**一：中间件概念在分布式系统中所扮演的角色和位置**

**中间件：**

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，该层在逻辑上位于由用户和应用程序组成的高层与由操作系统组成的低层之间，这样的分布式系统又称为中间件。

中间件层延伸到了多台机器上，且为每个应用程序提供了相同的接口。它的**重要的目的是提供一定程度的透明性**，也就是一定程度上向应用程序隐藏数据处理的分布性。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**，又能够具有分布式操作系统的**透明性**和与之相关的**易用性**。

中间件模型：

* **1. 基于分布式文件系统(distributed file system)**

**其中一种相对简单的模型是件任何东西都作为文件来处理。**

* **2. 基于RPC (Remote Procedure Call)**

**重点在于通过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。**

* **3. 基于分布式对象**

**以一种透明的方式调用驻留在远程机器上的对象也应该是可以做到的。**

**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。**

**二：分布式系统的进程通信，基本问题 远程调用的基本原理和步骤 十个步骤**

进程通信的四种模式：

1. 远程过程调用
2. 远程对象调用
3. 面向消息的通信
4. 面向流的通信

分布式系统的进程通信基本问题：

1. 在局域网中，发送者一般并不需要了解接收者所在的位置

2. 广域网中, 选择最佳路径的问题

3. 发送者发给接收者的包有可能在途中丢失

远程调用基本原理：

当机器A上的进程调用机器B上的进程时，A上的调用进程被挂起，而B上的被调用进程开始执行。调用方可以通过使用参数将信息传送给被调用方，然后可以通过传回的结果得到信息

远程过程调用的十个步骤：

(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；

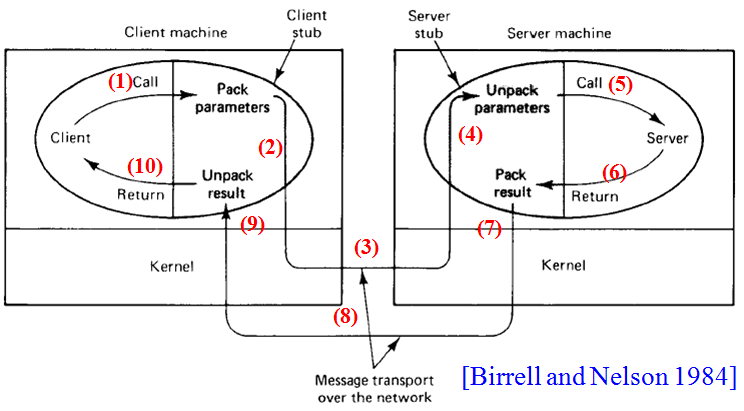
(6) 服务器执行要求的操作，操作完成后将结果返回给服务器存根

(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。



**三：移动agent技术的特点 和 名称解析**

Agent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。

移动agent概念：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的优势：**

* + 1. 减少网络负担They reduce the network load. 移动agent允许用户将会话打包转移至资源请求的目标主机，这样远程回话变为本地会话。
    2. 克服网络延迟问题They overcome network latency. 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They encapsulate protocols.
    4. 异步并自治执行They execute asynchronously and autonomously.
    5. 动态调整They adapt dynamically.
    6. 天然的异构性They are naturally heterogeneous.
    7. 是健壮和容错的They are robust and fault-tolerant.

**移动agent技术的特点：**

* 自主移动性：不拘泥于初始执行结点，可再网络各主机间自主移动。移动Agent可以将自身的状态和代码从一个环境迁移到另一个环境，并恢复执行。
* 协作性：移动Agent可以和其他Agent进行自主通信，协作完成任务。
* 安全性
* 移动功能而不移动数据，减少网络负担和延迟

**命名服务**

**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。命名服务由名称服务器实现。**

**全局层**

**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，特点就是它通常是稳定的。**

**行政层**

**由那些在单个组织内一起被管理的目录节点组成。行政层中的目录节点所具有的特点是它们代表属于同一组织或行政单位的实体组**

**管理层**

**由那些经常改变的节点组成。**

**名称解析：**

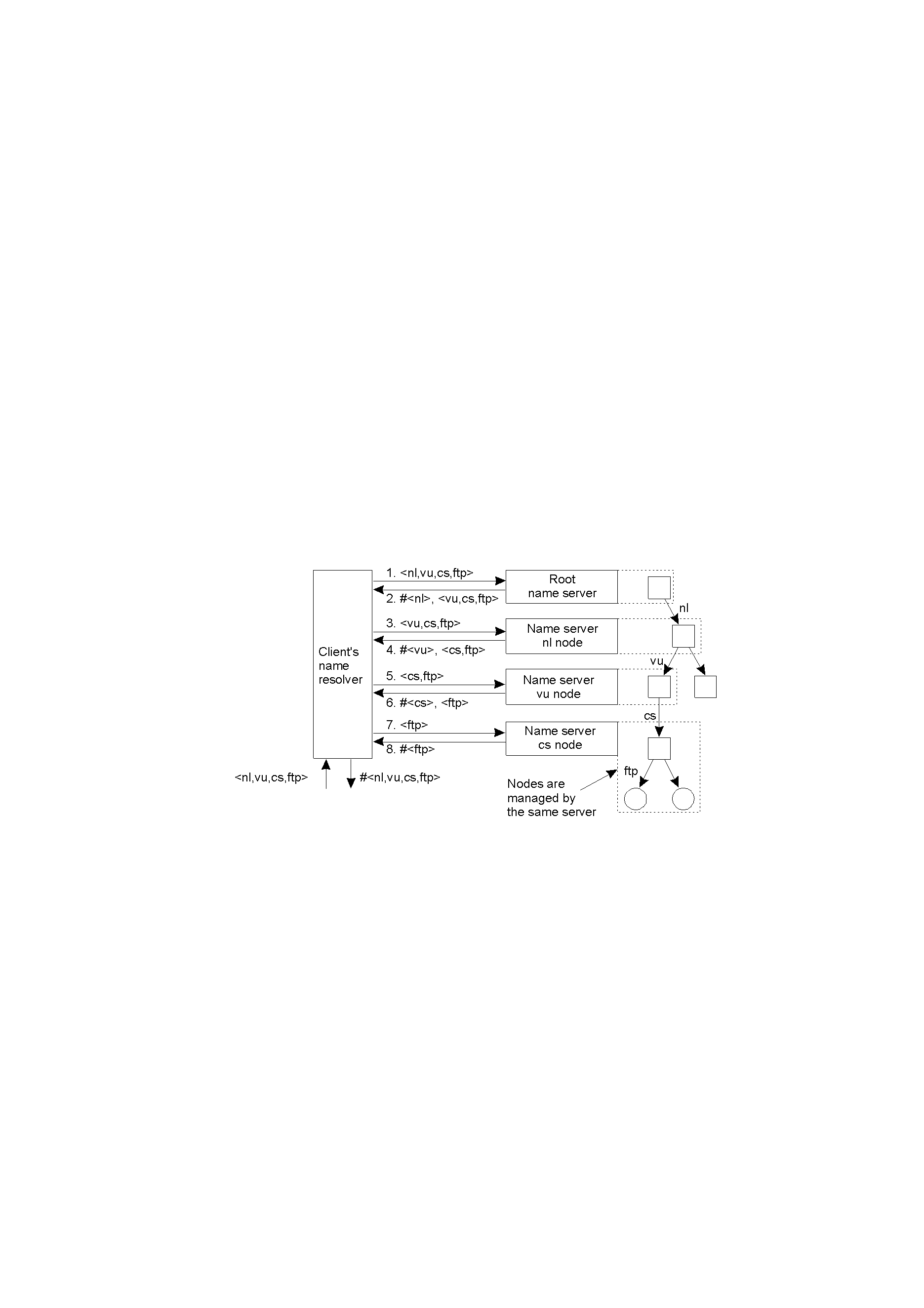
名称解析就是查询名称的过程。考虑一个路径名，如N: <lable-1, label-2, …, label-n>。 该名称的解析是从命名图的节点N开始的，在其目录表中查询名称label-1，并返回label-1所指向的节点的标识符。然后，解析过程在标识出的节点的目录表中继续查询名称label-2，依此类推，假设该命名路径确实存在，那么解析会在由label-n指向的最后一个节点上停止，并返回该节点的内容。

**名称解析分为两种：迭代名称解析和递归名称解析。**

1. **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。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，客户把剩下的路径名发送给根服务器解析到的服务器，该服务也尽量深入地解析路径名并把结果返回给客户。这一过程迭代的进行，最终含有请求地址的服务器会将地址返回给客户。每次迭代过程都是客户将路径名交给一台服务器，服务器把解析到的地址反馈给客户，客户再向下一级服务器发送解析请求。

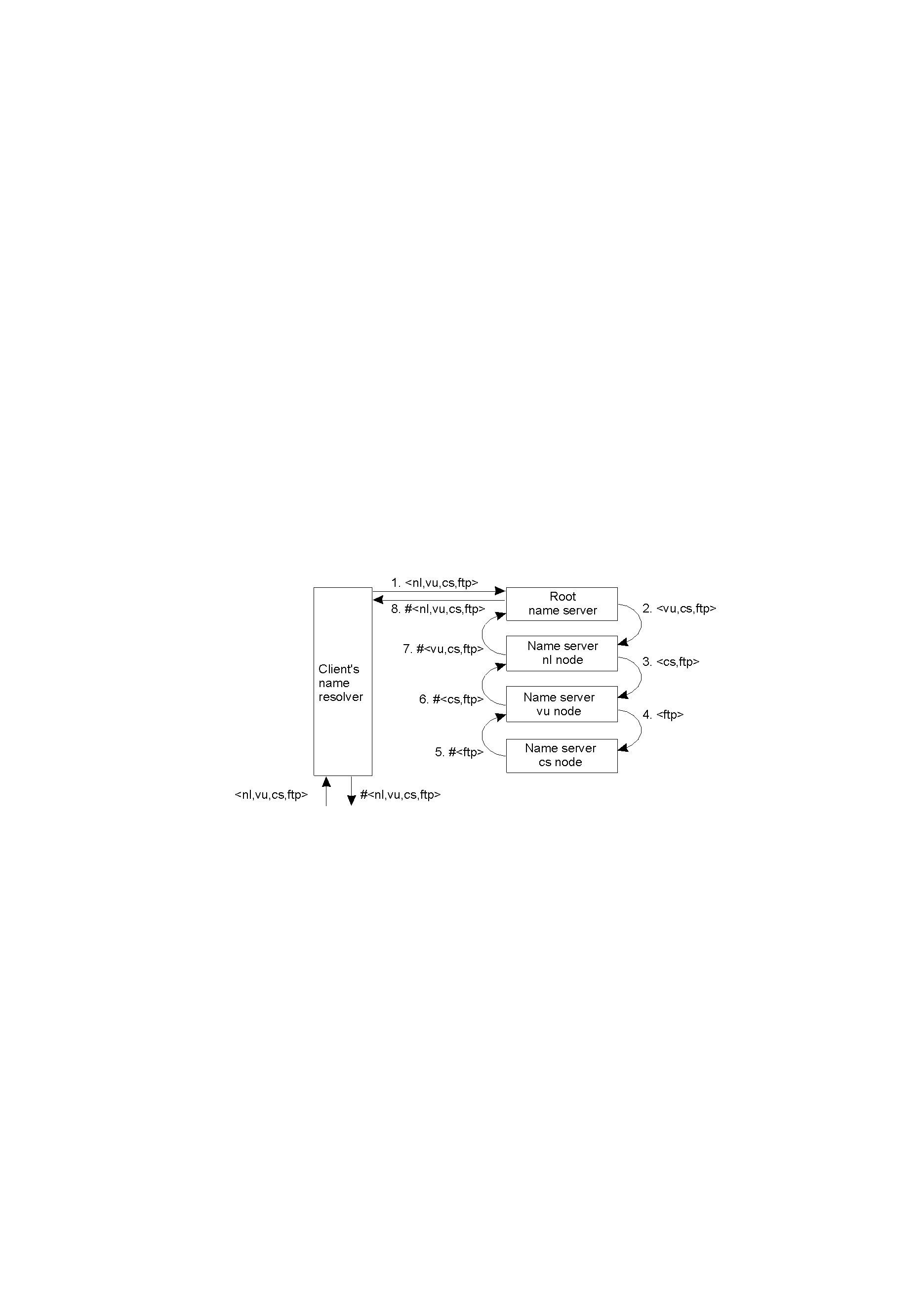
原理如图所示：



1. **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。举例来说，当根名称服务器找到实现nl节点的名称服务器所用的地址后，它会请求该名称服务器解析路径名nl: <vu, cs, ftp, pub, globe, index.txt>。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index.txt文件返回根服务器，根服务器再依次把该文件传递给客户的名称解析程序

