一致性和复制

分布式系统中的一个重要问题是数据的复制。通常复制数据是为了增强可靠性或提高性能。其中一个主要问题是保持副本的一致性。非正式地说，这意味着当一个副本被更新时，我们需要确保其他副本也被更新;否则，副本将不再相同。在本章中，我们将详细讨论复制数据的一致性实际上意味着什么，以及实现一致性的各种方法。

我们从一个一般性的介绍开始，讨论为什么复制是有用的，以及它与可伸缩性的关系。然后我们继续关注一致性到底意味着什么。一致性模型的一个重要假设是多个进程同时访问共享数据。当读取和更新共享数据时，知道其他人也在访问该数据时，可以根据进程的期望来制定这些情况的一致性。

在大规模分布式系统中，共享数据的一致性模型通常很难有效地实现。此外，在许多情况下可以使用更简单的模型，这些模型通常也更容易实现。一个特定的情况是由以客户端为中心的一致性模型构成的，该模型从单个(可能是移动的)客户端角度关注一致性。

一致性只是问题的一半。我们还需要考虑一致性实际上是如何实现的。我们基本上需要考虑两个或多或少独立的问题。首先，我们将重点放在管理副本上，它不仅考虑副本服务器的位置，还考虑内容如何分布到这些服务器。

第二个问题是副本如何保持一致。在大多数情况下，应用程序需要很强的一致性。非正式地说，这意味着更新将在副本之间或多或少地立即传播。有多种实现强一致性的方法，我们将在单独的部分中讨论。此外，还注意了缓存协议，它是一致性协议的一种特殊情况。

作为最大的分布式系统，我们分别关注基于web的系统中的缓存和复制，特别是内容交付网络和边缘服务器缓存技术。

7.1 简介

在本节中，我们首先讨论复制数据的重要原因。我们专注于将复制作为一种实现可伸缩性的技术，并解释了为什么一致性的推理如此重要。

**复制的原因**

复制数据有两个主要原因。首先，复制数据以提高系统的可靠性。如果一个文件系统已经被复制，那么在一个副本崩溃之后，只要切换到另一个副本，就可以继续工作。此外，通过维护多个副本，可以更好地保护数据免受损坏。例如，假设有一个文件的三个副本，每个副本都执行读写操作。通过将至少两个副本返回的值视为正确的值，我们可以防止一次失败的写操作。

复制数据的另一个原因是性能。当分布式系统需要按大小或所覆盖的地理区域进行伸缩时，性能复制非常重要。当越来越多的进程需要访问但一个服务器时，就会产生性能问题。在这种情况下，可以通过复制服务器并在访问数据的进程之间分配工作负载来改进性能。

相对于地理区域的扩展也可能需要复制。其基本思想是，通过将数据副本放在使用它们的进程附近，访问数据的时间就会减少。因此，该进程所感知到的性能会增加。这个例子还说明，复制对性能的好处可能很难评估。尽管客户机进程可能会感知到更好的性能，但也可能会消耗更多的网络带宽来保持所有副本都是最新的。

如果复制有助于提高可靠性和性能，那么谁会反对呢？不幸的是，在复制数据时要付出代价。复制的问题是，拥有多个副本可能会导致一致性问题。每当修改一个副本时，该副本就与其他副本不同。因此，必须对所有副本进行修改，以确保一致性。究竟什么时候以及如何进行这些修改决定了复制的价格。

要理解这个问题，可以考虑改进Web页面的访问时间。如果不采取特殊措施，从远程Web服务器获取页面有时甚至需要几秒钟。为了提高性能，Web浏览器通常在本地存储以前获取的Web页面的副本。如果用户再次需要该页面，浏览器将自动返回本地副本。用户感知的访问时间非常好。然而，如果用户总是想要一个页面的最新版本，这种方法可能不会生效。问题是，如果同时修改了页面，修改将不会传播到缓存副本，从而使这些副本过期。

将过期副本返回给用户的一个解决方案是首先禁止浏览器保存本地副本，从而有效地让服务器完全负责复制。但是，如果在用户附近没有放置副本，则此解决方案仍然可能导致较差的访问时间。另一个解决方案是让Web服务器使每个缓存副本失效或更新，但这要求服务器跟踪所有缓存并向它们发送消息。这进而可能降低服务器的整体性能。下面我们回到性能与可伸缩性的问题。

在接下来的文章中，我们将主要关注性能方面的复制。第8章讨论了可靠性的复制。

**Replication as scaling technique**

为提高性能而进行的复制和缓存作为扩展技术得到了广泛的应用。可伸缩性问题通常以性能问题的形式出现。将数据副本放在使用它们的进程附近可以通过减少访问时间提高性能，从而解决可伸缩性问题。

需要做出的一个可能的权衡是，使副本保持最新可能需要更多的网络带宽。考虑一个进程P每秒访问一个本地副本N次，而副本本身每秒更新M次。假设更新完全刷新本地副本的前一个版本。如果N << M，也就是说，访问更新比非常低，我们就会遇到这样的情况:P永远不会访问本地副本的许多更新版本，这使得这些版本的网络通信毫无用处。在这种情况下，最好不要安装接近P的本地副本，或者应用不同的策略来更新副本。

然而，一个更严重的问题是，保持多个副本的一致性本身可能存在严重的可伸缩性问题。直观地说，当副本总是相同时，副本集合是一致的。这意味着在任何副本上执行的读取操作总是返回相同的结果。因此，当对一个副本执行更新操作时，无论在哪个副本上启动或执行该操作，应该在后续操作发生之前将更新传播到所有副本。

这种类型的一致性有时称为非正式地(而且不精确地)一致性，由同步复制提供。(在第7.2节中，我们将提供一致性的精确定义，并介绍一系列一致性模型。) 关键思想是，更新作为单个原子操作或事务在所有副本上执行。不幸的是，当操作也需要快速完成时，实现涉及大量副本的原子性本质上是困难的，这些副本可能广泛分布在大型网络中。

困难在于我们需要同步所有副本。实际上，这意味着所有副本首先需要就何时在本地执行更新达成一致。例如，副本可能需要使用Lamport时间戳决定操作的全局顺序，或者让协调器分配这样的顺序。全局同步只需要大量的通信时间，特别是当副本分布在广域网中时。

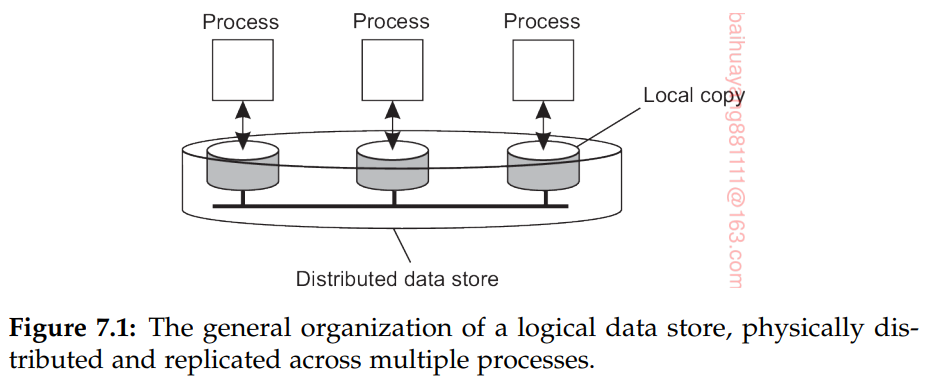
我们现在面临着进退两难的局面。一方面，通过应用复制和缓存可以缓解可伸缩性问题，从而提高性能。另一方面，为了保持所有副本的一致性，通常需要全局同步，这在性能方面有内在的代价。

