# 分布式计算系统

## 中间件的概念及其角色、位置

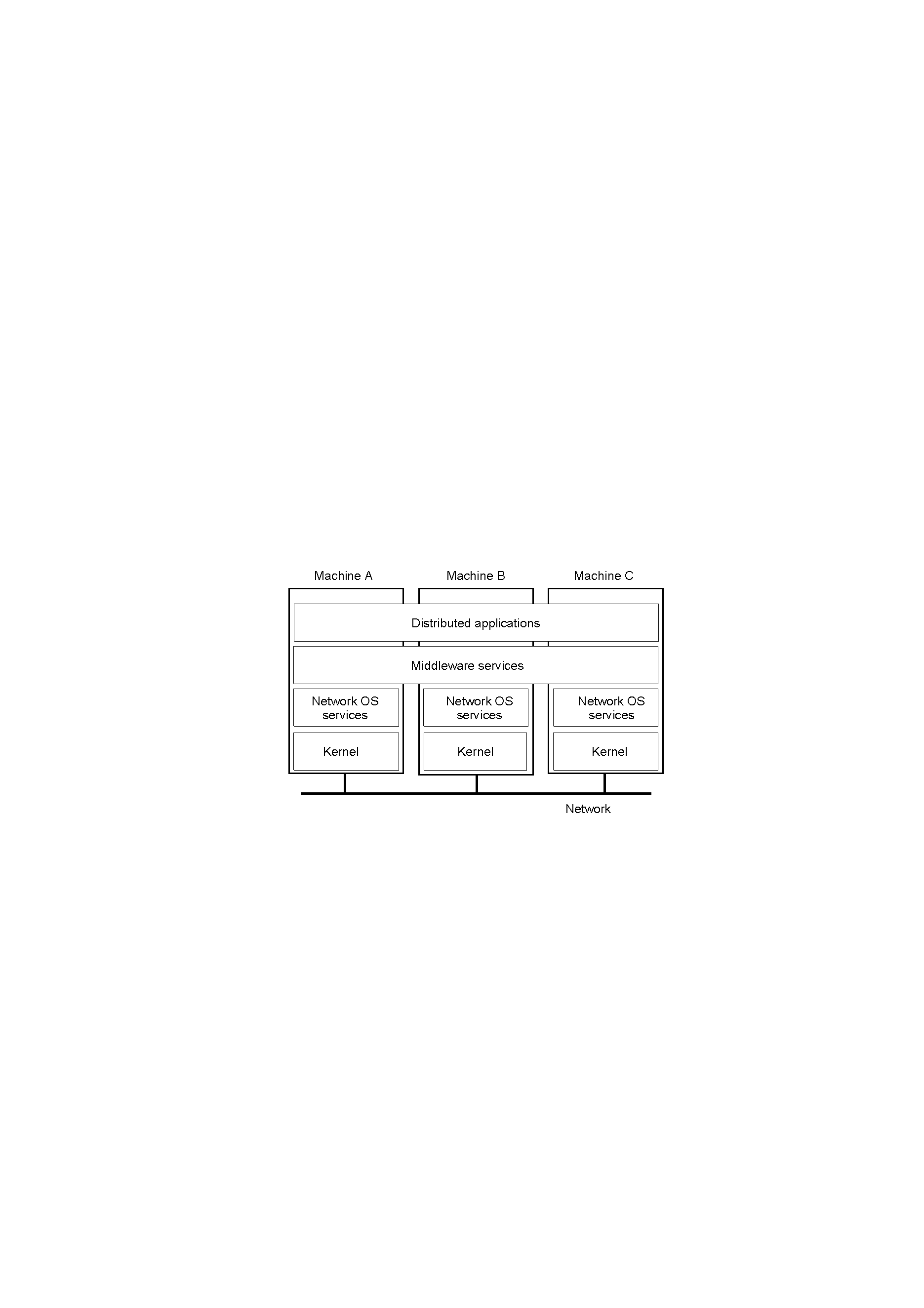
1. **分布式操作系统与网络操作系统**

分布式操作系统并不是用来管理一组“独立”的计算机的；

网络操作系统也没有提供“单个一致的系统” 的视图。

1. **中间件的定义**

为了使种类各异的计算机和网络都呈现为单个的系统，分布式系统常常通过一个“软件层”组织起来，该层在逻辑上位于有用户和应用程序组成的高层与有操作系统组成的低层之间，这样的分布式系统又成为中间件。



1. **中间件目标**

中间件系统的一个重要目标是对应用程序隐藏底层平台的异构性，因此许多中间件系统都提供一组完整程度不同的服务集；

它的重要目的是提供一定的透明度，也就是一定程度上向应用程序隐藏数据处理的分布性。

1. **中间件特点**

中间件延伸到多台机器上，且为每个应用程序提供了相同的接口；

中间件系统都提供一组完整程度不同的服务集，这些服务必须使用系统提供接口来访问，除此以外没有别的访问方法，一般禁止跳过中间件层直接调用底层操作系统的服务。提高系统安全性。

中间件集分布式操作系统与网络操作系统的优点于一身，既能够具有网络操作系统的可扩展性和开放性，又能具有分布式操作系统的透明性和与之相关的高可用性。

1. **中间件的模型：**

**基于分布式文件系统：**只对传统文件支持分布透明性

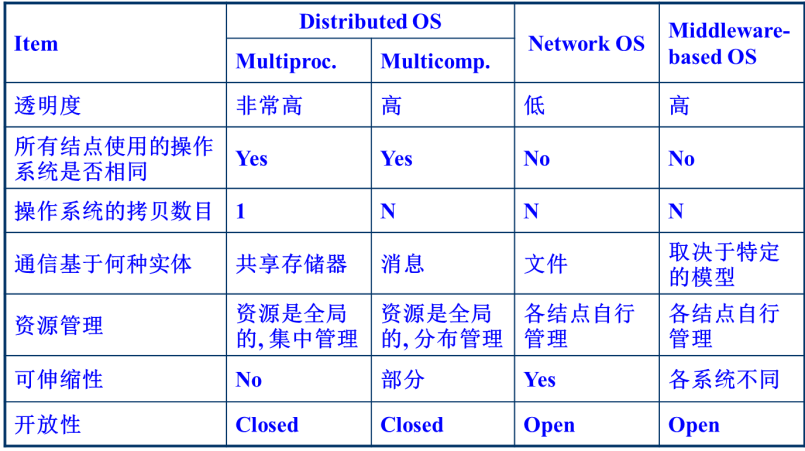
**基于RPC：**允许进程调用位于远程机器上的过程实现来隐藏网络通信，调用参数被透明地传输到远程机器上，随后在远程机器上执行该过程，执行完毕后将结果发送给调用者

**基于分布式对象：**本质上是实现了一种特殊的接口的对象，这种接口向对象隐藏了对象的所有内部细节。接口包含而且只包含由对象实现的方法。进程只能看到对象的接口

1. **中间件的功能：**

提供命名服务；提供持久性的存储方面的特殊功能；提供一些分布式事务的功能；提供安全功能

1. **操作系统之间的比较：**



## 分布式系统进程通信原理和步骤（RPC步骤）

1、客户进程以正常的方式调用客户存根(stub)；

2、客户存根生成一个消息，然后调用本地操作系统；

3、客户端操作系统将消息发送给远程操作系统；

4、远程操作系统将消息交给服务器存根；

5、服务器存根将参数提取出来，然后调用服务器；

6、服务器执行要求的操作，操作完成后将结果返回给服务器存根

7、服务器存根将结果打包成一个消息，然后调用本地操作系统；

8、服务器操作系统将含有结果的消息发送回客户端操作系统；

9、客户端操作系统将消息交给客户存根；

10、客户存根将结果从消息中提取出来，返回给调用的客户进程。

## 3、移动Agent技术

**1、Agent基本概念：**

Agent是驻留于环境中的实体，它可以解释从环境中获得的、反映环境中所发生的事件的数据，并执行对环境产生影响的行为。

**2、移动Agent的定义：**

移动Agent 就是一种可以移动的软件Agent，它可自主地在异构的网络上，按照一定的规程移动，寻找合适的计算资源、信息资源或软件资源，利用与这些资源处于同一主机或网络的优势，就近处理或使用这些资源，完成特定的任务。

移动agent是一个运行于开放、动态网络环境中的封装良好之计算实体，它代表用户自主地在网络上移动，完成指定的任务。移动agent由数据、操作和行为规则封装而成。

**3、移动Agent特性：**自主性、反应性、主动/面向目标、推理/学习/自适应能力、可移动性、协作性、社会性、安全性等。

**4、智能软件Agent概念：**是能为用户执行特定的任务、具有一定程度的智能、以允许自主执行部分并以一种合适的方式与环境相互作用的软件程序。

**5、移动Agent优势：**

（1）减少网络负担：移动Agent允许用户将会话打包转移至资源请求的目标主机，这样远程会话变为本地会话

（2）客服网络延迟问题，将资源从中心控制器转移到本地，这样就可以从本地直接执行中心控制指令

（3）封装协议

（4）异步并自治执行

（5）动态调整

（6）天然的异构型

（7）是健壮和可容错的

**6、Agent为何会通信失效？**

通信失效是指在特定条件约束下的信件不能或不能及时到达接受者的一种现象。

原因是在路由信件和实际信件传输过程中，目标Agent发生了物理位置的变化。这种变化是随机的、不可预料的。

**7、Mogent解决通信失效采取方案**

在Mogent系统中，每一个Agent都是由特定的结点产生，称之为本Agent的Home结点，该结点记录该结点上“出生”的Agent的动态信息。在发生Agent迁移时，需要及时信息更新。Agent迁移必须通知其Home结点，每一个结点都有一个Communicator，它可以用来记录当前主机上的所有Agent信息，包括本地结点的Agent信息，以及从其他结点迁移来的Agent信息。Agent迁移时要在Communicator上注册或注销。

通信失效问题本质上是通信和移动所共享的“位置”信息未同步造成的。Mogent系统中引入了“状态”的概念，每个Agent的状态分为迁移态和静止态。 同时引入两个信号量进行集中式同步控制策略，通过Home-Communicator集中管理对目标Mogent的地址信息的互斥访问，从根本上避免通信失效的发生，保证通信的可靠性。

**迁移状态：**由Home构件记录Agent的状态，当Mogent在一个Host上时状态为静止态，当按旅行计划准备迁移时Mogent要通知Home更新状态为迁移态。

**在途信件数：**保存在Host上，记录当前时刻以该Mogent为通信对象的在途信件数，发出信件，在途信件数增加，信件到达，在途信件数减一

**消息发送机制：**Mogent通过控制“在途信件数”和“迁移状态”这两个信号量，确保消息发送者仅向处于静止状态的Agent发送消息 Mogent迁移机制：Agent只有在没有消息发送给他的情况下才能移动。

**8、Mogent局限性：**频繁迁移，Home的地址注册开销大，会拥挤。Agent迁移受到信号量的限制,影响Agent的自主性和移动性

## 4、命名服务

**1、名称解析的定义：**名称解析就是查询名称的过程。如N: <lable-1, label-2, …, label-n>。 该名称的解析是从命名图的节点N开始的，在其目录表中查询名称label-1，并返回label-1所指向的节点的标识符。然后，解析过程在标识出的节点的目录表中继续查询名称label-2，依此类推，假设该命名路径确实存在，那么解析会在由label-n指向的最后一个节点上停止，并返回该节点的内容。