递归解析的原理图：



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。

但是递归名称解析拥有两个重要的优点。

一是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；

二是可以减少通信开销。递归名称解析让名称服务器逐步负责实现更底层节点名称解析的名称服务器的地址，这样可以有效地使用缓存来提高性能。

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。

**四：互斥算法，选举算法**

**选举算法：**

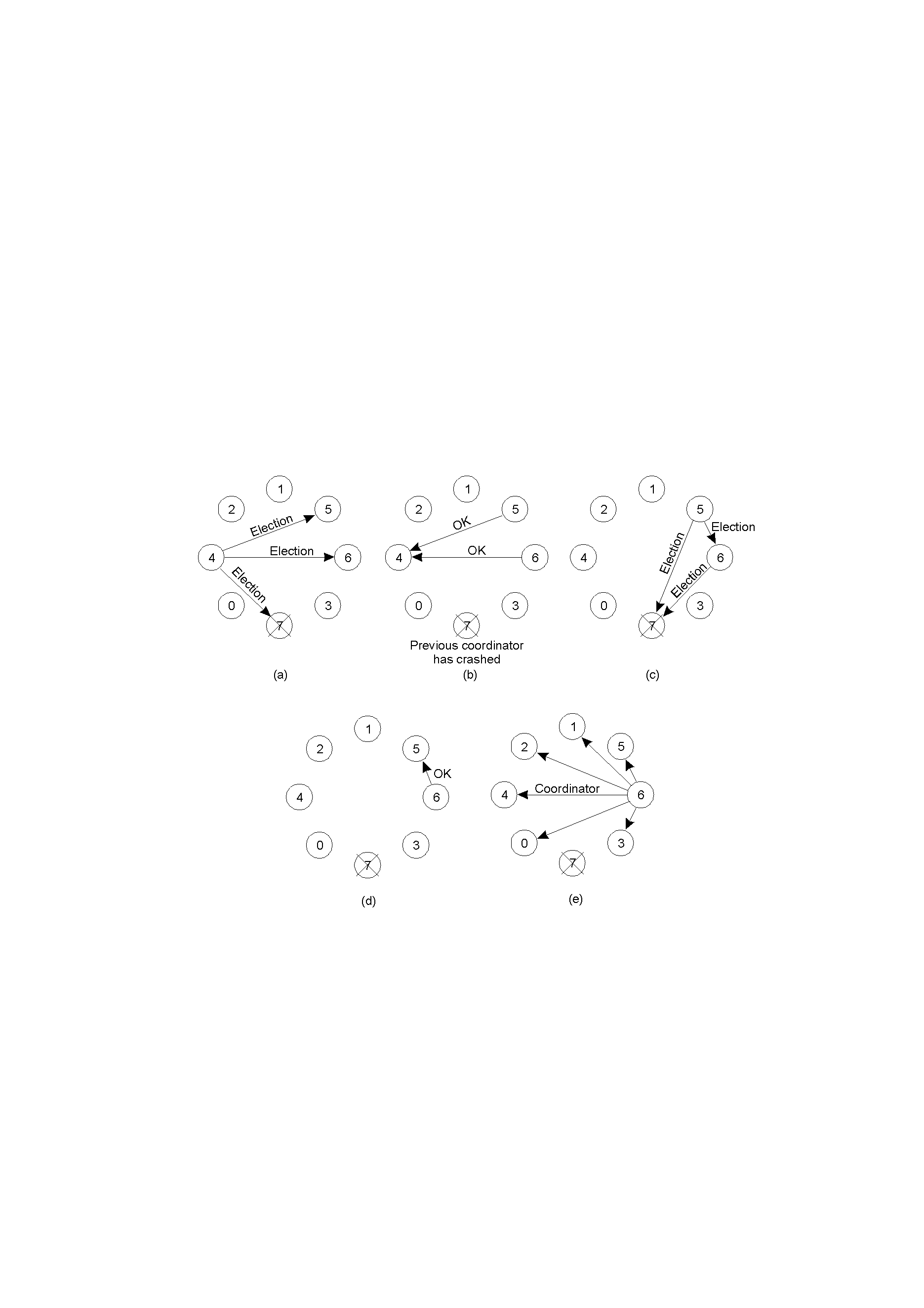
1. **欺负(Bully)算法**

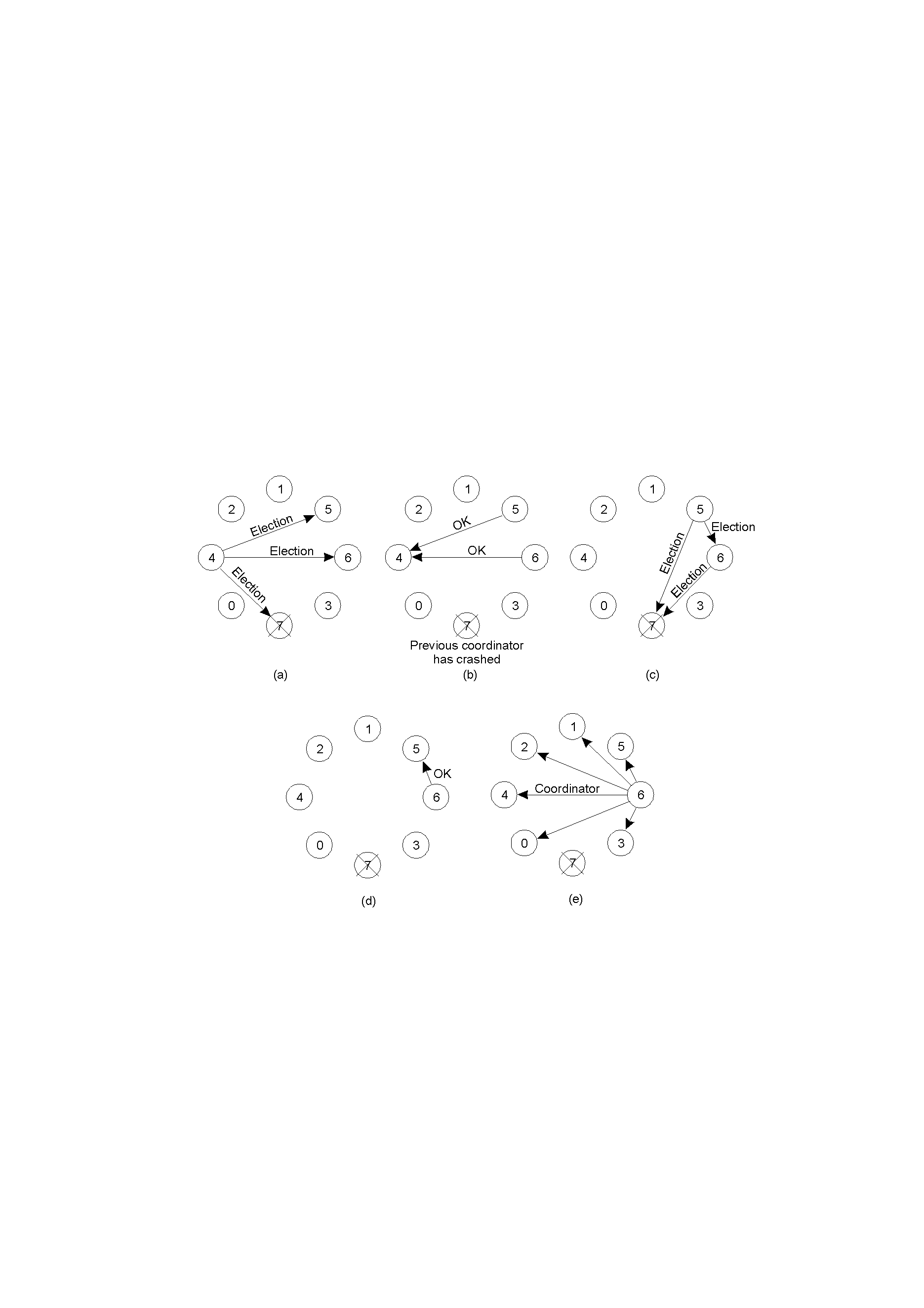
1.1当任何一个进程发现协调者不再响应请求时，它就发起一个选举。进程P按如下过程主持一次选举：

