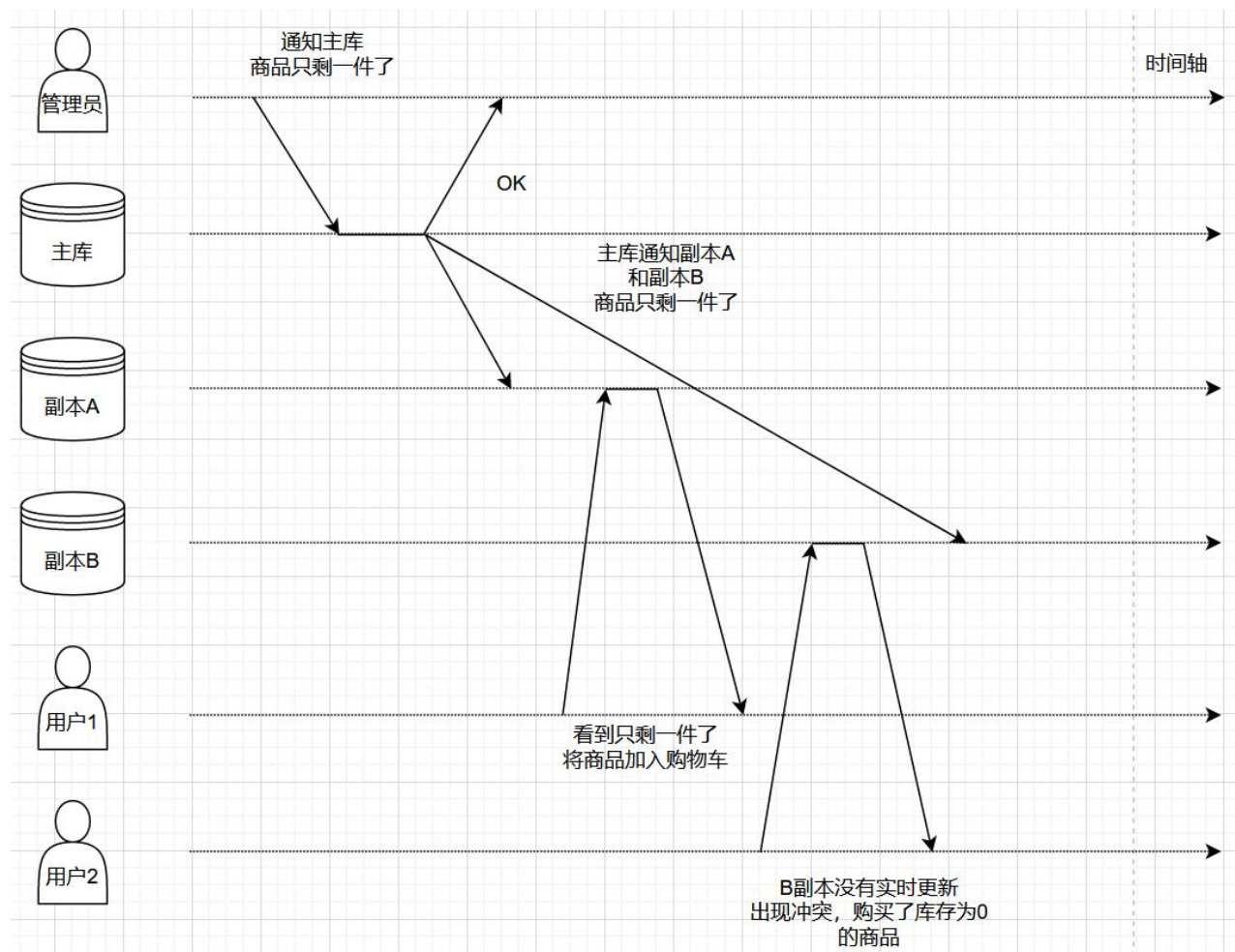
线性一致性



线性一致性 (Linearizability) 是分布式系统中最强的一致性级别，它要求系统在并发操作中能够保持线性化的顺序，一旦某个客户端成功提交了写请求，所有客户端发出的读请求一定都能看到刚刚写入的新值，就像是在操作唯一一个副本一样。

线性一致性有两个要求：第一，任何一次读都能读到某个数据的最新版本；第二，系统中的所有进程看到的操作顺序都和全局时钟下的顺序一致。

线性一致性



违背了线性一致性的要求

- 用户1在A副本上购买了一个商品，导致A副本的商品库存减少了。
- 由于B副本上的商品库存没有实时更新，用户2在稍后的时间尝试购买相同的商品，B副本并没有及时反映出商品已经售罄，所以用户2可能会成功购买到一个实际上已经没有库存的商品。
- 用户2的购买操作似乎在用户A的购买操作之后发生，但实际上是因为B副本的库存信息没有及时更新

线性一致性

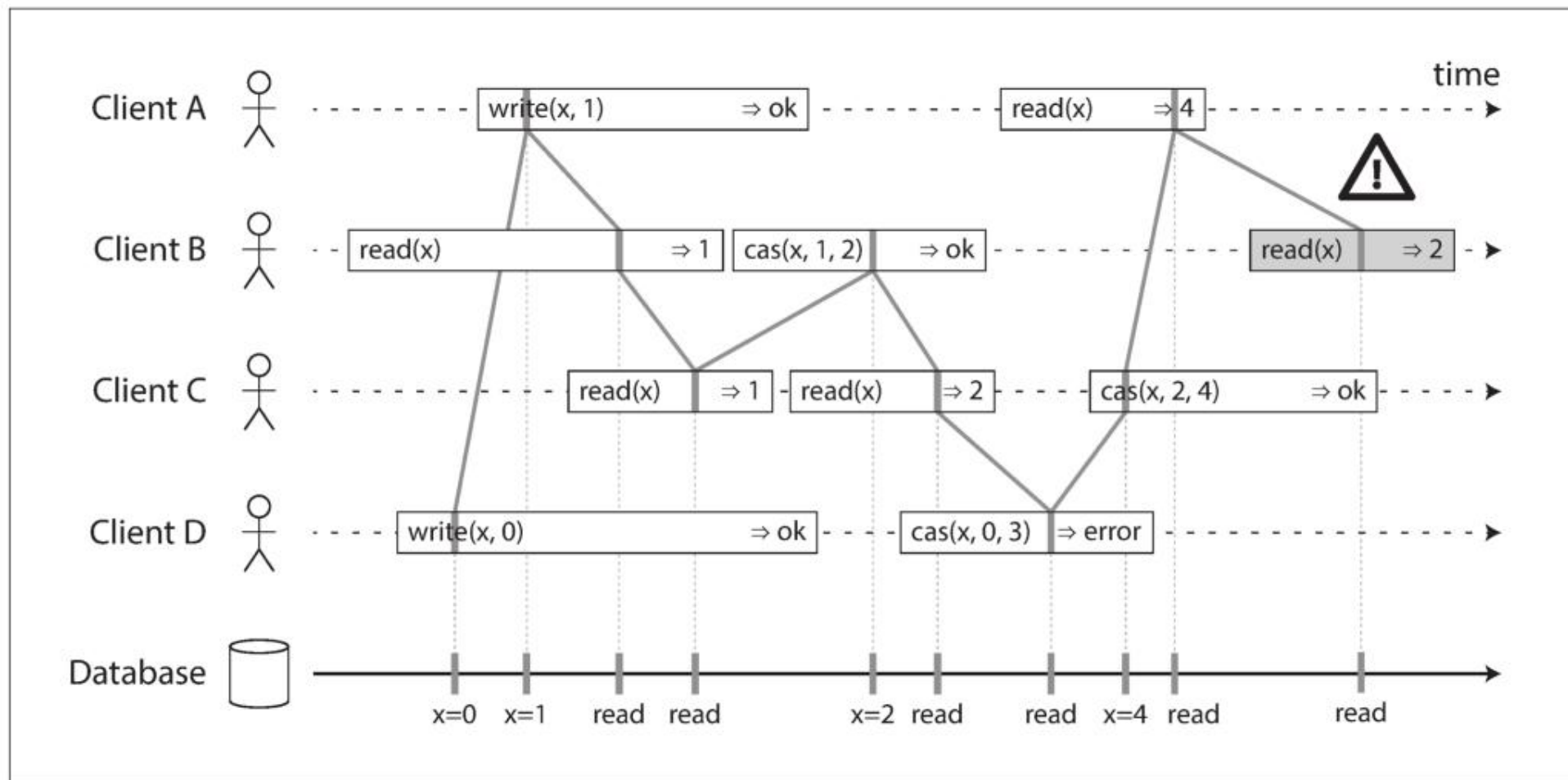


Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

线性一致性

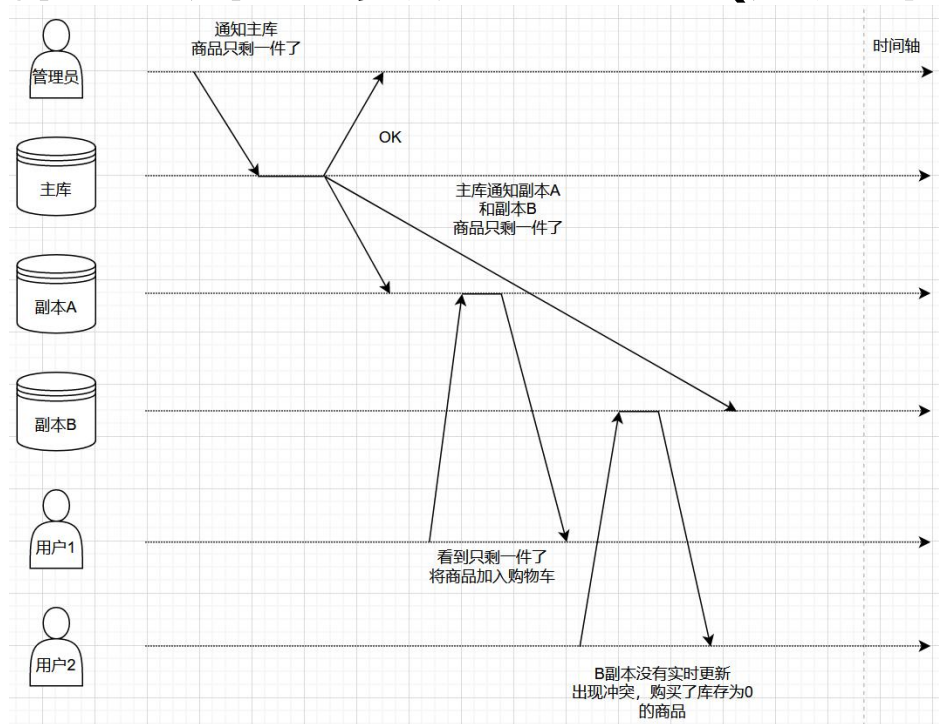
很难实现线性一致性：时钟漂移与不可靠问题

网络延迟与不稳定性：全局时钟同步在分布式系统中也受网络延迟的影响。NTP同步的准确性会受到网络延迟、拥塞以及数据包传输变化的影响。

不可信时间源与误差范围：分布式系统中的节点可能使用不同的时间源，例如NTP服务器或本地硬件时钟。然而，时间源的误差以及不可靠性。即使墙上的时钟返回微秒或纳秒级的信息，由于时钟的不准确性和漂移问题，这些精确度也并不可靠。

线性一致性

线性一致性的实现--TiDB（通过集中授时的全局时间戳来判断线性顺序）



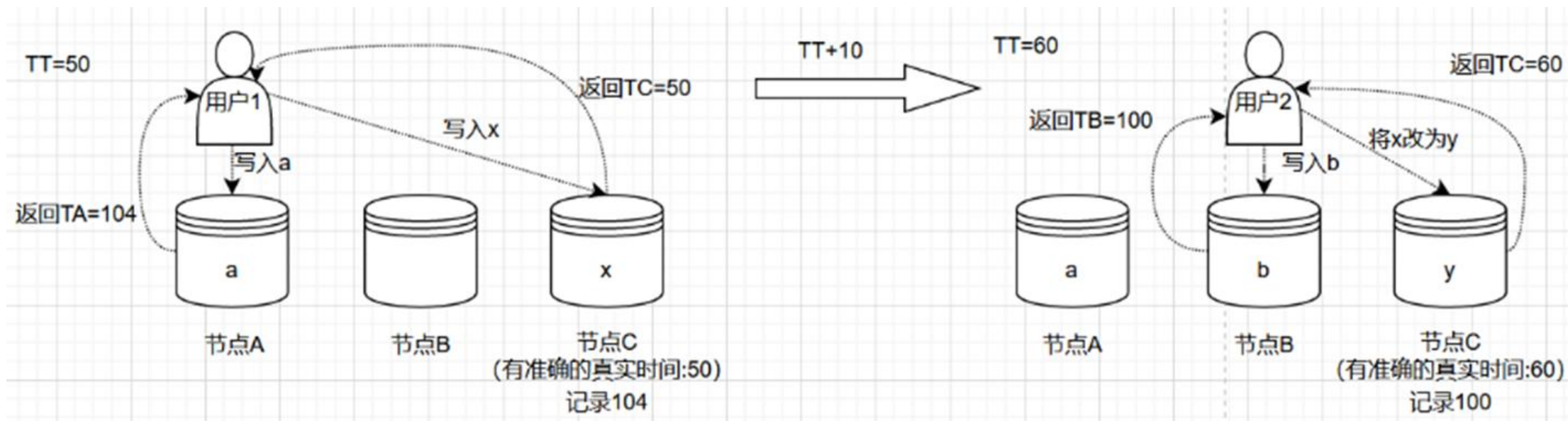
TiDB 的 Follower Read：当客户端对一个 Follower 发起读请求的时候，这个 Follower 会请求此时 Leader 的时间戳 t_1 ，拿到 t_1 后，本地副本可能会继续不断更新（更新都附带Leader时间戳），等本地副本 Apply 到 Leader最新的时间戳 t_1 后，这条数据 t_1 之前的最新版本返回给客户端

TiDB 使用**集中授时**来为分布式数据库集群提供一个统一的时间基准，以确保各个节点上的操作顺序和时间戳的一致性。其工作过程如下：

- **集中授时服务器配置：**TiDB 集群会配置一台或少数几台集中授时服务器，这些服务器提供高度可信赖的时间戳。
- **时间戳分配：**当TiDB 集群中的任何操作发生时，例如插入数据或执行事务，集群都会使用当前的集中授时服务器时间戳来为操作分配时间戳。由于所有节点都使用相同的集中授时服务器提供的时间戳，TiDB 能够准确地维护操作的顺序。无论用户在哪个节点上执行操作，操作的顺序和时间戳都能保持一致，从而实现了线性一致性。

线性一致性

线性一致性的实现—Spanner (通过有误差范围的全局时间戳来判断线性顺序)