**2、别名的实现方式：**

（1）允许多个绝对路径名来指向命名图中的同一节点

（2）用叶节点来表示实体

**3、分布式系统中挂载外部空间所需要的信息：**

（1）访问协议的名称

（2）服务器的名称

（3）外部名称空间中挂载点的名称

**4、挂载的作用：**挂载是合并不同名称空间时所使用的一种方式，另一种方式来自DEC公司的全局名称服务器(GNS)。它的做法是添加一个新的根节点，然后把现有的根节点变成它的子节点

**5、名称空间与命名服务：**名称空间形成了命名服务核心，命名服务就是一种允许用户和进程添加、删除和查找名称的服务

**6、名称空间的实现：**

名称空间通常分层组织，为了有效地实现这样的名称空间，把它分成逻辑上的层：

（1）全局层由最高级别的节点组成，即由根节点以及其他逻辑上靠近根节点的目录节点

（2）行政层由那些在单个组织内一起被管理的目录节点组成

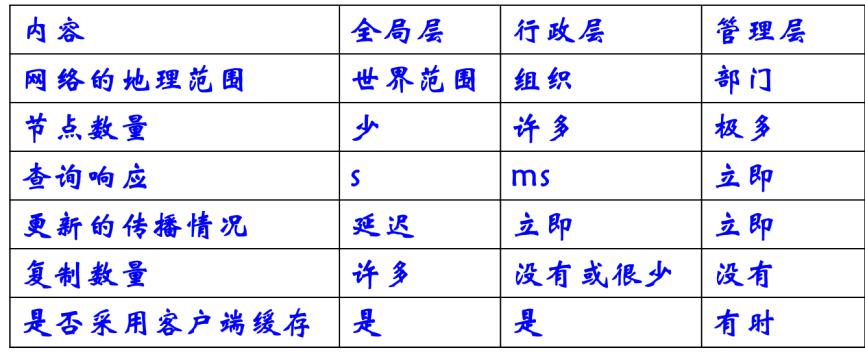
（3）管理层由那些经常改变的节点组成

在可用性和性能方面，每层的名称服务器都必须满足不同的要求：

在全局层中，良好的可用性对名称服务器特别重要；全局层的名称服务器并不需要快速响应单一的查询请求，相反，重要的可能是其吞吐能力。

行政层与全局层类似，但是在性能的要求上稍高于全局层。

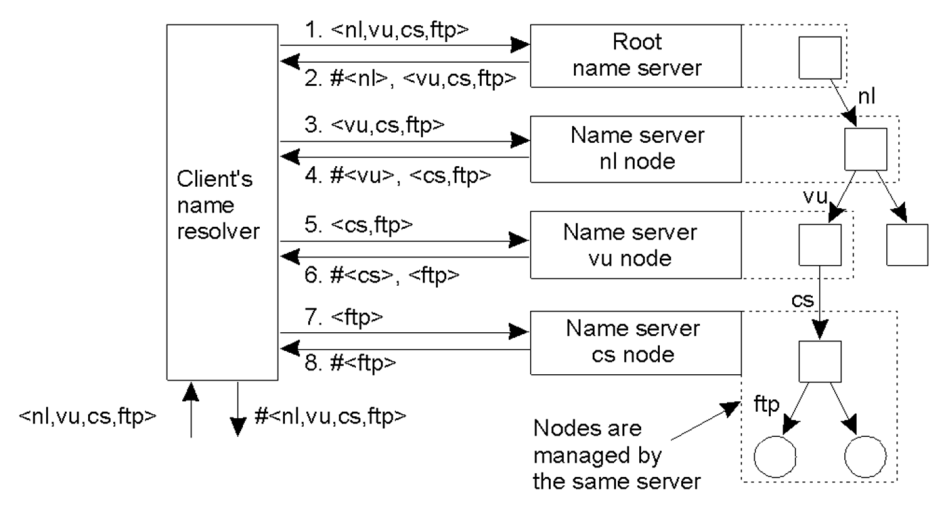
对管理层名称服务器的可用性要求一般比较低，但是对管理层的性能要求较高。



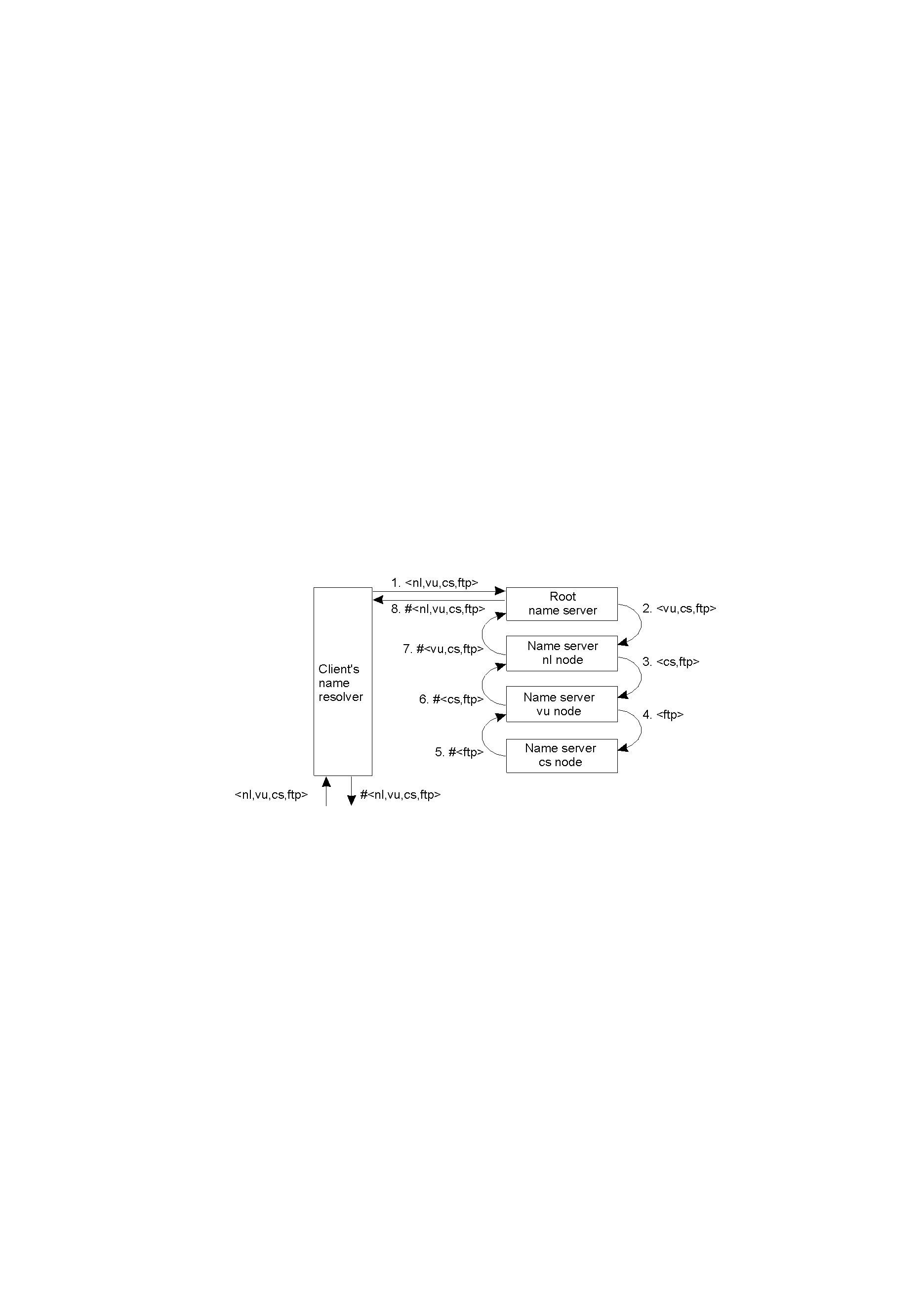
**7、名称解析的实现：**

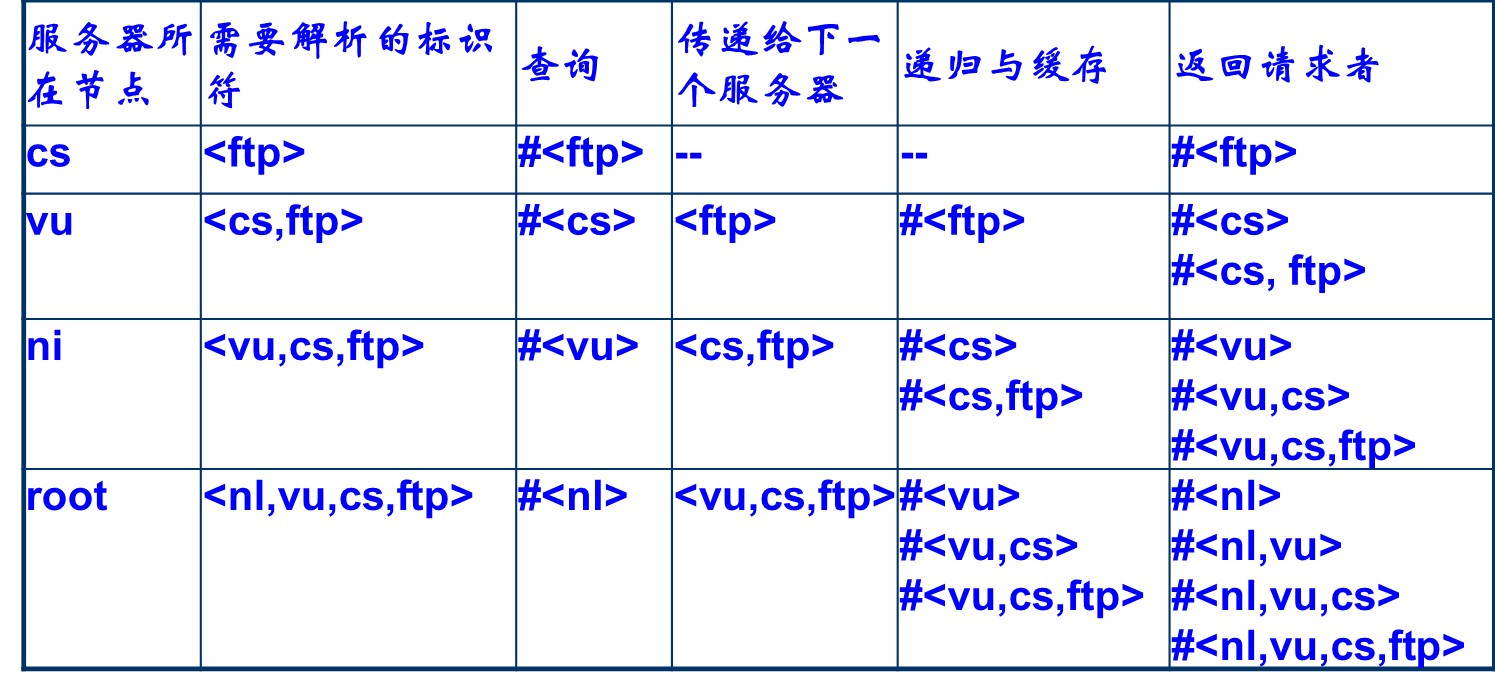
可以用两种方法实现名称解析：迭代名称解析、递归名称解析

**8、迭代名称解析：**在迭代名称解析中，名称解析程序把完整的名称转发给根名称服务器。假定可以联系根服务器的地址是众所周知的。根服务器会尽量深入地解析路径名，然后把结果返回给客户。接着，客户会把剩下的路径名发送给一台名称服务器，该服务器也尽量深入地解析路径名并把结果返回给客户，这一过程迭代地进行，最终含有请求地址的服务器将地址返回给客户，每次迭代过程都是客户将路径名交给一台服务器，服务器吧iexidao的地址反馈给客户，客户在想下一集服务器发送解析请求。



**9、递归名称解析：**与返回客户解析程序每个中间结果不同，在使用递归名称解析时，名称服务器会把结果传递给它找到的下一台服务器。在同样使用递归名称解析的情况下，下一台服务器将解析完整的路径名，最终会把解析得到的完整名返回给根服务器，根服务器再一次把该文件传递给客户的名称解析程序。





**10、迭代名称解析与递归名称解析的比较：**

（1）递归名称解析的主要缺点是它要求每台名称服务器具有较高的性能。（需要名称服务器对路径名进行完整的解析，这种额外的负担非常大，以至于在名称空间的全局层中，名称服务器只支持迭代名称解析）

（2）递归名称解析拥有两个重要的优点：与迭代名称解析相比，递归名称解析的缓存结果要更为有效一些；减少通信开销。

从根本上说，递归名称解析方法的优点就是可以非常有效地完成查询操作。如果使用迭代名称解析，那么就需要把缓存限制在客户的名称解析程序上。作为一种折衷的方案，许多组织都使用一台本地的中间名称服务器，所有客户都共享它。这台本地名称服务器处理所有的名称请求并缓存结果。

**11、域名系统DNS：**

当今使用的最大的分布式名称系统之一是Internet域名系统()。DNS主要用来查找主机地址和邮件服务器。DNS名称空间是分层组织的，就像一颗有根的树。标识符是区分大小写的字符串，字符串由字母和数字组成。标识符的最大长度是63个字符，完整路径名的最大长度是255个字符。

## 5、选举算法

算法前提：由哪个进程充当这个特殊的角色并不重要，重要的时它们中要有一个进程来充当；每个进程有一个唯一的编号；每个进程都知道所有其他进程的进程号。

**1、欺负算法：**

**步骤：**当任何一个进程发现协调者不再响应请求时，它就发起一个选举。进程P按如下过程主持一次选举：

（1）P向所有编号比它大的进程发送一个election消息

（2）如果无人响应，P获胜成为协调者；

（3）如果有编号比它大的进程响应，则响应者接管选举工作。P的工作完成

当一个以前崩溃了的进程现在恢复过来时，它将主持一次选举。如果该进程恰好是当前正在运行的进程中进程号最大的进程，它将赢得此次选举，接管协调者的工作。这样，最大的进程总是取胜。

**正确性：**任何时刻，一个进程只能从编号比它小的进程得到一个election消息。当该消息到达时，接收者发回一个OK消息给发送者，表明它仍然在运行，并且接管选举工作。然后接收者主持一个选举。最终，除了一个进程外，其他所有进程都将放弃，那个进程就是新的协调者。它将选举获胜的消息发送给所有进程，通知它们自己是新的协调者。

**2、环算法：**

**步骤：**

（1）当任何一个进程注意到协调者不工作时，它就构造一个带有它自己的进程号的election消息，并将该消息发送给它的后继者。

（2）如果后继者崩溃了，发送者沿着此环跳过它的后继者发送给下一个进程，或者再下一个，直到找到一个正在运行进程。在每一步中，发送者都将自己的进程号加到该消息列表中，以使自己成为协调者的候选人之一。

（3）最终，消息返回到发起此次选举的进程。当发起者进程接收到一个包含自己进程号的消息时，它识别出这个事件。此时，消息类型变成coordinator消息，并再一次绕环运行，向所有进程通知谁是协调者(成员列表中进程号最大的那个)以及新环中的成员都有谁。这个消息再循环一周后被删除，随后每个进程都恢复原来的工作。

## 6、互斥算法

**1、集中式算法：**仿照单处理器系统中的方法，选举一个进程作为协调者，无论何时一个进程1要进入临界区，它都要向协调者发送一个请求消息，说明它想要进入哪个临界区并请求允许。如果当前没有其他进程在该临界区内，协调者就发送允许进入的应答消息。假设另一个进程2请求进入同一个临界区。协调者知道一个与此不同的进程已经在临界区内，所以它不能同意该请求。协调者将进程2的请求放到队列中，并等待更多的消息请求。

**特点：**

（1）保证了互斥的实现

（2）公平，允许请求的顺序同接收它们的顺序是一致的，没有进程会处于永远等待状态

（3）易于实现

（4）不仅能用于管理临界区，也可以用于更一般的资源分配

**缺点：**

（1）协调者是一个单个故障点，所以如果它崩溃了，整个系统就可能瘫痪

（2）在规模较大的系统中，单个协调者会成为性能的瓶颈

**2、分布式算法（以Recard为例）：**一个进程想进入一个临界区时，它构造一个消息（包含它要进入的临界区的名字、它的进程号和当前时间）。然后它将消息发送给所有其他的进程，理论上讲也包括它自己。假设消息的传送是可靠的，也就是说，每个消息都能得到确认。如果可能，可以使用可靠的组通信来代替单个消息。

当一个进程接收到来自另一个进程的请求消息时，它根据自己与消息中的临界区相关的状态来决定它要采取的动作。可以分为三种情况：

（1）若接收者不在临界区也不想进入临界区，它就向发送者发送一个OK消息

（2）若接收者已经在临界区，它不进行应答，而是将该请求放入队列中

（3）如果接收者想进入临界区但尚未进入时，它将对收到的消息的时间戳和包含在它发送给其余进程的消息中的时间戳进行比较。时间戳最早的那个进程获胜。如果收到的消息的时间戳比较早，那么接收者向发送者发回一个OK消息。如果它自己的消息的时间戳比较早，那么接收者将接收到的请求放入队列中，并且不发送任何消息

在发送了请求进入临界区的请求消息后，进程进行等待，直到其他所有进程都发回允许进入消息为止。一旦得到所有进程的允许，它就可以进入临界区了。当它退出临界区时，它向其他队列中的所有进程发送OK消息，并将它们从队列中删除

**特点：**

（1）不会发生死锁或者饿死现象

（2）不存在单个故障点

**缺点：**

（1）单个故障点被n个故障点所取代。如果任何一个进程崩溃，它就不能回答请求。只是设法用一种糟糕了n倍，并且要求更多网络通信的算法来代替前面的那种集中式算法（n个进程之一发生故障的可能性至少是单一协调者发生故障的n倍）。

解决方法：请求到达时，接收者无论是允许还是拒绝都发送应答

（2）要么必须使用组通信原语，要么每个进程都必须自己维护组成员的清单

（3）处理所有请求时会产生瓶颈问题（所有进程要参与做出与进入临界区有关的所有决定，强迫每个进程都做完完全相同的事情也是不可能的）

**3、令牌环算法：**

**条件：**总线式网络

**内容：**用软件的方法构造出一个逻辑环，环中为每个进程都分配了一个位置，每个进程要知道谁在它的下一个位置上。当环初始化时，进程0得到一个令牌token。该令牌绕着环运行，用点对点发送消息的方式把它从进程k传递到进程k+1(以环大小为模)。进程从它邻近的进程得到令牌后，检查自己是否要进入临界区。如果自己要进入临界区，那么它就进入临界区，做它要做的工作，然后离开临界区。在该进程退出临界区后，它沿着环继续传递令牌。不允许使用同一个令牌进入另一个临界区。如果一个进程得到了邻近进程传来的令牌，但是它并不想进入临界区，那么它只是将令牌沿环往下传递

**正确性：**该算法的正确性是显而易见的。

（1）在任何时候都只有一个进程有令牌，所以实际上只有一个进程能进入临界区。

（2）由于令牌以固定顺序在进程间循环传递，所以不会发生饿死现象。

（3）如果一个进程想进入一个临界区，那么最差的情况是等待其他所有进程都进入这个临界区然后再从中退出后它再进去

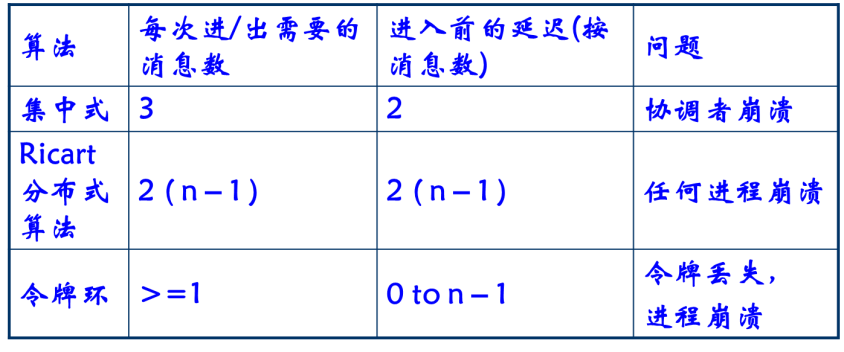
缺点：

（1）如果令牌丢失了，那么它必须重新生成令牌（检测令牌丢失是很困难的）

（2）如果有进程崩溃，该算法也会出现麻烦（恢复起来比其他算法容易）

**解决方案：**要求每个进程在收到令牌后发出确认信息，那么当一个进程的邻近进程试图把令牌传递给它但是没有成功时，这个崩溃的进程就会被检测到。将这个崩溃的进程从组中删除，令牌的持有者将令牌沿环传给这个崩溃进程的下一个进程，如果必要，传给再下一个（需要每个进程都维护当前环的配置信息）

4、三种算法比较：



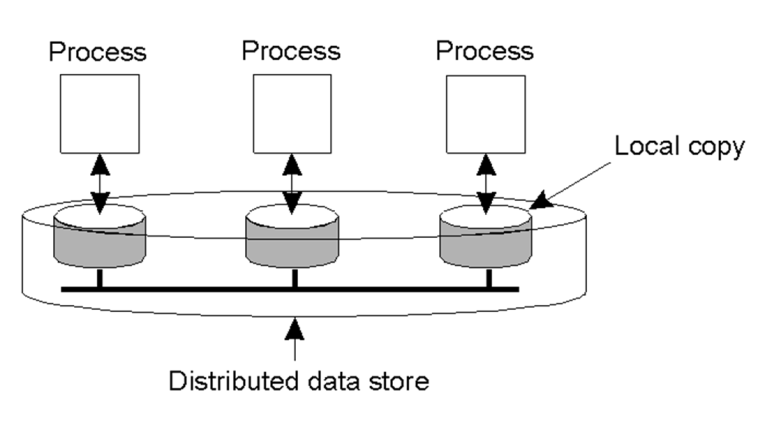
对于三种算法来说，从进程想进入一个临界区到它真正进入临界区这段延迟时间是不同的：