1.1.1 P向所有编号比它大的进程发送一个election消息；

1.1.2如果无人响应，P获胜成为协调者；

1.1.3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。





(a) 进程4主持一个选举；

(b) 进程5和6进行响应，告诉进程4停止选举；

(c) 进程5和6此时各自主持一个选举；

(d) 进程6通知进程5停止选举；

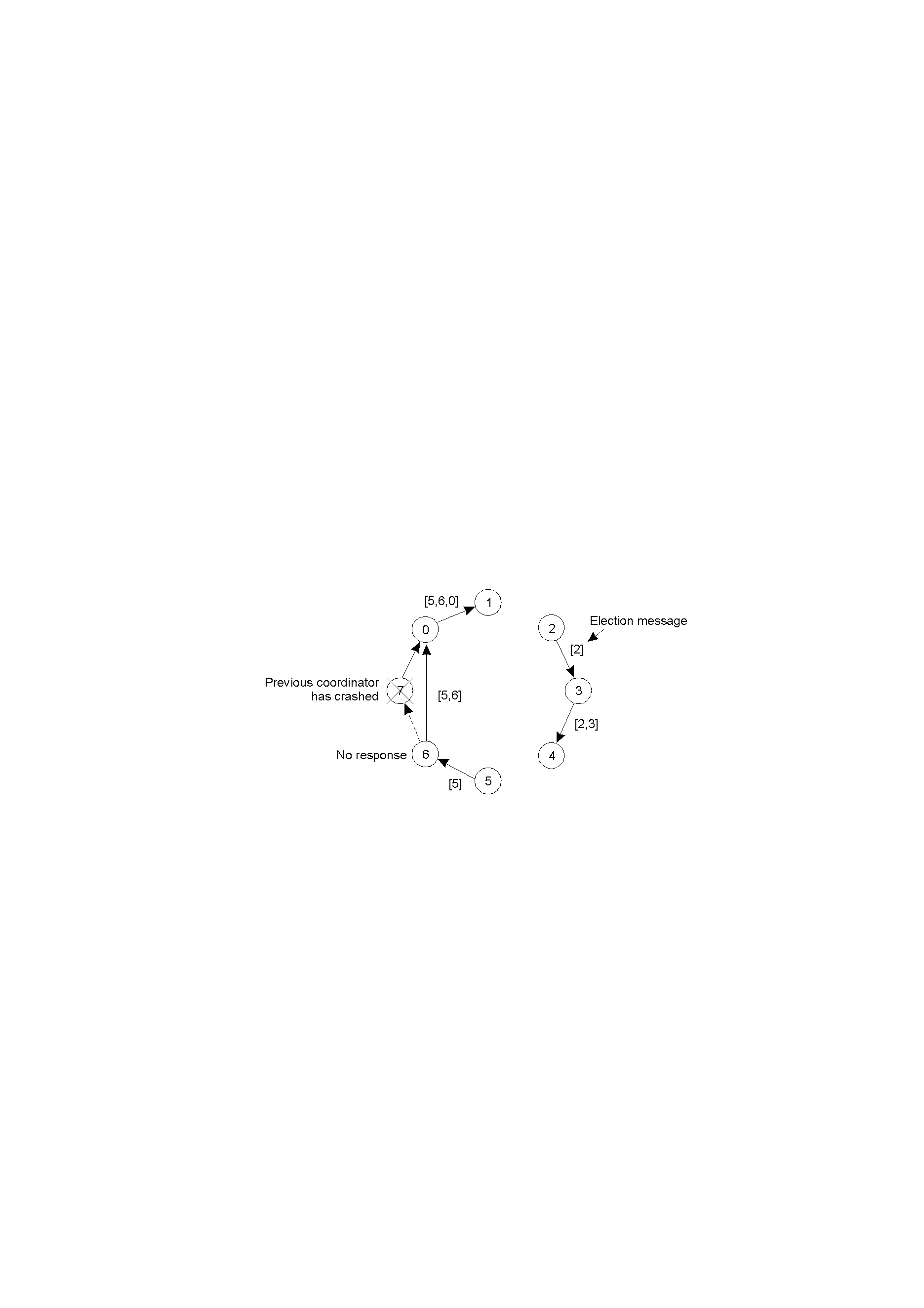
(e) 进程6获胜，并通知每个进程。

**2、环算法**

假设进程按照物理和逻辑顺序进行了排序，那么每个进程就都知道它的后继者是谁了。

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。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。

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。

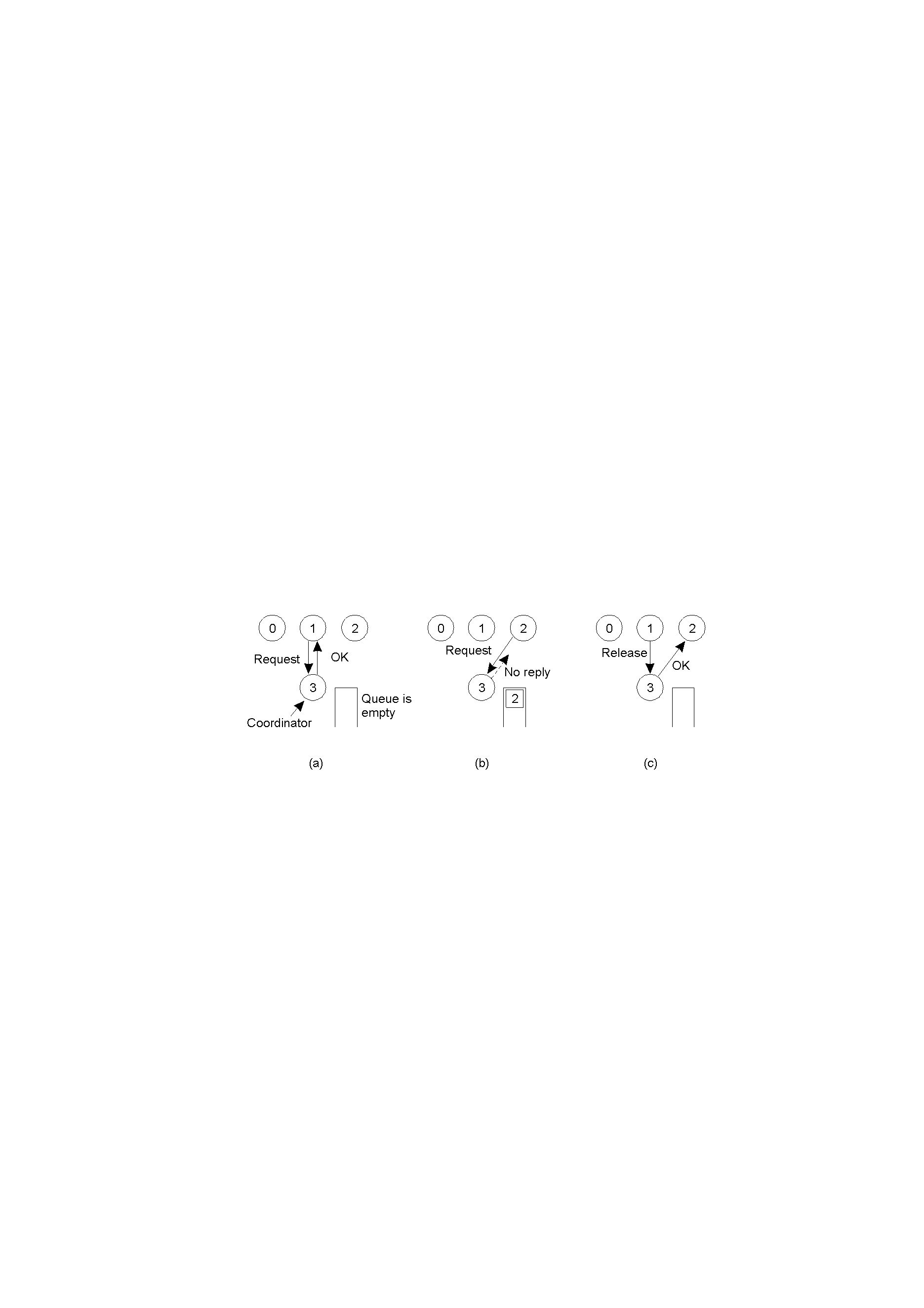


上图示意了当进程2和进程5同时发现以前的协调者进程7崩溃了时，这两个进程各自构造一个election消息，并且它们相互独立地绕环运行。最后，两个消息都将绕环走过全程，进程2和进程5分别将它们转化为coordinator消息，这两个消息拥有相同的成员，相互顺序也相同。两个消息绕环一周后都被删除。有多余的消息的循环会花费了一点带宽。

**互斥：**

**1、集中式算法**

只有一个协调者，无论何时一个进程要进入临界区，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。



(a) 进程1请求协调者允许它进入一个临界区。请求得到了批准；

(b) 进程2也请求进入同一个临界区。协调者不应答；进程2进入等待队列

(c) 进程1在退出临界区时通知协调者，协调者然后做出应答。协调者再通知等待队列中的排在最前面的进程2进入临界区

**优点：**没有进程会处于永远等待状态(不会出现饿死的情况)；易于实现，每使用一次临界区只需3条消息(请求、允许和释放)；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：**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。在一般情况下，如果进程在发出请求之后被阻塞，那么请求者就不能区分“拒绝进入”和协调者已经崩溃这两种情况，因为上述两种情况都没有消息返回。此外，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**

该算法的工作过程如下：当一个进程想进入一个临界区时，它构造一个消息，其中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。

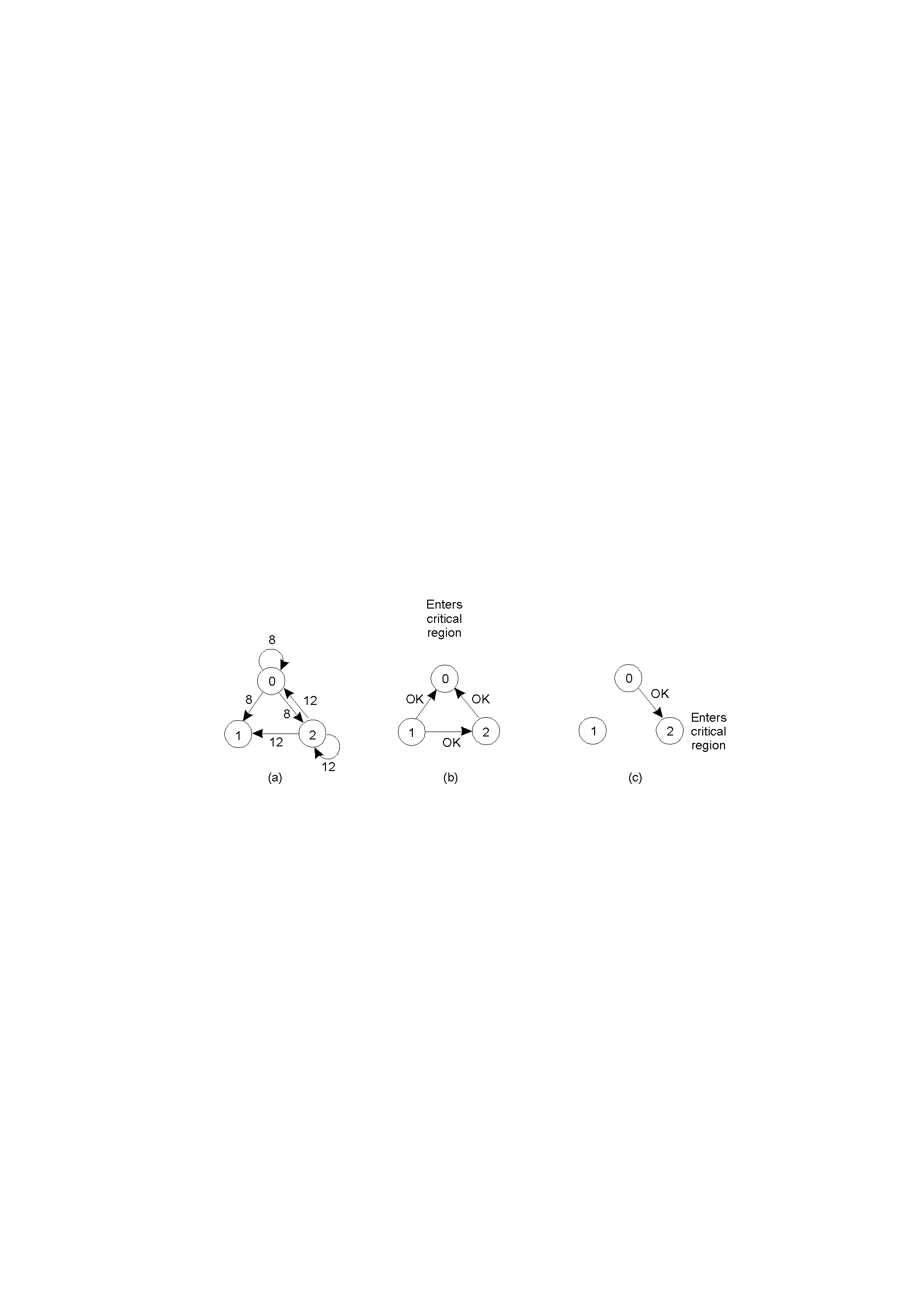
当一个进程接收到来自另一个进程的请求消息时，它根据自己与消息中的临界区相关的状态来决定它要采取的动作。可以分为三种情况：