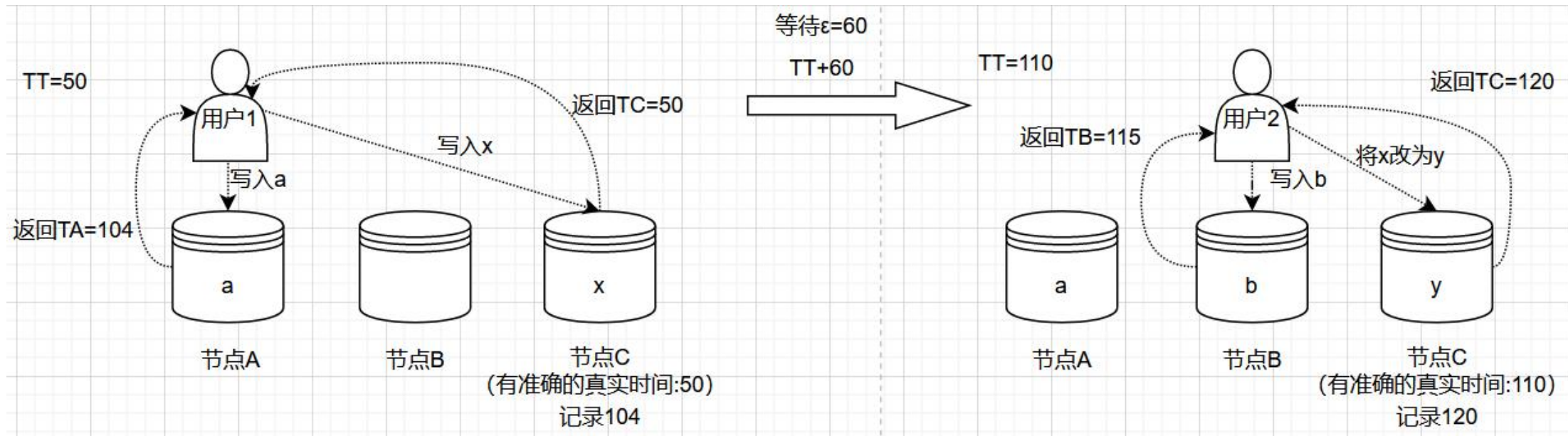


a、b、x、y均为数据

假设三个节点A、B、C，其中C具有准确时间。在时间点 $TT=50$ ，用户1更新了A和C的值，并从A和C获得时间戳 $TA=104$ 和 $TC=50$ 。协调者选择了最大的时间戳104作为操作的提交时间戳。10单位时间后，用户2将B和C的值更新，得到较大的TB的时间戳100，该时间戳成为操作2的提交时间戳。因此，虽然操作1先于操作2，但由于时间戳的处理，结果导致在读取C的值时，较旧的值 x （时间戳104）比较新的值 y （时间戳100）看起来更新。这种时间戳处理方式引起了数据的不一致问题

线性一致性

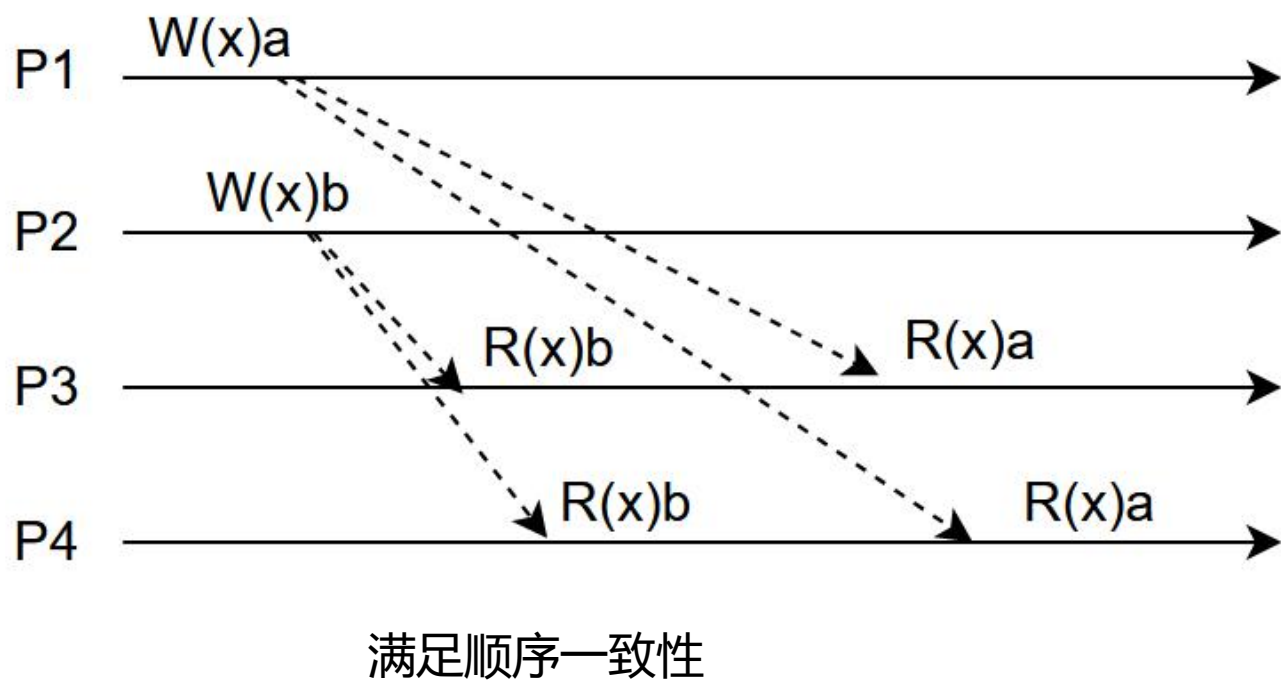
线性一致性的实现—Spanner (TrueTime: 利用时间偏差上限上限)



Spanner的TrueTime利用原子钟，保证了机群里的任意机器的时间戳差距不会超过一个上限值： ϵ 。

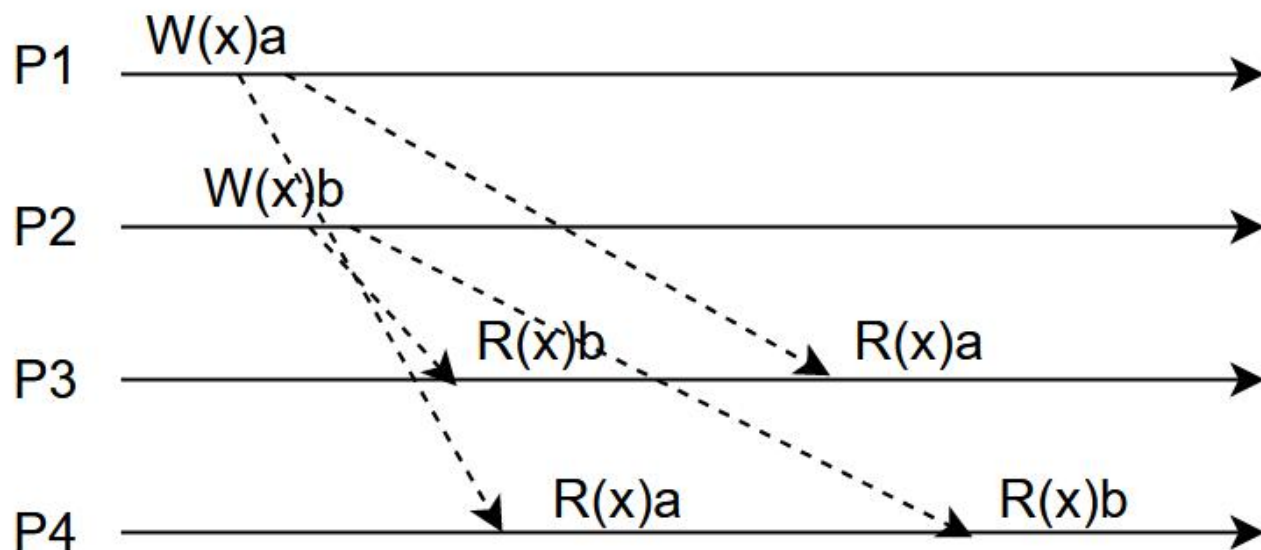
- 等待60个单位时间。在这60个单位时间里，C会被上锁，不能接受任何新的操作。
- 60个单位时间过去，操作1提交结束，操作1的时间戳为104。C接到操作1提交的信息，解除锁，C此时的本地时间变成110 ($50+60$)。记录x的时间戳为104。
- 由于C的锁，在操作1之后发生的操作2最快也只能在真实时间110发生，此时系统收集B和C关于操作2的时间戳，C的时间戳为110。不管B的时间戳是什么，都保证操作的时间戳不会小于110，这样y的时间戳必定大于110，也必定大于在操作1里C写入x的时间戳104。假设TB=115，TC=120，则操作2的时间戳为120，

顺序一致性



- 顺序一致性确保所有节点上的读写操作结果看起来就像是按某个特定顺序执行的。
- 这种一致性模型并不保证全局的时间顺序，即操作的执行顺序不一定与实际的时间顺序相匹配。
- 顺序一致性要求节点内的操作顺序必须符合该节点的时间顺序，保证一个节点上后续的操作能看到前面操作的结果。

顺序一致性

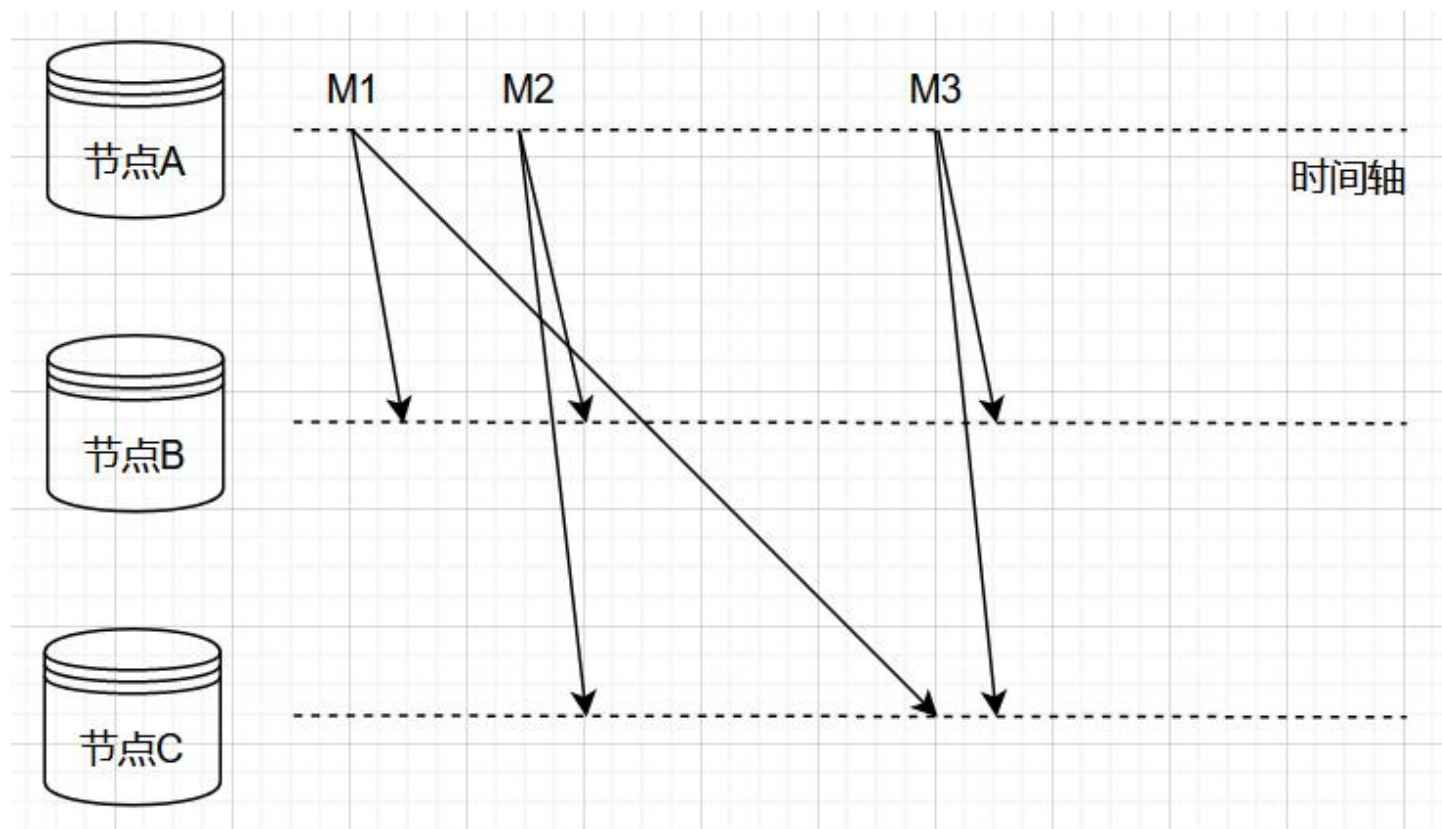


违反顺序一致性

违反了顺序一致性，因为并非所有进程都看到相同的写入操作顺序。对于P3而言，看起来好像数据项首先更改为b，然后更改为a。然而，由于可能出现的网络延迟等原因，P4将得出最终值为b的结论。

顺序一致性

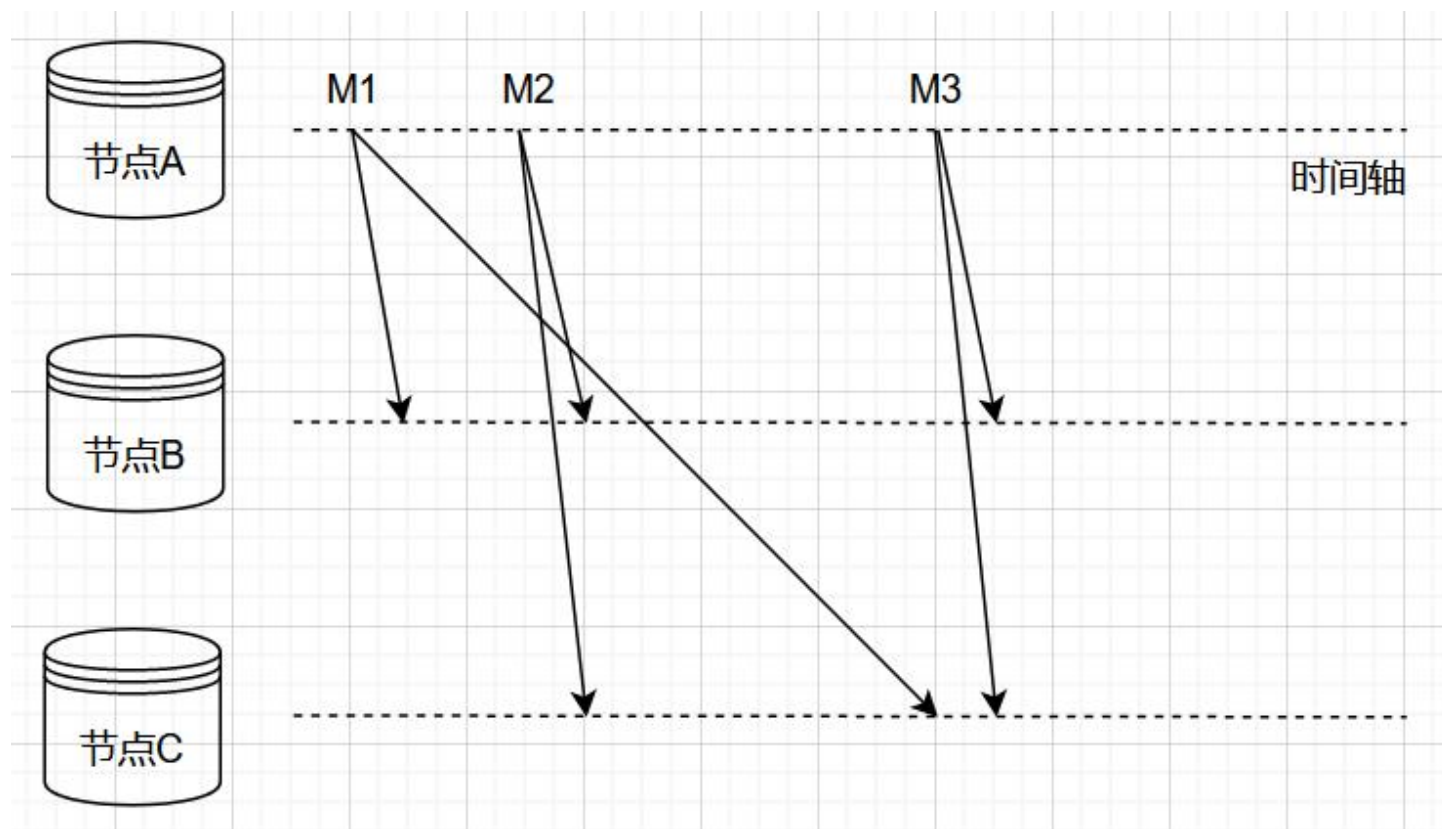
全序广播（原子广播）



全序广播是一种分布式系统通信模型，其中消息在系统中以特定的顺序被所有节点接收。这意味着每个节点都会按照相同的顺序接收相同的消息，从而保持消息的全局顺序一致性。

顺序一致性

全序广播 (原子广播)