在许多情况下，唯一真正的解决方案是放松一致性约束。换句话说，如果我们能够放松更新需要作为原子操作执行的要求，我们就能够避免(瞬时)全局同步，从而获得性能。付出的代价是，副本可能并不总是在所有地方都一样。事实证明，可以在多大程度上放松一致性，在很大程度上取决于对复制数据的访问和更新模式，以及使用这些数据的目的。

有一系列的一致性模型，以及通过所谓的分布和一致性协议实现模型的许多不同方法。

7.2以数据为中心的一致性模型

传统上，一致性是在共享数据的读写操作上下文中讨论的，这些操作通过(分布式)共享内存、(分布式)共享数据库或(分布式)文件系统来实现。这里，我们使用更广泛的术语**数据存储**。数据存储可以物理地分布在多台机器上。特别是，可以假定每个数据都存有本地副本，所以每个进程都可以进行访问。写操作被传播到其他副本，如图7.1所示。数据操作在更改数据时被分类为写操作，否则被分类为读操作。



**一致性模型**本质上是进程和数据存储之间的约定。它说，如果进程同意遵守某些规则，存储的数据承诺将正常工作。通常，对数据项执行读操作的进程期望该操作返回一个值，该值显示上次对该数据执行写操作的结果。

在没有全局时钟的情况下，很难精确地定义哪个写操作是最后一个写操作。作为一种替代方法，我们需要提供其他定义，从而产生一系列一致性模型。每个模型都有效地限制数据项上的读取操作可以返回的值。正如预期的那样，具有主要限制的应用程序很容易使用，例如在开发应用程序时，而具有次要限制的应用程序通常被认为很难在实践中使用。当然，代价是，易于使用的模型的性能不如难使用的模型。生活就是这样。Such is life~

**连续的一致性**

没有复制数据的最佳解决方案。复制数据会带来一致性问题，而这些问题通常无法有效地解决。只有我们放松一致性，才能有希望获得有效的解决方案。不幸的是，也没有放松一致性的一般规则:确切的说，只有高度依赖的应用程序才是可以容忍一致性的。

应用程序有不同的方法来指定它们可以容忍哪些不一致。Yu和Vahdat[2002]采用了一种通用的方法，通过区分三个独立的轴来定义不一致性:副本之间数值的偏差、副本之间陈旧程度的偏差以及更新操作顺序的偏差。他们将这些偏差称为形成**连续一致性**范围。

数据具有数值语义的应用程序可以使用数值偏差来测量不一致性。一个明显的例子是复制包含股票市场价格的记录。在这种情况下，应用程序可以指定两个副本的偏差不应超过0.02美元，这将是一个绝对的数值偏差。或者，可以指定一个相对的数值偏差，说明两个副本之间的差异不应超过0.5%。在这两种情况下，如果股票上涨(并且其中一个副本立即更新)而没有违反指定的数值偏差，那么副本仍然被认为是相互一致的。

数值偏差也可以理解为已应用于给定副本但尚未被其他人看到的更新的数量。例如，Web缓存可能没有看到由Web服务器执行的一批操作。在本例中，值的相关偏差也称为其权重。

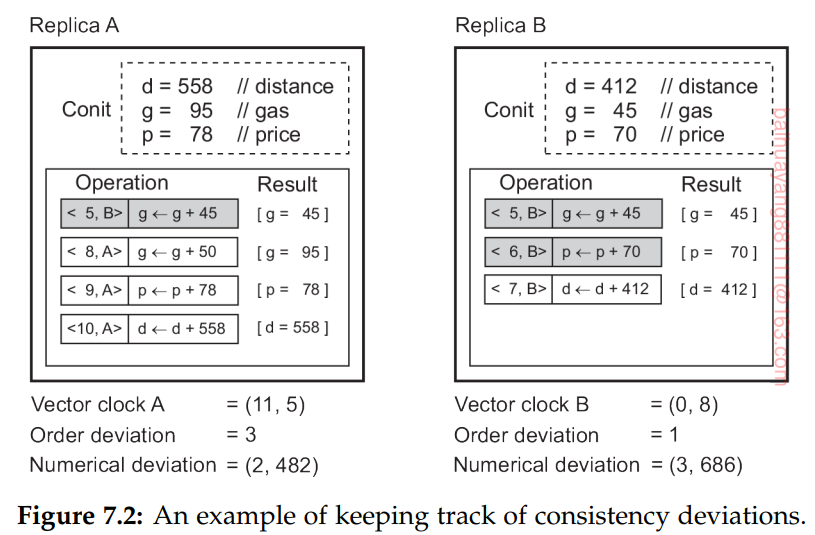
新旧偏差与副本更新的最后时间有关。对于某些应用程序，只要旧数据不太旧，就可以容忍副本提供旧数据。例如，天气报告通常在一段时间内保持相当准确，比如几个小时。在这种情况下，主服务器可能会收到及时的更新，但是可能决定只在一段时间内将更新传播到副本。

最后，在一些 应用程序类中，允许在不同的副本上执行不同的更新次序，只要这些差别是有限度的。查看这些更新的一种方法是，它们暂时应用于本地副本，等待所有副本的一起同步。因此，一些更新可能需要回滚并以不同的顺序应用，然后才能成为永久性的。直观地说，顺序偏差比其他两个一致性度量更难理解。

The notion of a conit

为了定义不一致性，Yu和Vahdat引入了一个**一致性单元**，缩写为**conit。**conit指定要测量一致性的单位。例如，在我们的股票交易所示例中，conit可以定义为表示单个股票的记录。另一个例子是单独的天气报告。

为了给出一个conit的例子，并同时说明数值和顺序偏差，请考虑跟踪车队的情况。特别是，车队老板想知道他平均为汽油付多少钱。为此，每当司机加满汽油时，他都会报告加满的汽油量(记为g)、支付的价格(记为p)以及自上次加满以来的总距离(记为变量d)。从技术上讲，g、p和d这三个变量构成了一个conit。这个conit被复制到两个服务器上，如图7.2所示，司机通过单独更新每个变量，定期向其中一个服务器报告他的汽油使用量(无需进一步考虑问题车辆)。



服务器的任务是保持conit的“一致性”被复制。为此，每个复制服务器维护一个二维矢量时钟。我们使用符号<T,R>， Ri来表示由副本R在(其)逻辑时间T执行的操作。在本例中，我们看到两个副本操作一个conit，其中包含本例中的数据项g、p和d。假设所有变量都初始化为0。副本A从副本B收到操作：



我们用灰色阴影表示该操作，表示A已将该操作提交到其本地存储。换句话说，它已经成为永久性的，不能回滚。副本A还列出了三个暂定的更新操作：<8,A>,<9,A>和<10,A>。就连续一致性而言，A有三个待提交的暂定操作，order deviation是3。类似地，在总共进行了三次操作(其中两次操作)的情况下，B的顺序为1。

从这个例子中，我们看到A的逻辑时钟值现在是11。我们假设向量的第一个分量用于A，第二个分量用于B。因为A接收到的B的最后一个操作的时间戳是5，所以A处的向量时钟将是(11,5)。同理，B处的逻辑时钟为(0,8)。在一个复制R处的数值偏差由两个部分组成:在所有其他尚未被R看到的复制上的操作数，以及相应的缺失值的总和(当然，也可能有更复杂的方案)。（就是别的机器的操作还没看到）。在我们的例子中，A看不到<6,B>和<7,B>操作，总共两个操作，值的和为70+412=486，所以值差为(2,482)。同样的，B也看不到A的三个操作，总差值是686，所以B的值差为(3,686)。