（1）当使用临界区的时间很短且很少使用时，延迟的主要因素是如何进入临界的实际机制

（2）当临界区使用时间较长且经常使用时，延迟的主要因素时等待其他进程使用临界区的时间

## 7、复制与一致性

**数据存储的特点：**数据存储可以物理地分布在多台机器上。特别是，假设每个可以访问数据存储中的数据的进程都有整个数据存储的一个本地或邻近的拷贝，写操作将传播到其他拷贝。

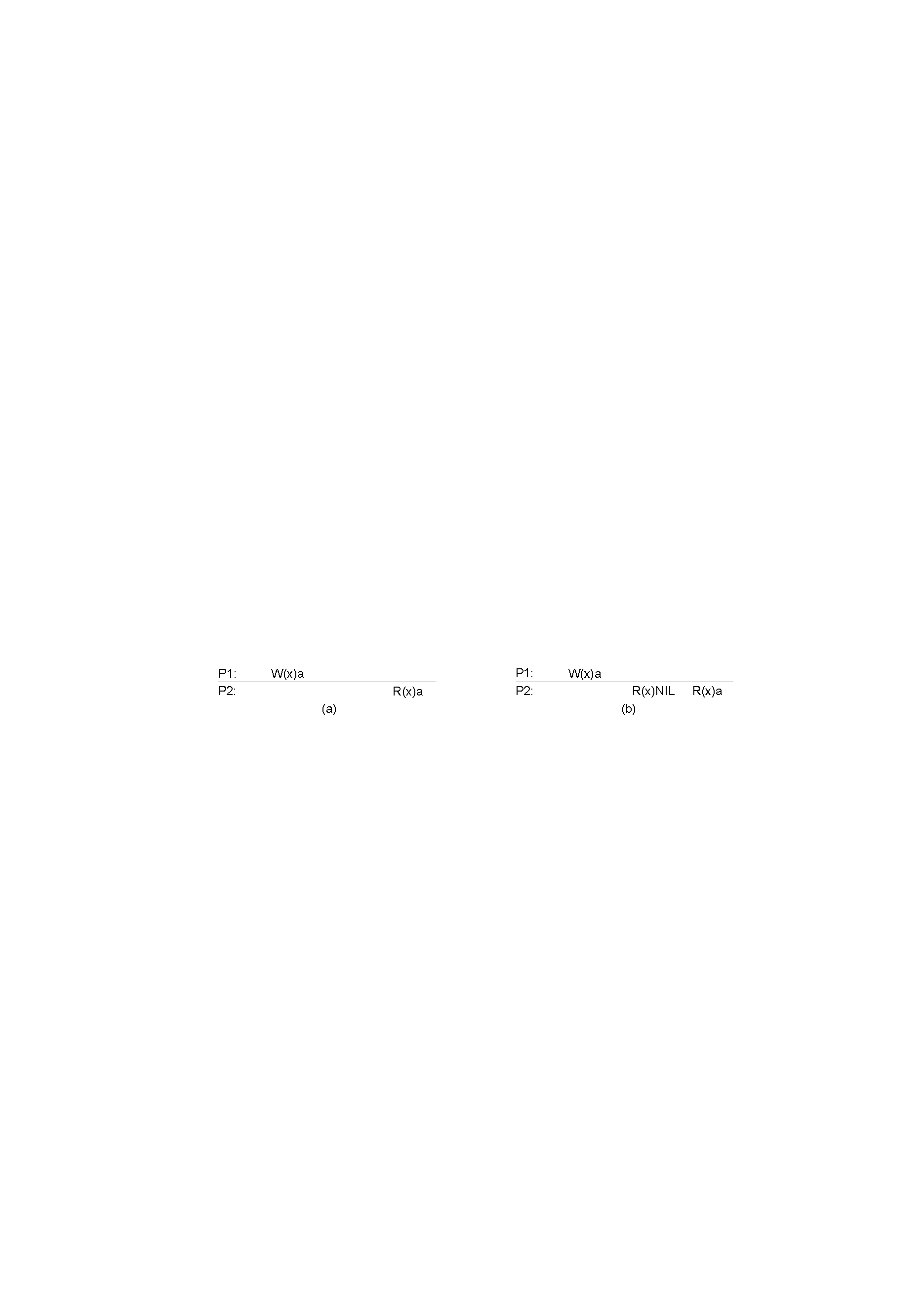


### 7.1以数据为中心的一致性

**1、严格一致性：**

**条件：**对于数据项x的任何读操作将返回最近一次对x进行的写操作的结果所对应的值。

**实现方法：**严格一致性中存在的问题是它依赖于绝对的全局时间（每个与准确的全局时间对应的操作分配唯一的时间戳本质上不可能），可以放宽条件，将时间分割成一系列连续的、不重叠的时间间隔。



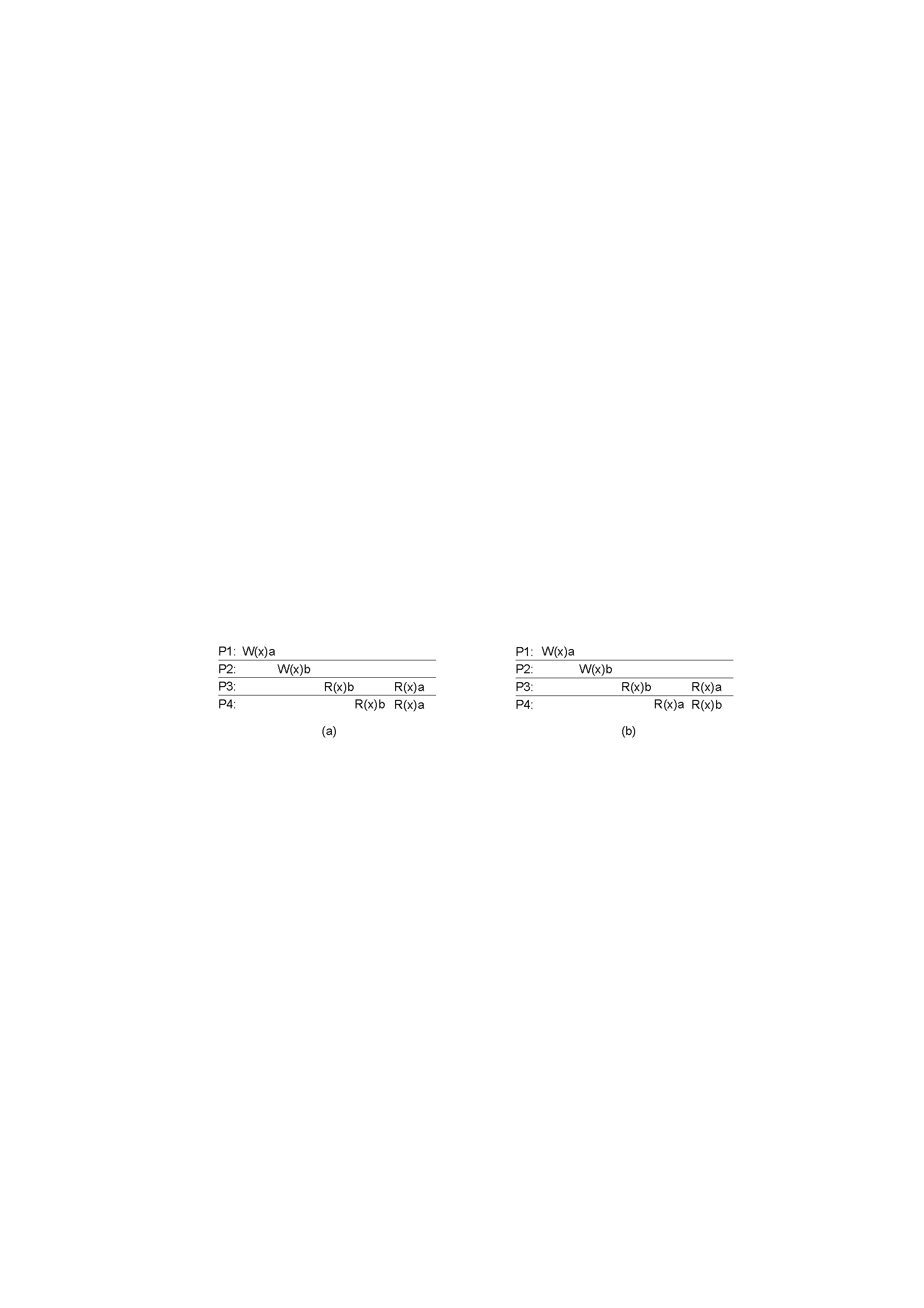
**总结：**当数据存储是严格一致的时候，对于所有的进程来说，所有写操作是瞬间可见的，系统维持着一个绝对的全局时间顺序。尽管严格一致性是理想的一致性模型，但是在分布式系统中，它无法实现。

**2、线性化与顺序一致性：**

**条件：**任何执行结果都是相同的，就好像所有进程对数据存储的读、写操作时按照某种序列顺序执行的，并且每个进程的操作按照程序所制定的顺序出现在这个序列中。

**实现方式：**

（1）线性化一致性（强于顺序一致性，弱于严格一致性）：假设操作具有一个全局有效时钟的时间戳，但是这个时钟仅具有有限的精确度，根据一系列同步时钟确定序列顺序。线性化的实现比顺序一致性的实现开销更大

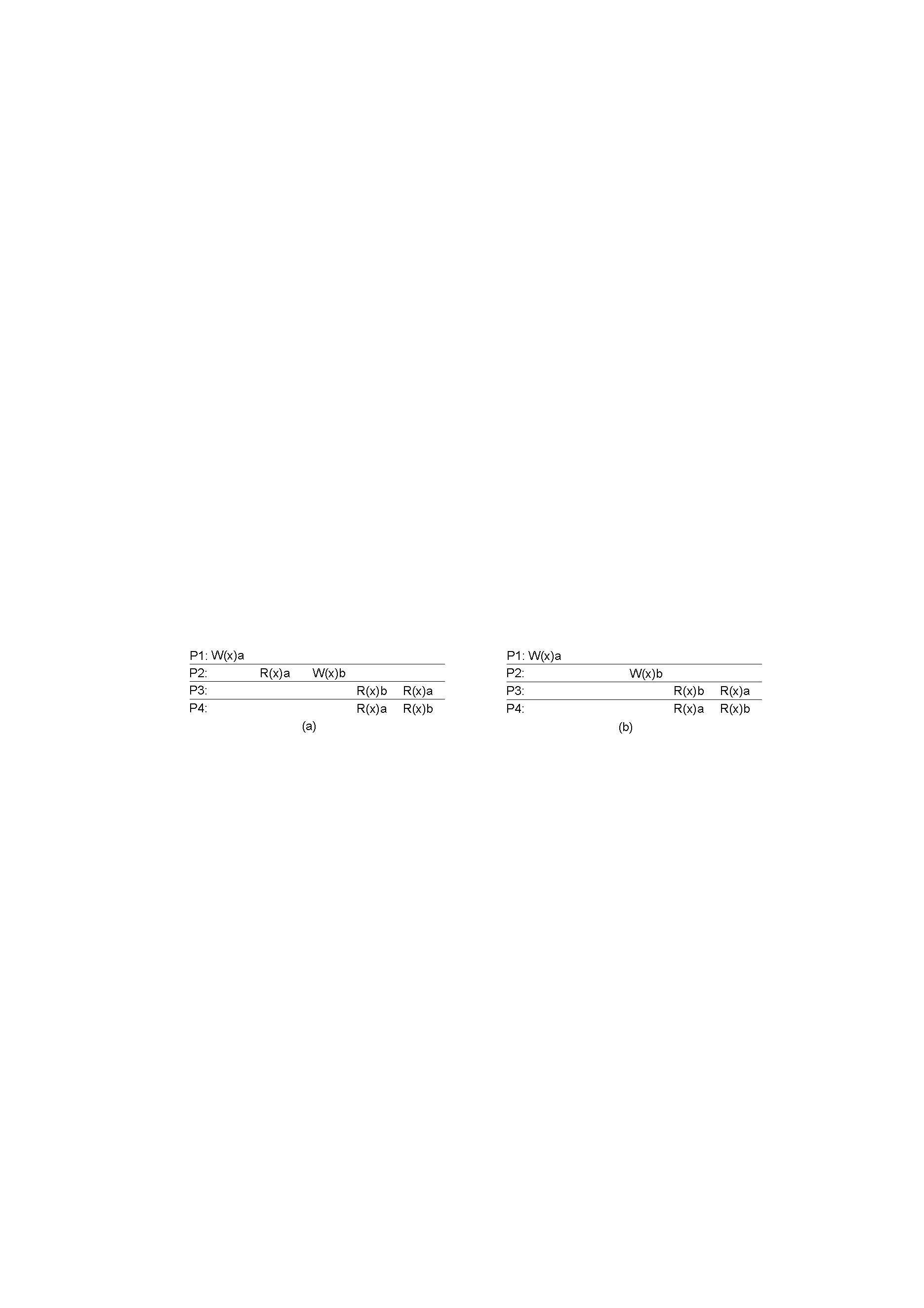
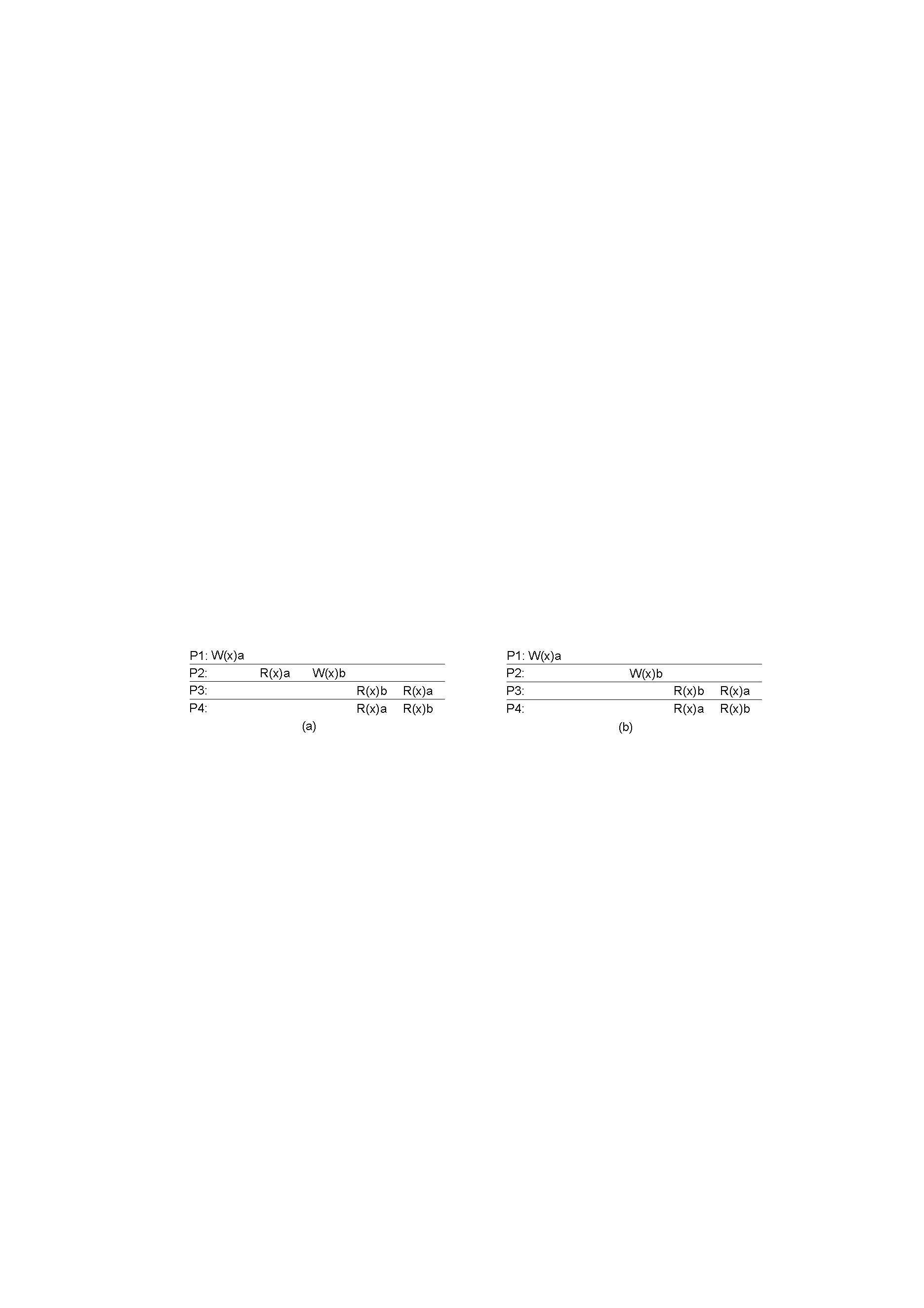