网络延迟未能保证全序广播

- 一个由三个节点A、B、C组成的分布式系统通过网络通信，以达成操作的一致性。
- 全序广播保证所有节点以相同的顺序接收相同的消息。
- 由于网络延迟，节点A发送的消息M1首先被节点B接收，但节点C因延迟较晚接收到此消息。
- 在节点C接收到M1之前，节点A发送了另一条消息M2，该消息被节点B和C几乎同时接收。
- 随后，节点A发送第三条消息M3，节点B和C都接收到这条消息。
- 由于M1的延迟到达，节点C的消息接收顺序 (M2, M1, M3) 与其他节点不同，违反了全序广播的原则。

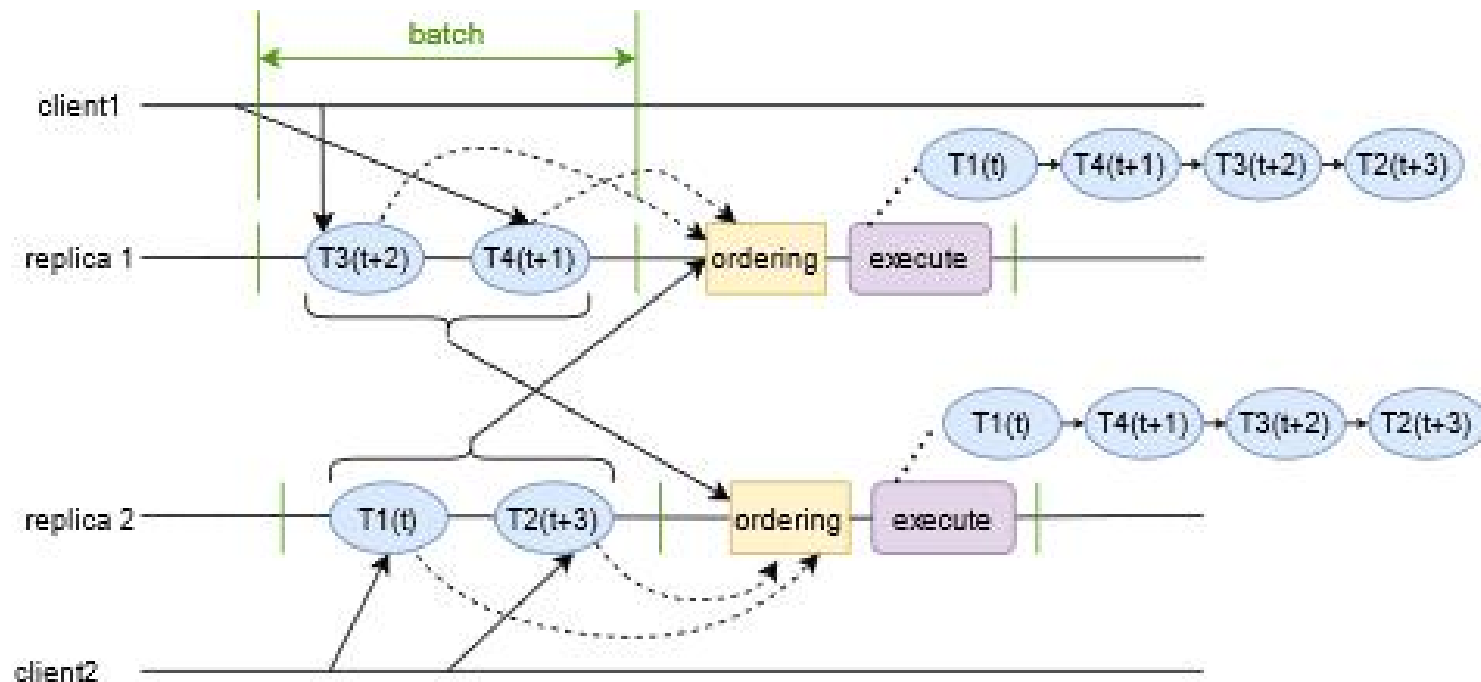
顺序一致性

全序广播（原子广播）

- 为了解决这些问题，全序广播采用了一些方法来保证在分布式系统中实现有序的消息传递。主要利用共识协议来实现全序广播，确保所有节点按照相同的顺序接收和处理消息，从而维护系统的一致性和可靠性。
- 全序广播的一个典型的应用场景是金融交易系统。在金融交易中，多个节点可能同时发送交易请求，例如买卖股票、债券等。确保所有节点以相同的顺序接收并处理这些交易请求至关重要。如果交易的顺序在不同节点之间不一致，可能会导致不正确的资金处理、订单错误等严重问题。通过使用全序广播，交易系统可以确保所有节点以相同的顺序接收交易请求，从而维护交易的全局顺序一致性。这有助于避免潜在的金融风险 and 错误，并确保整个交易系统的正确运行。

顺序一致性

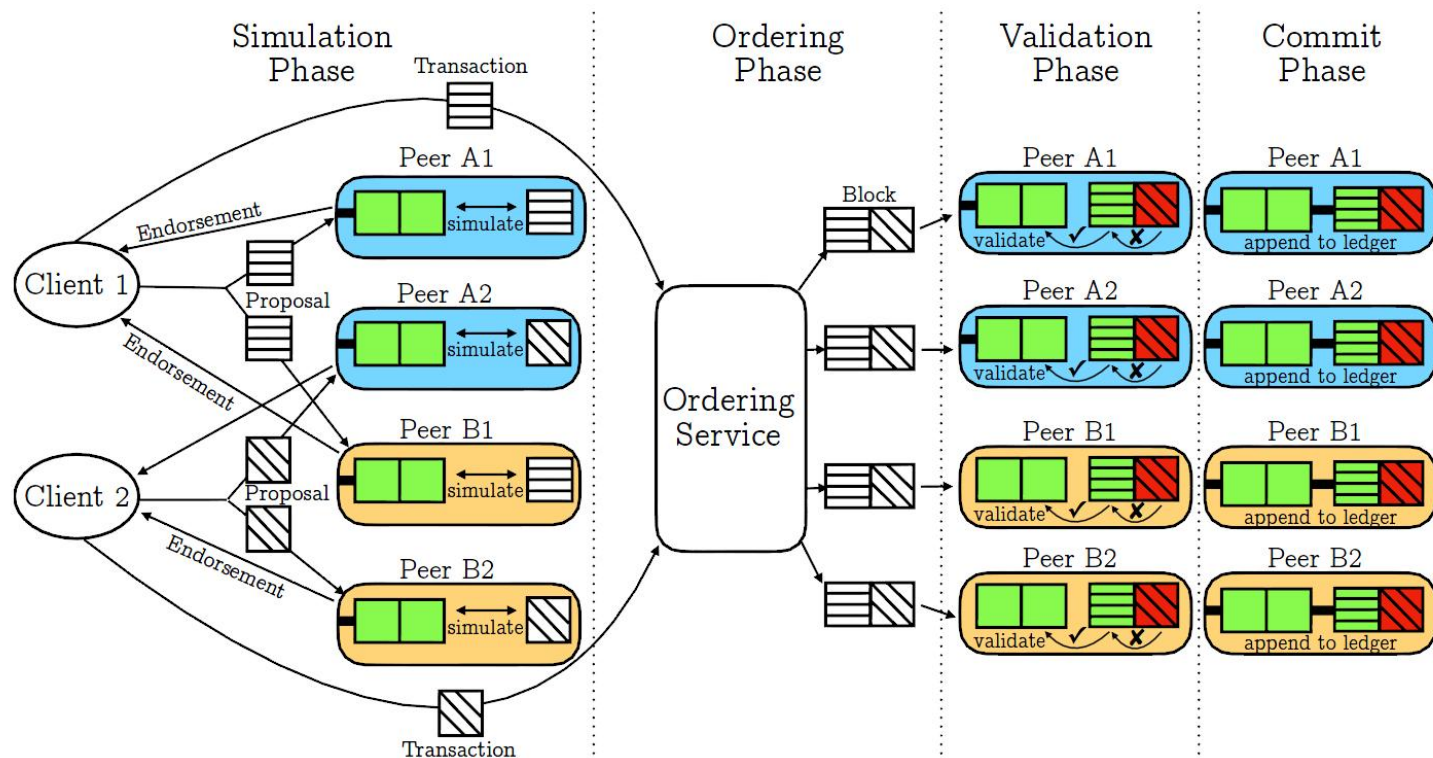
顺序一致性的典型实现—确定性数据库Aria



Aria实现了副本顺序一致性。Aria是基于批次(batch)周期性地复制同步的事务处理机制,即各个主节点将一个batch的事务读写集打包并广播给除自己外的其他节点。同时,主节点也会收到其他节点发来的事务请求.当一个batch的事务收全之后,各个节点的排序层根据特定的确定性排序规则(如各事务的本地时间戳等)对收到的事务请求进行确定性排序。

顺序一致性

顺序一致性的典型实现—Hyperledger Fabric联盟区块链



Hyperledger Fabric

先预执行再排序再验证。预执行是可以并发执行的，用户的client端可以并行的提交交易到不同的背书节点进行预执行和交易合法性的背书，然后背书节点将预执行的结果返回给客户端，客户端将预执行结果和交易发送到排序服务，排序服务负责排序共识，可以是可信的单独一个排序节点做，也可以是多个排序节点做并用拜占庭共识达成交易顺序的一致，然后再发到记账节点进行验证阶段。

因果一致性

评论朋友圈的用户

用户A  

A: 商店在哪里呀?

A回复B: 谢谢! 我也这么觉得

A回复C: 看着像礼品。你看B的回复, 他也觉得很好看

用户B  

B回复A: 在XX路。这个东西看起来好漂亮

用户C  

C: 这个是什么呀?

朋友圈

这家商店的东西真不错!

→  读者1

A: 商店在哪里呀?
C: 这个是什么呀?
B回复A: 在XX路。这个东西看起来好漂亮
A回复B: 谢谢! 我也这么觉得
A回复C: 看着像礼品。你看B的回复, 他也觉得很好看



→  读者2

A: 商店在哪里呀?
B回复A: 在XX路。这个东西看起来好漂亮
A回复B: 谢谢! 我也这么觉得
C: 这个是什么呀?
A回复C: 看着像礼品。你看B的回复, 他也觉得很好看



→  读者3

A: 商店在哪里呀?
A回复B: 谢谢! 我也这么觉得
B回复A: 在XX路。这个东西看起来好漂亮
C: 这个是什么呀?
A回复C: 看着像礼品。你看B的回复, 他也觉得很好看



→  读者4

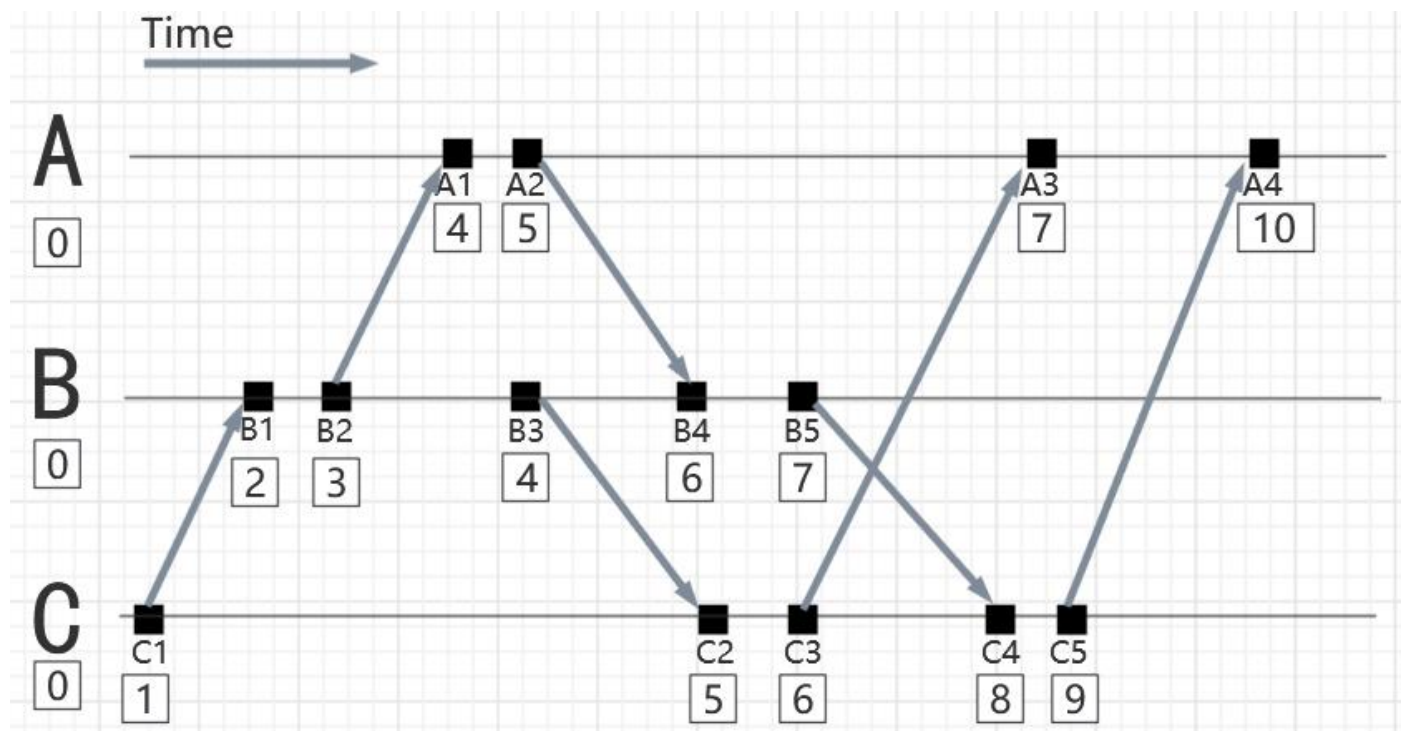
A: 商店在哪里呀?
C: 这个是什么呀?
A回复C: 看着像礼品。你看B的回复, 他也觉得很好看
B回复A: 在XX路。这个东西看起来好漂亮
A回复B: 谢谢! 我也这么觉得



- 因果一致性用于确保操作的因果关系在所有节点上得到保持。
- 如果一个操作在逻辑上导致了另一个操作的发生, 该模型确保所有进程观察到这两个操作的顺序是一致的。
- 任何因果相关的操作在系统的任何节点上都将按照其发生的逻辑顺序被观察到