使用这些概念，可以指定特定的一致性方案。例如，我们可以通过指定一个可接受的最大值来限制这些偏差。同样，我们可能希望两个副本的数值偏差不超过1000。拥有这样的一致性方案确实需要一个副本知道它与其他副本的差异有多大，这意味着我们需要单独的通信来通知副本。基本的假设是，这种通信比保持副本同步的通信要便宜得多。诚然，如果这个假设也适用于我们的例子，那就值得怀疑了（怀疑啥？）。

虽然从概念的角度来看conits是捕获一致性需求的一种吸引人的方法，但是在将它们投入实际使用之前，有两个重要的问题需要处理。首先，为了加强一致性，我们需要协议。本章后面将讨论连续一致性协议。第二个问题是程序开发人员必须为他们的应用程序指定一致性需求。实践表明，获得这些需求可能极其困难。程序员通常不习惯处理复制，更不用说理解提供关于一致性的详细信息意味着什么。因此，必须有简单易懂的编程接口。

**操作顺序一致性**

在过去的几十年里，有大量以数据为中心的一致性模型的工作。一类重要的模型来自并行编程领域。在并行和分布式计算中，多个进程需要同时共享资源和访问这些资源，面对这一事实，研究人员试图在复制共享资源时表达并发访问的语义。我们在这里讨论的模型都处理对共享、复制数据的一致排序操作。原则上，当需要提交副本上的临时更新时，副本需要就全局(即更新的一致顺序)达成一致，因此模型增强了连续一致性。

**顺序一致性**

在下面，我们将使用一个特殊的符号，在这个符号中，我们沿着时间轴绘制进程的操作。时间轴总是水平绘制的，时间从左到右递增。我们使用符号Wi(x)来表示进程Pi将值a写入数据项x。类似地，Ri(x)b表示进程读取x并返回值b。我们假设每个数据项的初始值为NIL。当没有混淆哪个进程正在访问数据时，我们从符号W和R中省略索引。

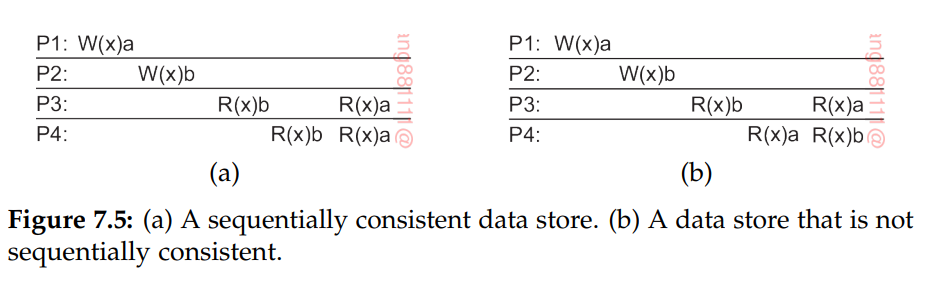
在图7.4中，p1向x写入数据a。注意，根据我们的系统模型，W1(x)a操作首先在P1本地的数据存储副本上执行，然后才传播到其他本地副本。在我们的示例中，P2稍后读取值NIL，在此之后的一段时间内读取值a(从存储的本地副本)。我们在这里看到的是，将x的更新传播到P2花费了一些时间，这完全可以接受。

**顺序一致性**是一种重要的以数据为中心的一致性模型，Lamport[1979]在多处理器系统共享内存的上下文中首次定义了该模型。当数据存储满足以下条件时，它被称为顺序一致的。

*The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*

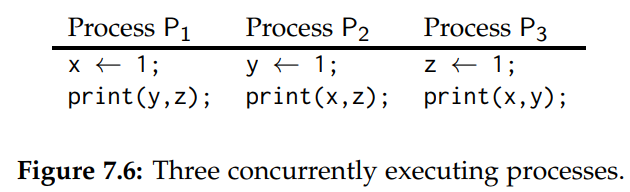
这个定义的意思是，当进程在(可能)不同的机器上并发运行时，任何有效的读写操作交错都是可接受的行为，但是所有进程都看到相同的操作交错（还是不懂，后面例子会讲解）。注意，这里没有提到时间;也就是说，数据项上没有对“最近”写操作的引用。此外，进程可以通过自己的读操作看到其他所有进程的写操作。

从图7.5中可以看出，时间没有发挥作用。考虑在相同数据项x上运行的四个进程。在图7.5（a），进程P1首先执行对x的写操作，写入a。之后（绝对时间上），进程P2同样执行x的写操作，写入b。然而，P3和P4都是先读到b值，然后读到a值。换句话说，写入b的操作先于写入a的操作。

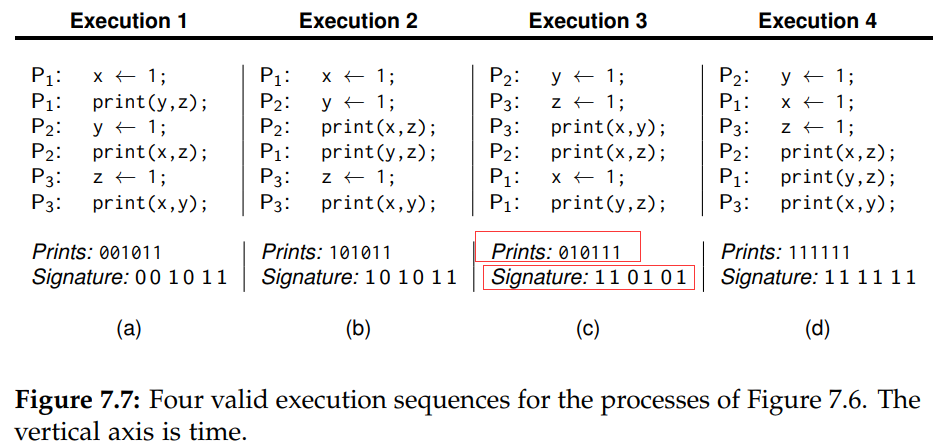


相反，图7.5(b)违反了顺序一致性，因为不是所有进程都看到相同的写操作交错。特别是在处理P3时，数据项似乎首先被更改为b，然后又被更改为a。另一方面，P4会得出最终的值是b。

为了使顺序一致性的概念更加具体，考虑三个并发执行的进程P1、P2和P3，如图7.6所示。本例中的数据项由三个整数变量x、y和z组成，它们存储在(可能是分布式的)共享的顺序一致的数据存储中。我们假设每个变量都初始化为0。在本例中，赋值对应于写操作，而print语句对应于它的两个参数的同时读取操作。所有的语句都被认为是不可分割的。



各种交错执行序列是可能的。使用6个独立语句，可能有720(6!)个可能的执行序列，尽管其中一些违反了程序顺序。考虑从x1开始的120(5!)个序列。其中一半在y =1之前已经打印(x,z)，因此违反了程序顺序。其中一半在z =1之前已经打印(x,y)，因此违反了程序顺序。因此只剩下四分之一，即30组有效的。另外30个有效序列可以从y =1开始，另外30个可以从z= 1开始，总共有90个有效的执行序列。其中的4种如图7.7所示。



在图7.7(a)中，这三个进程依次运行，首先是P1，然后是P2，然后是P3。另外三个例子展示了在时间上的交错不同但同样有效的语句。这三个进程中的每个进程打印两个变量。由于每个变量只能接受初值(0)或赋值(1)，所以每个进程生成一个2位字符串。打印后的数字是出现在输出设备上的实际输出。

如果我们按照这个顺序将P1、P2和P3的输出连接起来，就会得到一个6位的字符串，它表示语句之间的特定交错。这是作为图7.7中的签名列出的字符串。下面我们将通过每个订单的签名而不是打印输出来描述每个订单。