2.1若接收者不再临界区也不想进入临界区，它就向发送者发送一个OK消息。

2.2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。

2.3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。



(a) 两个进程同时希望进入同一个临界区；

(b) 进程0具有最早的时间戳，所以它获胜；

(c) 当进程0退出临界区时，它发送一个OK消息，所以进程2现在可以进入临界区

优点：不会发生死锁或者饿死现象；最大的优点是不存在单个故障点。

缺点：单个故障点被n个故障点所取代；要求更多网络通信的算法；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，清单中包括进入组的进程、离开组的进程以及崩溃的进程。

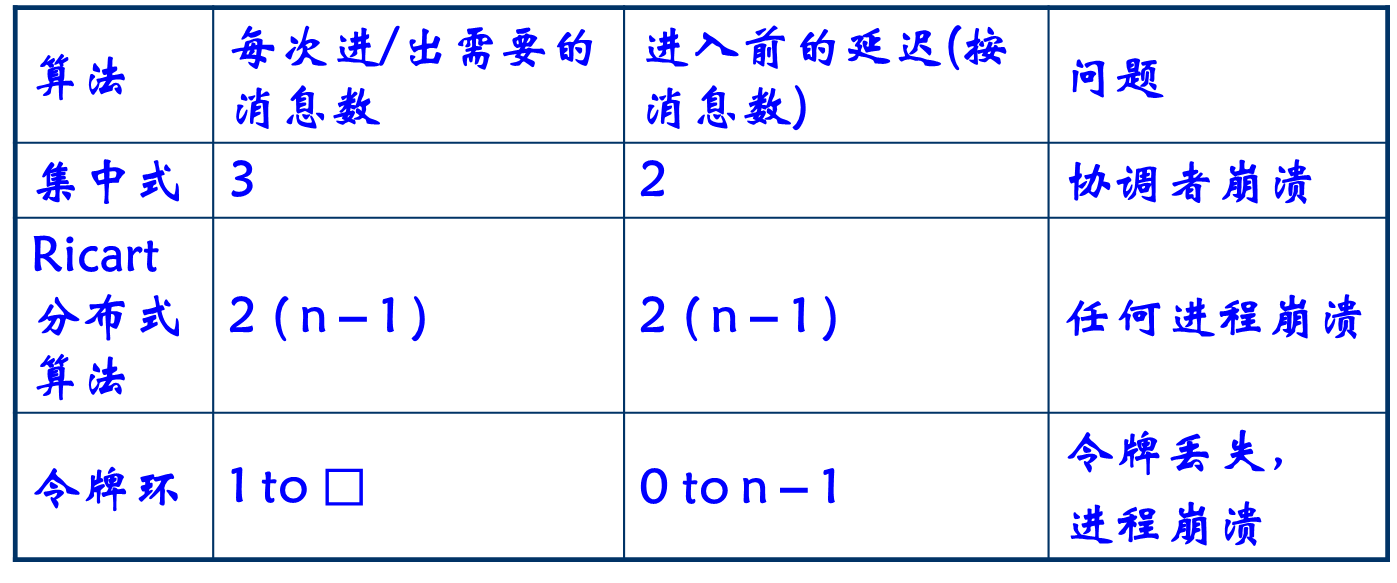
**3、令牌环算法**

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。

优点：不会发生饿死现象，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：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；如果有进程崩溃，该算法也会出现麻烦，但是恢复起来比其他算法容易。

三个算法的比较:



**六：以数据为中心的一致性**

**1.严格一致性:**

条件定义：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。

严格一致性中存在的问题是它依赖于绝对的全局时间(注意由于技术的限制,我们需要处理同一时间间隔内所发生的多个操作)

图例如下:

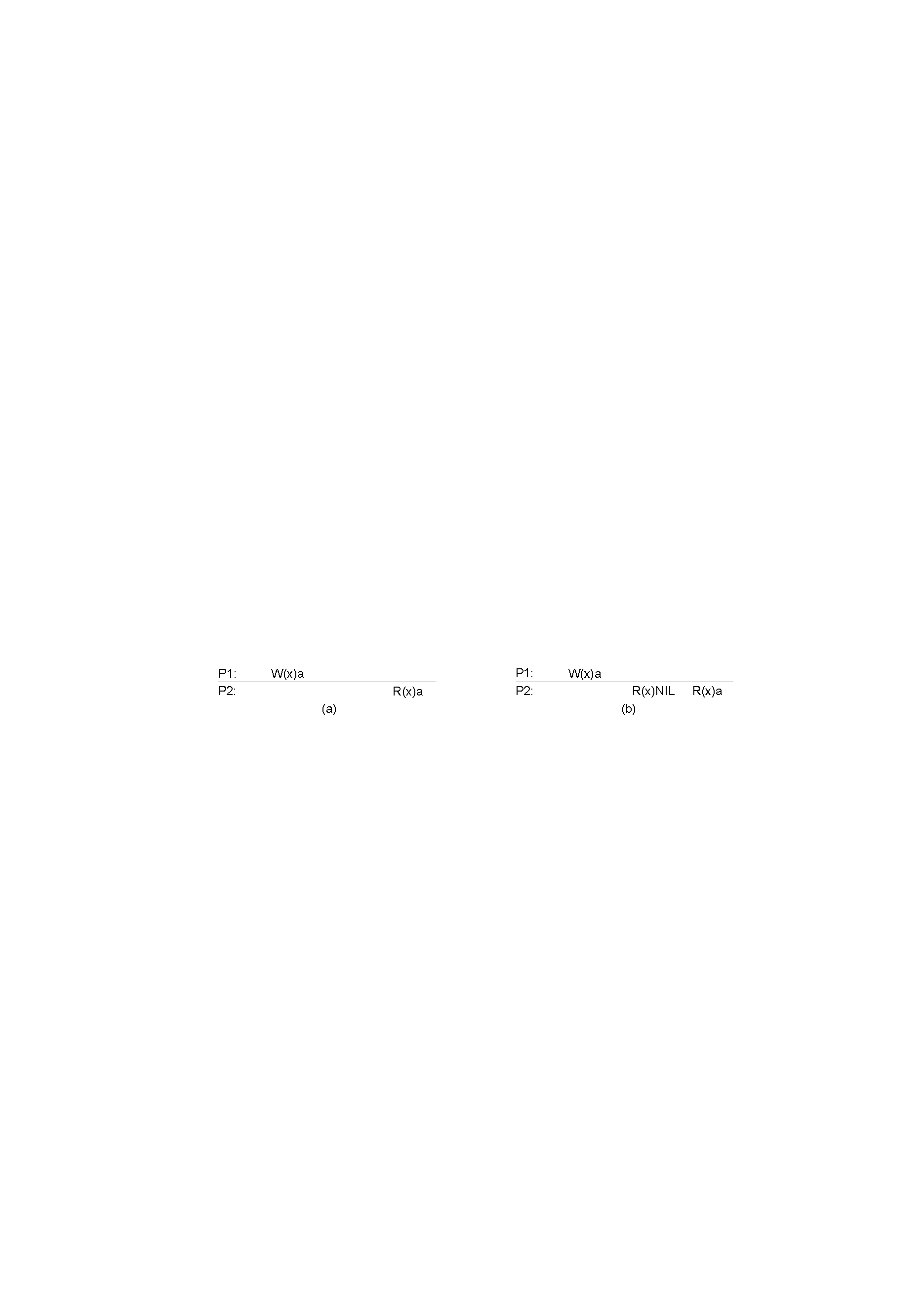


图 6.5 两个对同一数据项进行操作的进程。

(a)严格的一致性存储; (b) 非严格的一致性存储

总之，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。(如果一个数据项被改变了，那么无论数据项改变之后多久执行读操作，无论哪些进程执行读操作，无论这些进程的位置如何，所有在该数据项上执行的后续读操作都将得到新数值。同样，如果执行了读操作，那么无论多快地执行下一个写操作，该读操作都将得到当前的值。 )

**2.顺序一致性:**

条件定义:

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。

图例如下:

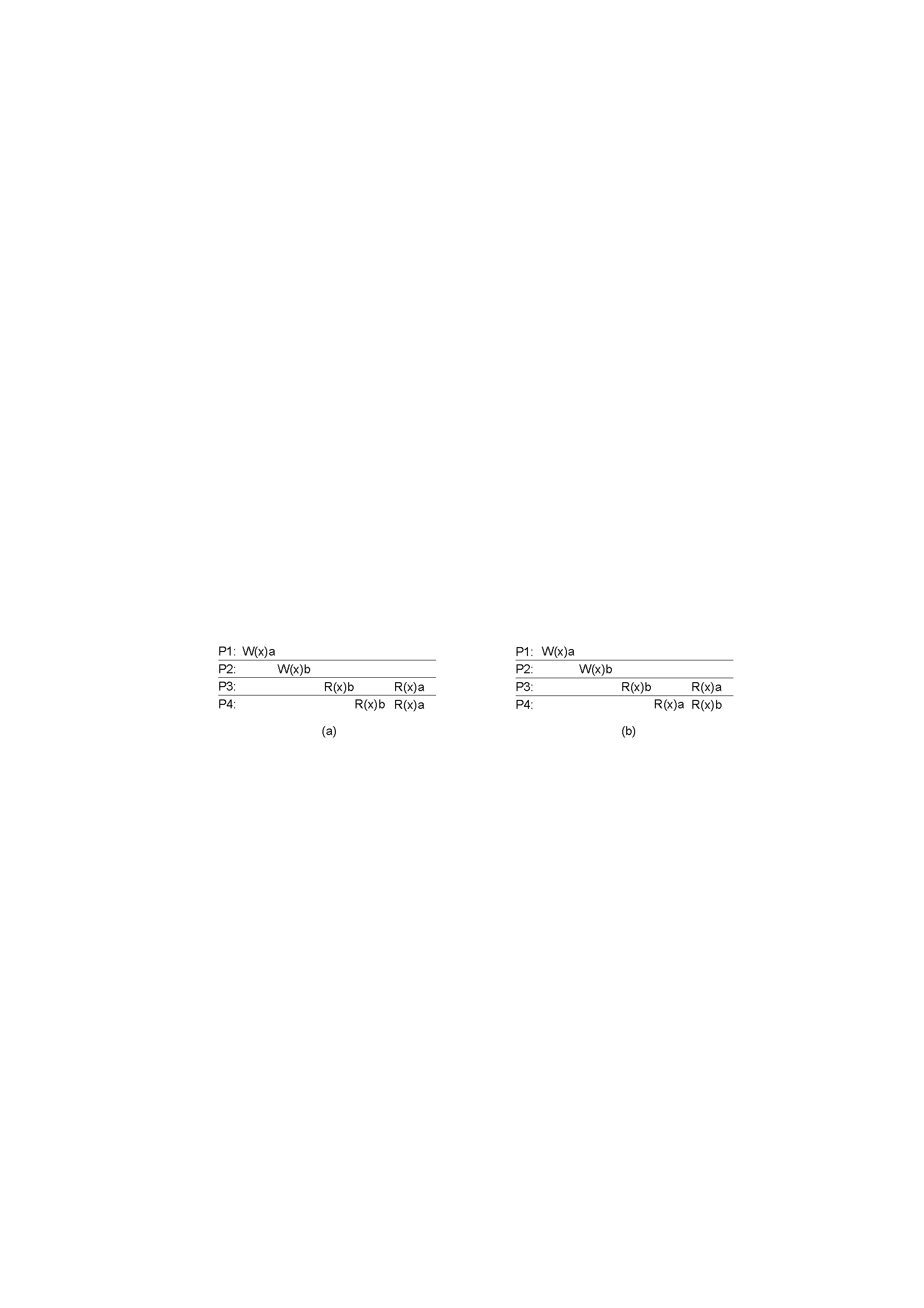


图 6.6 顺序一致的与非顺序一致的数据存储

(a) 顺序一致的数据存储； (b)非顺序一致的数据存储

**3.线性一致性**（弱于严格一致性但又强于顺序一致性）这种模型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。

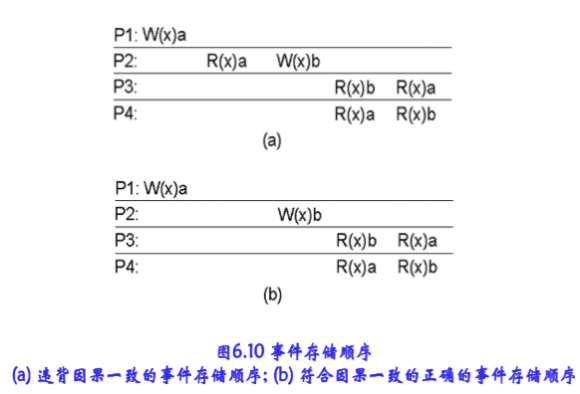
条件定义：当数据存储上的每个操作都具有时间戳并满足以下条件时，称这个数据存储是可线性化的。任何执行结果都是相同的，就好像所有进程对数据存储的读、写操作是按某种顺序执行的，并且每个进程的操作按照顺序所执行的顺序出现在这个顺序中。另外，如果tsop1(x)<tsop2(y)，那么在这个顺序中，操作OP1(x)出现在操作OP2之前。 （注意，可线性化的数据存储也是顺序一致的。它们的区别在于：线性化是根据一系列同步时钟确定序列顺序的。在实际应用中，线性化主要用于开发算法的形式验证。关于根据时间戳维护顺序的附加限制使得线性化的实现比顺序一致性的实现开销更大。Tsop(x)---时间戳）。

**4．因果一致性**(causal consistency)模型(Hutto和Ahamad 1990)代表一种弱化的顺序一致性模型，因为它将潜在具有因果关系的事件和没有因果关系的事件区分开来。

因果关系理解：考虑一个存储器的实例。假设进程P1对变量x执行了写操作。然后进程P2先读取x，然后对y执行写操作。这里，对x的读操作和对y的写操作具有潜在的因果关系，因为y的计算可能依赖于P2所读取的x值。没有因果关系的操作被称为并发的。

条件定义：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。一种实现方法是使用上一章所讨论的向量时间戳。

[](http://www.cyanny.com/wp-content/uploads/2013/11/Distributed-System04.png)

**5. FIFO一致性**（因果一致性允许不同的机器以不同的顺序看到并发的写操作，但是所有机器必须以相同的顺序看到因果相关的写操作。如果放弃后一条要求，这样就得到FIFO一致性。）条件定义：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。 （解释：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。）

[](http://www.cyanny.com/wp-content/uploads/2013/11/Distributed-System05.png)

FIFO的限制就是这些应用要求从任何位置都可以按顺序看到某个单一进程所产生的写操作。

然而，现实中有些应用并不这样要求。例如：在多副本数据库中，一个在临界区的进程正在对记录进行修改操作时，其它进程对该记录的访问得不到相应，当该进程离开临界区之后，其它进程对临界区的访问才得到响应。对于诸如数据库这样的系统，处于临界区中的进程对数据库的写操作只在本地副本上进行，当要推出临界区时，才将最终结果传播到其它副本。所以在其它机器上的进程并不能也不必见到该进程的所有操作。对这类应用的一般操作是引入一个同步变量，与该变量相对应的操作只有一个，即synchronize（S),该操作对于所有的数据副本进行同步。一个进程p只对本地数据副本进行操作，当数据被同步时，p进程对数据的最终操作结果才被传播到其它副本上。当然其它进程对数据的最终操作结果在同步时，会传播到进程P的副本上。

* 一个较好的解决方法是让进程结束它对临界区的占用，然后确保向所有地方发送最后结果，这种方法不太关心，甚至毫不关心所有中间结果是否已经按照顺序传播到所有拷贝。
* 通常，引入同步变量(synchronous variabiles)可以实现这种方法。同步变量S仅有一个关联操作synchronize(S)，该操作同步数据存储的所有本地拷贝。让我们回忆一下，进程P仅在存储的本地有效拷贝上进行操作。数据存储同步时，进程P的所有本地写操作都将被传播到其他拷贝，而其他进程的写操作也将被传播到P的本地拷贝。

**6.弱一致性**

使用同步变量来部分地定义一致性就得到称为弱一致性(weak consistency)模型 (Dubois 等1988)。弱一致性模型具有三个属性：

* + 对数据存储所关联的同步变量的访问是顺序一致的；
  + 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；
  + 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。

**7.释放一致性**

通常，如果一个分布式的数据存储满足以下规则，那么它就是释放一致的。

* + (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)。

**释放一致性提供两种类型的同步变量。**

**获取(acquire) 操作是用于通知数据存储进程进入临界区的操作**

**释放(release)操作是表明进程刚刚离开临界区的操作。**

**这两个操作可以通过以下两种方法实现：**

**(1) 对特殊变量的一般操作；**

**(2) 特殊操作。无论哪种方法，程序员都将负责在程序中加入显式的代码，用以说明执行操作的时间，**

**8.入口一致性**

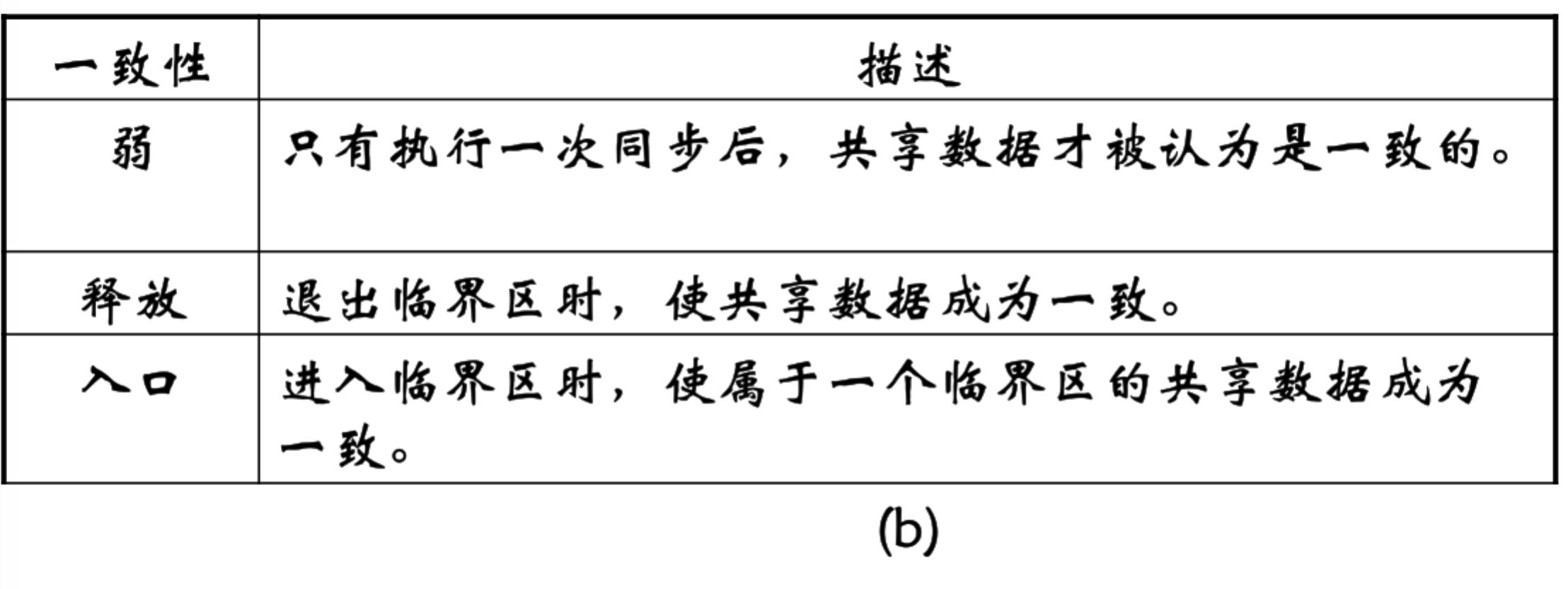
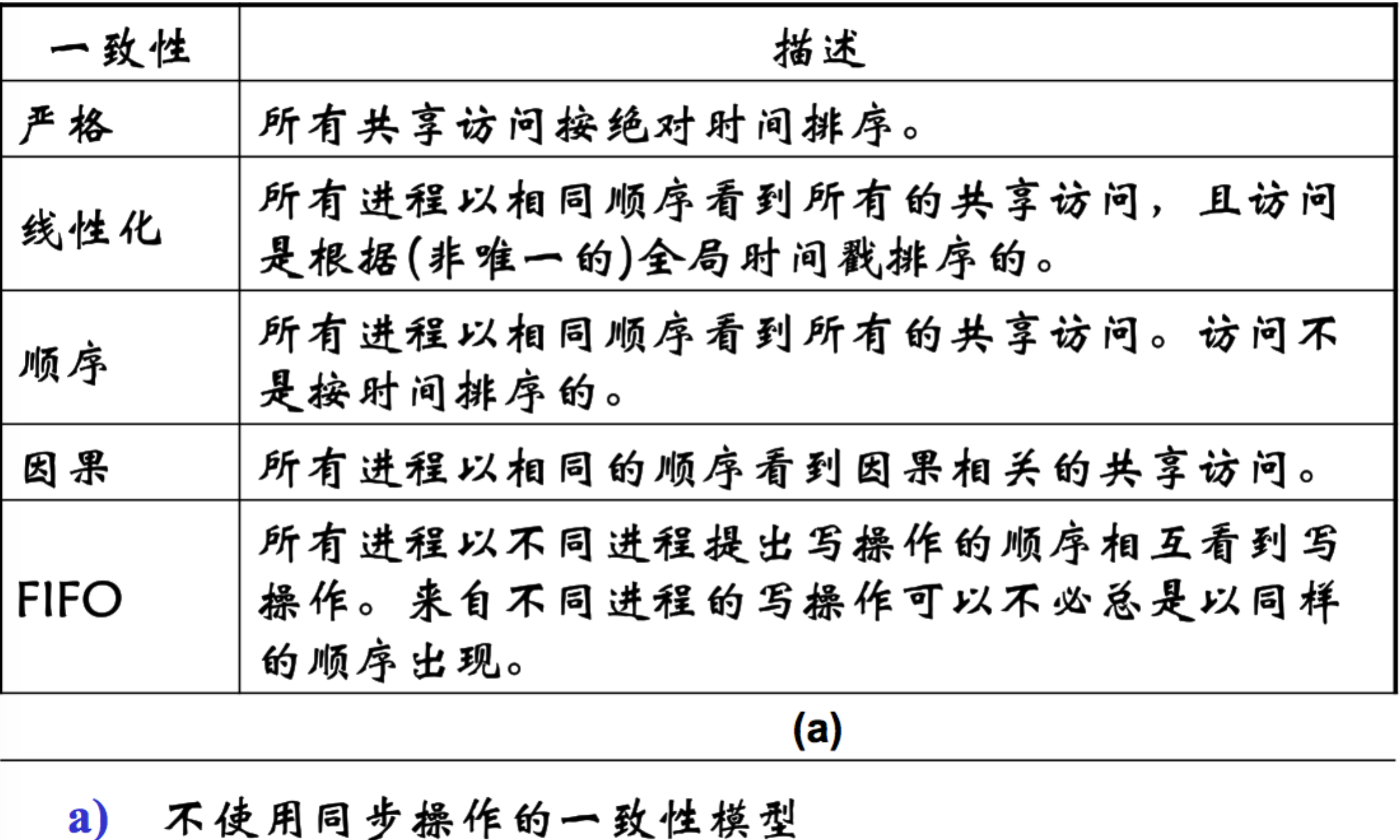
形式上，如果一个数据存储满足以下所有条件，那么它符合入口一致性(Bershad和Zekausas 1991):