因果一致性

逻辑时钟



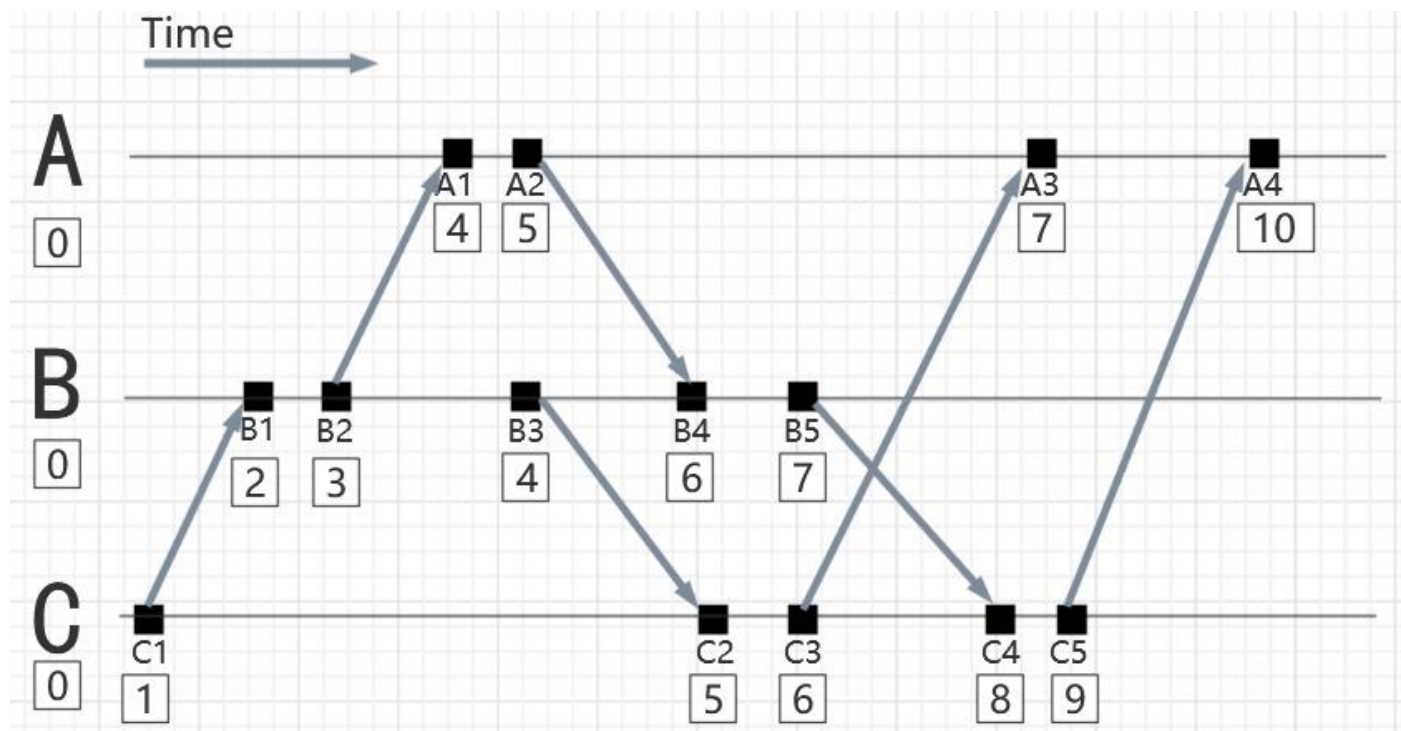
逻辑时钟是一种在分布式系统中用于跟踪事件顺序的机制，而不是实际的时间。

每个事件对应一个Lamport时间戳，初始值为0：

- 如果事件在节点内发生，本地进程中的时间戳加1
- 如果事件属于发送事件，本地进程中的时间戳加1并在消息中带上该时间戳
- 如果事件属于接收事件，本地进程中的时间戳=Max(本地时间戳，消息中的时间戳)+1

因果一致性

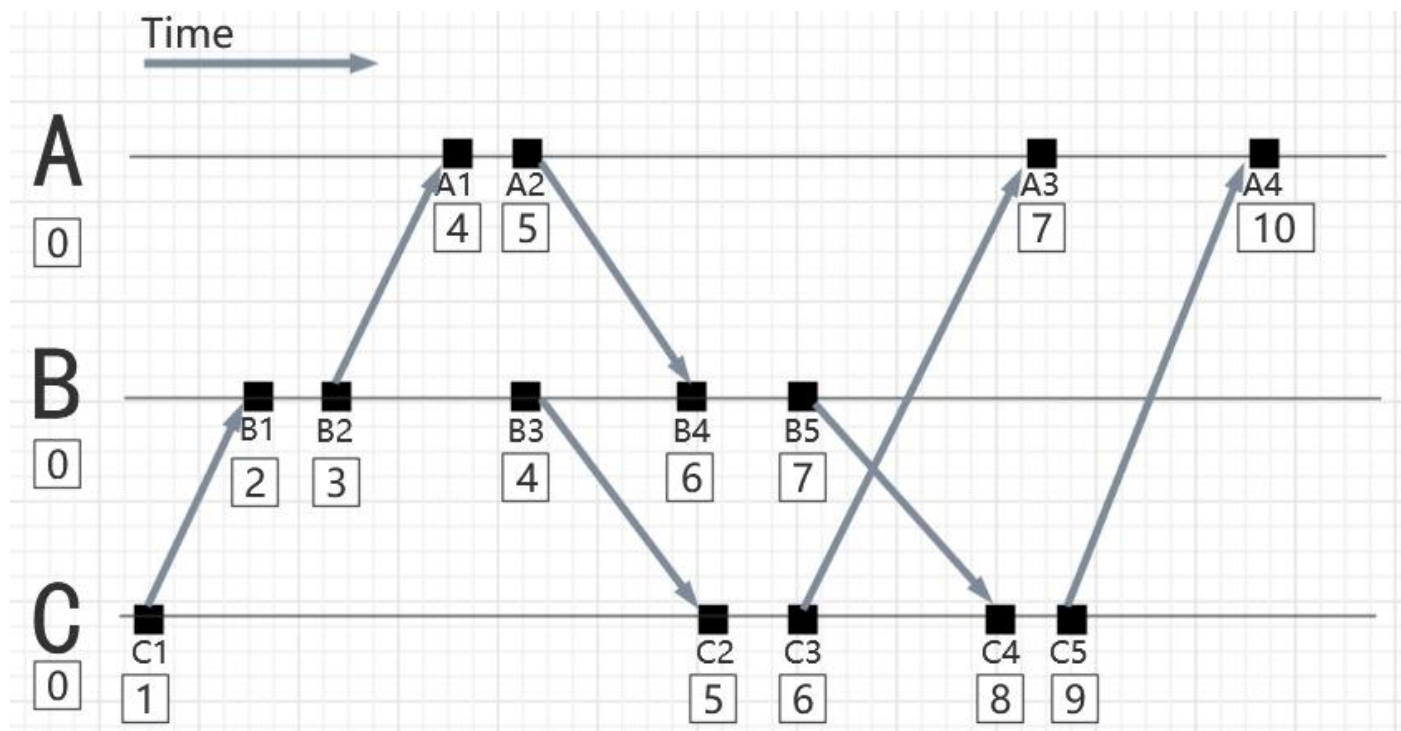
逻辑时钟



- 假设有事件a和b，其中C(a)和C(b)分别表示事件a和b对应的Lamport时间戳。
- 如果事件a发生在事件b之前（记作 $a \rightarrow b$ ），则有 $C(a) < C(b)$ 。
- 通过这一定义，我们可以比较事件集中Lamport时间戳不同的事件，从而得到事件的偏序关系（partial order）。

因果一致性

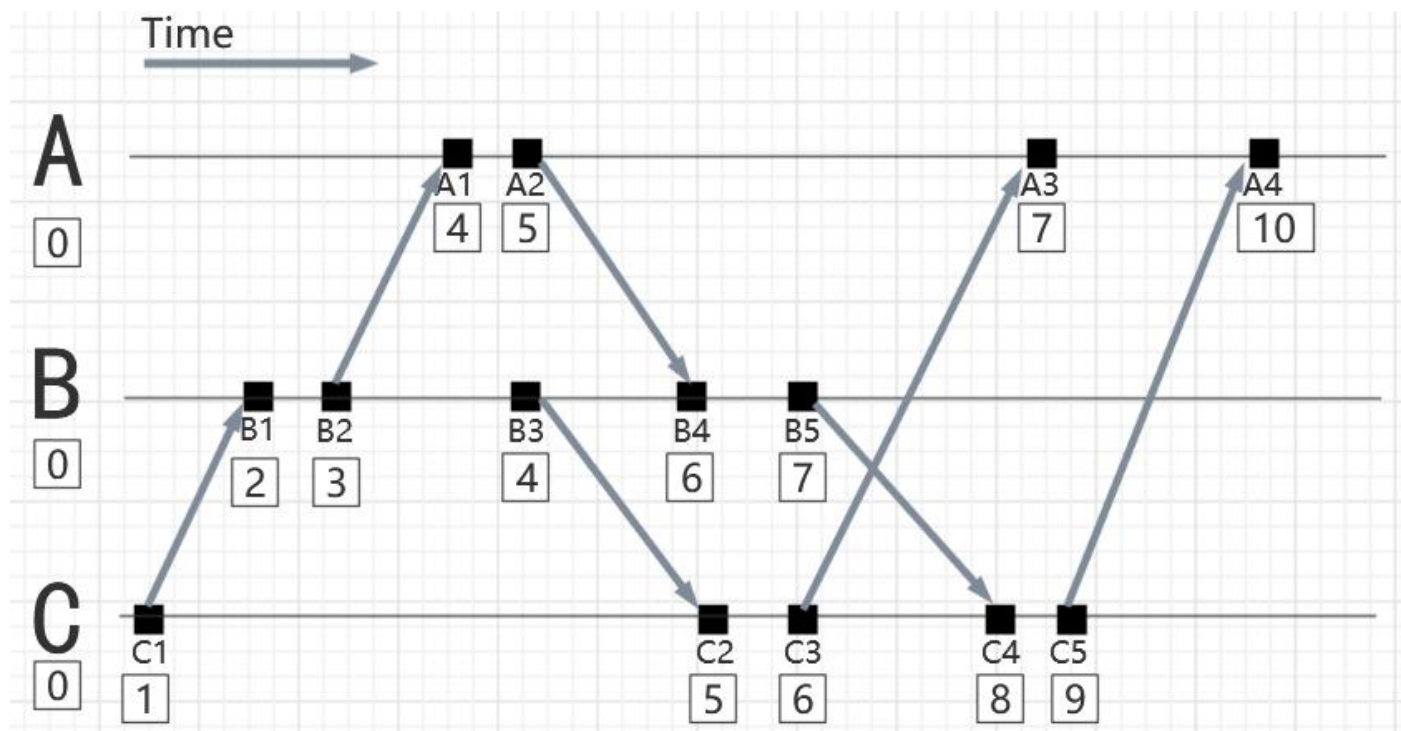
逻辑时钟



- 当 $C(a) = C(b)$ 时，它们之间一定不是因果关系，可能是分别在两条逻辑链上的事件。
- Lamport逻辑时钟只保证因果关系（偏序），而不保证绝对时序。
- 在这种情况下，若要得到全序关系，我们需要定义一种确定的方式来界定它们的顺序。

因果一致性

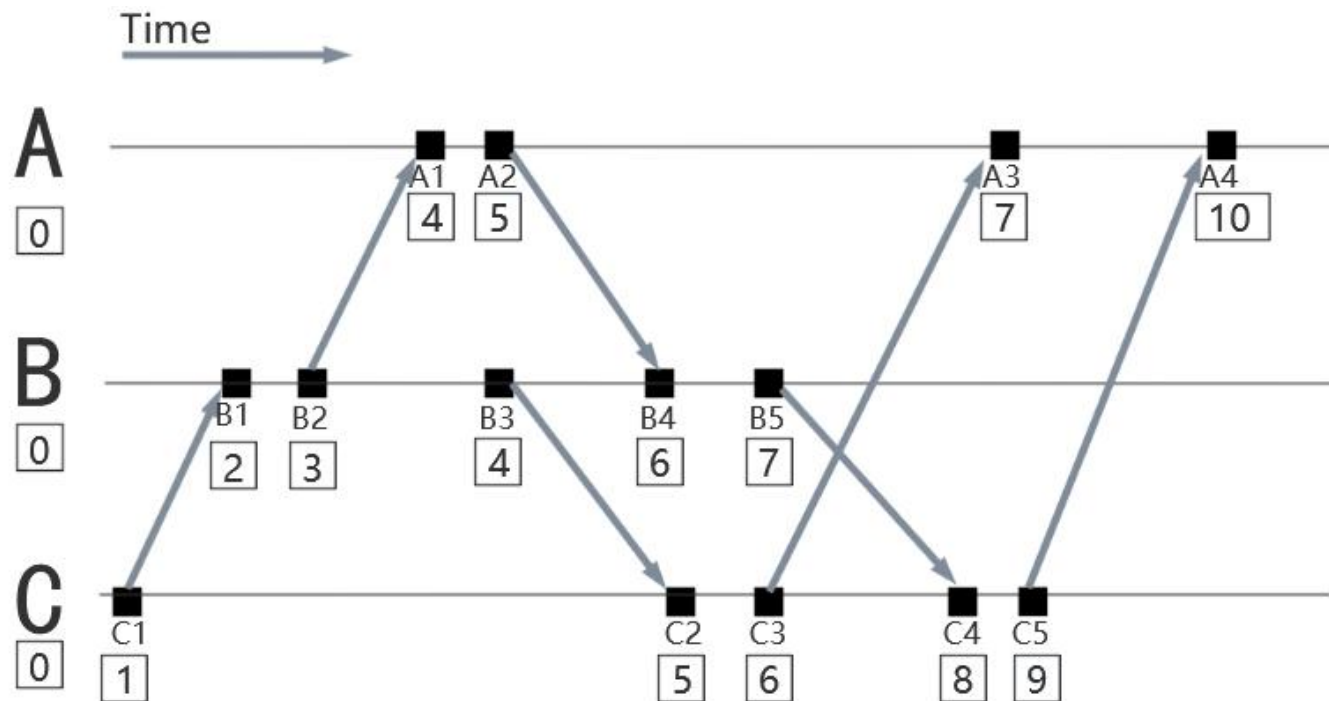
逻辑时钟



- 实现全局顺序：利用进程编号的大小来排序事件。假设事件a、b分别在节点P、Q上发生， P_i 、 Q_j 分别表示给P、Q的编号。当 $C(a) = C(b)$ 并且 $P_i < Q_j$ 时，定义a发生在b之前，记作 $a \Rightarrow b$ （全序关系）。
- 例如，如果对图中的事件A、B、C分别编号为 $A_i = 1$ 、 $B_j = 2$ 、 $C_k = 3$ ，而且因为 $C(B4) = C(C3)$ 并且 $B_j < C_k$ ，则有 $B4 \Rightarrow C3$ 。从而得到全序关系
- 在给定的图例中，可以进行排序：
 $C1 \Rightarrow B1 \Rightarrow B2 \Rightarrow A1 \Rightarrow B3 \Rightarrow A2 \Rightarrow C2 \Rightarrow B4 \Rightarrow C3 \Rightarrow A3 \Rightarrow B5 \Rightarrow C4 \Rightarrow C5 \Rightarrow A4$ 。