并不是所有64个签名模式都被允许。作为一个简单的例子，00 00 00是不允许的，因为这意味着print语句在赋值语句之前运行，违反了按程序顺序执行语句的要求。一个更微妙的例子是00 10 01。前两位00表示p打印时y和z都是0。只有当P1在P2或P3 \*之前执行两个语句时才会出现这种情况。接下来的两位，10，意味着P2必须在P1启动之后，P3启动之前运行。最后两位，01，意味着P3必须在P1开始之前完成，但我们已经看到P1必须先完成。因此，00 10 01是不允许的。

简而言之，90种不同的有效语句排序会产生各种不同的程序结果(不过小于64)，这些结果在序列一致性的假设下是允许的。进程与分布式共享系统之间的约定是进程必须接受这些结果作为有效的结果。换句话说，进程必须接受图7.7中所示的四个结果和所有其他有效结果作为有效的结果，并且必须在其中任何一个出现时正确工作。对其中一些结果有效而对其他结果无效的程序违反了与数据存储的约定，并且是不正确的。

**因果一致性**

因果一致性模型[Hutto and Ahamad, 1990]代表了弱化的顺序一致性，因为它区分了可能与因果相关的事件和不相关的事件。在前一章讨论向量时间戳时，我们已经遇到了因果关系。

如果事件b是由较早的事件a引起或影响的，因果关系要求其他人首先看到a，然后看到b。

考虑通过分布式共享数据库进行的简单交互。假设进程P1写了一个数据项x。进程p2读取x并写入y。这里读x和写y可能有因果关系，因为y的计算可能依赖于P2读x时的值。

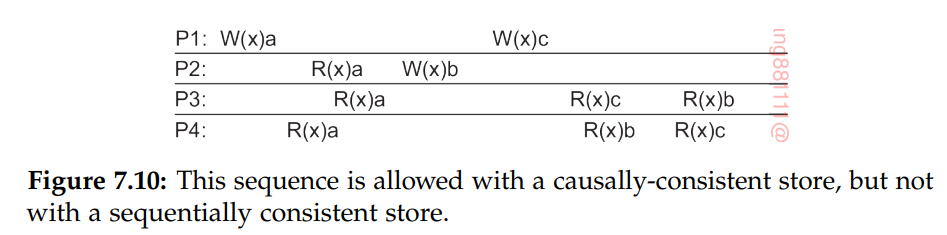
另一方面，如果两个进程同时自发地编写两个不同的数据项，则它们之间没有因果关系。没有因果关系的操作称为并发操作。要使数据存储被认为是因果一致的，必须使存储遵守以下条件：

*Writes that are potentially causally related must be seen by all processes*

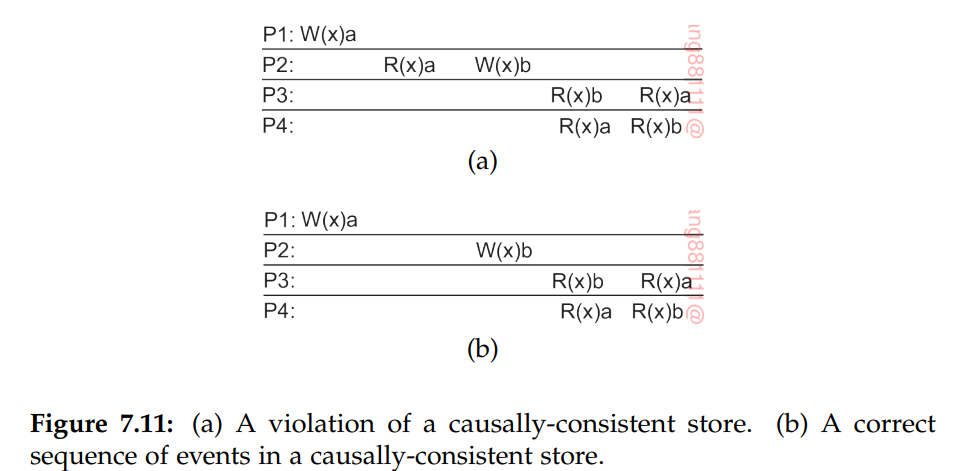
*in the same order. Concurrent writes may be seen in a different order on*

*different machines.*

作为因果一致性的一个例子，请考虑图7.10。在这里，我们有一个事件序列，它允许使用与原因一致的存储，但是禁止使用顺序一致的存储或严格一致的存储。需要注意的是，写入W2(x)b和W1(x)c是并发的，所以并不要求所有进程都以相同的顺序查看它们：



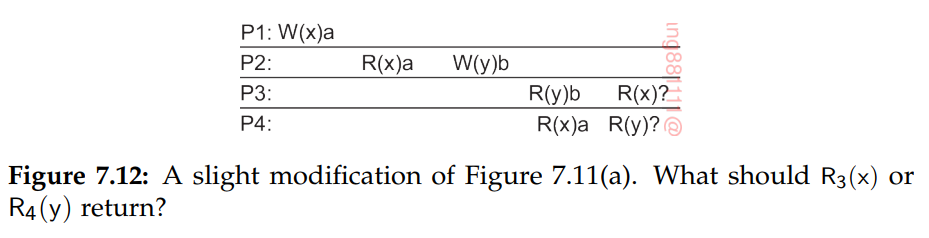
现在考虑第二个例子。在图7.11(a)中，我们有W2(x)b，它可能依赖于W1(x)a，因为将值b写到x中可能是R2(x)a先前读取值的计算结果。这两个写是因果相关的，所以所有进程必须以相同的顺序查看它们（即现在写入是p1->x(a) p2->x(b)，其他进程读取x 必须是 先a 后 b）。因此，图7.11(a)是不正确的。（3,4 必须以相同顺序读取a和b）。另一方面，在图7.11(b)中已经删除了读操作，所以W1(x)a和W2(x)b现在是并发写操作。原因一致的存储不需要对并发写进行全局排序，因此图7.11(b)是正确的。注意，图7.11(b)反映了顺序一致的存储不能接受的情况。



实现因果一致性需要跟踪哪些流程看到了哪些写入。有许多微妙的问题需要考虑。为了说明这一点，假设我们将图7.11(a)中的W2(x)b替换为W2(y)b，同样地，R3(x)b替换为R3(y)b。这种情况如图7.12所示。

首先来看R3(x)。进程P3在R3(y)b之后执行此操作。在这一点上我们确定W(x)a发生在W(y)b之前。特别是W(x)a->R (x)a->W(y)b，意思是如果我们要保持因果关系，从y中读取b后读取x只能返回a。如果系统将nil返回到P3，就违反了因果关系的保存。

R4 (y)呢?它能否返回y的初始值，即NIL? 答案是肯定的:虽然我们在没有从y中读取b的情况下 有W(x)a->W(y)b这层关系，进程P4仍然可以合理地观察到W(x)a的发生独立于y的初始化。



在实现方面，保留因果关系引入了一些有趣的问题。例如，考虑图7.12中的中间件基础进程P3。当这个中间件从读取y返回值b时，它必须知道关系W(x)a ->W (y) b。换句话说，当y的最新值传播到P3的中间件时，至少也应该传播关于y依赖关系的元数据（这里指W(x)a->W(y)b）。或者，传播也可以与更新P3节点上的x一起完成。总的来说，底线是我们需要一个依赖关系图，哪个操作依赖于哪个操作。当相关数据也在本地存储时，可以删除这样的图。

注：和上文非结果依赖顺序一致性区别：

非结果依赖，顺序一致性强调所有的进程必须执行出来的结果都保持相同的顺序，可以a b 也可以 b a，而结果依赖如7.11(a)，由于p2读取到了p1的写入值，则p2的写入b必须位于p1的写入a之后，则 p3 p4 读取的值 必须保持 先a后b。