**总结：**当进程在多台(可能)不同机器上并发运行时，任何读、写操作的有效交叉都是可接受的行为，但是所有进程都看到相同的操作交叉。存在严重的性能问题，对于任何的顺序一致性存储，改变协议以提高读操作性能必将降低写操作性能，反之亦然。

**3、因果一致性：**

**条件：**所有进程必须以相同的顺序看到具有潜在因果关系的写操作。不同机器上的进程可以以不同的顺序被看到并发的写操作

**实现方式：**实现因果一致性要求跟踪哪些进程看到了哪些写操作。这意味着必须构建和维护一张记录哪些操作依赖于哪些操作的依赖关系图。一种实现方法是向量时间戳。

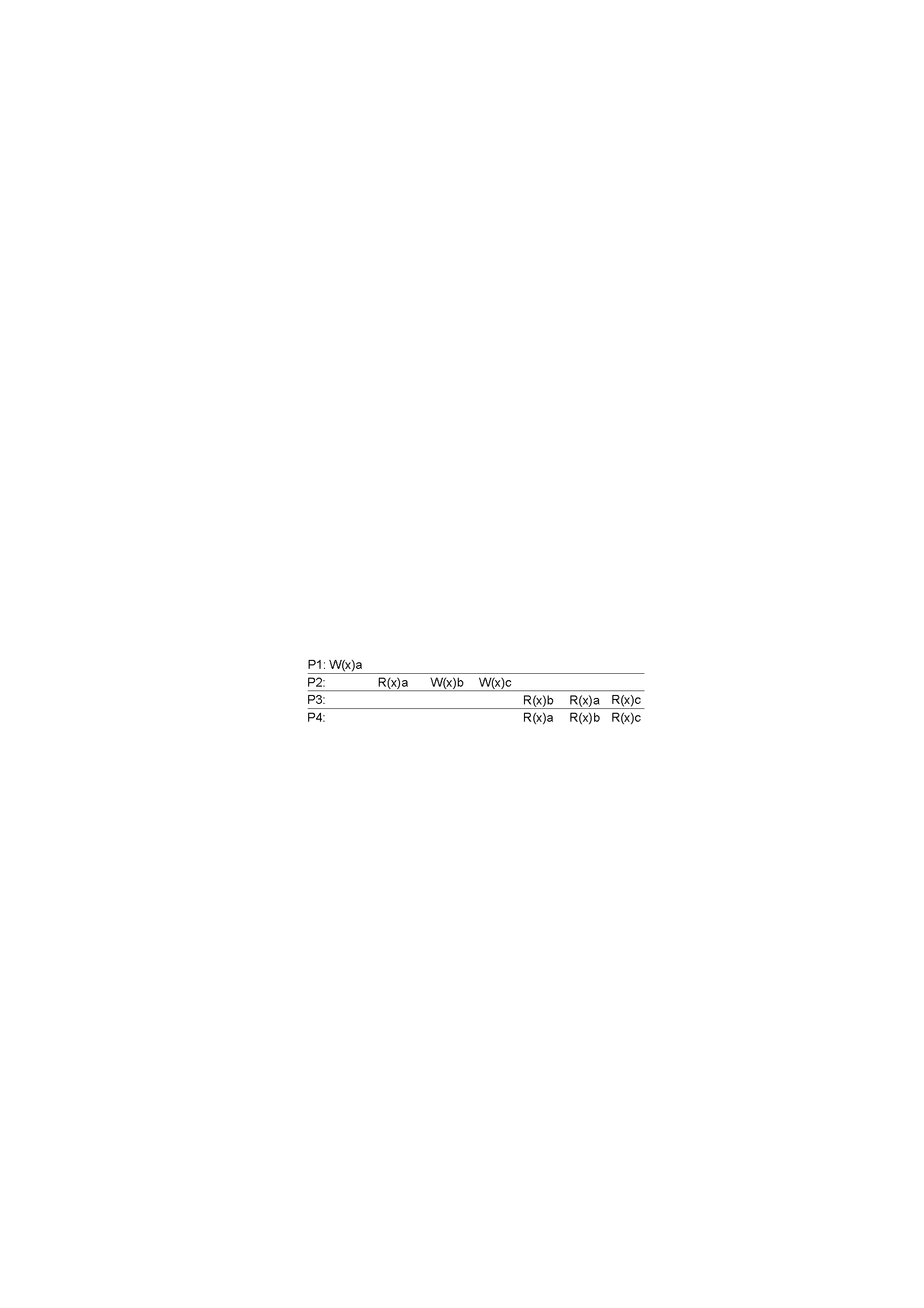
 

**总结：**当一个读操作后面跟着一个写操作时，这两个事件就具有潜在的因果关系。同样，读操作也与为读操作提供数据的写操作因果相关。没有因果关系的操作被称为并发的。因果一致的存储并不要求并发的写操作是全局有序的

**4、FIFO一致性：**

**条件：**所有进程以某个单一进程提出写操作的顺序看到这些写操作，但是不同进程可以以不同的顺序看到不同的进程提出的写操作。

**实现方式：**简单地使用(进程，序列号)对作为每个操作的标签，并根据序列号执行每个进程的写操作



**总结：**实际上，这种模型不需要保证不同进程看到写操作的顺序，除非两个以上的写操作是同一个进程提出的。这种情况下，写操作必须按顺序达到。

**与顺序一致性的区别：**

（1）对于顺序一致性而言，尽管语句的执行顺序是非确定的，但是至少所有的进程都对某个顺序达成一致

（2）对于FIFO一致性而言，各个进程不需要达成一致，不同进程可以以不同的顺序看到操作

**5、采用同步变量的各类一致性：**

**弱一致性：**

**弱一致性的条件：**

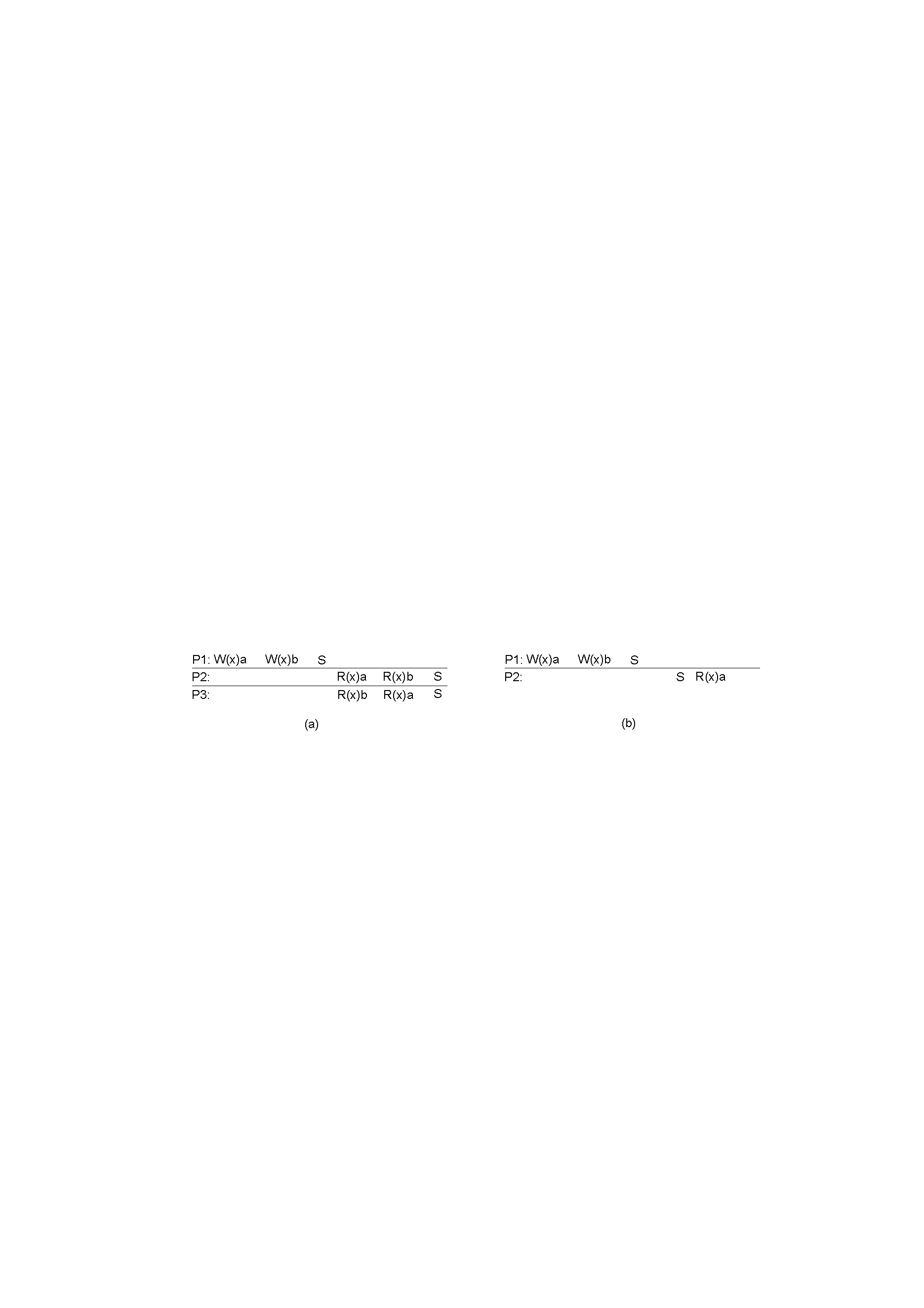
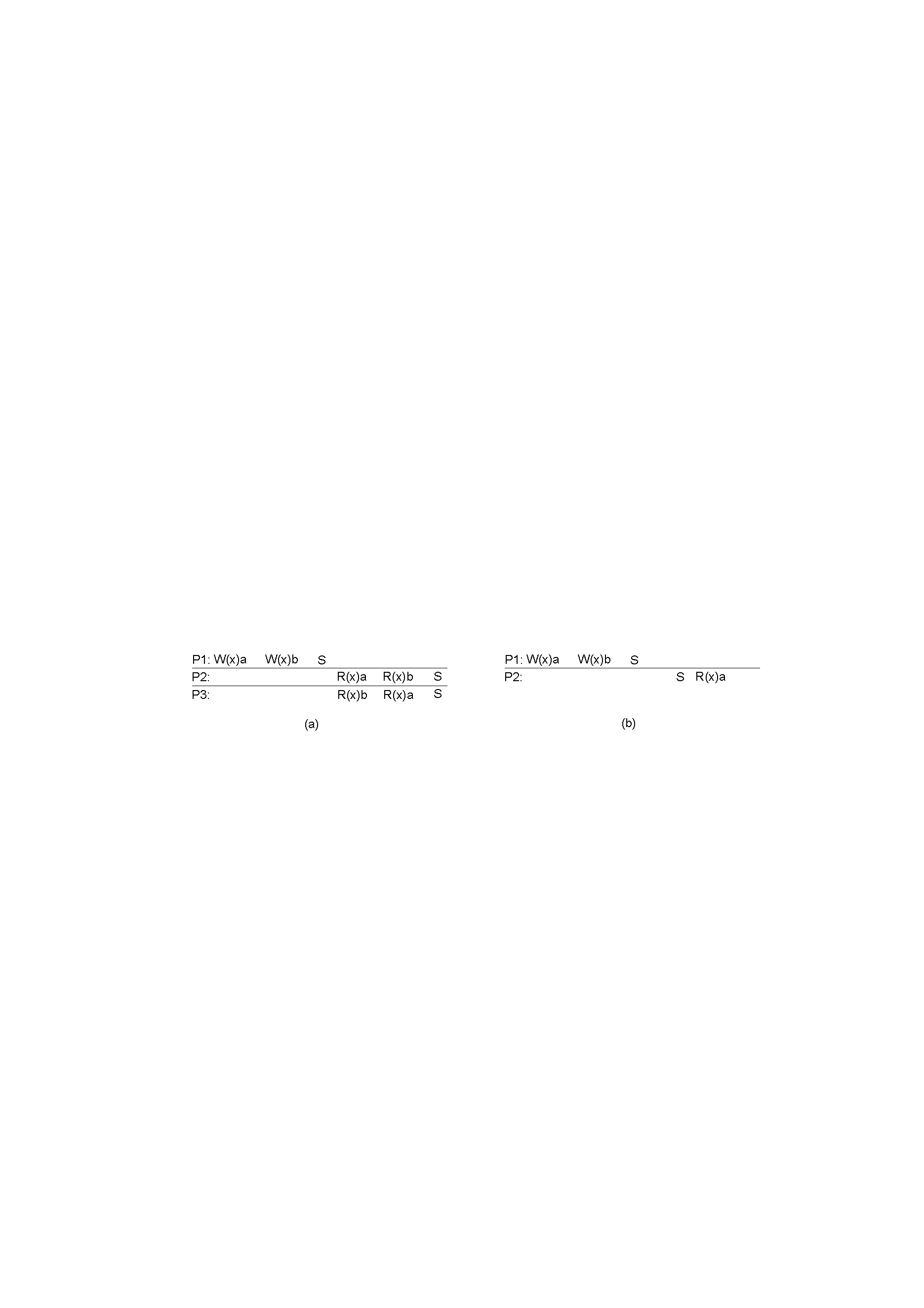
（1）对数据存储所关联的同步变量的访问是顺序一致的；

（2）每个拷贝完成所有先前执行的写操作之前，不允许对同步变量进行任何操作；

（3）所有先前对同步变量执行的操作都执行完毕之前，不允许对数据项进行任何读或者写操作

与前面的一致性模型不同，弱一致性不是使单个的读操作和写操作服从一致性，而是迫使一组操作服从一致性。如果对于共享数据来说，很少有访问是孤立的，大部分访问是群集的(即一段短时期内有很多访问，然后很长一段时间内没有访问)，那么这种情况下，弱一致性模型非常有用。

另一个重要不同点是，它只限制了保持一致性的时间，而没有限制一致性的形式。实际上，可以说，弱一致性是在一组操作，而非单个操作上强迫执行顺序一致性。同步变量用于划分操作的组

**问题：**即当同步变量被访问时，数据存储不知道此次访问是因为进程已经结束对数据存储的写操作还是因为进程将开始读数据而进行的

**释放一致性：**

**释放一致性的条件：**

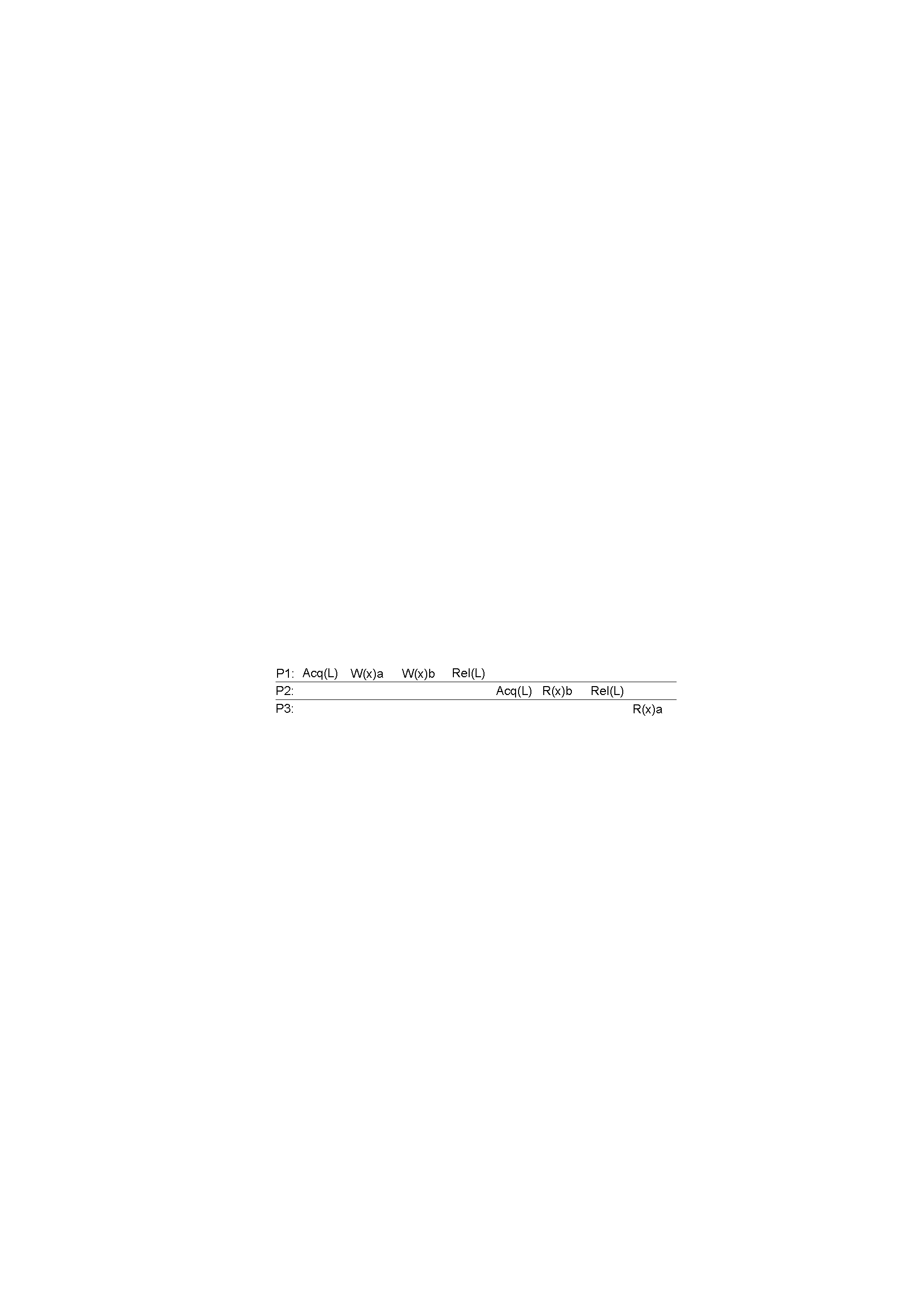
（1）对共享数据执行读操作或写操作之前，所有进程先前执行的获取操作都必须已经成功完成；

（2）在释放操作被允许执行前，所有进程先前执行的读操作和写操作都必须已经完成；

（3）对同步变量的访问是FIFO一致的(不需要顺序一致)。

使用两种类型的同步变量来代替原先弱一致性的一种类型的同步变量。获取(acquire) 操作是用于通知数据存储进程进入临界区的操作，而释放(release)操作是表明进程刚刚离开临界区的操作

释放一致性也可以使用障碍（一种同步机制）代替临界区。当一个进程到达一个障碍，它必须一直等待，直到其他所有进程也都到达这个障碍。当最后一个进程到达这个障碍时，说明所有的共享数据都已经同步，然后所有进程才可以继续执行。获取操作表明进程离开障碍，而释放操作表明进程到达障碍。



**懒惰释放一致性：**在懒惰释放一致性中，在执行释放操作时，不向任何地方发送任何数据。相反，执行获取操作时，试图执行获取操作的进程必须从持有数据的进程那里获得数据的最新值。可以使用一个时间戳协议来确定哪些数据项是实际必须被传输的。

**入口一致性：**

**入口一致性的条件：**

（1）在一个进程可以获取一个同步变量之前，所有的由此同步变量保护的共享数据的更新都必须已经相对于该进程执行完毕

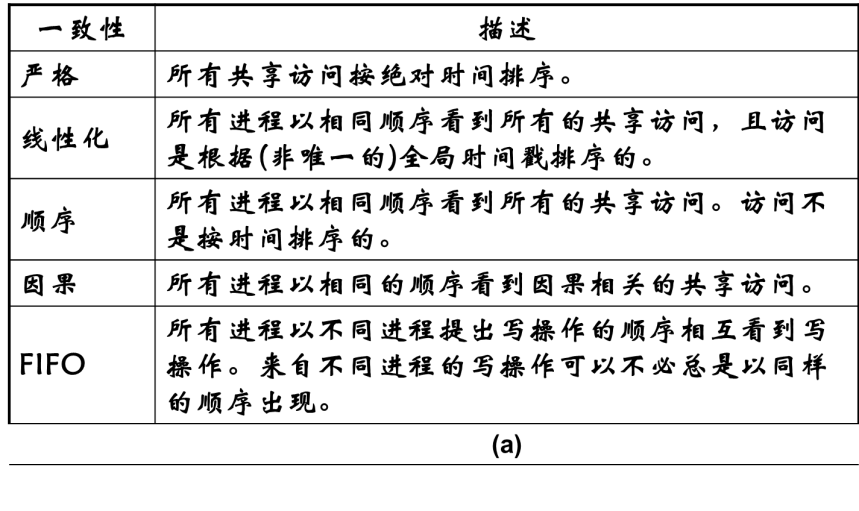
（2）在一个进程的一个同步变量的独占访问被允许执行之前，其他的进程不可以拥有这个同步变量，甚至也不能以非独占的方式拥有这个同步变量

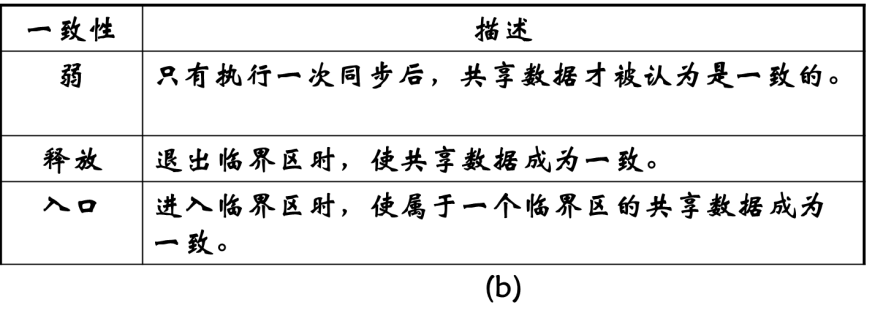
（3）一个进程对一个同步变量执行独占访问之后，在对该同步变量的所有者进行检查之前，任何其他的进程都不能执行下一个非独占访问

释放一致性不同的是，入口一致性要求每个普通的共享数据项都要与某种同步变量(如锁或障碍)关联。如果需要并行地、独立地访问数组的多个元素，只有该同步变量保护的那些数据是保持一致的。

每个同步变量都有一个当前的所有者，即最后获取它的进程。所有者不必在网络上发送任何信息，就可以重复地进入、退出临界区。而一个当前不拥有一个同步变量的进程想要获取这个同步变量，它必须向当前的所有者发送消息，请求获得这个同步变量的所有权及其关联数据的当前值。多个进程也可以以非独占的方式同时拥有一个同步变量，此时这些进程只能读，而不能写这个同步变量关联的数据。