因果一致性

逻辑时钟

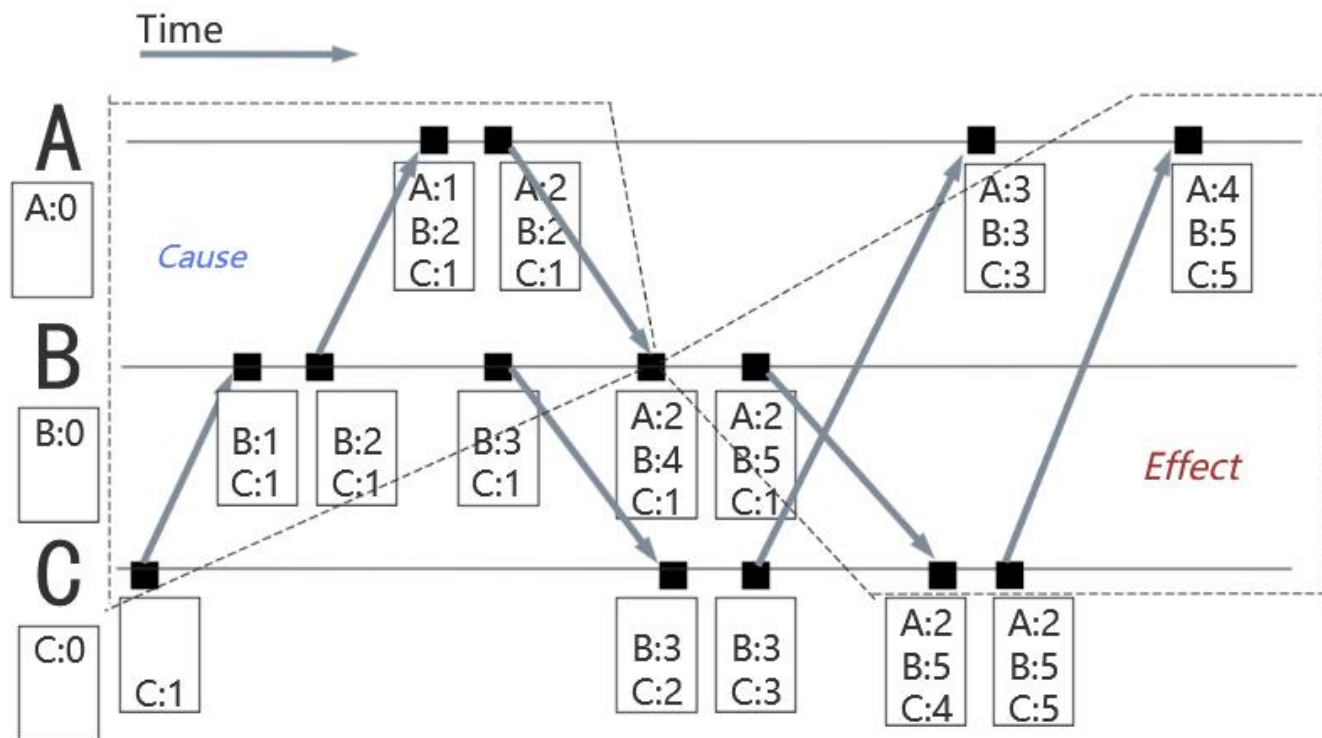


使用Lamport时间戳，只是比较事件a和b各自的时钟值 $C(a)$ 和 $C(b)$ ，无法说明它们之间实际的时序关系。也就是说， $C(a) < C(b)$ 不能说明事件a在事件b之前发生。比如事件C4与A3， $C(A3) < C(C4)$ ，但是在实际时间上事件A3在C4之后发生。

Lamport时间戳存在的问题

因果一致性

向量时钟



向量时钟

向量时钟是在Lamport时间戳基础上演进的另一种逻辑时钟方法，它通过vector结构不但记录本节点的Lamport时间戳，同时也记录了其他节点的Lamport时间戳。当系统有N个进程时，则存在N个逻辑时钟，一个进程对应一个时钟。具体如下：

- 初始时所有时钟都为0；
- 每一次处理完内部事件，将本进程对应的逻辑时钟+1；
- 每一次发送一个消息的时候，需要将自己的向量时钟和消息一起发送；
- 每一次接收到一个消息，需要将自己的逻辑时钟+1，同时更新每一个逻辑时钟，更新规则为取本地逻辑时钟和收到的逻辑时钟的最大值。

向量时钟的好处

- **支持偏序关系**：向量时钟通过向量大小关系定义偏序，使得系统能够识别事件之间的因果先后顺序，而不强制所有事件线性排序，保留了并发性的信息
- **检测并发事件**：当两个向量时间戳既不满足 $VC(a) < VC(b)$ 也不满足 $VC(b) < VC(a)$ 时，系统即可断定这两个事件并发发生，避免了单一标量时钟在并发场景下的歧义
- **记录多分支合并的因果历史**：向量时钟可记录事件合并的多条并发分支信息，这是逻辑时钟无法做到的

如何判定因果关系和并发关系

- **因果判定**

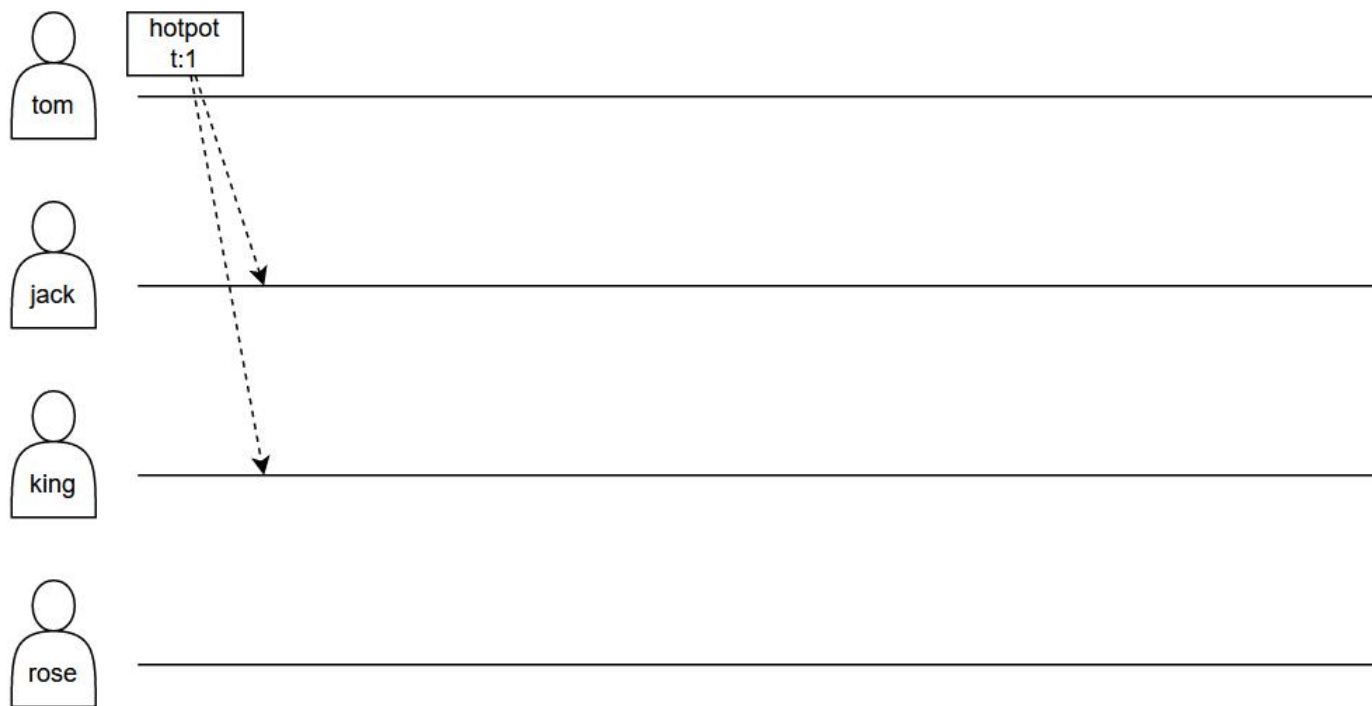
设事件 x 和 y 的向量时钟分别为 $VC(x)$ 和 $VC(y)$ ，若对所有进程 i 都有 $VC(x)_i \leq VC(y)_i$ ，且存在某个 j 使得 $VC(x)_j < VC(y)_j$ ，则可判定 $x \rightarrow y$ ，即 x 因果先于 y 发生。

- **并发判定**

若既不满足 $VC(x) < VC(y)$ 也不满足 $VC(y) < VC(x)$ ，则 x 与 y 在逻辑上互为并发事件，即它们各自的向量分量在某些位置上交叉领先，无法确定先后顺序

因果一致性

使用向量时钟解决冲突检测问题



tom准备和jack, king, rose四个人下周聚餐，但在吃什么上协调出现了问题。首先，tom提出想吃火锅(hotpot)，那么对提出这个提议的tom来说，此时他的时钟向量就是这样的：