**分组操作**

许多一致性模型是在基本读写操作级别定义的。这种粒度级别是出于历史原因：这些模型最初是为共享内存多处理器系统开发的，实际上是在硬件级实现的。

在许多情况下，这些一致性模型的细粒度与应用程序提供的粒度不匹配。我们看到共享数据的程序之间的并发性通常通过互斥和事务的同步机制得到控制。实际上，在程序级，读写操作由一对操作ENTER\_CS和LEAVE\_CS括起来。成功执行ENTER\_CS的进程将确保其本地存储中的所有数据都是最新的。此时，它可以安全地在该存储上执行一系列读和写操作，然后通过调用LEAVE\_CS结束这些操作。ENTER\_CS和LEAVE\_CS之间的数据和指令表示为一个临界区。实际上，在一个程序中，由一系列读和写操作操作的数据受到保护，不受并发访问的影响，并发访问将导致看到的不是作为一个整体执行该系列的结果。换句话说，括号将一系列读写操作转换为原子执行的单元，从而提高了粒度级别。

为了达到这一点，我们确实需要有关于ENTER\_CS和LEAVE\_CS操作的精确语义。这些语义可以根据共享**同步变量**或简单的**锁**来表示。锁具有与其关联的共享数据项，并且每个共享数据项最多与一个锁关联。在粗粒度同步的情况下，所有共享数据项都只关联到一个锁。当每个共享数据项都有自己独特的锁时，就可以实现细粒度同步。当然，这只是将共享数据与锁关联的两个极端。当一个进程进入临界区时，它应该获得相关的锁，同样地，当它离开临界区时，它释放这些锁。

每个锁都有一个当前所有者，即最后获得它的进程。当前不拥有锁但希望获得锁的进程必须向当前所有者发送一条消息，要求获得与锁关联的数据的所有权和当前值。虽然具有对锁的独占访问权，但允许进程执行读和写操作。几个进程也可能同时具有对锁的非排他性访问，这意味着它们可以读，但不能写关联的数据。当然，当且仅当没有其他进程具有独占访问权时，才可以授予非独占访问权。

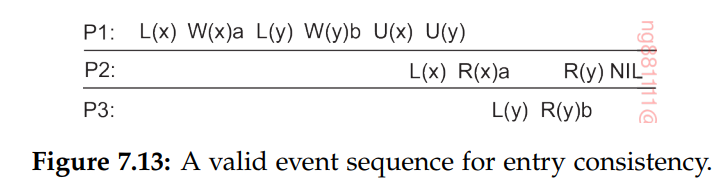
我们现在要求满足下列标准[Bershad et al.， 1993]:

只有当对其关联的共享数据的所有更新都完成时，获取锁才能成功。

只有当没有其他进程具有对锁的独占或非独占访问时，对锁的独占访问才能成功。

只有在完成了以前的任何独占访问(包括更新锁的相关数据)之后，才允许对锁进行非独占访问。

请注意，为了坚持顺序一致性，我们实际上要求将锁的使用线性化。图7.13显示了一个称为条目一致性的示例。我们将锁与每个数据项单独关联。我们使用符号L(x)作为获取x锁的缩写，即锁定x。同样，U(x)表示释放x上的锁，或解锁它。在这种情况下，P1锁定x，改变x一次，然后锁定y。进程P2还为x而不是y获取锁，因此它将为x读取值a，但可能为y读取NIL。但是，由于进程P3首先为y获取锁，所以当y被P1解锁时，它将读取值b。这里需要注意的是，每个进程都有一个变量的副本，但是这个副本不需要立即或自动更新。当锁定或解锁一个变量时，进程显式地告诉底层分布式系统，需要同步该变量的副本。因此，没有锁定的简单读取操作可能会导致读取实际上已过期的本地值。



条目一致性的编程问题之一是正确地将数据与锁关联起来。一种直接的方法是显式地告诉中间件将访问哪些数据，通常通过声明哪些数据库表将受到事务的影响来实现。在基于对象（object-based）的方法中，我们可以将一个惟一的锁与每个声明的对象关联起来，从而有效地序列化对此类对象的所有调用。

**一致性和连贯性**

在这一点上，澄清两个密切相关的概念之间的区别是有用的。到目前为止，我们所讨论的模型都处理这样一个事实:许多进程对一组数据项执行读写操作。一致性模型描述了当多个进程同时对该数据进行操作时，对该集合的期望。然后，如果集合符合模型所描述的规则，那么它就是一致的。

当数据一致性涉及到一组数据项时，**一致性模型**描述了只能为单个数据项保存的内容[Cantin et al.， 2005]。在本例中，我们假设复制了一个数据项;当不同的副本遵守由其关联一致性模式定义的规则时，它被称为是一致的。流行的模型是顺序一致性模型，但现在只应用于单个数据项。实际上，这意味着在并发写的情况下，所有进程最终将看到相同的更新顺序。

**最终一致性**

进程实际在多大程度上以并发方式运行，以及需要在多大程度上保证一致性，这些可能会有所不同。在许多示例中，并发性仅以受限制的形式出现。例如，在许多数据库系统中，大多数进程几乎从不执行更新操作;他们主要从数据库中读取数据。只有一个或很少的进程执行更新操作。只有一个或很少的进程执行更新操作。接下来的问题是，对于只读进程，应该以多快的速度提供更新。在全球运营内容交付网络的时代，开发人员常常选择缓慢地传播更新，隐式地假设大多数客户机总是重定向到相同的副本，因此永远不会出现不一致。

另一个例子是网络。实际上，在所有情况下，Web页面都由一个权威机构更新，例如Web管理员或页面的实际所有者。通常不需要解决写-写冲突。另一方面，为了提高效率，通常将浏览器和Web代理配置为将获取的页面保存在本地缓存中，并在下一次请求时返回该页面。这两种类型的Web缓存的一个重要问题是，它们可能返回过期的Web页面。换句话说，与实际Web服务器上可用的页面相比，返回给请求客户机的缓存页面是一个旧版本。事实证明，许多用户发现这种不一致是可以接受的(在一定程度上)，只要他们只能访问相同的缓存。实际上，他们仍然不知道已经发生了更新，就像以前的内容交付网络一样。

另一个例子是一个世界性的命名系统，比如DNS。DNS名称空间被划分为域，每个域被分配给一个命名机构，该机构充当该域的所有者。只有该权限的拥有者才能更新名称空间的一部分。因此，由于两个操作都希望对相同的数据执行更新而产生的写-写冲突，永远不会发生。惟一需要处理的情况是读写冲突，在这种情况下，一个进程希望更新数据项，而另一个进程同时尝试读取该数据项。事实证明，在这种情况下，以延迟的方式传播更新通常是可以接受的，这意味着读取过程只有在更新发生一段时间之后才会看到更新。