**6、一致性比较：**





### 7.1以客户为中心的一致性

**1、以客户为中心的一致性特点：**不会出现同时发生的更新操作，或者当出现同时发生的更新操作时，可以容易地化解它们。大部分操作是读取数据的操作。

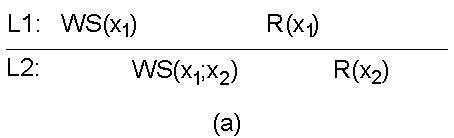
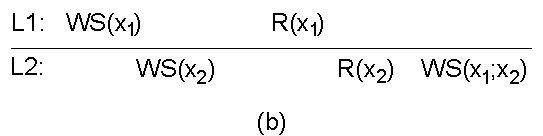
**2、最终一致性**进程实际以并发方式操作的程度与所需保证的一致性的程度可能不同。并发操作只以有限的形式出现的实例很多。那么，现在的问题是应该以多快的速度进行更新操作，才能使更新对只读进程有效。

**最终一致性条件：**没有更新操作时，所有副本逐渐成为相互完全相同的拷贝。（开销较小）

**问题：**只要客户总是访问同一副本，最终一致性的数据存储就会工作得很好。但是，当客户访问不同的副本时，看到的可能是好像根本没有做过任何改变一样。

**3、单调读：**

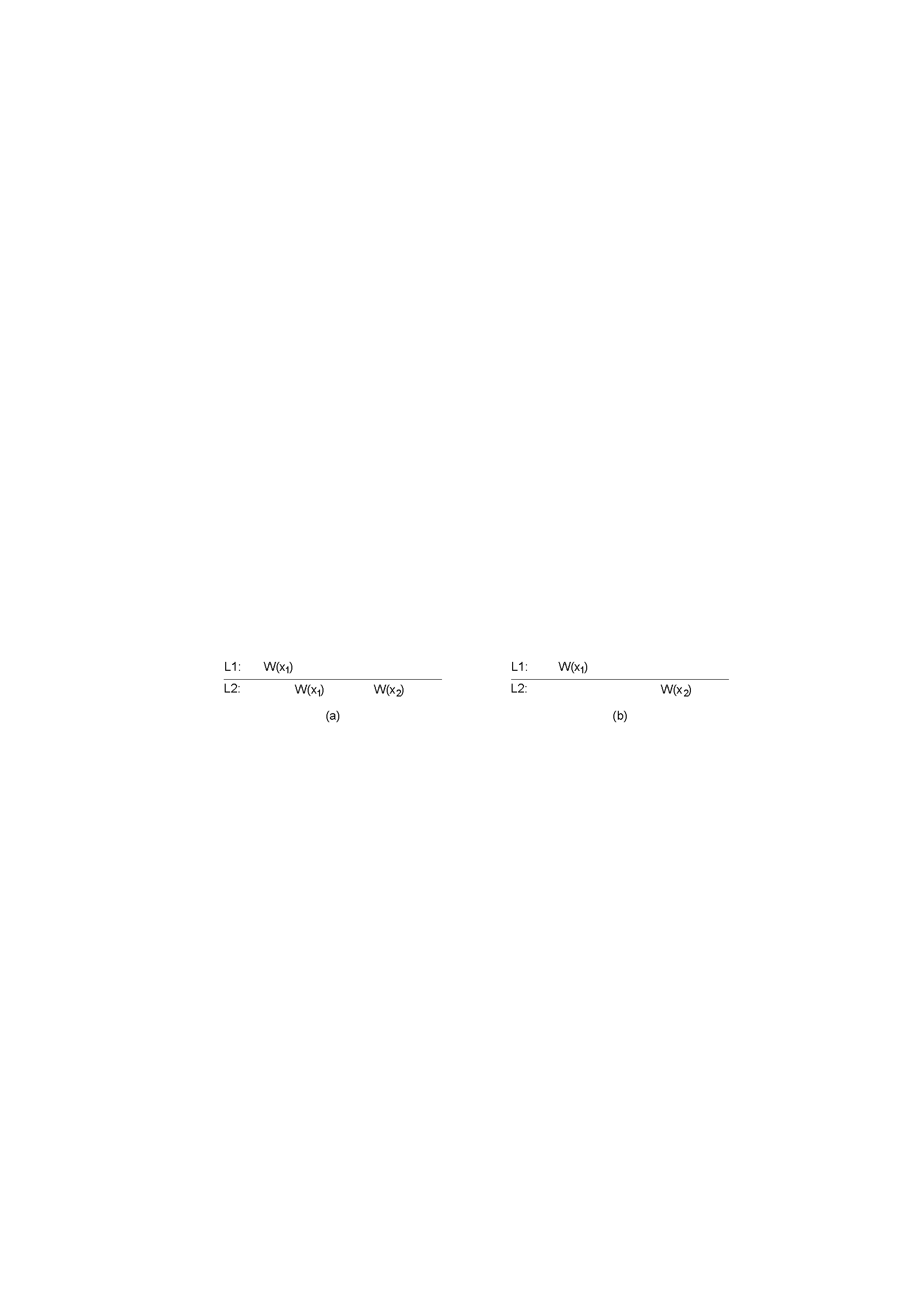
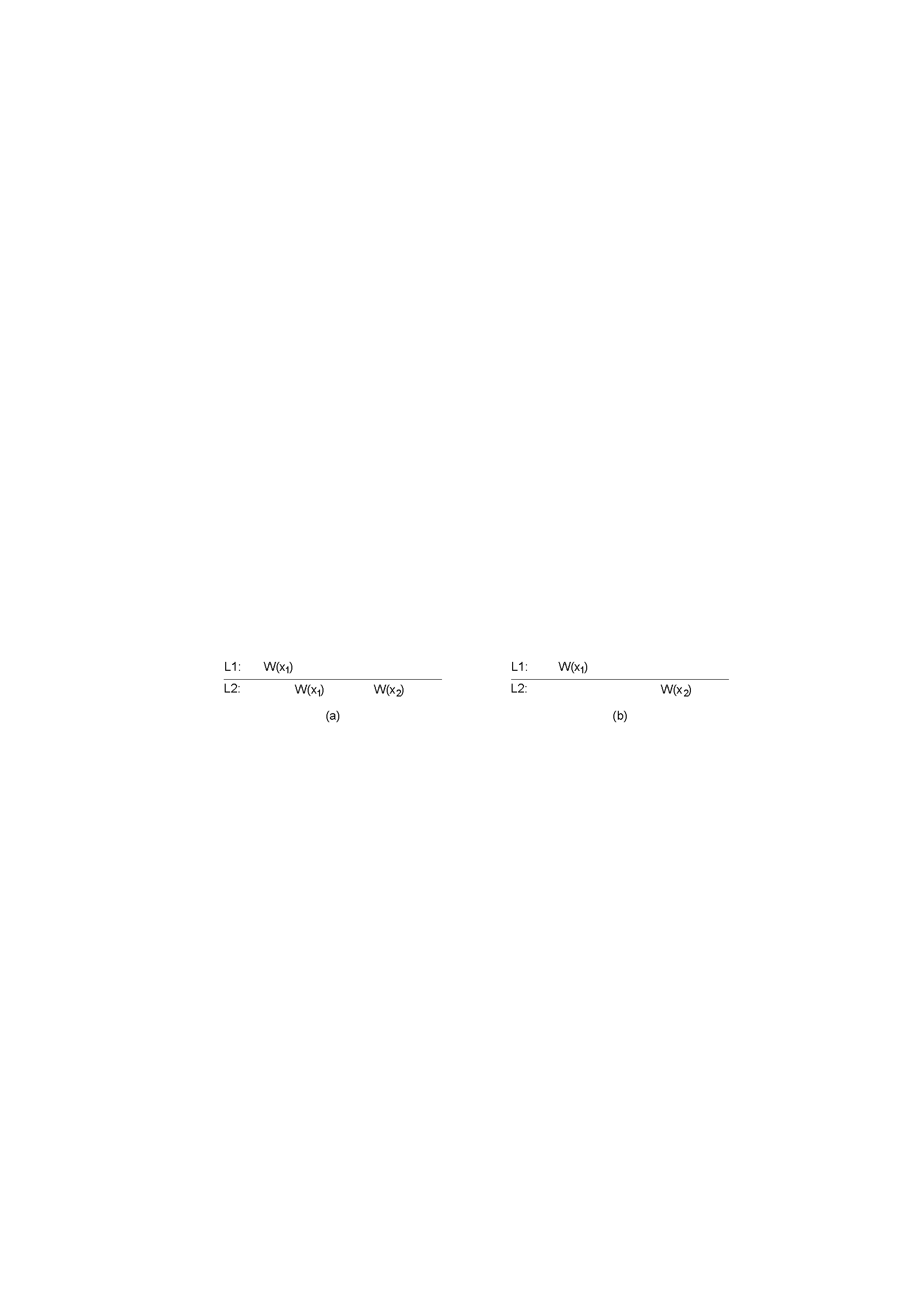
**单调读的条件：**如果一个进程读取数据x的值，那么该进程对执行任何后续读操作将总是得到第一次读取的那个值或更新的值。

**4、单调写：**

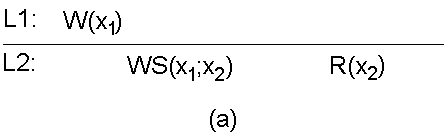
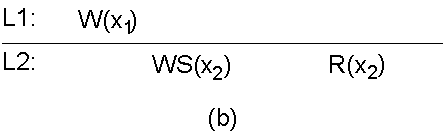
**单调写的条件：**一个进程对数据项x执行的写操作必须在该进程对x执行任何后续写操作之前完成。

**单调写的特点：**单调写一致性与数据为中心的FIFO一致性相似。FIFO一致性的本质是，同一进程执行的写操作必须在任何地方以正确的顺序执行。这一顺序限制也适用于单调写一致性，只是我们这里考虑的是仅为单一进程维持的一致性，而不是为许多并发进程维持的一致性。

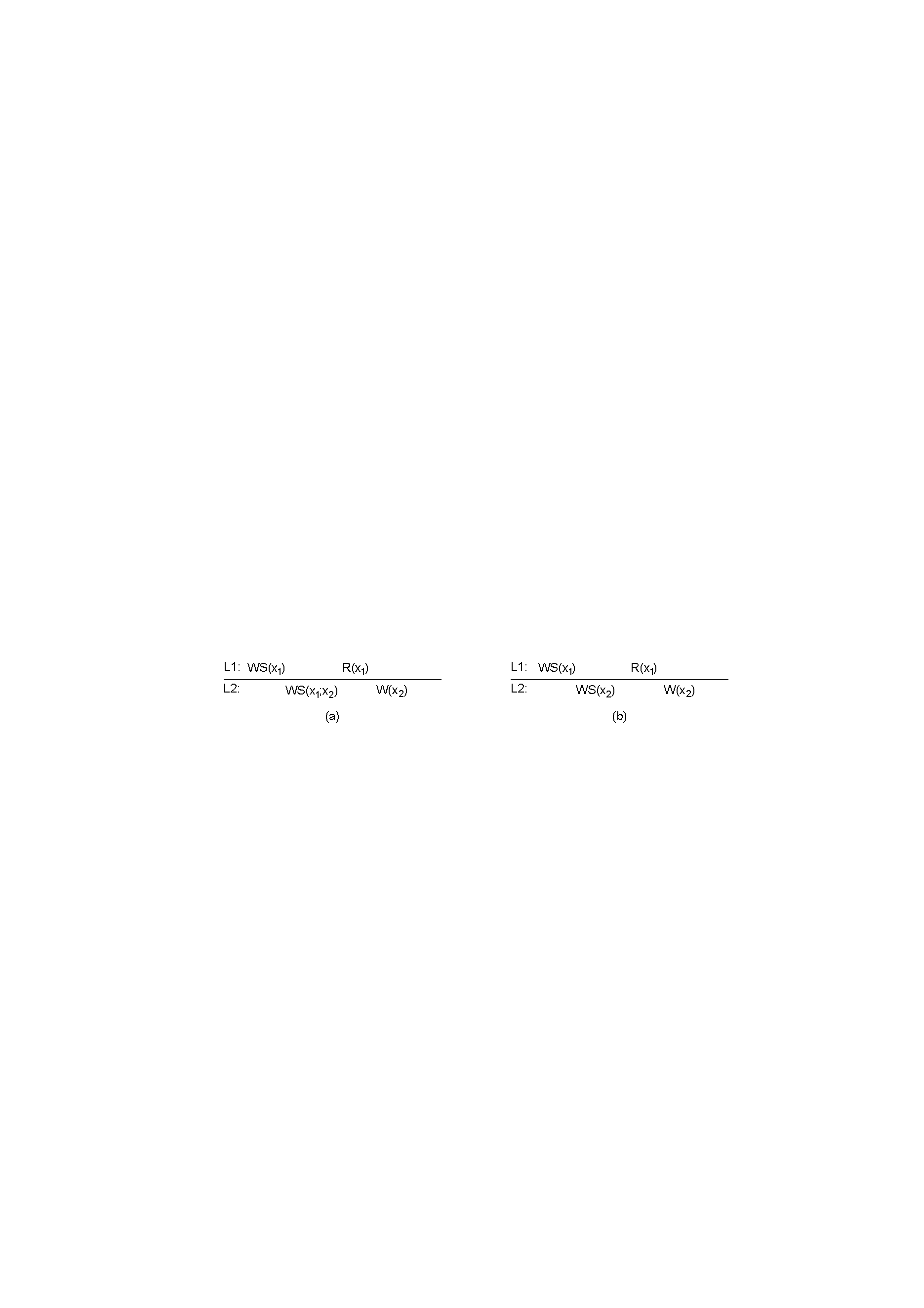
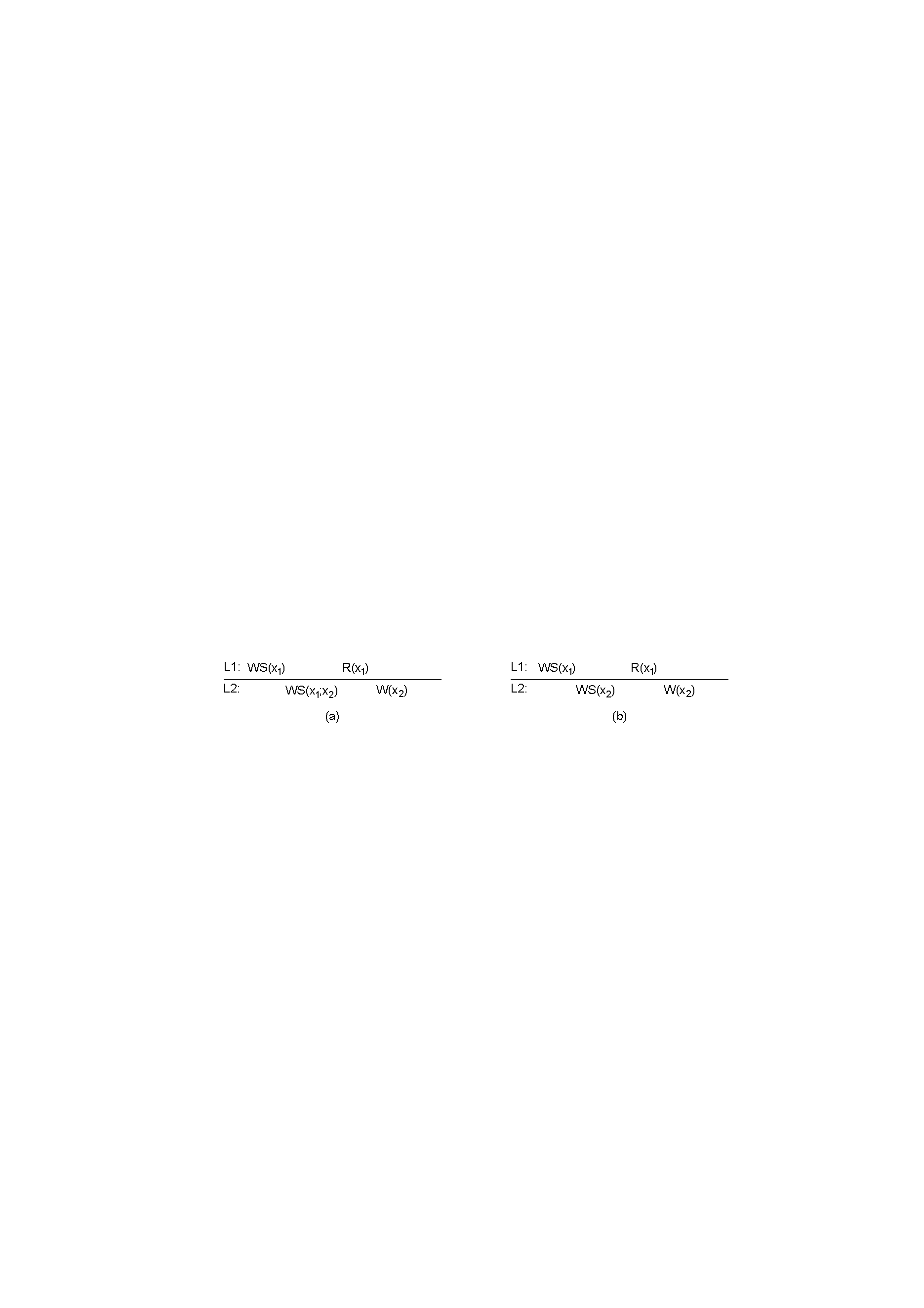
**5、写后读：**

**写后读的条件：**一个进程对数据项x执行一次写操作的结果总是会被该进程对x执行的后续读操作看见。

1. **读后写：**

**读后写条件：**同一个进程对数据项x执行的读操作之后的写操作，保证发生在与x读取值相同或比之更新的值上。

**** ****

**7、简化实现：**在以客户为中心的一致性的一种简化实现中，每个写操作都被分配一个全局唯一的标识符。该标识符是第一次接受操作的服务器为其分配的。我们也称该操作是在那台服务器上启动的。

这样，对于每个客户，我们跟踪写标识符的两个集合。一个是客户的读集合，它由与客户所执行的读操作相关的写标识符组成；一个是客户的写集合，它由客户所执行的写操作的标识符组成。

**单调读一致性实现：**一个客户在一台服务器上执行一个读操作时，这台服务器获得客户的读集合，检查所有标识的写操作是否已经在本地执行。如果没有执行，那么它就联系其他服务器以确保执行读操作之前将本地拷贝更新。

另一种方法是将读操作转发到已经执行了这些写操作的服务器那里执行。读操作执行后，在所选择的服务器上执行的写操作以及与读操作的写操作会被加入客户的读集合。

**单调写一致性实现：**每当一个客户在服务器上启动一个新的写操作，客户的写集合都会转交到这台服务器(同样，面对性能需求，这个写集合的大小可能过大。下面讨论另一个可选择的解决方案)。然后，这台服务器确保先执行被标识的写操作，并按正确的顺序执行这些写操作。执行新操作之后，该操作的写标识符将被加入写集合。