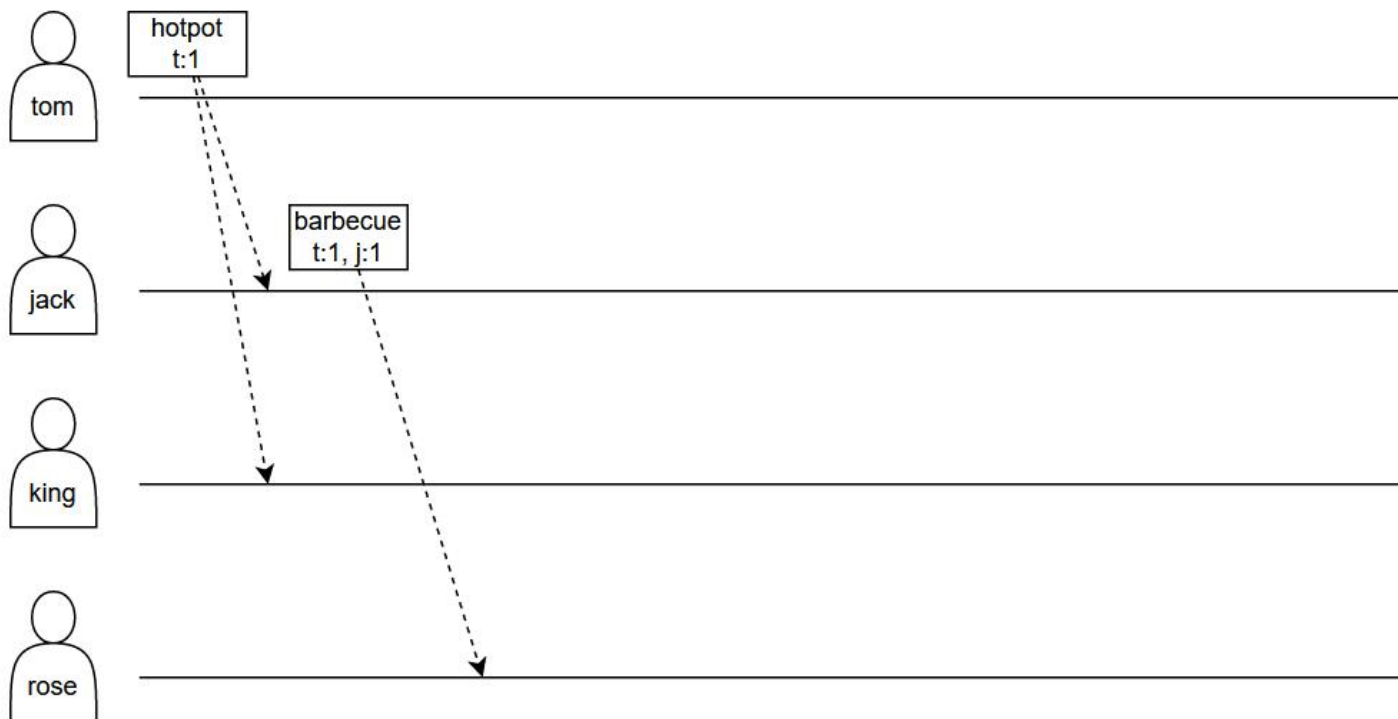
food = hotpot

vclock = tom:1

由于tom和rose刚刚认识，tom还没有rose的好友，于是tom在给jack的消息中让他通知rose

因果一致性

使用向量时钟解决冲突检测问题



jack在收到tom发出的提议之后，提出自己的提议烤肉（barbecue），此时jack的时钟向量就是：

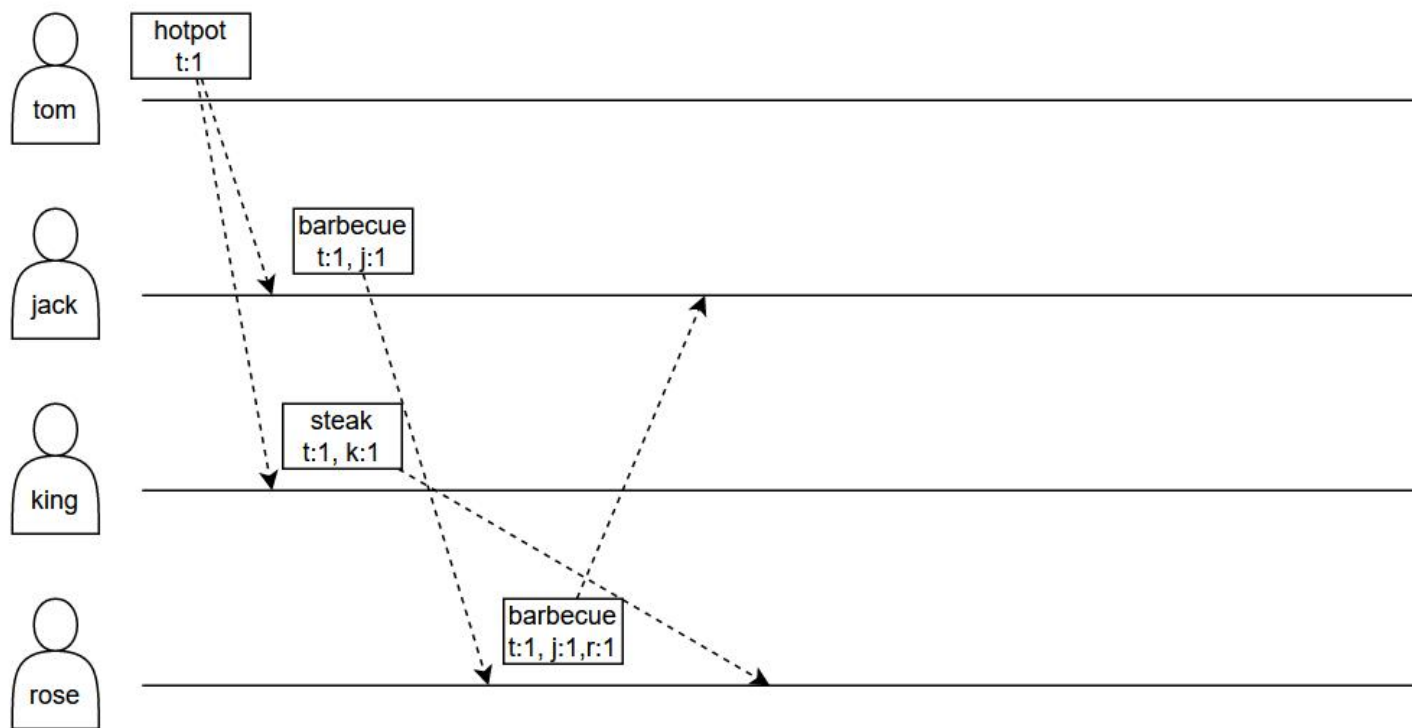
food = barbecue

vclock = tom:1,jack:1

然后将消息发送给rose，如图所示

因果一致性

使用向量时钟解决冲突检测问题



rose收到之后，回复jack表示同意
barbecue，此时rose的时钟向量是：

food = barbecue
vclock = tom:1,jack:1,rose:1

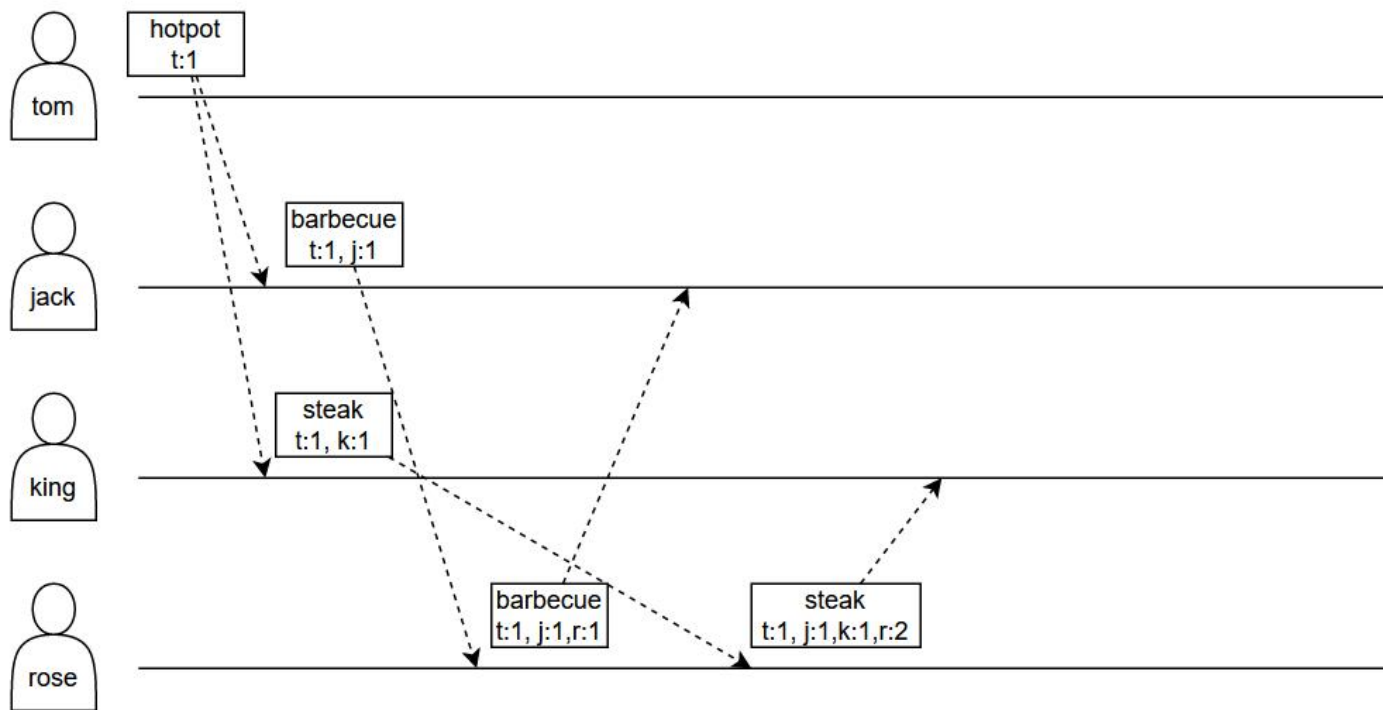
现在king接到tom消息后参与进来，建议
吃牛排（steak），此时king的时钟向量
是：

food = steak
vclock = tom:1,king:1

然后将提议发送给了rose

因果一致性

使用向量时钟解决冲突检测问题



rose此时就有了两份有冲突的数据（冲突的原因是jack和king “同时” 修改了最初tom的数据）：

food = barbecue

vclock = tom:1,jack:1,rose:1

和

food = steak

vclock = tom:1,king:1

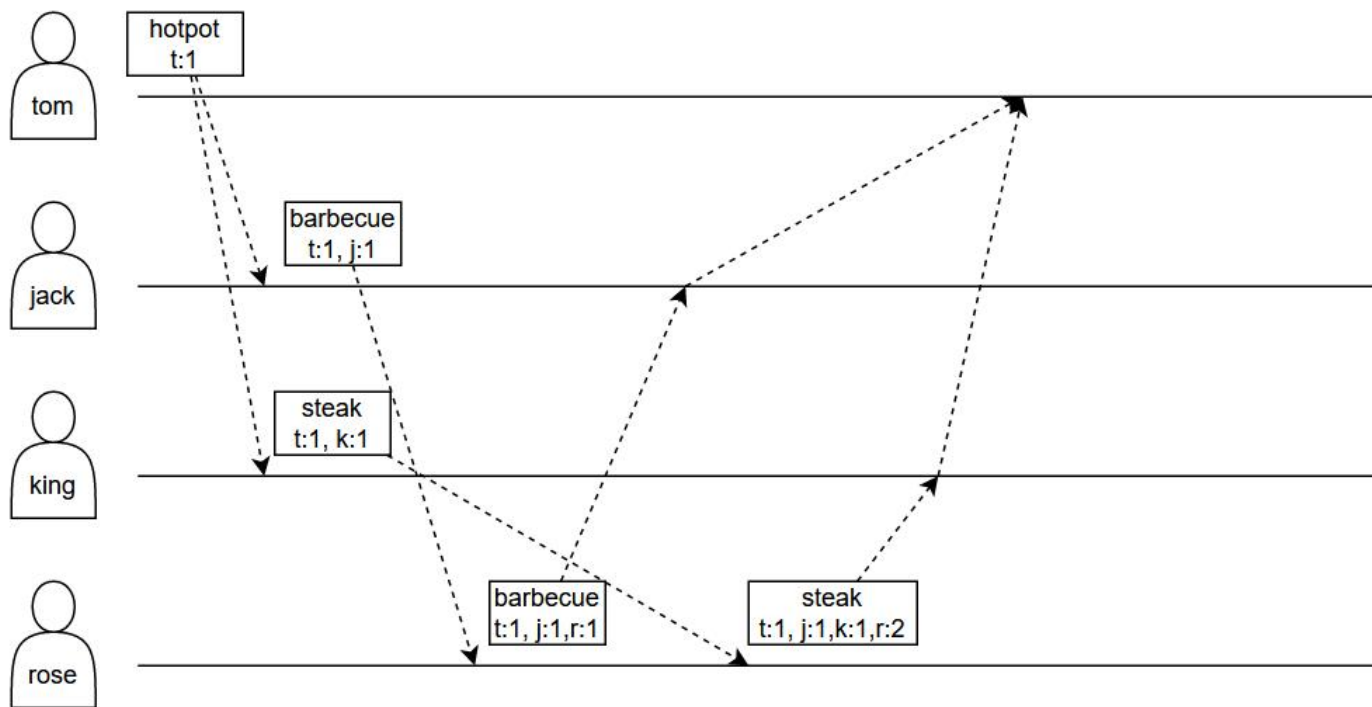
rose发现了冲突，因为当前的向量时钟和新接收到的向量时钟上的各个逻辑时钟并不是全部小于或大于。rose选择了steak解决冲突并回复给king和jack，此时她的向量时钟为：

food = steak

vclock = tom:1,jack:1,king:1,rose:2

因果一致性

使用向量时钟解决冲突检测问题



至此，当tom问jack和king最新的决定，
会收到来自jack的：

food = barbecue

vclock = tom:1,jack:1,rose:1

和来自king的：

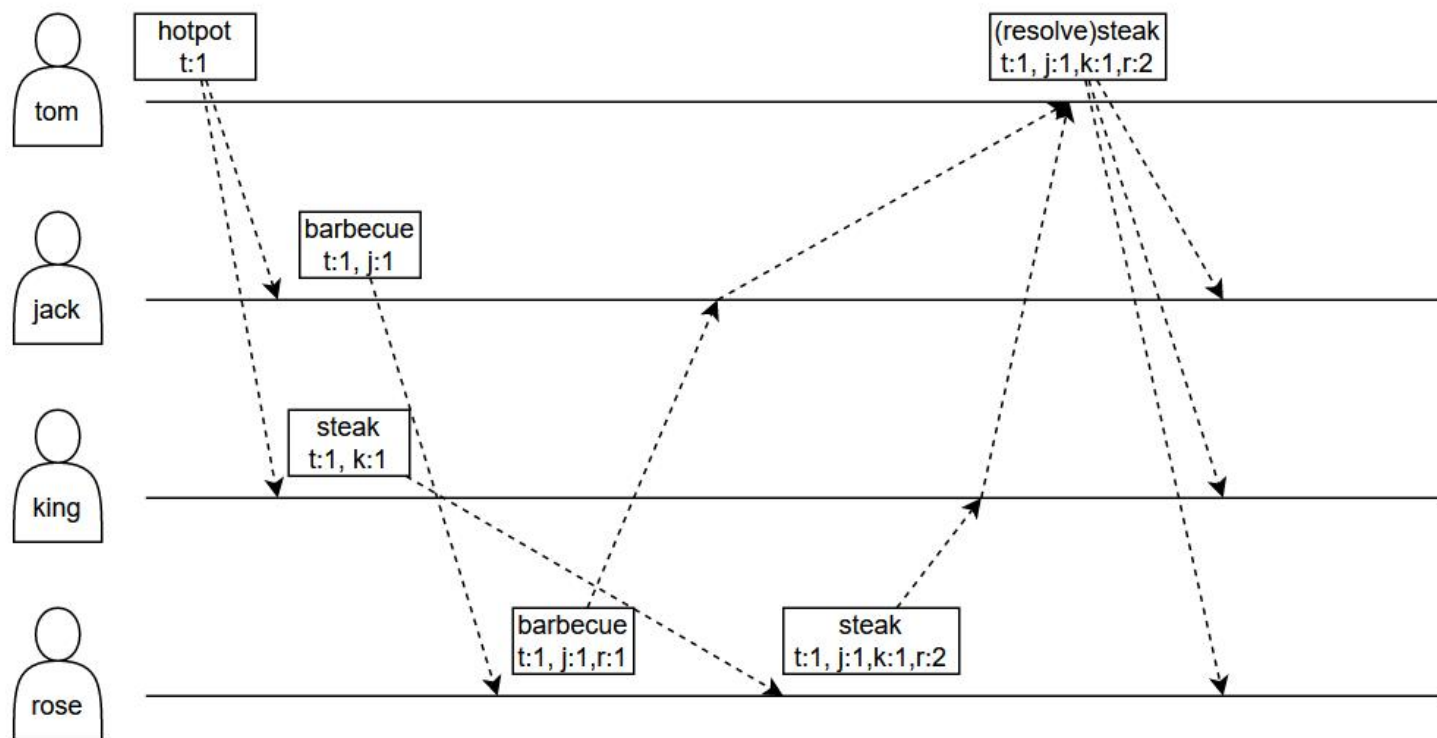
food = steak

vclock = tom:1,jack:1,king:1,rose:2

这时rose也已经和tom加了好友

因果一致性

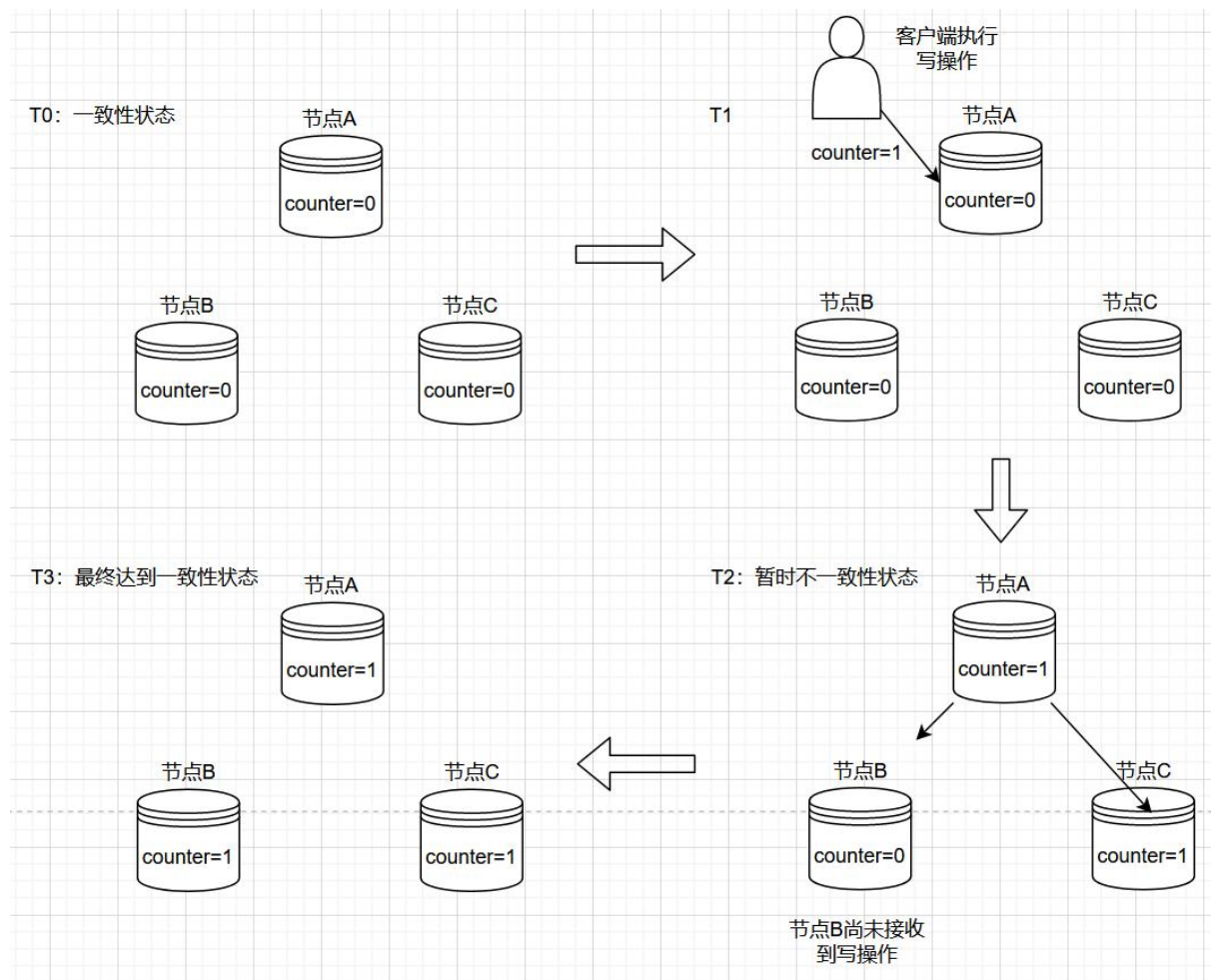
使用向量时钟解决冲突检测问题



这样，tom可以说rose与king一致修改了自己和jack的提议，并将king的消息给jack，jack就知道它的提议已经被修改，如图所示。

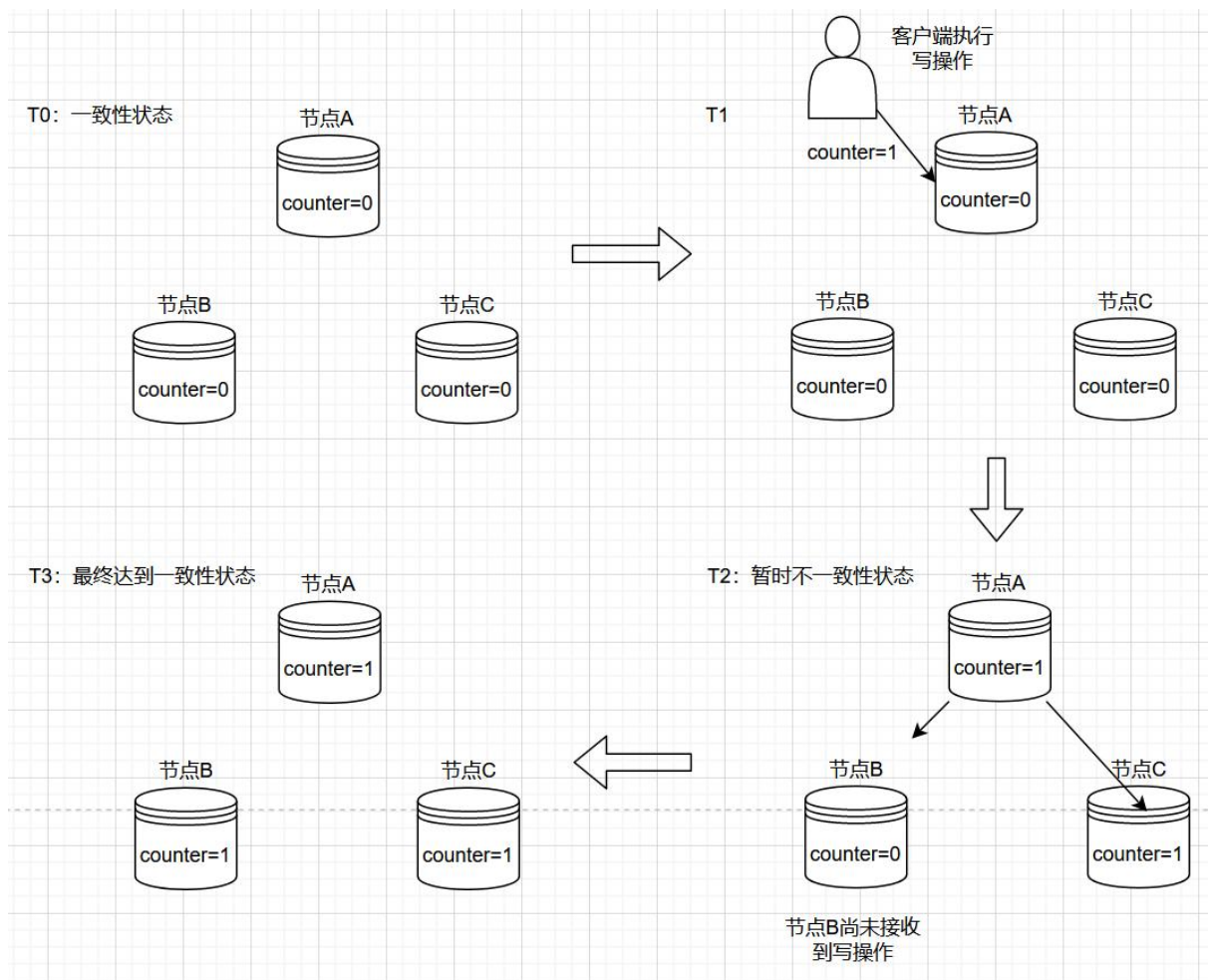
现在它们愉快地达成了一致。对于整个协商的过程，可以看到冲突的根源在于jack和king都“同时”修改了tom的数据，而rose通过向量时钟发现了数据冲突，并解决了它，最后他们达成一致。