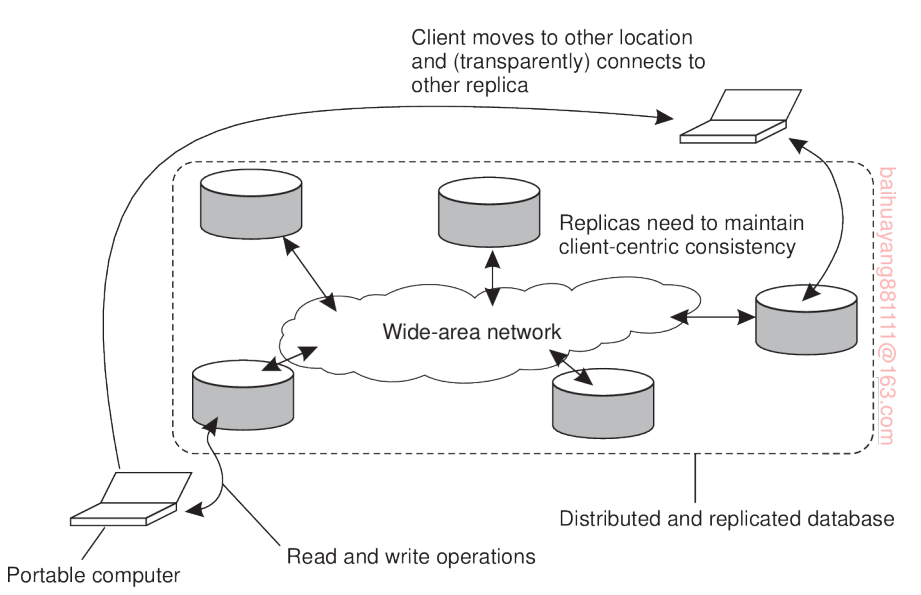
这些例子可以看作是(大规模)分布式和复制数据库的案例，这些数据库能够容忍相对较高程度的不一致性。它们的共同点是，如果长时间不进行更新，所有副本将逐渐变得一致，也就是说，存储的数据完全相同。这种形式的一致性称为最终一致性[Vogels, 2009]。

最终一致的数据存储具有这样的特性:在没有写-写冲突的情况下，所有副本都将收敛到彼此的相同副本。最终的一致性本质上只需要保证更新传播到所有副本。当假定只有一小组进程可以执行更新时，写-写冲突通常相对容易解决。在实践中，我们经常看到在冲突的情况下，一个特定的写操作(全局)被声明为“赢家”，覆盖了任何其他冲突写操作的效果。因此，最终一致性的实现成本通常很低。

7.3以客户为中心的一致性模型

以数据为中心的一致性模型旨在为数据存储提供系统范围的一致视图。一个重要的假设是，并发进程可能同时更新数据存储，并且有必要在面对这种并发时提供一致性。例如，在基于对象的条目一致性的情况下，数据存储确保在调用对象时，向调用过程提供对象的副本，该副本反映到目前为止对对象所做的所有更改，可能是由其他进程所做的更改。在调用期间，还保证没有其他进程可以干扰，即为调用进程提供互斥访问。能够在保持强一致性的同时处理共享数据上的并发操作是分布式系统的基础。出于性能原因，只有当流程使用诸如事务或同步变量之类的机制时，才可能保证强一致性。

在本节中，我们将研究一类特殊的分布式数据存储。我们所考虑的数据存储的特性是不考虑同步更新，或者说当更新发生时，可以假定这些更新能被容易的解决。大多数操作都需要读取数据。这些数据存储提供了一个弱一致性模型，比如最终一致性。通过引入特殊的以客户为中心的一致性模型，可以以一种相对节省成本的方式隐藏许多不一致性。



如果客户能总是访问同一个副本，最终一致性的存储将会工作的很好。然而，当在短时间内访问不同的副本时，就会出现问题。通过考虑移动用户访问分布式数据库，可以很好地说明这一点，如图7.15所示。

例如，移动用户Alice通过透明方式连接到其中一个副本来访问数据库。换句话说，在Alice的移动设备上运行的应用程序并不知道它实际在哪个副本上运行。假设Alice执行了几个更新操作，然后再次断开连接。稍后，她可能在移动到不同的位置或使用不同的访问设备之后再次访问数据库。此时，她可能连接到一个与以前不同的副本，如图7.15所示。但是，如果之前执行的更新尚未传播，Alice将注意到不一致的行为。以前做过的更改，但现在似乎什么都没有发生。

这个例子是最终一致的数据存储的典型例子，这是因为用户有时可能在更新还没有完全传播的情况下操作不同的副本。可以通过引入以**客户为中心的一致性**来缓解这个问题。本质上，以客户端为中心的一致性为单个客户端提供了关于该客户端访问数据存储的一致性的保证。对于不同客户机的并发访问，不提供任何保证。如果Bob修改与Alice共享但存储在不同位置的数据，我们可能很容易创建写-写冲突。而且，如果Alice和Bob都不能在一段时间内访问同一个位置，那么这种冲突可能需要很长时间才能被发现。

以客户端为中心的一致性模型源于Bayou上的工作，更普遍的是来自移动数据系统。Bayou是为移动计算而开发的数据库系统，它假定网络连接不可靠，并且会受到各种性能问题的影响。无线网络和跨大区域的网络，如因特网，属于这一类。

Bayou基本上区分了四种不同的一致性模型。为了解释这些模型，我们再次考虑物理上分布在多台机器上的数据存储。当一个进程访问数据存储时，它通常会连接到本地(或最近的)可用副本，尽管在原则上，任何副本都可以。所有读写操作都在该本地副本上执行。更新最终会传播到其他副本。

以客户端为中心的一致性模型使用以下符号进行描述。版本xi是从初始化到一系列写操作的集合，WS(xi)。通过将写操作附加到该系列中，我们获得了另一个版本xj，并说xj遵循xi。我们用符号WS(xi;xj)表示xj遵循xi。如果我们不清楚xj是否遵循xi，使用WS(xi|xj)。

**单调读取**

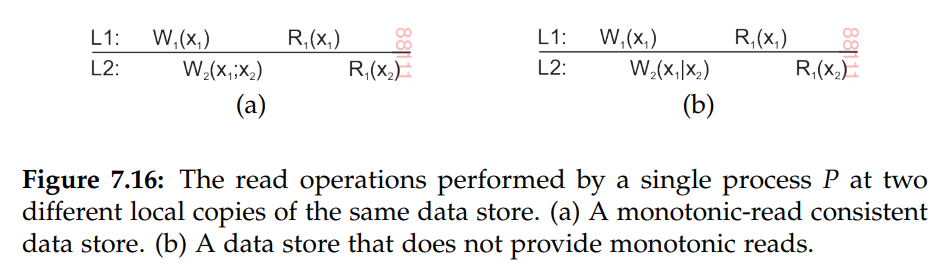
第一个以客户端为中心的一致性模型是单调读取模型。如果下列条件成立，则(分布式)数据存储提供**单调读一致性**：

*如果进程读取数据项x的值，则该进程对x执行的任何后续读取操作都将始终返回相同的值或更近期的值*

换句话说，单调读一致性保证了一个进程一旦看到x的值，它就永远不会看到旧版本的x。

作为单调读取非常有用的例子，考虑一个分布式电子邮件数据库。在这样的数据库中，每个用户的邮箱可以跨多台机器分布和复制。邮件可以进入任何位置的邮箱。然而，更新会发生的迟一些。只有当副本需要某些数据以保持一致性时，这些数据才传播到该副本。假设一个用户在旧金山阅读他的邮件。假设只读取邮件不会影响邮箱，即不会删除消息、将消息存储在子目录中，甚至标记为已经读取，等等。当用户稍后飞往纽约并再次打开邮箱时，单调读一致性保证了旧金山邮箱中的消息在纽约打开时也在邮箱中。

使用类似于以数据为中心的一致性模型的符号，单调读取一致性可以用图形表示，如图7.16所示。我们现在在示例L1和L2中显示本地数据存储，而不是沿着垂直轴显示进程。一个写或读操作由执行该操作的进程索引，即W1(x) A表示进程P1将值A写入x。由于我们对共享数据项的特定值不感兴趣，而是对它们的版本感兴趣，所以我们使用符号W1(x2)来表示进程P1在不了解其他版本的情况下生成了版本x2。W2(x1;x2)表示进程P2负责生成x1之后的版本x2。W2(x1,x2)表示在生产x1版本时，进程p2并发的生产版本x2（这可能会产生写写冲突）。R1(x2)表示P1读取x2版本。