* + (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。
* 同步:严格一致性, 线性一致性,,顺序一致性,因果一致性,FIFO。
* 不同步:弱一致性，释放一致性，入口一致性。

本节中，讨论一类特殊的分布式数据存储。这里讨论的数据存储的特点是：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。

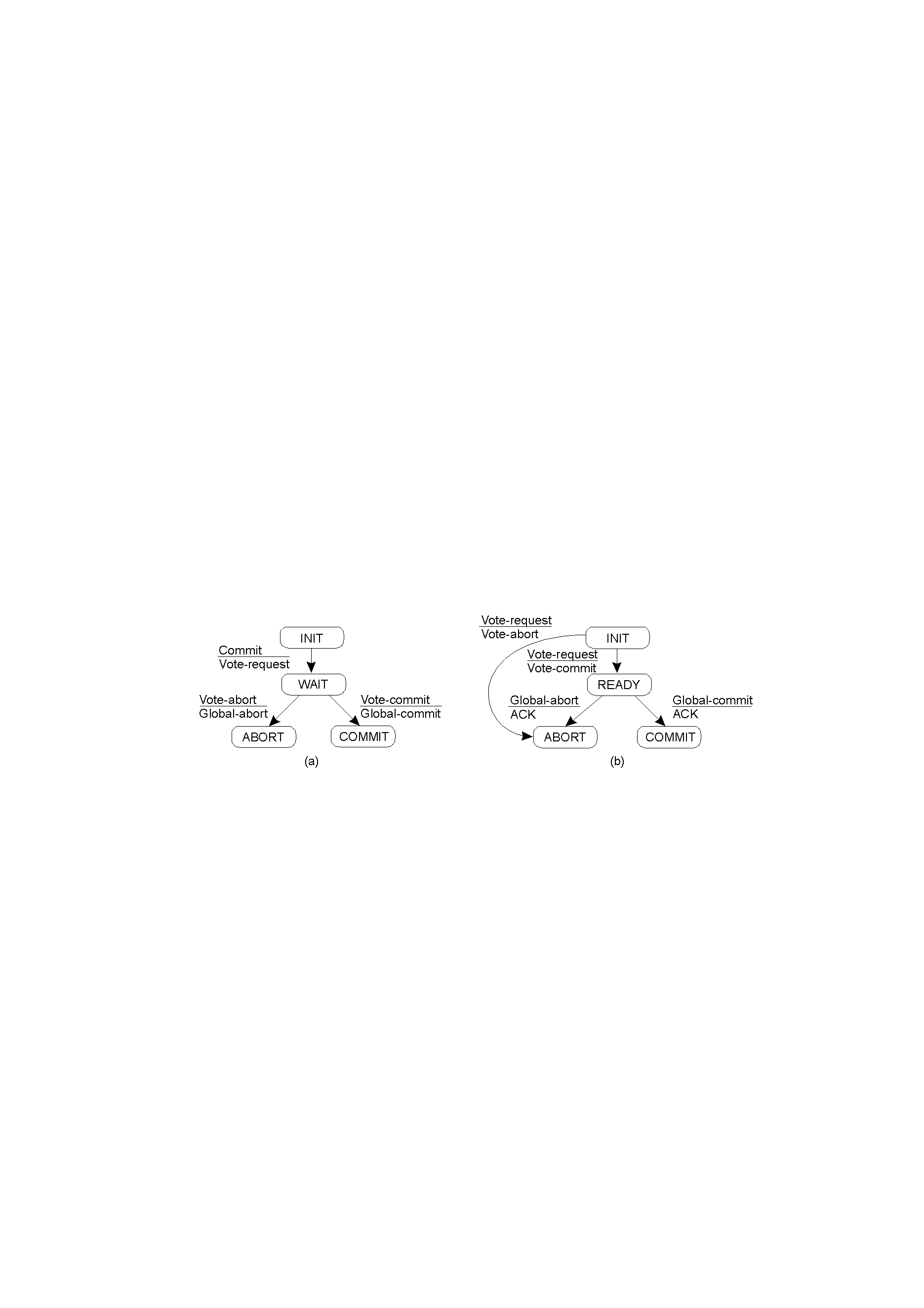
这些实例可以被看作是可以容忍相对较高程度的不一致性的(大规模的)分布式复制的数据库的案例。它们的共同之处是，如果在一段很长的时间内没有操作，那么所有的副本将逐渐地成为一致的。这种形式的一致性称为最终一致性(eventual consistency)。

因此，满足最终一致性的数据存储具有以下属性：没有更新操作时，所有副本逐渐成为相互完全相同的拷贝。



**七：事务提交(二阶段，三阶段提交)**

1. **两阶段提交协议(2PC)：**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。
   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。



1. The finite state machine for the coordinator in 2PC.（协调者）
2. The finite state machine for a participant.（参与者）

两阶段提交的一个问题在于当协调者崩溃时，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。**三阶段提交协议(3PC)**，避免了在出现故障停机时的阻塞过程。

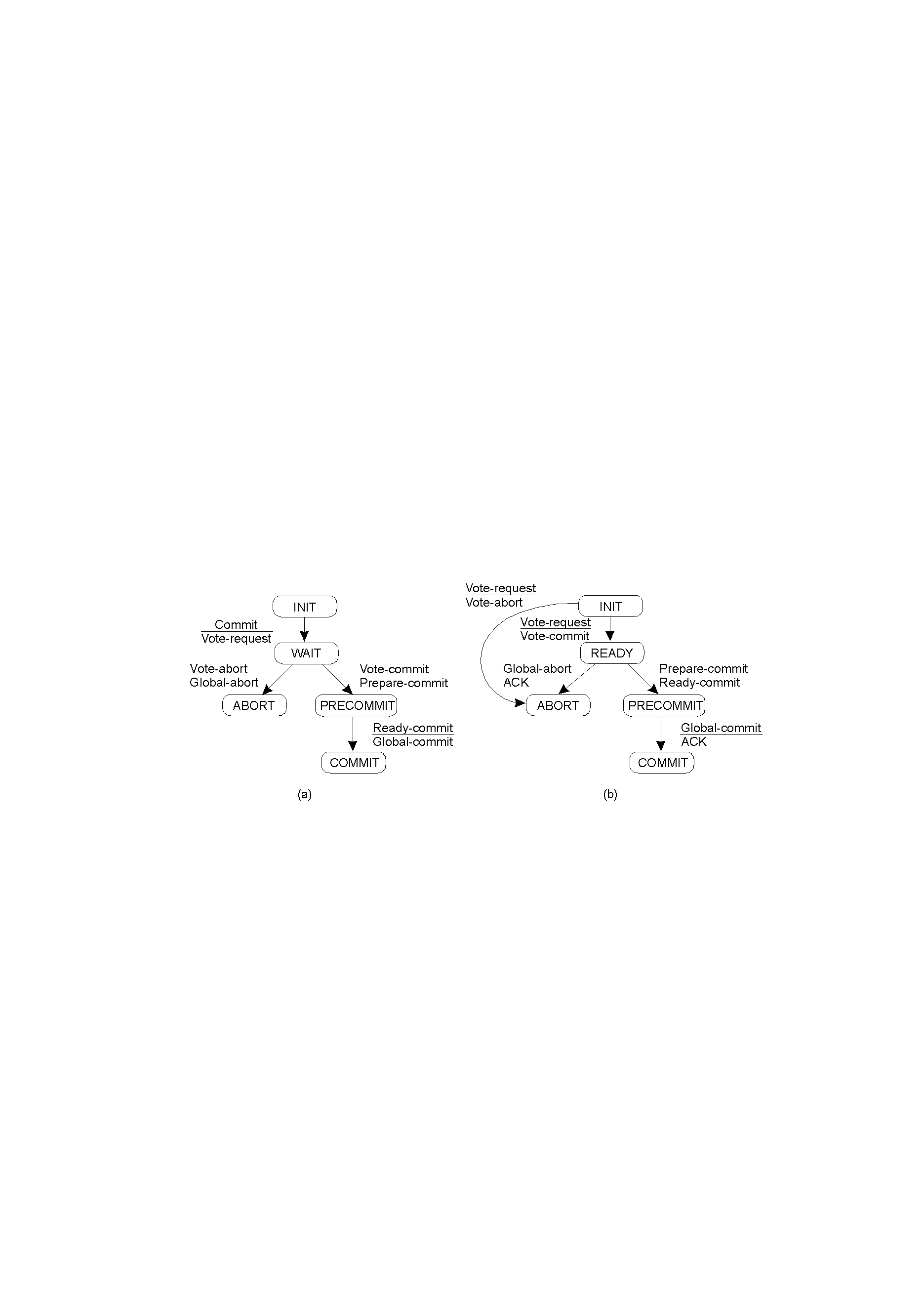
**2． 三阶段提交（3PC）**

3PC的本质在于协调者和每个参与者都满足以下两个条件：

* + 没有一个可以直接转换到Commit或者Abort状态的单独状态。
  + 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。