最终一致性



- 最终一致性是可用分布式系统中常用的弱一致性模型，允许存在短暂的数据不一致，但保证最终所有节点的副本会达到相同的状态。
- 此模型优先考虑系统的可用性和性能，牺牲即时一致性，以满足大数据系统对高性能的需求。
- 在实际应用中，尤其是在大数据领域，最终一致性因其高效和适应性而成为最常用的一致性模型。

最终一致性



- 分布式KV存储系统包括节点A、节点B和节点C，目标是存储键为“counter”的键值对，记录计数器的值。
- 初始时，“counter”的值为0，分布在所有节点上的副本都是一致的。
- 客户端在节点A上发起写请求，将“counter”的值增加1，此操作被记录在节点A上并传播至其他节点。
- 写操作被复制到节点B和节点C后，它们也会对各异的“counter”副本执行相同的写操作，将值增加到1。
- 在写操作传播的过程中，可能会出现一段时间的不一致性，但随着时间推移，所有节点的“counter”副本最终达到一致状态（值为1）。

最终一致性

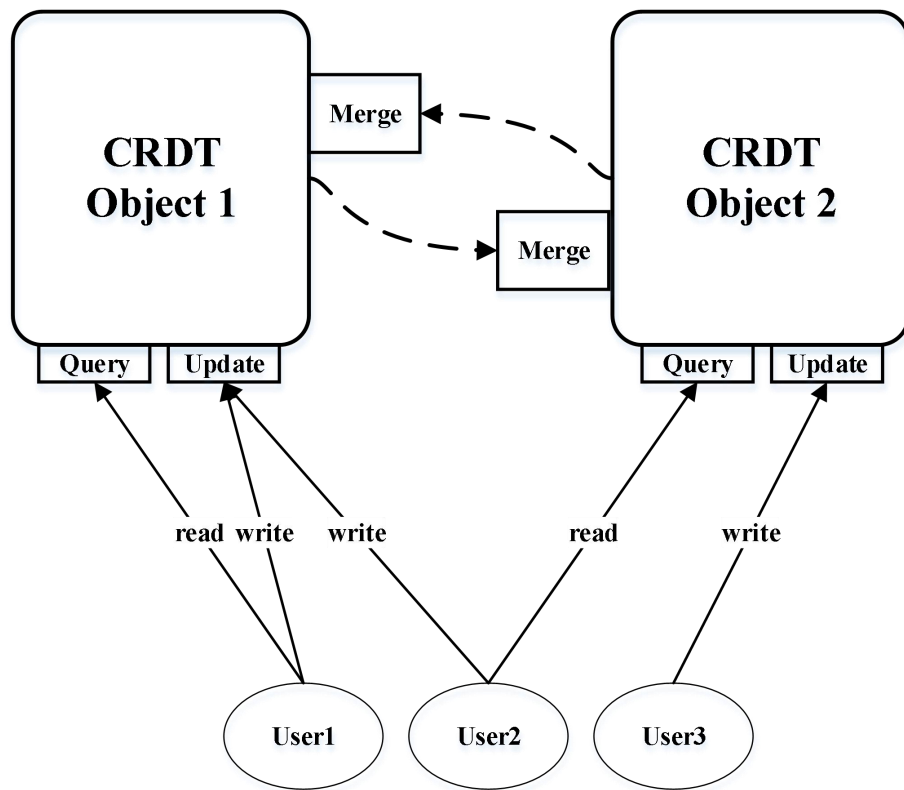
BASE理论

BASE理论是分布式系统设计中的一个指导原则，它强调在面对大规模分布式系统的设计时，可以放松一些传统的强一致性要求，以换取更好的可用性和性能。BASE 是三个术语的缩写：Basically Available（基本可用）、Soft State（软状态）和Eventually Consistent（最终一致性）。

- **Basically Available**（基本可用）：系统在出现故障或分区时仍然保持基本的可用性。这意味着系统不一定要保证一直处于正常工作状态，但它至少应该能够提供部分功能。
- **Soft State**（软状态）：系统中的状态信息可能会随着时间的推移而发生变化，即系统状态不是严格一致的。这种情况下，不同节点的数据副本可能不同步，但最终会达到一致状态。与之对应的是硬状态（Hard State）：数据库状态必须一直保持数据库状态一致性，就是指任意时刻数据必须时正确的。
- **Eventually Consistent**（最终一致性）：尽管系统在某个时刻不一定立即达到一致状态，但经过一段时间后，系统最终会达到一致状态。这种一致性级别更弱，但在大多数情况下是可以接受的。

最终一致性

CRDT无冲突复制数据类型

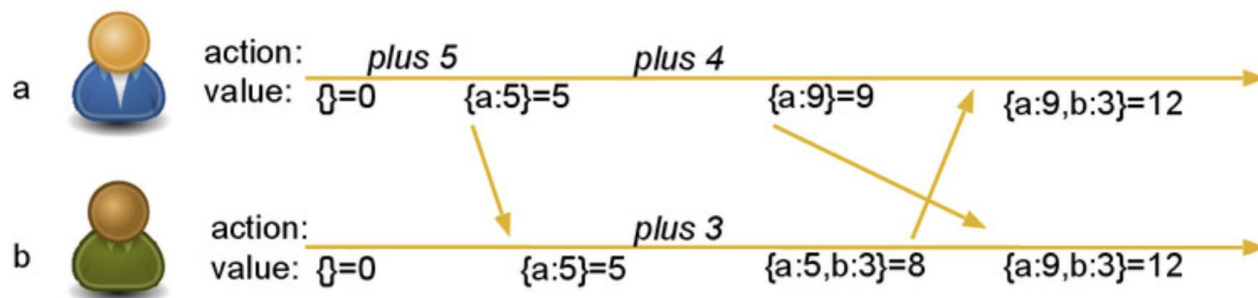


- CRDT (Conflict-Free Replicated Data Type, 无冲突复制数据类型) 是一种特定的数据结构, 允许数据在网络中多台计算机上复制, 并能独立并行更新而无需副本间协调。
- 更新操作设计为不会产生冲突, 因为副本间的操作可以独立进行, 且合并顺序具有可交换性。
- CRDT分为两类: 面向状态的CRDT (State-based CRDT) 和 面向操作的 CRDT (Operation-based CRDT), 两者都支持无冲突的数据复制和同步。

最终一致性

面向状态的CRDT

CRDT: State-based counter



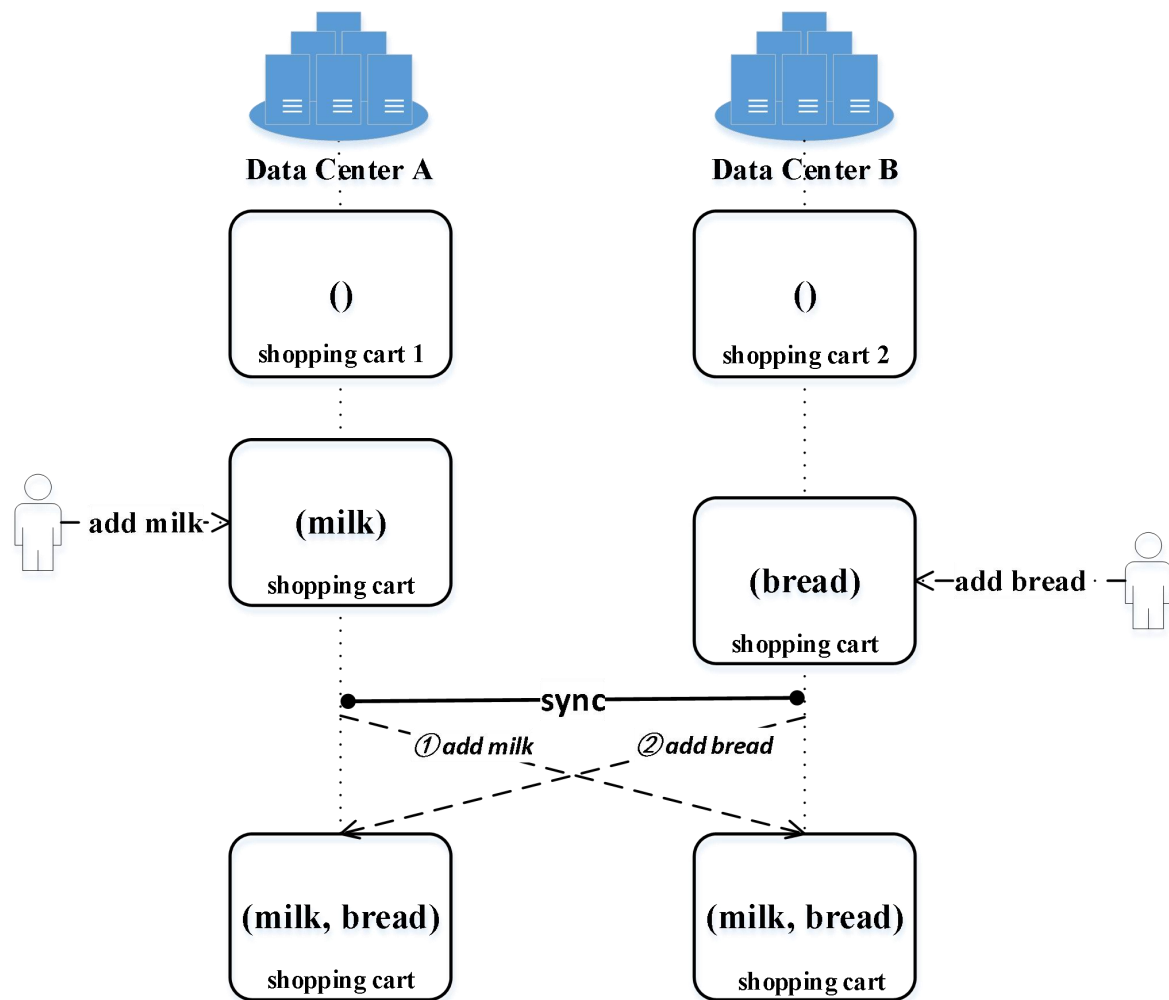
State-based CRDT:

- Local *update*
- Send state and *merge*

- 面向状态的CRDT通过传播完整的数据状态来在副本之间保持一致性。
- 每个副本维护自己的状态，使用从其他副本收到的状态进行更新。
- 状态更新过程通常涉及合并或取并集操作，确保所有副本最终达到相同的状态。

最终一致性

面向操作的CRDT



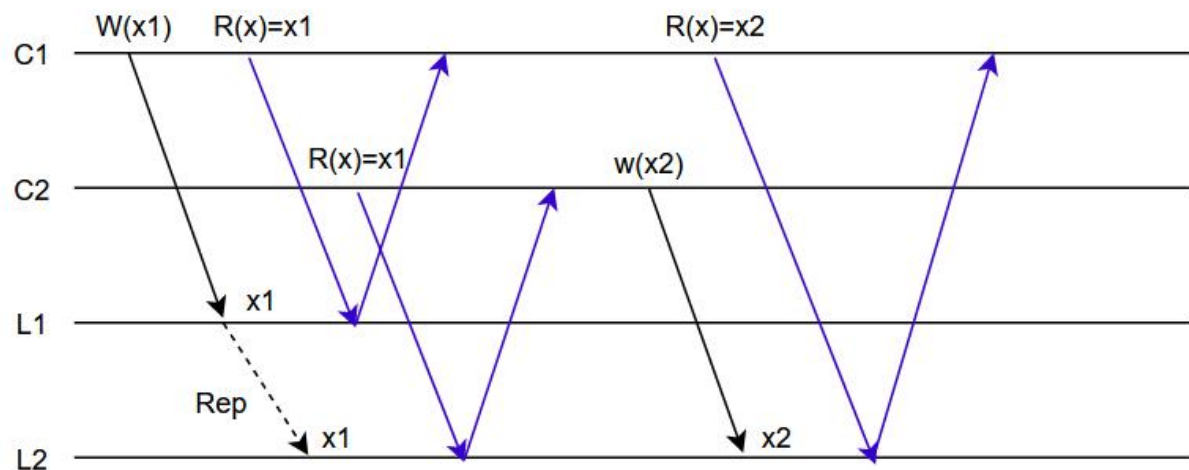
- 面向操作的CRDT通过在每个数据副本上应用操作来实现数据的复制和同步。
- 所有操作在不同副本间传递，并以相同顺序应用，确保所有副本状态一致。
- 操作具有可交换性和并发性，允许副本间同步时不发生冲突。
- 通过这种方式，面向操作的CRDT能够实现无冲突的复制数据类型。