在图7.16(a)中，进程P1首先在L1处对x执行写操作，生成版本x1，然后读取这个版本。在L2进程P2跟随版本x1，生成版本x2。当进程P1移动到L2并再次读取x时，它会发现一个更近期的值，至少考虑了之前的写操作。

图7.16(b)为违反单调读一致性的情况。当进程P1在L1处读取x1之后，它将在L2处执行操作R1(x2)。然而，已知进程P2在L2上执行的前一个写操作W2(x1|x2)生成的版本与x1不同。因此，已知P1在L2处的读操作不包括在L1处执行R1(x1)时的写操作的影响。

（个人理解：a是单调读表示 p1在L1的写记录会在L1处读取到，所以根据单调读规则，其结果应在其他存储地方读取到，而这里L2是可以读取到的，对x2的写入是依据x1的。对于b则有可能在L2读取不到x1的结果，所以不是单调读）。

**单调写**

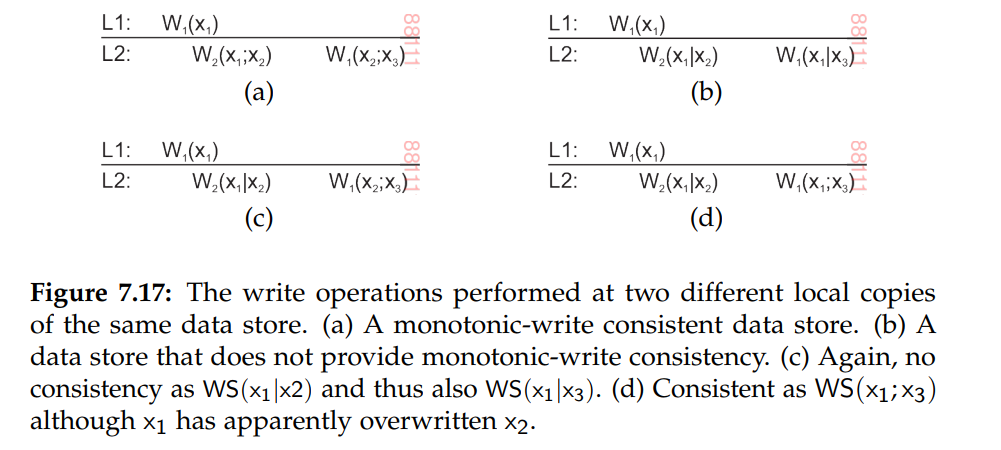
在许多情况下，按正确的顺序将写操作传播到数据存储的所有副本是非常重要的。此属性以单调写一致性表示。在单调写一致性存储中，有以下条件:

*数据项x上的进程的写操作先于同一进程对x上的任何后续写操作完成*

更正式的说，如果通过Pk进程有两个连续的操作Wk(xi)和Wk(xj)，那么无论Wk(xj)在哪里发生，我们都将有WS(xi;xj)。因此，完成一个写操作意味着执行后续操作的副本反映了由相同进程执行的前一个写操作的效果，无论该操作是在何处开始的。换句话说，对项目x的副本执行写操作的条件是，该副本必须通过该进程之前的任何写操作(可能发生在x的其他副本上)更新到最新。如果需要，新写的必须等待旧的写完。

注意，单调写入一致性类似于以数据为中心的FIFO一致性。FIFO一致性的本质是相同进程的写操作在任何地方都按照正确的顺序执行。这种顺序约束也适用于单调写，只是我们现在只考虑单个进程的一致性，而不是并发进程的集合。

当每个写操作完全覆盖x的现值时，不需要将x的副本更新到最新。然而，写操作通常只对数据项的部分状态执行。例如，考虑一个软件库。在许多情况下，通过替换一个或多个函数来更新这样的库，从而生成下一个版本。在单调写入一致性下，保证如果对库的副本执行更新，则将首先执行前面的所有更新。最终得到的库将成为最新的版本，并包含所有导致该库以前版本的更新。



单调写一致性如图7.17所示。在图7.17(a)中，进程P1在L1处对x执行写操作，表示为操作W1(x1)。之后，P1对x执行另一个写操作，但这次是在L2处，表示为W1(x2;x3)。P1在L2处生成的版本来自于流程P2的更新，后者基于版本x1，这个操作表示为W2（x1;x2）。为了保证单调写一致性，有必要将之前在L1处的写操作传播到L2，并进行可能的更新。

相比之下，图7.17(b)显示了一种不保证单调写入一致性的情况。与图7.17(a)相比，缺少的是在生成另一个版本的x之前x1到L2的传播，用W2操作表示(x1 | x2)。在本例中，进程P2生成了一个到x1的并发版本，在此之后，进程P1只生成版本x3，但仍然与x3产生并发关系。图7.17(c)所示的情况稍微微妙一些，但仍然违反单调写入一致性。进程P1现在生成版本x3，而这是依据x2开始的。而，因为x2没有包含导致x1的写操作，即WS(x1|x2)，所以我们也有WS(x1|x3)。

图7.17(d)显示了一个有趣的情况。操作W2(x1|x2)与x1并发生成x2版本。然而，后来的进程P1生成了版本x3，但显然是基于版本x1在L2中可用的事实。x1是如何以及何时被转移到L2的还没有确定，但是在任何情况下，都是与x2版本创建了write-write冲突，并以有利于x1的方式解决了这个冲突（因为x3是基于x1的）。其结果是，图7.17(d)所示的情况符合单调写一致性的规则。但是，请注意，进程P2在L2上的任何后续写入(没有读取版本x1)都将立即再次违反一致性。

注意，根据单调写一致性的定义，相同进程的写操作按初始顺序执行。单调写的一种较弱的形式是，只有在前面所有的写操作也都执行过之后才能看到写操作的效果，但可能不是按照它们最初开始的顺序执行的。这种一致性适用于写操作是可交换的情况，因此实际上没有必要进行排序。

**Read your writes**

如果符合以下条件，则数据存储将提供“read-your-writes”一致性:

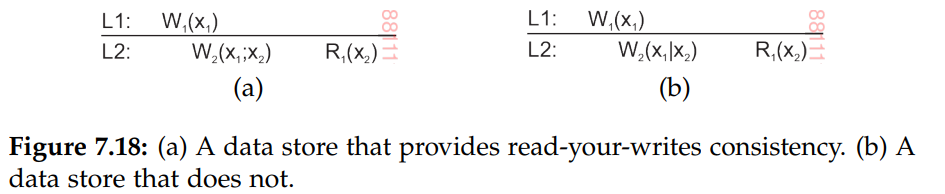
*同一个进程中，对数据项x的写操作，总是能够被后继读操作观察到*

换句话说，无论读操作发生在哪里，写操作总是在同一进程的后继读操作之前完成。

在更新Web文档并随后查看效果时，有时会体验到读写一致性的缺失。更新操作经常通过标准编辑器或文字处理器进行，这些操作可能嵌入到内容管理系统中，然后将新版本保存到Web服务器共享的文件系统中。用户的Web浏览器可能在从本地Web服务器请求该文件之后访问该文件。然而，一旦获取了文件，服务器或浏览器通常会缓存一个本地副本以供后续访问。因此，当Web页面更新时，如果浏览器或服务器返回缓存的副本而不是原始文件，用户将不会看到效果。Read-your-wirtes 一致性可以保证，如果编辑器和浏览器集成到一个程序中，那么当页面更新时缓存将失效，从而获取和显示更新后的文件。

更新密码时也会出现类似的效果。如果需要访问网页图书馆，则需要一个带密码的账户。然而，更改密码可能需要一些时间才能生效，其结果是用户可能在几分钟内无法访问库。延迟的原因是使用一个单独的服务器来管理密码，然后可能需要一些时间将(加密的)密码传播到构成库的各个服务器。