三阶段提交协议(3PC)的基本原理为：在2PC的参与者投票和协调者决策之间增加了“预提交”阶段。协调者在接收到所有参与者的提交票后发送一个全局预提交命令，当参与者接收到全局预提交命令之后，它就得知其他的参与者都投了提交票，从而确定自己在稍后肯定会执行提交操作，除非它失败了。每个参与者都对全局预提交发出确认消息，协调者一旦接收到所有参与者的确认消息就再发出“全局性提交”。3PC协议在站点失败，甚至是所有的站点都失败的情况下也不会带来阻塞。

它们各自的状态机如图所示。（a 协调者，b 参与者）



**二者比较：**

与2PC相比，3PC的主要不同点在于以下情况：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。

**八：代码迁移**

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。其基本的思想是：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。（迁移的是计算程序本身，而非数据）

分类（强弱）：

* + 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。
    2. 强可移动性(strong mobility)：它还可以迁移执行段。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。

分类（主动方）：

1. 发送者启动(sender-initiated)迁移：在这种模型中，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。一般来说，在向计算服务器上载程序时进行的就是发送者启动的迁移。
2. 接收者启动(receiver-initiated)迁移：代码迁移的主动权掌握在目标机器手中。Java小程序是这种迁移的一个例子。

**九：移动定位问题**

用于用户友好命名系统不适合经常移动的实体。使用位置无关的标识符可以更为有效地实现对移动实体的定位。大体上有4种用于定位移动实体的方法。

* + 第1种方法是使用**广播和多播**。实体的标识符被广播给分布式系统的每个进程。为实体提供访问点的进程通过提供访问点的地址进行响应。

缺点： 随着网络的膨胀，广播变得低效，浪费带宽。

可扩展性有限。

* + 第2种方法是使用**转发指针**。每当一个实体转移到另一个位置时，它就会留下一个指针，说明它下一步所在的位置。定位实体需要遍历转发指针形成的路线。为了避免形成太长的指针链，定期缩短指针链是很重要的。

缺点： 指针链可能会过长，性能问题，定期缩短指针链是很重要的。

还要维护中间位置的转发指针链。链很脆弱。

* + 第3种方法就是**给实体指定一个起始位置**。每当实体转移到另一个地方时，它都会通知起始位置，告诉起始位置自己当前的位置。在定位实体时，首先询问起始位置，以便了解实体的当前位置。

缺点： 大型网络中，起始位置可能与实体本身处于完全不同的位置，通信延迟。

必须保证起始位置始终存在。

* + 第4种方法是**创建一颗搜索树**。网络划分成不重叠的域。域可以组合成更高层的(不重叠)域，以此类推。只有一个顶级域，它覆盖了整个网络。每个层次的每个域都有关联的目录节点。如果一个实体位于域D中，那么更高一层的域所用的目录节点将拥有一个指向D的指针。最低层目录节点存储了实体的地址。最高层的目录节点知道所有的实体。

**缺点**： 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，瓶颈。

十：**以客户为中心的一致性模型**

**以客户为中心**的一致性为单一的客户提供一致性保证，保证该客户对数据存储的访问的一致性。它并不为不同客户的并发访问提供任何一致性保证。

**以下四个模型:**

1.单调读一致性(monotonic-read consistency)：

* + 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。

也就是说，单调读一致性保证，如果一个进程已经在t时刻看到x的值，那么以后他不再会看到较老的版本的x的值。

2. 单调写一致性(monotonic-write consistency):保证写操作以正确的顺序传播到数据存储的所有拷贝。其应该满足以下条件：

* + 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。

3.写后读一致性(read-your-writes consistency)。

* + 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。

也就是说，一个写操作总是在同一进程执行的后续读操作之前完成，而不管这个后续的读操作发生在什么位置。

4. 读后写一致性(writes-follow-reads consistency): 更新是作为前一个读操作的结果传播的。

* + 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。

也就是说，进程对数据项上x所执行的任何后续的写操作都会在x的拷贝上执行，而该拷贝是用该进程最近读取的值更新的。

**十二：分布式系统并发理论（Petri网）**

**问题描述：**有五个哲学家围坐在一圆桌旁，桌子中央有一盘通心面，每人面前有一只空盘子，每两人之间放一把叉子。每个哲学家思考、饥饿、然后，欲吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子。

T1



释放筷子T3

获取筷子T2

**图解：**

h-hunger k-thinking f –fork/chopstick e-eating

1.圆圈：hi,ki,ei,fi为库所

h：表示为哲学家饥饿状态库所 : k表示为哲学家思考状态的库所 e:表示为哲学家吃饭状态的库所 f:表示筷子处于备用状态的库所

2.黑色实心点为token。初始状态时：f0-f4中的是筷子，k0-k4中的是哲学家

3．方框为变迁

初始状态：筷子备用状态，处于f0-f4库所中。哲学家是思考状态，处于k0-k4中。

以哲学家0为例

T1 变迁1（感到饥饿）：哲学家在思考中感到饿了，从库所k0经过T1到达库所h0。哲学家从思考状态进入饥饿状态。

T2变迁2（获得筷子）：哲学家和筷子分别从h0，f4，f0库所经过T2迁移到库所e0。哲学家从饥饿状态进入吃饭状态。

T3变迁3（释放筷子）：吃完饭后，哲学家经过T3迁移到k0库所中，从吃饭状态转入思考状态。两只筷子经过T3，分别迁移到库所f0，f4，从使用状态进入备用状态。

**生产者消费者问题：**

生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。

消费者线程从缓冲区中获得物品，然后释放缓冲区。

当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放出一个空缓冲区。

当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻塞，直到新的物品被生产出来。

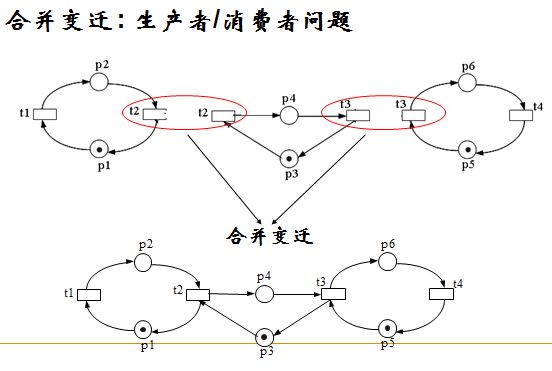


变迁 Transition

库所 Place

托肯 Token

弧 Arc



**十三：移动Agent系统通信失效问题**

* 分析
  + 无论本地通信还是异地通信，其通信模型是一致的
  + 通信失效本质上都是因为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，而这种变化是随机的，不可预计的。
* 结论
  + 通信失效现象并不是mogent系统特有的现象.它是一个因agent的移动而带来的可能会出现在任意一个移动agent系统中的普遍现象
* 其它系统的处理
  + Aglets：未见
  + Mole：
    - 将失效信件扔弃;
    - 扔弃的同时回送错误信息
    - 将失效信件就地保存,待被通信agent在回送时交付
* 根本原因：
  + 移动：随机改变位置信息
  + 通信：要求位置信息“暂时”不变
  + 通信和移动所共享的“位置”信息未进行同步控制是造成通信失效的根本原因
  + 从OS进程互斥考虑，接收者的“位置”在通信失效问题中具有决定性的意义，当通信和移动相矛盾时，该“位置”就成为了一个必须互斥使用的“资源”
* “位置”的互斥 ＝》 “状态”的互斥
* 在一个能够避免通信失效的移动agent系统中，必须且只要做到以下三条 ：
  + 准确纪录agent的状态信息
  + 只能向一个处于“静止态”的agent发送信件
  + 信件发送过程中必须限制接收者从“静止态”向“移动态”的状态转换

Mogent解决方案：

刻画agent状态的数据结构设计

agent状态值使用的“临界区”界定

agent状态的“临界区”管理

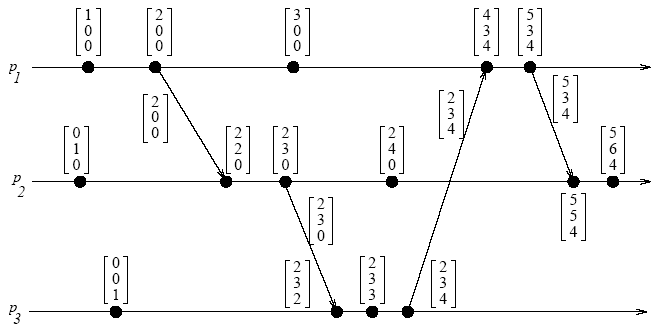
* 刻画agent状态的数据结构设计
  + 增添Mogent状态值：迁移态、静止态
  + Home将维护mogent状态值的变化
* Mogent状态值的使用
  + mogent在某个结点上执行时,其状态值定义为静止态
  + mogent按计划准备向另一个结点迁移时, 必须通知该mogent的Home, Home收到通知后应将状态值改为移动态
  + mogent到达某个新结点时,新结点必须向mogent的Home 通知该mogent的到达,Home收到通知,将状态值改为静止态

十四：同步Lamport逻辑时钟，**向量时间戳**

* 为了同步逻辑时钟，Lamport定义了一个称作 “先发生” (happens-before) 的关系。表达式a🡪b读作 “a在b之前发生”，意思是所有进程一致认为事件a先发生，然后事件b才发生。这种先发生关系有两种情况。
  + 如果a和b是同一个进程中的两个事件，且a在之前发生，则a🡪b为真。
  + 如果a是一个进程发送消息的事件，而b为另一个进程接收消息的事件，则a🡪b也为真。消息不可能在发送之前被修改，也不能在发送的同时被接收，这是因为消息需要一定时间才能到达接收端。
  + Lamport逻辑时间钟具有传递性和并发性；
* 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。
* 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。这样，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。
* 使用这种方法，我们现在有了一个为分布式系统中的所有事件分配时间的方法，它遵循下面的规则：
  + 若同一进程中a在b之前发生，则C(a)<C(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)。
  + 对于所有不同的事件a和b，C(a) ≠ C(b)。
* 这个算法为我们提供了一种对系统中所有事件进行完全排序的方法。许多其他的分布式算法都需要这种排序以避免混淆，所以文献中广泛引用此算法。

**为什么采用向量时间戳可以表示事件因果关系？**

* 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，该向量具有下面两个性质：
  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
  3. 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。
  4. 第二个性质时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。
* 使用这种方式，接收者可以得知进程Pi中已经发生的事件数。
* 更重要的是，接收者可以得知进程Pi发送消息m之前其他进程已经发生了多少个事件。换句话说，消息m的时间戳vt告诉接收者其他进程中有多少事件发生在它之前，并且消息m可能在因果关系上依赖于这些事件。
* 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。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。