**8、效率提升：**很容易看出，与每个客户关联的读集合和写集合可能变得非常大。为了维持这些集合的可控制性，客户的读操作和写操作根据会话分成组。通常，一个会话与一个应用程序关联。

## 8、事务提交

**1、单阶段提交：**协调者通知所有涉及到的称为参与者的其他进程是否(本地)按要求执行操作。这种方法称为单阶段提交协议。

**缺陷：**如果有一个参与者不能真正执行该操作，那么就说明它存在着明显的缺陷，因为它没有办法来通知协调者

**2、两阶段提交：**考虑一个分布式事务中有很多进程作为参与者，每个进程都运行在不同的机器上。假定没有故障发生，协议就由以下两个阶段组成，每个阶段又由两步组成。

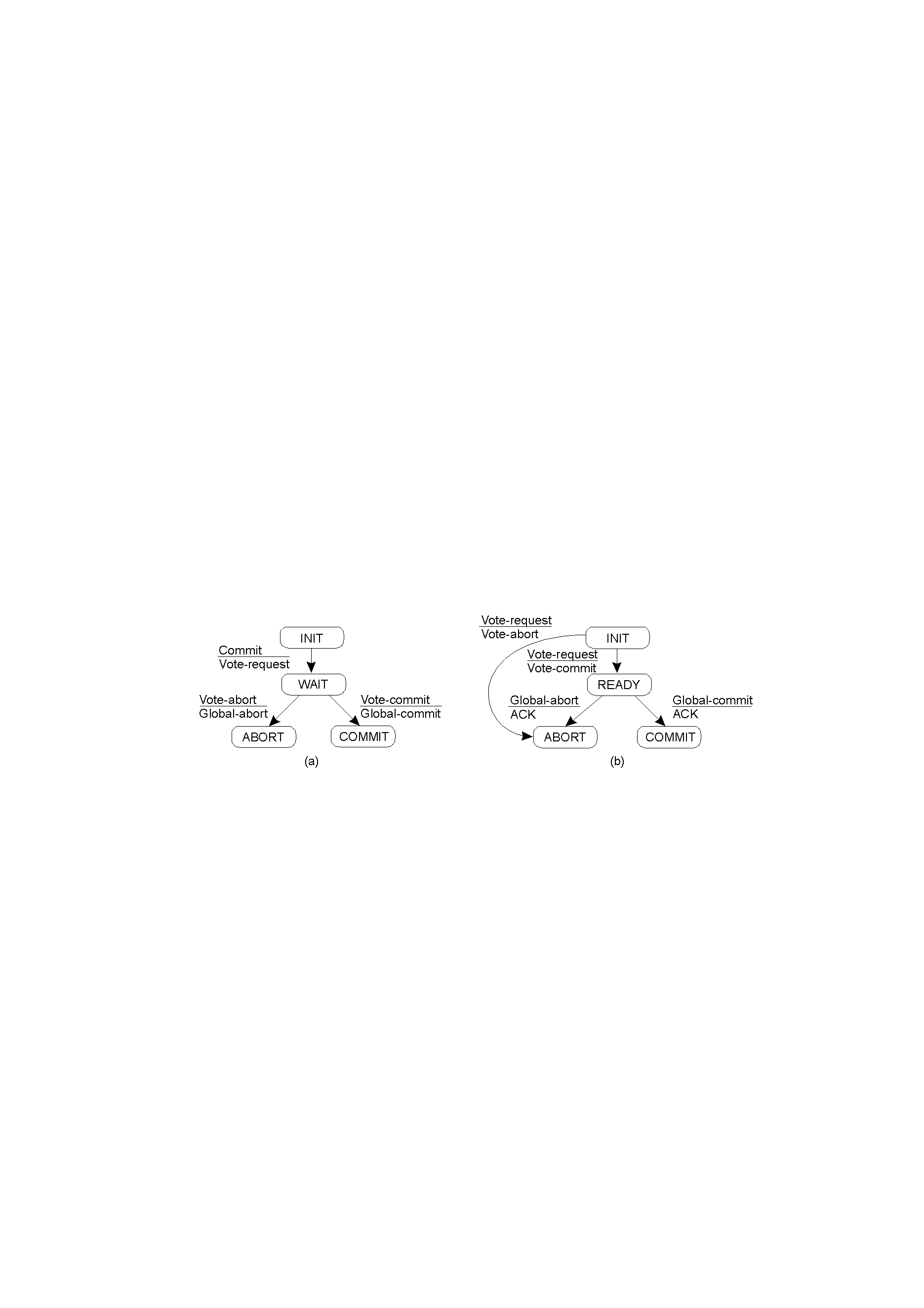
**步骤：**

（1）协调者向所有的参与者发送一个Vote\_Request消息

（2）当参与者接收到Vote\_Request消息时，就向协调者返回一个Vote\_Commit消息通知协调者它已经准备好本地提交事务中属于它的部分，否则就返回一个Vote\_Abort消息

（3）协调者收集来自参与者的所有选票。如果所有的参与者都表决要提交事务，那么协调者就进行提交。在这种情况下它向所有的参与者发送一个Global\_Commit消息。但是，如果有一个参与者表决要取消事务，那么协调者就决定取消事务并多播一个Global\_Abort消息

（4）每个提交表决的参与者都等待协调者的最后反应。如果参与者接收到一个Global\_Commit消息，那么它就在本地提交事务，否则接收到一个Global\_Abort消息时，就在本地取消事务



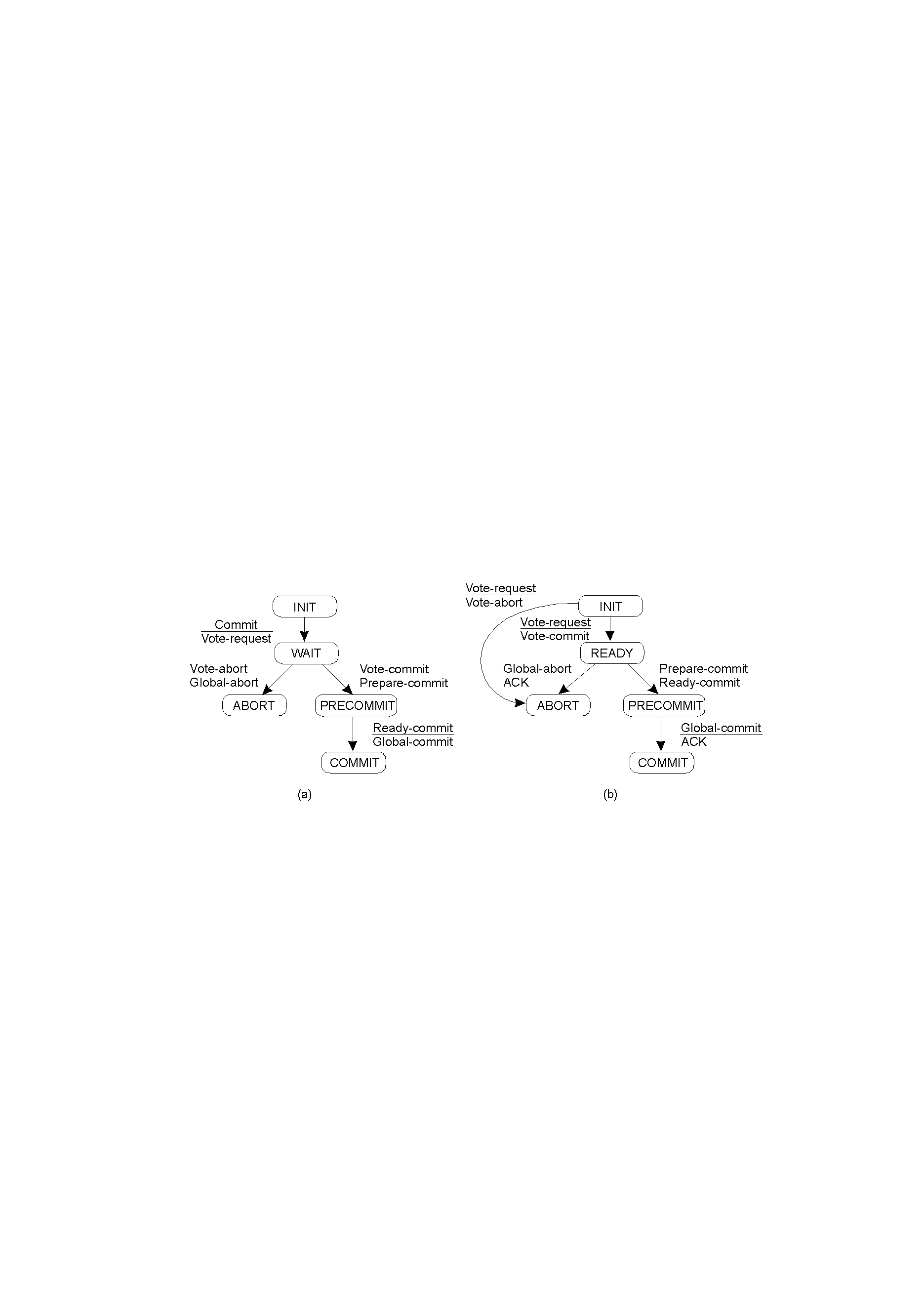
**问题：**协调者崩溃时，参与者不能做出最后的决定。因此参与者可能在协调者恢复之前保持阻塞。

**3、三阶段提交：**

**在两阶断提交基础上满足的条件：**

（1）没有一个可以直接转换到Commit或者Abort状态的单独状态。

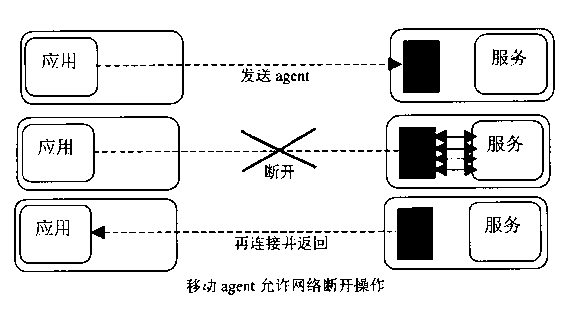
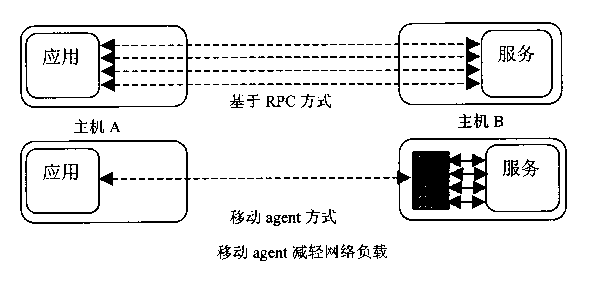
（2）没有一个这样的状态：它不能做出最后决定，而且可以从它直接转换到Commit状态



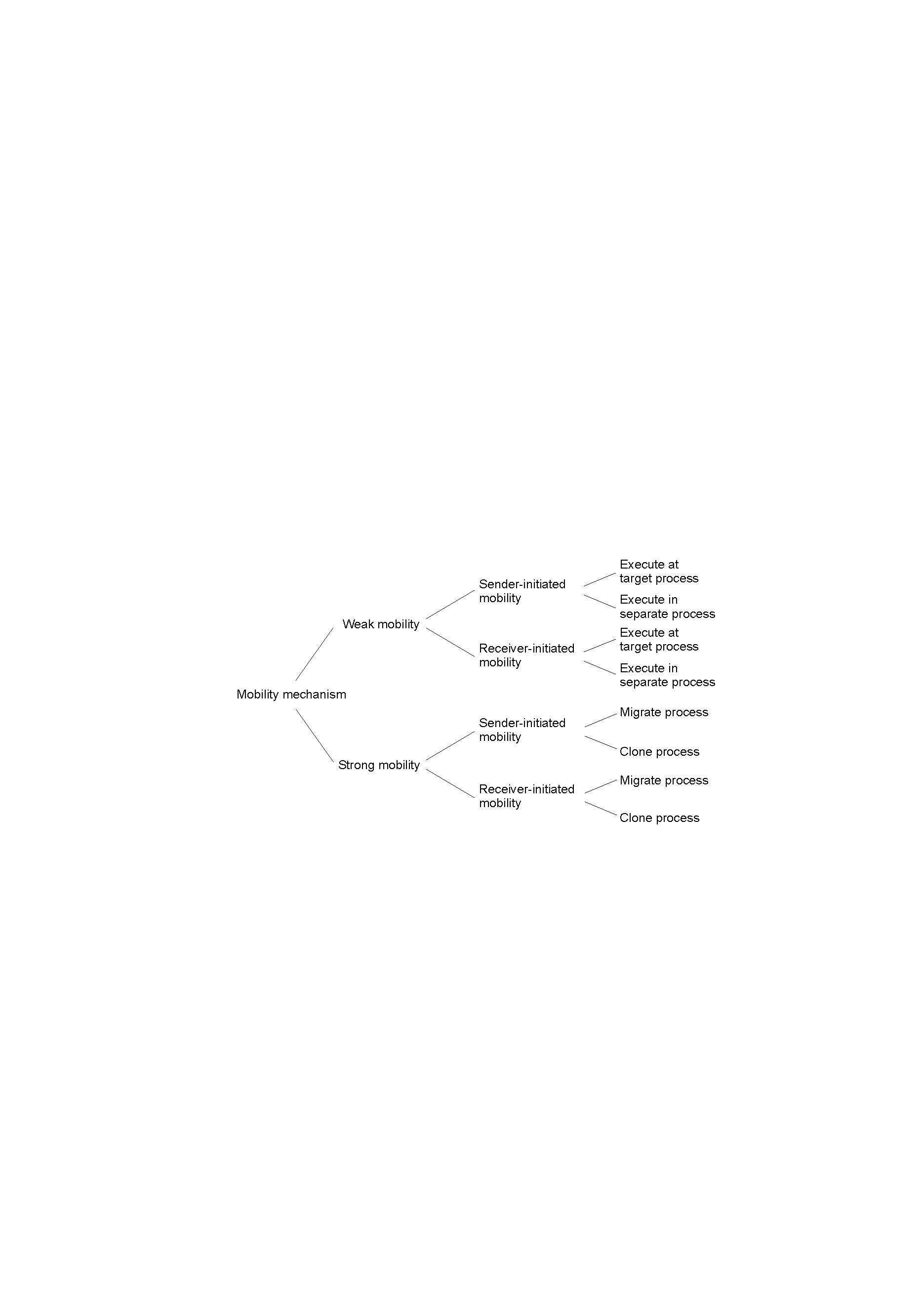
**与2PC的区别：**崩溃的参与者可能恢复到了Commit状态而所有参与者还处于Ready状态。在这种情况下，其余的可能操作进程不能做出最后的决定，不得不在崩溃的进程恢复之前阻塞。在3PC中，只要有可操作的进程处于Ready状态，就没有崩溃的进程可以恢复到Init、Abort或Precommit之外的状态。因此存活进程总是可以做出的最后决定。

## 9、代码迁移技术

**1、迁移的理由：**分布式系统中的代码迁移是以进程迁移(process migration)的形式进行的，在这种形式下整个进程被从一台机器搬到另一台机器上去。基本的思想是：如果把进程由负载较重的机器上转移到负载较轻的机器上去，就可以提升系统的整体性能。



**2、迁移的分类：**



**3、代码迁移的框架结构：**

（1）代码段部分包含构成正在运行的程序的所有指令。

（2）资源段包含指向进程需要的外部资源的指针，这些外部资源包括文件、打印机、设备、其他进程等。

（3）执行段(execution segment)，它用来存储进程的当前执行状态量，这些状态量包括私有数据、栈和程序计数器等。

**4、若可迁移性&强可迁移性：**

**（1）弱可迁移性：**在这种模型中，可以只传输代码段以及某些初始化数据。弱可移动性的典型特征是，传输过来的程序总是以初始状态重新开始执行的（如Java Applet）。这种方法的好处在于其简单性。弱可迁移性只要求目标机器上能执行代码，这在本质上可以归结为使用代码可移植。

**（2）强可迁移性**（支持传输运行段）：强可移动性的典型特征是，可以先停止运行中的进程，然后将它搬到另一台机器上去，再从刚才中断的位置继续执行。很明显，强可移动性要比弱可移动性强大得多，但是也更加难以实现。

**5、接收者启动的迁移&发送者启动的迁移：**

**（1）发送者启动的迁移：**在发送者启动的迁移中，代码当前驻留在哪台机器上或者正在哪台机器上执行，就由该机器来启动迁移（如计算服务器上载程序）。发送者启动的迁移要求安全地向服务器上载代码，常常要求客户端预先在服务器上注册并通过验证（即要求服务器了解它的所有客户），这是因为客户很可能会访问服务器的资源，比如磁盘，而对这些资源的保护极为重要。

**（2）接收者启动的迁移：**代码迁移的主动权掌握在目标机器手中（如Java Applet）。接收者启动的迁移一般比发送者启动的迁移要更容易实现。在许多情况下，在客户和服务器之间进行的代码迁移的主动权由客户端掌握。

## 10、移动实体定位

**1、实体名称类型：**易于理解的名称、标识符、地址

**2、传统命名方法的局限性：**

考虑把ftp.cs.vu.nl转移到一台名为ftp.cs.unisa.edu.au的机器上，有两个传统方法：

（1）在cs.vu.nl的DNS数据库中记录新机器的地址：当再次把ftp.cs.vu.nl转移到另一台机器上时，必须同时更新它在cs.vu.nl所用DNS数据库中的项。重要的是这种更新不再是本地操作，完成这种操作实际上可能需要几百ms（违背了一条假设：对管理层节点进行的操作是高效的）