按客户端角度区分的一致性

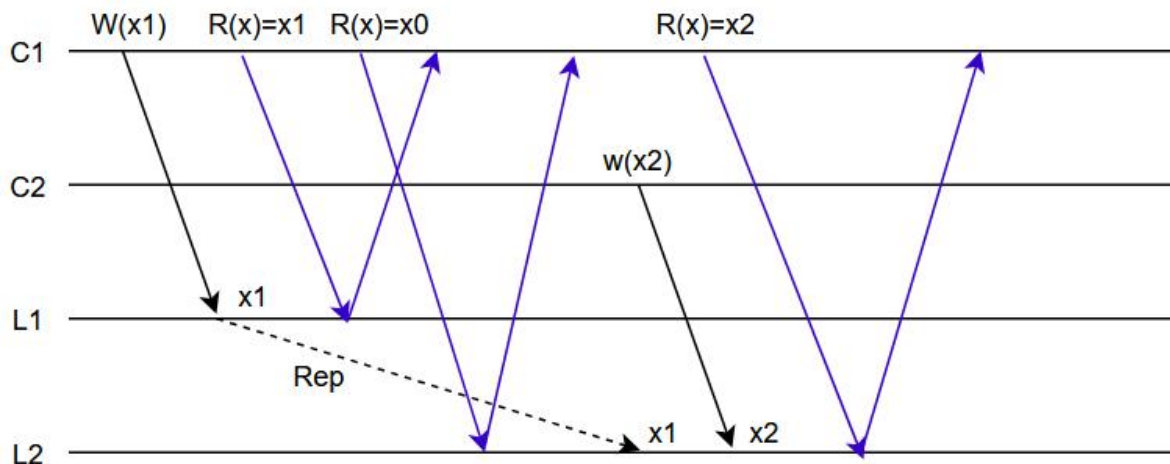
单调读一致性

- 当一个客户端读了数据项 x 的值后，该客户端所有后续的对 x 的读操作，都将返回相同的值，或者更新的值
- 例：读email|旧金山--> 纽约)

C1满足单调读一致性



C1不满足单调读一致性

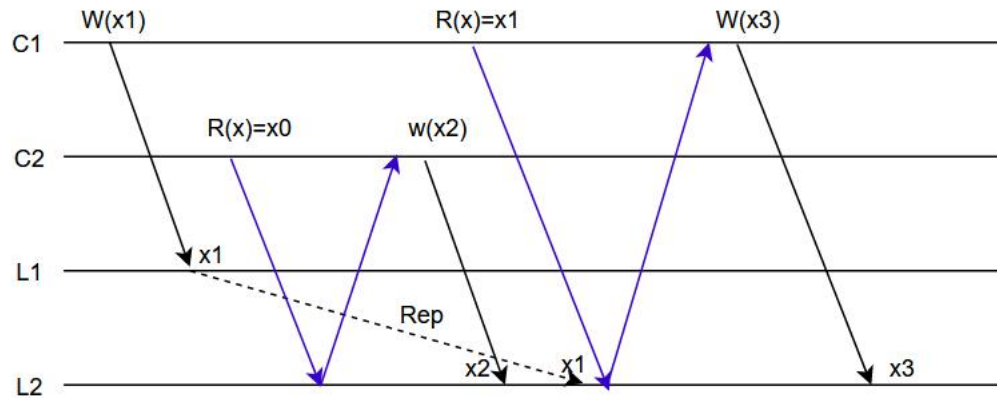
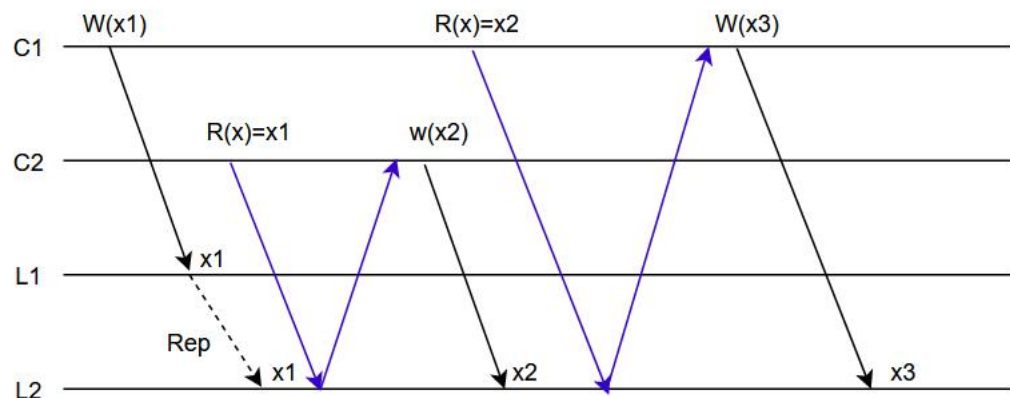


按客户端角度区分的一致性

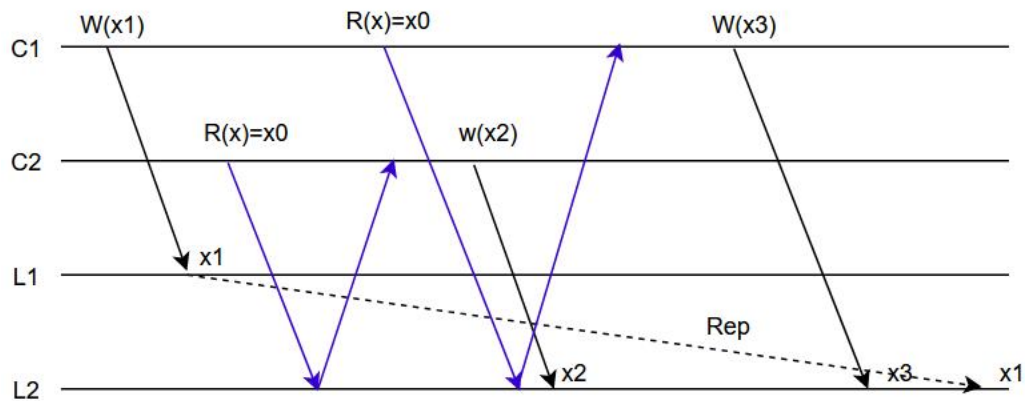
单调写一致性

- 来自同一个客户端的写操作被顺序地执行。
- 例：软件库更新（版本1,...,n）

C1满足单调写一致性



C1不满足单调写一致性

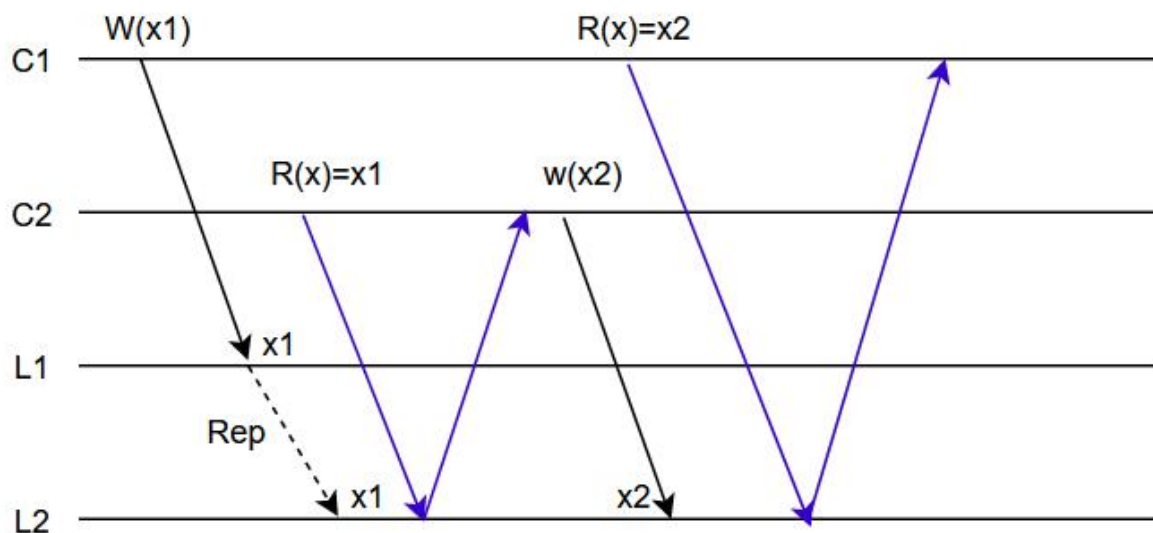


按客户端角度区分的一致性

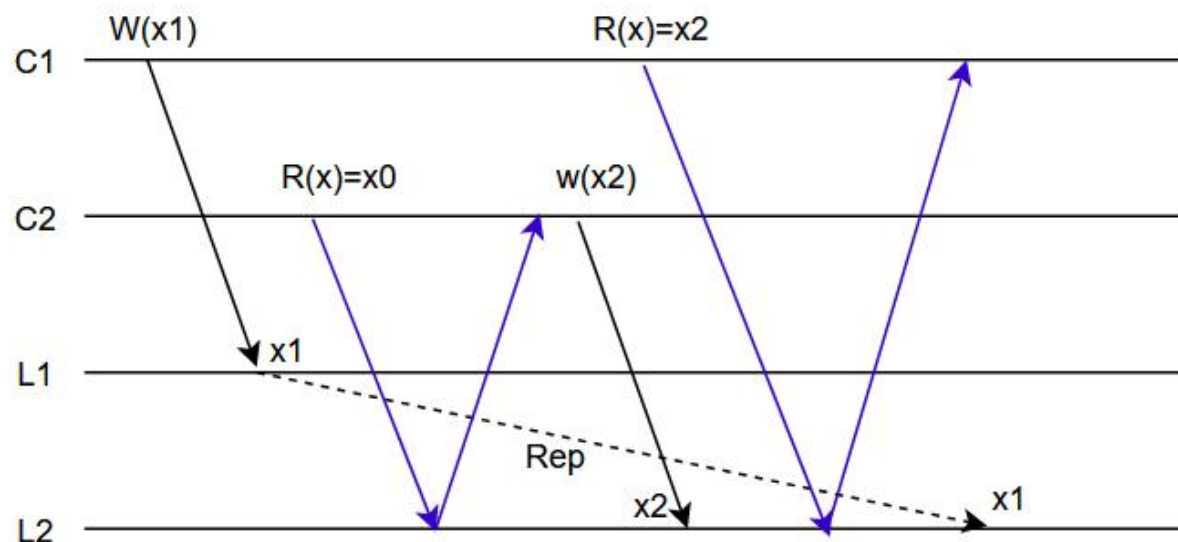
读自己写一致性

- 一个客户端对数据项 x 的写操作结果，总能被该客户端对 x 的后续读操作读到。
- 例1：Web网页更新、浏览
- 例2：数字图书馆密码更新

C1满足读自己写一致性



C1不满足读自己写一致性

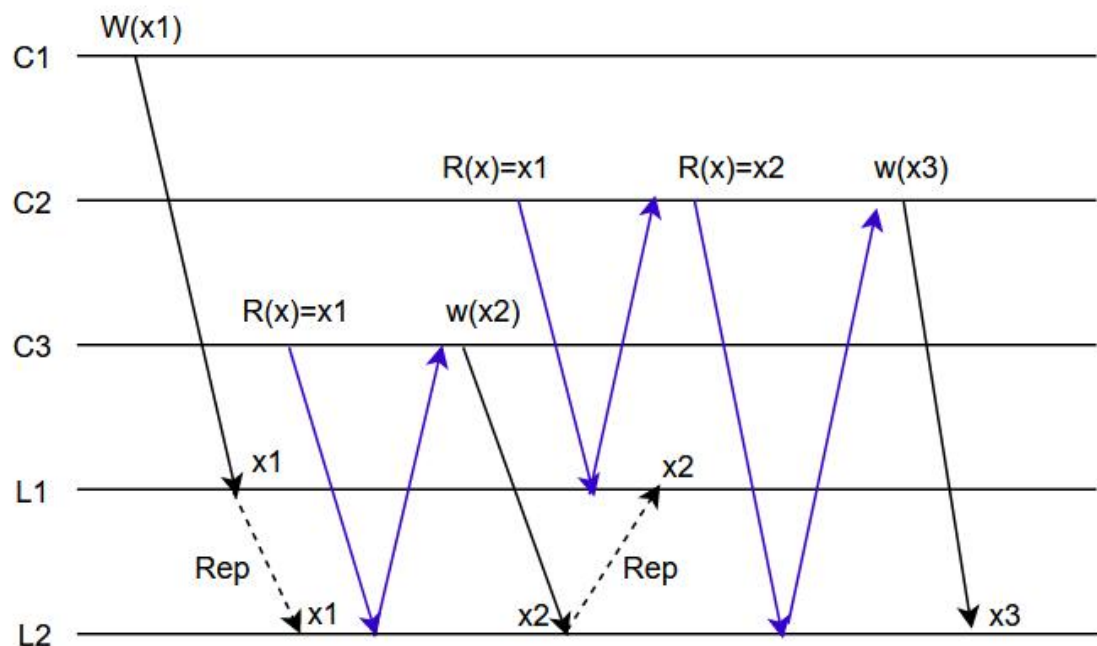


按客户端角度区分的一致性

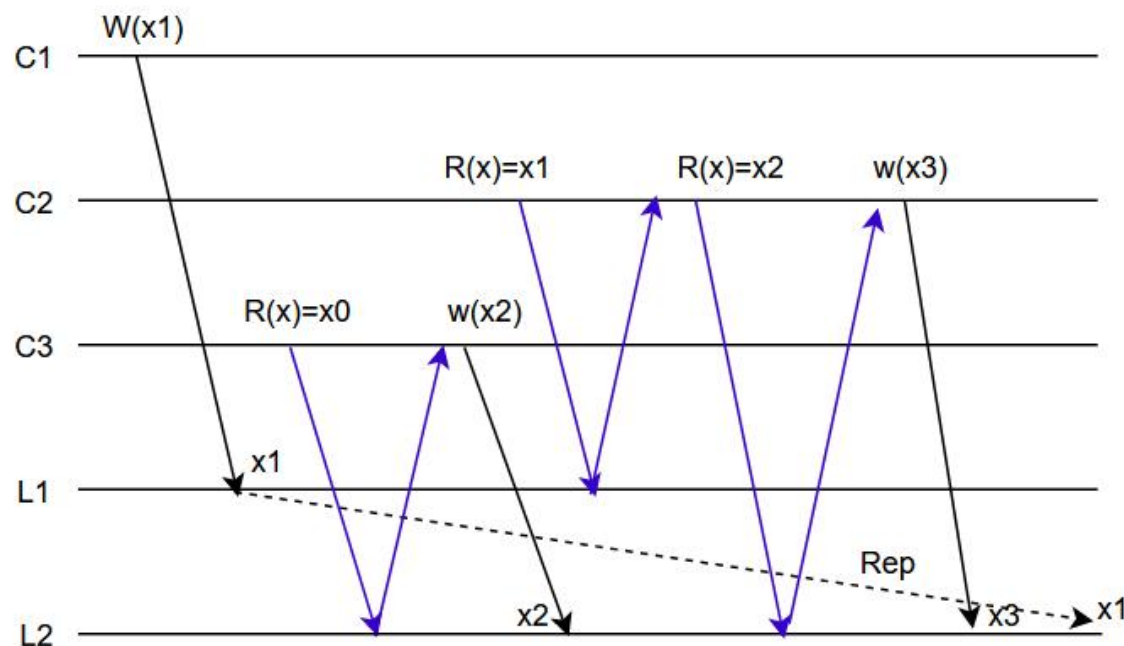
写跟随读一致性

- 一个（客户的）进程在对数据项x读操作之后，客户对x的写操作，必须在x已读出的相同值或者更近的值之上进行
- 例：BBS跟帖（读文章A, 跟帖写文章B）

C2满足写跟随读一致性



C2不满足写跟随读一致性



本章小结

- 4.1节：详细介绍了主从复制、多主复制和无主复制三种架构，揭示了这些架构的原理、可能出现的冲突，以及目前采用这些架构的系统。
- 4.2节：介绍了同步复制、异步复制、半异步复制等不同策略，同时还探讨了物理复制和逻辑复制等不同复制方式。
- 4.3节：针对一致性模型，介绍了线性一致性、顺序一致性、因果一致性、最终一致性等不同的一致性模型，同时探讨了全局时钟同步问题、谷歌Spanner TrueTime、TiDB的工作原理、全序广播、逻辑时钟、向量时钟以及CRDT等相关内容。按客户端角度区分，介绍了单调读一致性、单调写一致性、读自己写一致性、写跟随读一致性等不同的一致性要求。

第四章 完