补充：

拜占庭将军问题

通信良好，进程确不好，正如拜占庭的将军有叛徒的情况。

在这个问题中，红军还是在山谷中扎营，但是在附近的山上有n个带领部队的蓝军将领。通信是通过电话双向进行的，及时而且质量很好。但是有n个将军是叛徒(故障)，他们通过给忠诚的将军发送错误的和矛盾的信息(模拟故障进程)来阻止他们达成协议。现在问题在于忠诚的将军是否还能达成协议。

Lamport采用递归算法来让无故障的进程达成协议。

N个进程，其中M个故障进程：只有当N>=3m+1时，才能达成一致，即有三分之二的进程是正常的。

拜占庭问题的最初描述是：n 个将军被分隔在不同的地方，忠诚的将军希望通过某种协议达成某个命令的一致（比如一起进攻或者一起后退）。但其中一些背叛的将军会通过发送错误的消息阻挠忠诚的将军达成命令上的一致。Lamport 证明了在将军总数大于3m ，背叛者为m 或者更少时，忠诚的将军可以达成命令上的一致。

为了保证上面的需求，必须满足下面两个条件：

1. 每两个忠诚的将军必须收到相同的值 v(i)（v(i)是第i 个将军的命令）

2. 如果第i 个将军是忠诚的，那么他发送的命令和每个忠诚将军收到的v(i)相同

为了简化以上模型，我们使用一个将军发送命令给多个副官的形式来证明，发送命令的将军称为发令者，接收命令的将军为副官，那么上面的两个条件可以表述为：

IC1. 所有忠诚的副官遵守相同的命令

IC2. 如果发出命令的将军是忠诚的，那么所有忠诚的副官遵守司令（发出命令的将军）的命令

特别提示：发送命令的每次只有一个将军，将其命令发送给n-1 个副官。m 代表叛国者的个数，因为将军总数为n，所以副官总数为n-1 个。IC2 中副官遵守实际上是指忠诚的将军能够正确收到忠诚将军的命令消息。

通过口头消息

通过口头消息传递达到一致，如果有m 个叛国将军，则将军们的总数必须为3m+1 个以上。下面是口头消息传递过程中默认的一些条件：

A1. 每个被发送的消息都能够被正确的投递

A2. 信息接收者知道是谁发送的消息

A3. 能够知道缺少的消息

A1 和A2 假设了两个将军之间通信没有干扰，既不会有背叛者阻碍消息的发送（截断）也不会有背叛者伪造消息的情况（伪造）。即是每个将军都可以无误地将自己的消息发送给其他每个将军。（下一节中可以不需要这个必要条件）

我们定义口头消息[算法](http://baike.baidu.com/view/7420.htm" \t "_blank)OM(m) 。对于所有的非负整数m ，每个发令者通过OM(M) 算法发送命令给n-1 个副官。下面将说明OM(m) 算法在最多有m 个背叛者且总将军数为3m+1 或者更多的情况下可以解决拜占庭将军问题。（考虑到网络应用实际环境，原文使用了收到值代替收到命令，本文不做修改）

算法定义一个函数：majority(com1,com2,…,comn)等于多数派命令。

OM(0)[算法](http://baike.baidu.com/view/7420.htm" \t "_blank)

（1）发令者将他的命令发送给每个副官。

（2）每个副官采用他从发令者发来的命令，或者默认使用撤退命令，如果没有收到任何命令。

OM(m)[算法](http://baike.baidu.com/view/7420.htm" \t "_blank)

（1）发令者将他的命令发送给每个副官。

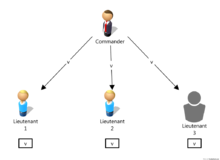
（2）对于每个i ，vi 是每个副官i 从发令者收到的命令，如果没有收到命令则为撤退命令。副官i 在OM(m-1) 中作为发令者将vi 发送给另外n-2 个副官。

（3）对于每个i，并且j\neq i，vj 是副官i 从第（2）步中的副官j 发送过来的命令（使用OM(m-1)[算法](http://baike.baidu.com/view/7420.htm" \t "_blank)），如果没有收到第（2）步中的副官j 的命令则默认为撤退命令。最后副官i 使用majority(v1,…,vn-1)得到命令。

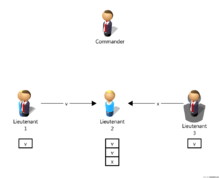
接下来实际的考虑一个n=4，m=1 的情况：

1. 当副官3 是背叛者，以副官2 为例

第一步发令者执行算法OM(1)将自己的命令发送给三个副官，三个副官都正确地收到了命令。

[](http://baike.baidu.com/picview/2815670/2815670/0/ac345982b2b7d0a29fc2ec3ccbef76094b369a0c.html)

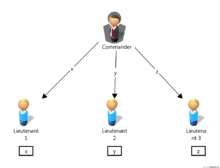
第二步每个收到命令的副官都作为发令者执行算法OM(0)，将自己的命令转发给其余副官，因为副官3是背叛者，所以他给副官2 传递的消息会是假消息。副官2 最后根据majority 函数来决定命令。

[](http://baike.baidu.com/picview/2815670/2815670/0/c83d70cf3bc79f3d1c960c91baa1cd11728b2918.html)

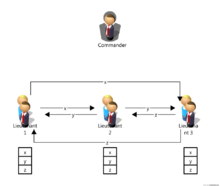
这样背叛的副官3 同理也干扰不了忠诚的副官1 和发令者的决定。下面看看如果发令者是背叛者。

2. 发令者是背叛者，其余副官为忠诚的

第一步：发令者向副官发送了不同的命令，实际情况中是一个攻击者向不同方发送了不同的值进行欺骗。

[](http://baike.baidu.com/picview/2815670/2815670/0/203fb80e7bec54e7959cf2bfb9389b504fc26a1b.html)

第二步：副官收到命令后，摇身一变为发令者执行OM(0) 向所有的副官发送命令，这样所有副官的到命令会都相同（<x,y,z>），这样所有命令都达不到大部分。

[](http://baike.baidu.com/picview/2815670/2815670/0/e1fe9925bc315c608af7834e8db1cb1349547735.html)

[1]

文章接着就证明了OM(m)[算法](http://baike.baidu.com/view/7420.htm" \t "_blank)对任意m 都是可以满足，首先引入一个引理（归纳法证明）：

引理1：对于任意m 和k ，如果有超过2k+m 个将军和最多k 个背叛者，那么算法OM(m) 满足IC2 （回顾下IC2 指的是，如果将军是忠诚的，所有的副官遵守将军命令）。

证明：当m=0 的时候，OM(0) 在将军是忠诚的时候满足IC2。当m>0 时，首先将军将命令传递给 n-1 个副官，然后每个副官对n-1 个将军执行OM(m-1) 算法。因为假设了n>2k+m（引理中有将军数大于2k+m），所以 n-1 > 2k+(m-1) >= 2k（即每一轮中副官总数不小于背叛者的两倍），这样每轮OM(m-k) 算法中忠诚的副官收到的命令都是majority(v1,v2,...,v(n-1))，其中忠诚副官发送的命令大于或者等于一半。

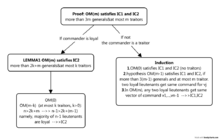
接着解决拜占庭将军问题。

定理1：对于任意m，如果有超过3m 个将军和最多m 个背叛者，算法OM(m) 满足条件IC1 和条件IC2。

证明：通过m 的归纳法证明，我们通过假设OM(m-1) 成立来证明OM(m) m>0。首先考虑发送命令的将军是忠诚的。那么将引理中k 设为m 则OM(m) 满足IC2 ，IC1 在发令将军是忠诚的情况下也满足。

接着考虑m 个背叛者中有一个是发令者，那最多就只有m-1 个副官是背叛者了，又因为有3m 个将军，所以副官的总数超过3m-1，且有3m-1>3(m-1) 。因此通过归纳法假设 OM(m-1) 满足IC1 和IC2（最多3(m-1) 个将军和最多m-1 个背叛者）。那么任意两个忠诚的副官j 在OM(m-1) 获得相同命令vj，那么在OM(m) 算法中每个忠诚的副官都会收到(v1,v2,...,\v(n-1))，可知满足条件IC1 和IC2。

看完这段证明其实我也挺糊涂的~~~~，画了张图来看看lamport 是怎么证明的：

[](http://baike.baidu.com/picview/2815670/2815670/0/a9d3fd1f4134970ae83df88e95cad1c8a6865dc1.html)

通过签名消息

签名消息在除了满足口头消息A1-A3 三点要求外还应该满足下面A4：

A4 （a）签名不可被伪造，一旦被篡改即可发现

（b）任何人都可以验证将军签名的可靠性

下面定义一个类似于上面majority() 函数的choice() 来决定副官的选择：1.当集合V 只包含了一个元素v ，那么choice(V)=v ；2. choice(o)=RETREAT。

有了上面A4 和choice() 基础我们给出SM(m) 方法：

SM(m) 算法

初始化Vi=空集合

（1）将军签署命令并发给每个副官

（2）对于每个副官i ：

（A）如果副官i 从发令者收到v:0 的消息，且还没有收到其他命令序列，那么：

（i）使Vi 为{v}

（ii）发送v:0:i 给其他所有副官

（B）如果副官i 收到消息v:0:j1:...:jk 且v 不在集合Vi 中则

（i）添加v 到Vi

（ii）如果k<m ，那么发送v:0:j1:...:jk:i 给每个不在j1,..,jk 中的副官

（3）对于每个副官i ，当不再收到消息，则遵守命令choive(Vi)

算法的几点说明：

算法第三步并没有说明副官如何判断没有新的消息，可以通过两个解决方法：一是要求每个副官收到消息后要么签名转发该消息，要么发送一个通知他将不发送这个消息；二是设置一个超时时间，如果在该时间段还没有收到消息，则认为不再收到消息。

[算法](http://baike.baidu.com/view/7420.htm" \t "_blank)SM(m) 中，让副官签名是确认其收到了该消息（有点像来了份文件给每个领导批阅）。在SM(1) 中副官收到消息就不用签字了，因为不用转发给其他副官。

定理2：对于任意m，最多只有m 个背叛者情况下，算法SM(m) 解决拜占庭将军问题

Proof：首先证明IC2，如果发令者是忠诚的，那么所有的副官一定收到相同的命令，因为签名无法篡改，且IC1 也就满足了。证明IC1 只用考虑发令者是背叛者的情况（重新回顾下IC1 是指所有忠诚的副官执行相同的命令），IC1 要求两个忠诚的副官（i，j）必须收到相同的命令集合Vi、Vj，也就是Vi 中每个v 都在Vj 中。会存在两种情况，其一当副官i 收到v 命令的序列后，加入到Vi，并将其发送给副官j ，j 将命令v 保存；其二副官i 收到命令v:0:j1:...:jk:i，其中有jr=j，所以 由A4 可知副官j 也收到了该命令。如果没有，则有：

k<m。这种情况下，i 发送信息v:0:j1:...:jk:i 给副官j ，那么j 一定收到v 。（这点感觉和上面有重复）

k=m。发令者是背叛者，最多只有m-1 个副官是背叛者。因此，最少有一个序列j1,...,jm是忠诚的。那么j 一定收到这个忠诚的副官序列发来的值v ，所以副官j 收到v 。

证明完毕。