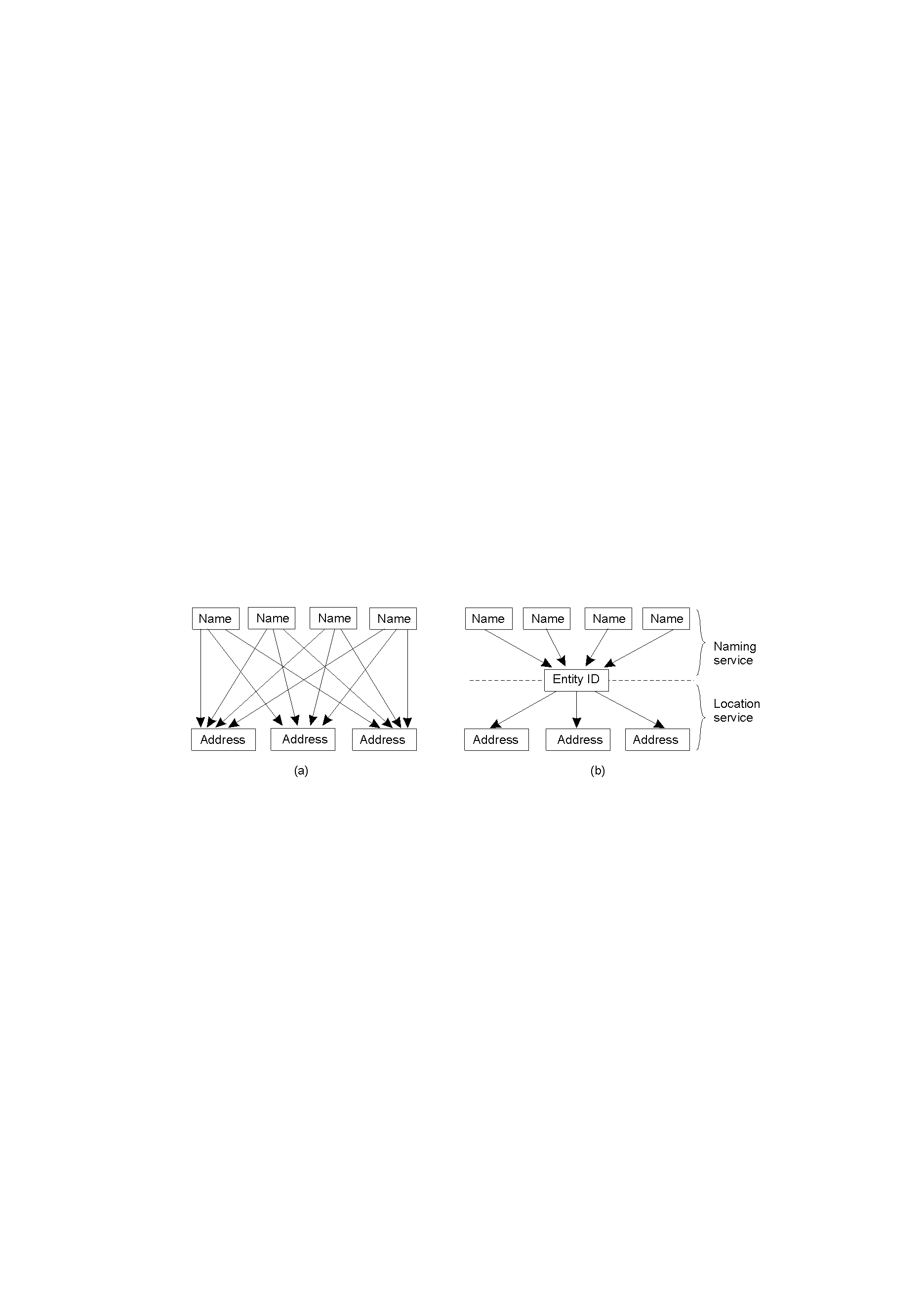
（2）记录新机器的名称，而不是它的地址，同时有效地把ftp.cs.vu.nl转化成一个符号链接：每次查询分两步（1、找到新机器的名称； 2、查找与该名称相关联的地址）。然而，如果ftp.cs.vu.nl再次被转移，必须在查询操作中添加一个步骤。

**传统方法：**（1）对于非常容易移动的实体来说，问题会变得更加糟糕。每当实体转移的时候，要么需要执行非本地更新操作，要么需要向查询操作添加另一个步骤。（2）还存在另一个严重的问题，那就是名称ftp.cs.vu.nl不允许改变。因此，为实体选择合适的名称是及其重要的，选择的名称在它所代表实体的生存周期内应该不发生改变。（3）特别是，许多实体以不同的名称为人所知，并且所有这些名称应该保持有效，更精确地说，就是应该始终指向同一个实体，即使是面对易移动实体也是如此。

**3、使用标识符的两级映射：**

一种更好的解决方法是通过引入标识符来把命名实体与查找实体分开。在通过命名服务查找实体时，命名服务会返回一个标识符。只要有必要就可以在本地存储该标识符，这是因为已经知道它永远不会指向其他实体，也永远不会改变。

实体的定位由独立的定位服务(location service)处理。定位服务器实际上是以标识符为输入，然后返回被标识实体的地址。如果地址存在多份副本，那么可能会返回多个地址。



**4、移动实体定位：广播与多播**

**方法简述：**包含该实体所用标识符的消息会广播到每台机器上，并且请求每台机器查看它是否拥有该实体。只有能够为实体提供访问点的机器才会发送回复消息，回复消息中包含访问点的地址。

**问题：**随着网络的膨胀，广播开始变得低效

**解决方案：**一种可能的解决方案是转换成多播，通过使用多播，只有符合条件的一组主机才会接收到请求；使用多播地址的另一种方式就是让它与一个复制的实体相关联，并且使用多播查找最近的复制实体。向该多播地址发送请求时，每个复制实体都会用它当前的(通常的)IP地址进行响应。选择最近的复制实体的一种粗糙方法就是选择最先回复的复制实体。

**5、移动实体定位：转发指针**

**方法简述：**当实体从A移动到B时，它将在后面留下一个指针，这个指针指向它在B中的新位置。这种方法的主要优点是它很简便；一旦找到实体以后(比如使用传统的命名服务)，客户就可以顺着转发指针形成的链查找实体的当前地址。

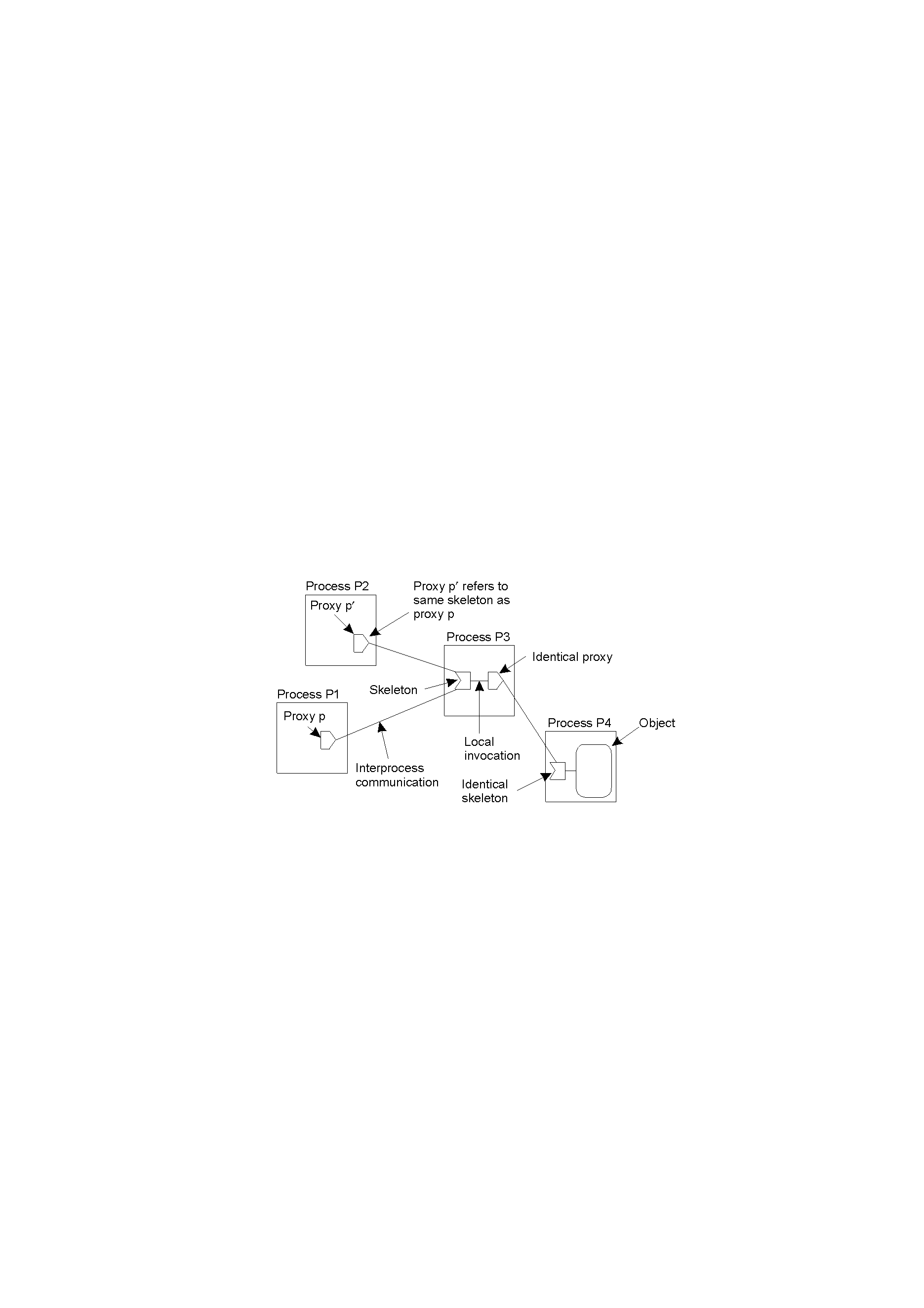
**缺点：**

（1）如果不采取特殊措施，那么链可能会特别长，以致定位实体的开销会变得很大

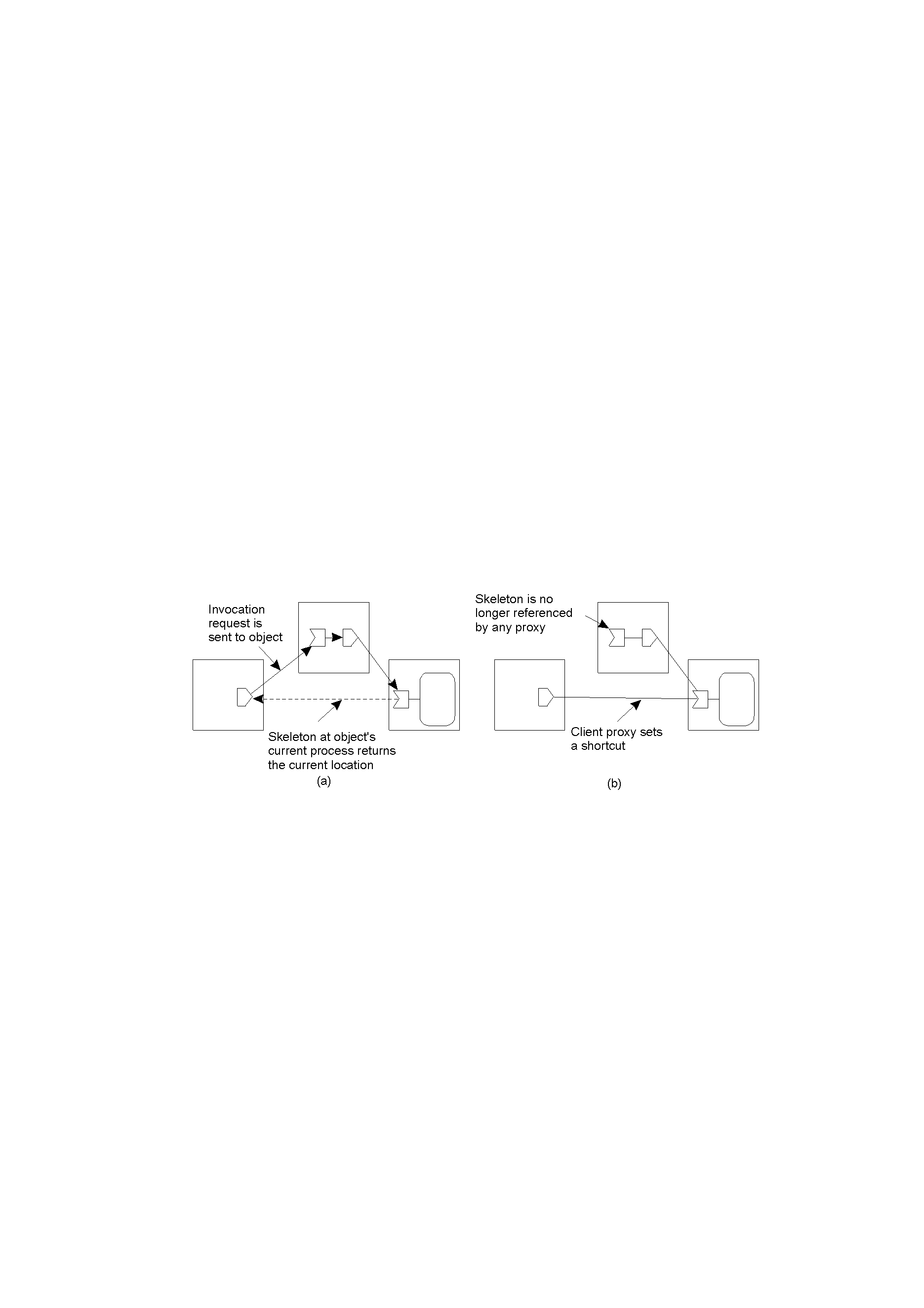
（2）只要需要，链中的所有中间位置就必须维护它们的那一部分转发指针链

（3）一个相关的缺点是它所在链很脆弱，易于断开。不管是什么原因，只要有一个转发指针丢失，就无法再到达实体（一个重要的事情就是让指针链相对短一些，并且确保转发指针是健壮的）。

**优化方案：**每个转发指针都以(代理, 骨架)值对的形式实现。在SSP中，代理被称为存根，而骨架被称为后裔，结果就变成了(stub, scion)值对，即(存根, 后裔) 。骨架(即服务器端存根)要么包含实际对象的本地引用，要么包含该对象所用代理(即客户端存根)的本地引用。



当对象从地址空间A移到地址空间B时，它会把一个代理留在A中，并且在B中安装一个引用代理的骨架。这种方法存在一个有趣的特点：移动细节对客户是完全透明的。客户唯一能看到的就是一个代理。客户无法看到代理转发调用的方法，也不知道代理把调用转发到了什么位置。



直接向起点代理发送响应和沿着转发指针的相反路线发送响应二者各有所长。

（1）对于前者来说，通信要快一些，这是因为需要经过的过程较少

（2）使用前者时只有起点代理可以调用，而后者可以调用所有中间代理

当骨架不再被任何代理引用时，它可以被删除。

如果(代理, 骨架)值对组成的链中有进程崩溃，或者因其他原因而无法到达，那么将会出现问题。解决方法是：链断开时，可以向对象的起始位置提出请求，询问对象当前所在的位置。为了允许对对象的起始位置进行修改，可以使用传统的命名服务来记录当前的起始位置。

**6、移动实体定位：基于起始位置的方法**

**广播多播和指针转发的局限性：**使用广播和转发指针带来了可扩展性的问题。在大型网络中，很难有效地实现广播和多播，而过长的转发指针链会导致性能问题，并且容易受到链断开的影响。

**方法简介：**起始位置代理位于局域网中，与包含在移动代理主机IP地址中的网络地址相对应。当一台移动主机转移到另一个网络中时，它会请求一个可以用来通信的临时地址。这种转交地址(care-of address) 在起始位置代理中注册。

当起始位置代理收到发给移动主机的数据包时，它会查找主机的当前位置。如果主机是在当前本地网络中，那么就简单地转发数据包。否则，它会建立一条通往主机当前位置的通道，准确地说，它会把数据重新组装成IP包，然后发送给转交地址。同时，将把主机的当前位置告诉数据包的发送者。



**缺点：**

（1）使用了起始位置，增加了通信延迟（为了与移动实体通信，客户首先必须与起始位置进行联系，而起始位置可能与实体本身处于完全不同的位置）

（2）必须保证起始位置始终存在，否则，将无法与实体联系

**解决方案：**

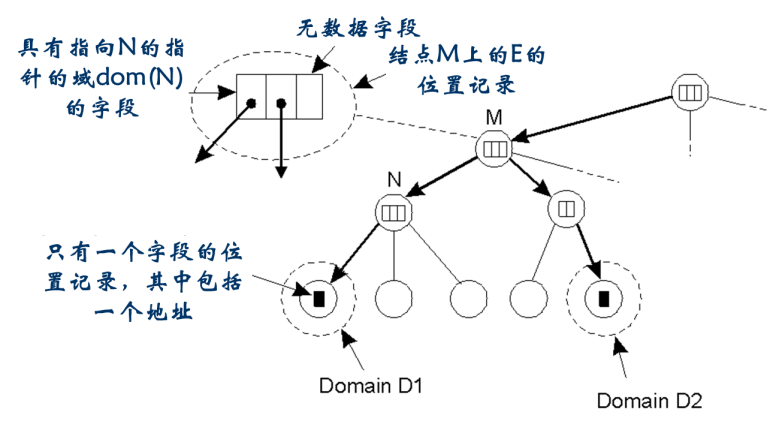
（1）在传统的命名服务中注册起始位置，然后让客户首先查找起始位置所在的位置（有效地缓存起始地址）

（2）双层设计：与移动实体建立连接时，客户首先查看本地注册机构，看看移动实体是否在本地。如果不是，那么就联系实体的起始位置，以便找到它的当前位置

**7、移动实体定位：分层方法**

**方法简介：**在分层设计中，网络被划分为一组域（与DNS类似）。有一个覆盖整个网络的顶级域。每个域可以进一步分成多个更小的子域。最低层的域称为叶域，叶域通常与计算机网络中的局域网相对应，或者对应于移动电话网络中的单元。每个域D都拥有关联的目录节点dir (D)。dir (D)会持续跟踪域的实体，这样就形成了一颗目录节点树。顶级域的目录节点称为根(目录)节点，它包括了全部实体。

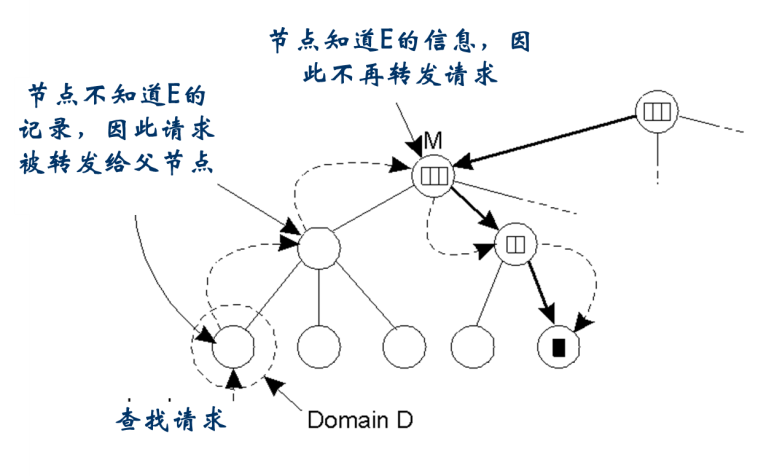
根节点将拥有每个实体的位置记录，其中每条位置记录都存储一个指向更低层子域目录节点的指针，而这里所说的更低层子域就是记录的关联实体当前所在的子域。实体可以拥有多个地址，比如说它被复制了，就会出现这种情况。如果实体分别在叶域D1和D2中拥有地址，那么同时包含D1和D2的最小域的目录节点将包含两个指针。每个指针都指向一个包含地址的子域。



**查询操作（查找操作是在局部进行的）：**

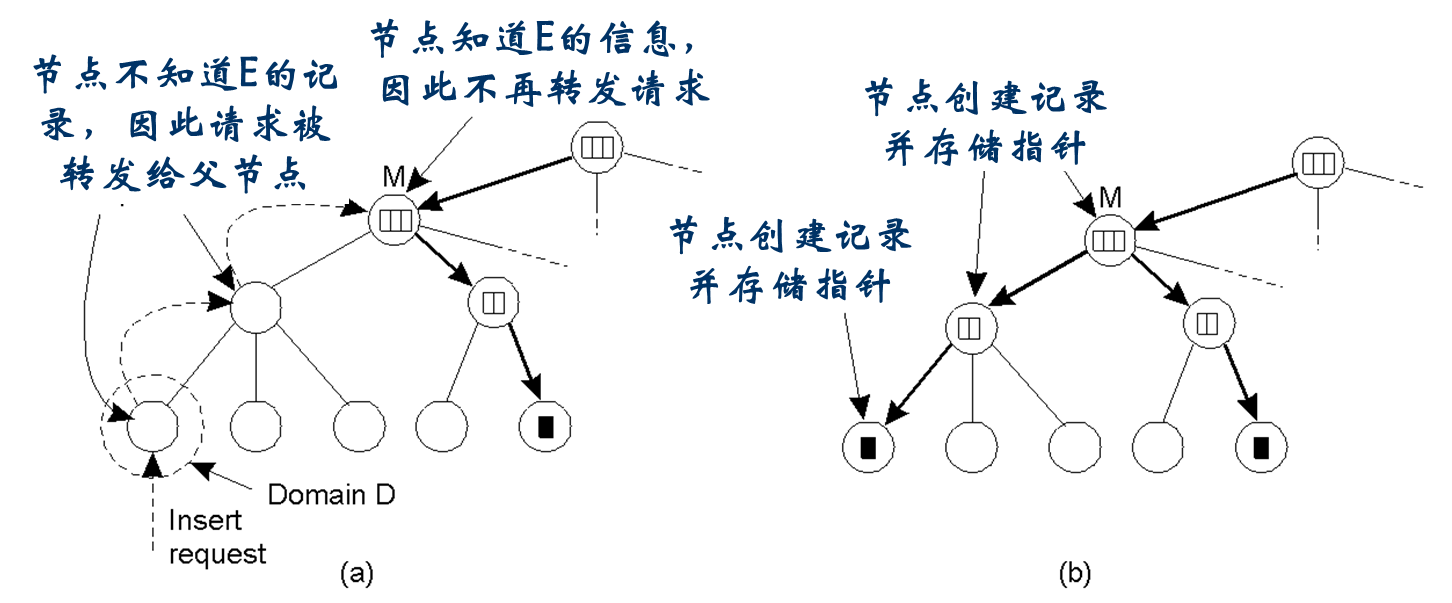
（1）如果这个目录节点没有存储该实体的位置记录，那么就该说明实体现在不在D中。因此，这个节点会把请求转发给它的父节点。注意，父节点代表一个比它的子域更大的域。

（2）如果父节点也没有E的位置记录，那么就会把查找请求转发给更高一层的域，依此类推。



**更新操作：**

**（1）复制：**假设实体E在叶域D中创建了一个复制实体，需要在这个复制实体中插入E的地址。插入操作从D的叶节点dir(D)开始，然后D会立即把插入请求转发给它的父节点。父节点同样会转发插入请求，直到插入请求到达已经为E存储了位置记录的目录节点M为止。

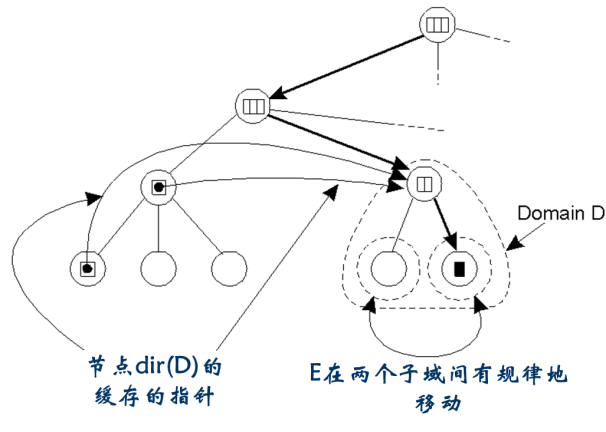


**（2）删除：**如果需要删除叶域D内实体E的地址，那么需要将目录节点dir(D)从它为E存储的位置记录中删除该地址。如果那条位置记录变成了空的，准确地说，就是说它不再包含其他E在D中的地址，那么这条记录也可以删除。在这种情况下，dir(D)的父节点需要删除指向dir(D)的指针。如果父节点中用于E的位置记录也变空了，那么也应该删除这条位置记录并通知更高一层的节点。依此类推，这个过程将连续进行，直到从一条位置记录中删除指针后该位置记录仍然非空或者到达根节点为止。

**指针缓存：**

对于定位服务器来说，在本地缓存地址一般不会非常有效（实体经常移动）。只有缓存数据很少改变时缓存才会有效。

一般情况下，如果D是一个最小的域，在这个域中有一个移动实体有规律地移动。那么在dir(D)中而不是任何其他节点中查找E的当前位置是有意义的。这种方法也称作**指针缓存**。原则上，从发起查询的叶节点开始，一路上经过的所有节点都可以缓存指向dir (D)的引用



通过不让dir(D)存储指向E当前所在子域的指针，而是让它直接存储E的实体地址，情况可以进一步改善。再加上指针缓存，查找操作只用两步就可以实现。第一步要求检查本地指针缓存，这样就会直接通往正确的目录节点。第二步要求这个节点返回E的当前地址。

**指针缓存需要解决的问题：**

（1）如何找到最合适的目录节点来存储移动实体的当前地址

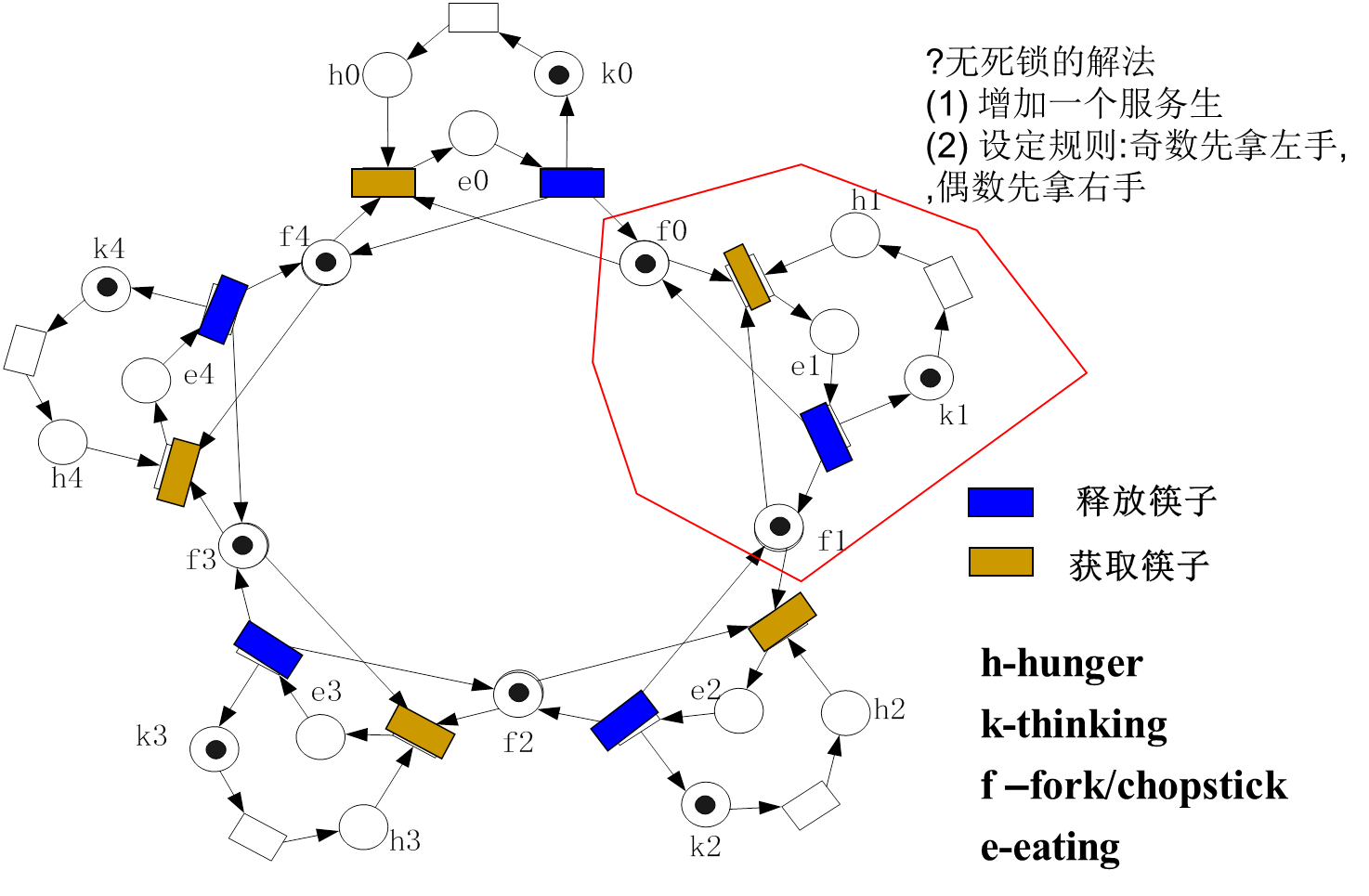
（2）什么时候让缓存项失效

**可扩展性问题：**

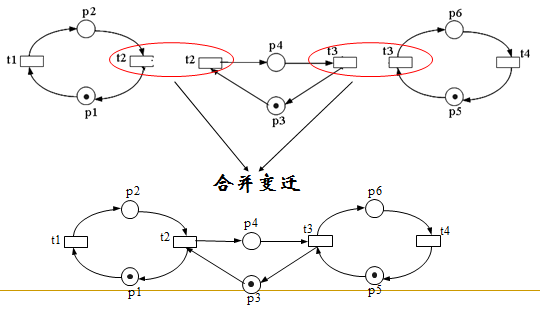
（1）根节点需要存储所有实体的位置记录并为每个实体处理请求（存储本身不是主要问题，真正的问题是如果不采用特殊技术，根节点需要处理太多的查找和更新请求，以致它会成为瓶颈）。**解决方案：**把根节点以及其他高层目录节点划分成多个子节点。每个子节点负责处理与定位服务支持的所有实体的特定子集相关的请求。把子节点均匀地扩散到网络中（让哪一个子节点处理哪个实体仍然是一个悬而未决的问题，一个可能的解决方法是考虑使用创建实体E的位置，不过，如果实体转移到一个很远的位置，那么问题依然存在）

## 11、并发建模（Petri网）

1、哲学家问题：



2、生产者/消费者问题：



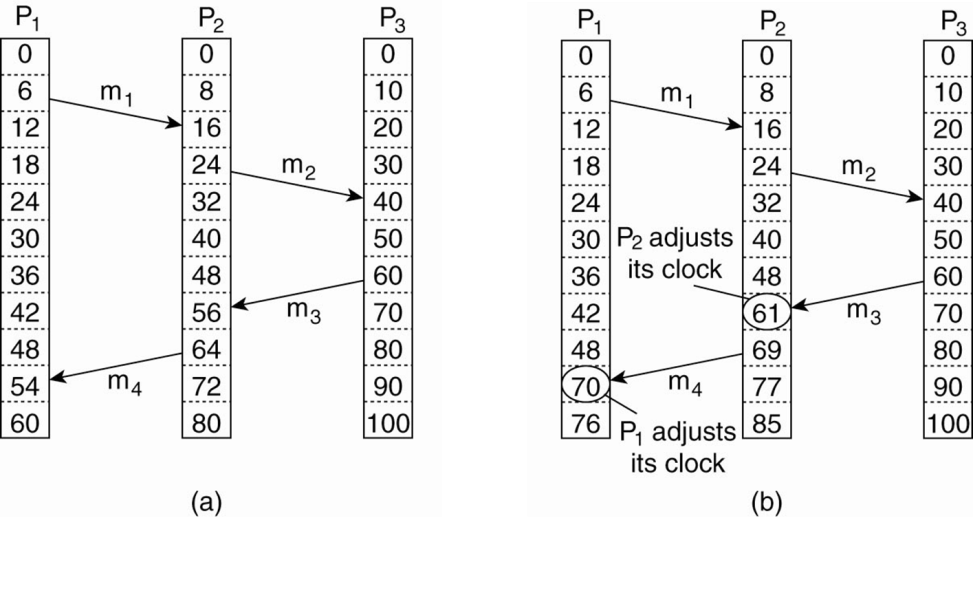
## 12、分布式系统同步

**1、Lamport算法：**

**逻辑时钟概念：**对于某类算法而言，重要的是时钟的内部一致性，而不是它们是否与真实的时间接近。这类算法通常将时钟称为逻辑时钟。

**Lamport算法的主要思想：**如果两个进程不进行交互，那么它们的时钟也无须同步，这是因为即使没有同步也察觉不出来，并且不会产生问题。它进一步指出，通常重要的不是所有的进程在时间上完全一致，而是它们在事件的发生顺序上要达成一致。

**Lamport算法简介：**



（1）当消息到达并且接收者时钟显示的时间值比消息的发送时间早时，接收者就将它的时钟调到一个比发送时间大1的值。（对这个算法稍作补充就可以满足全局时间的需要。即在每两个事件之间，时钟必须至少滴答一次。如果一个进程以相当快的速度发送或者接受两个消息，那么它的时钟必须在这之间至少滴答一次。）

（2）在某些情况下还需要一个附加条件，即两个事件不会精确地同时发生。为了达到这个目标，我们可以将事件发生所在的进程号附加在时间的低位后，并用小数点分开。（如果进程1和进程2中的事件都发生在时刻40，那么前者记为40.1后者记为40.2）

为分布式系统中的所有事件分配时间的方法遵循下面的规则：

a、若同一进程中a在b之前发生，则C(a)<C(b)

b、若a和b分别代表发送一个消息和接收该消息的事件，则C(a)<C(b)

c、对于所有不同的事件a和b，C(a) ≠ C(b)

**2、向量时间戳：**

**Lamport算法的局限性：**不能捕捉因果关系

**向量时间戳：**因果关系可以通过向量时间戳来捕获。分配给事件a的向量时间戳VT(a)具有下列性质：如果对某一事件b，有VT(a)<VT(b)，那么认为事件a在因果关系上处于事件b之前。向量时间戳的创建是通过让每个进程P维护一个向量V来完成的。

（1）当进程Pj接收到消息m时，它调整自己的向量，将每项Vj[k] 设置为max{Vj[k],vt[k]}。该向量现在反映了进程Pj必须接收的消息数，该消息数目至少是在发送消息m之前见到的消息。

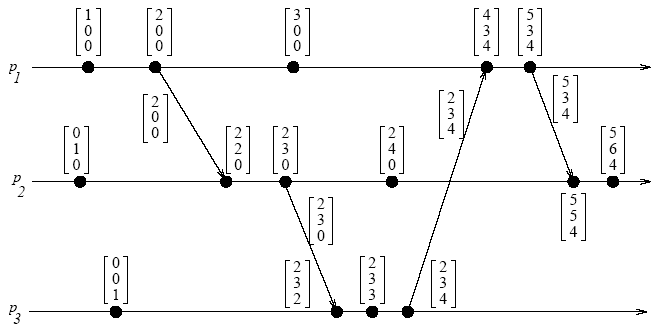
（2）此后将Vj[i]项增1，这表示接收消息m的事件是来自于进程Pi的下一个事件。

**向量时间戳的性质：**

（1）Vi[i]是到目前为止进程Pi发生的事件的数量

（2）如果Vi[j]=k，那么进程Pi知道进程Pj中已经发生了k个事件

第一个性质是通过在进程Pi中的新事件发生时递增Vi[i]来维护的。第二个性质是通过在所发送的消息中携带向量来维护的。当进程Pi发送消息m时，它将自己的当前向量作为时间戳vt一起发送。



**延时：**

假设进程Pj从进程Pi接受一个（向量）时间戳为ts的消息m，把该消息传送到应用层将被延时，直到以下两个条件满足为止：

（1）ts(m)[i] = VCj[k] + 1

（2）ts(m)[k] = VCj[k]，其中k!=i

**问题：**

（1）因为通信层不能判断消息包含的内容，所以只能捕获潜在的因果关系。例如，来源相同的两个完全独立的消息总是被通信层标记为因果相关的。这种方法过于严格，可能会导致效率问题

（2）并不是所有的因果关系都可以捕获。由于外部通信产生的因果关系可能不会被捕获