图7.18(a)显示了提供读写一致性的数据存储。注意，图7.18(a)与图7.16(a)非常相似，只是一致性现在由进程P1的最后一次写操作决定，而不是由进程P1的最后一次读操作决定。



在图7.18(a)中，进程P1执行一个写操作W1(x1)，然后在另一个本地副本上执行一个读操作。read -your-write一致性保证了写操作的效果可以被后续的读操作看到。它由W2（x1;x2）表示，它表示一个进程P2生成了一个新版本的x，但它是基于x1的。相反，在图7.18(b)中，进程P2与x1并发生成一个版本，表示为W2(x1|x2)。这意味着在生成x2时，进程P1之前的写操作的影响没有传播到L2。当P1读取x2时，它不会看到L1处自己写操作的效果。

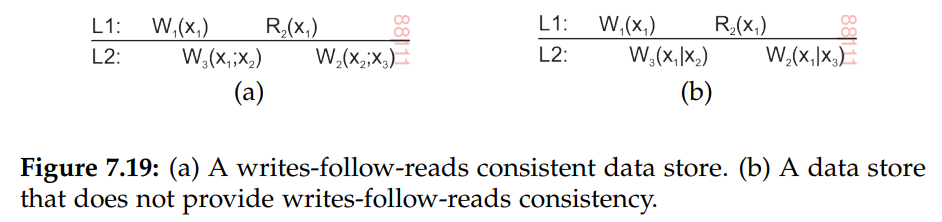
**Writes follow reads**

最后一个以客户端为中心的一致性模型是将更新作为之前读取操作的结果传播的模型。如果下列条件成立，则数据存储被称为提供 **writes-follow-reads** 一致性。

*一个进程对数据项x执行的写操作在该进程之前对x执行的读操作之后，将保证在被读的x的相同或更近的值上执行*

换句话说，一个进程对数据项x的任何后续写操作都将在一个x的副本上执行，该副本是最新的，且该进程最近读取的值是最新的。

write -follow-read一致性可用于确保网络新闻组的用户只有在看过原始文章之后才能看到对文章的反应。要理解这个问题，假设用户首先阅读文章a。然后，她发布了一个响应B。通过要求writes-follow-reads的一致性，只有在写入了A之后，才会将B写入新闻组的任何副本。注意，只阅读文章的用户不需要任何特定的以客户端为中心的一致性模型。write -follow-read的一致性保证了对文章的响应只存储在本地副本中，前提是原始副本也存储在本地副本中。



这个一致性模型如图7.19所示。在图7.19(a)中，进程P2读取本地副本L1上的版本x1。这个版本的x之前由进程P1通过操作W1(x1)在L1生成。该版本随后传播到L2，并由另一个进程P3使用，生成一个新的版本x2，表示为W3(x1;x2)。当进程P2在迁移到L2之后更新它的x版本时，我们知道它将操作从x1开始的版本，表示为W2(x2;x3)。因为我们还有W3(x1;x2)我们知道wS(x1;x3)。（R2在L1读取了x1，而W2在L2更新了x3，而x3依据x2，x2依据x1，所以满足）

图7.19(b)所示的情况有所不同。进程P3同时生成一个版本x2和版本x1。因此，当P2在读取x1之后更新x时，它将更新一个以前没有读取过的版本。这将违反Writes-follow-reads的读取一致性。（W2应该读取x2）

7.4副本管理

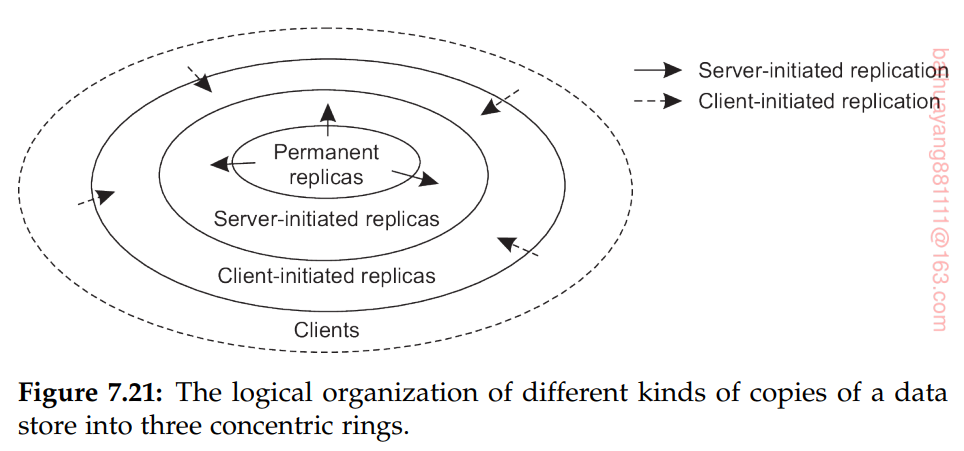
对于任何支持复制的分布式系统，一个关键问题是决定复制应该放在何处、何时、由谁放置，以及随后使用哪些机制来保持复制的一致性。安置问题本身应该分成两个子问题：放置复制服务器和放置内容。两者之间的差别是很微妙的，而且这两个问题常常不能清楚地分开。复制服务器放置涉及到寻找放置能够承载(部分)数据存储的服务器的最佳位置。内容放置涉及寻找放置内容的最佳服务器。注意，这通常意味着我们只寻找单个数据项的最佳位置。显然，在进行内容放置之前，必须先放置副本服务器。

**寻找最佳服务器位置**

也许十多年前，人们可能会关心在何处放置单个服务器，但随着Internet上许多大型数据中心的出现，情况发生了很大的变化。同样，连通性也在不断改善，使得精确定位服务器变得不那么重要。

**内容复制和放置**

当涉及到内容复制和放置时，可以从逻辑上区分三种不同类型的复制，如图7.21所示。



**永久的副本**

可以将永久副本视为构成分布式数据存储的初始副本集。在许多情况下，永久性副本的数量很少。例如，考虑一个Web站点。网站的分发通常有两种形式。第一种分布方式是将构成站点的文件复制到单个位置的有限数量的服务器上。每当有请求进来时，就使用循环策略将其转发到其中一个服务器。

分布式Web站点的第二种形式是所谓的**镜像**。在本例中，Web站点被复制到有限数量的服务器上，这些服务器称为**镜像**站点，它们在地理上分布在Internet上。在大多数情况下，客户只需从提供给他们的列表中选择一个镜像站点。镜像Web站点与基于集群的Web站点有一个共同点，即只有几个副本，这些副本或多或少是静态配置的。

类似的静态组织也出现在分布式数据库中。同样，数据库可以在多个服务器之间分布和复制，这些服务器共同组成一个服务器集群，通常称为无共享架构，强调磁盘和主内存都不是由处理器共享的（一个数据库服务器负责一部分数据？？）。或者，数据库是分布式的，并可能在许多地理位置分散的站点上复制。这种体系结构通常部署在联邦数据库中。

**Server-initiated replicas**

与永久副本不同，服务器发起的副本是数据存储的副本，存在于数据存储以提高性能，并由数据存储所有者主动创建。例如，考虑一个位于纽约的Web服务器。正常情况下，这个服务器可以很容易地处理传入的请求，但是可能会在几天内突然从远离服务器的意外位置发出大量请求。在这种情况下，在发出请求的区域安装一些临时副本可能是值得的。注意，只要保证每个数据项都由至少一个服务器承载，那么只使用服务器发起的复制就足够了，而不需要任何永久副本（why？？）。但是，永久性副本通常用作备份设施，或者用作惟一允许更改以保证一致性的副本。然后，服务器启动的副本用于将只读副本放置在客户机附近。

**Client-initiated replicas**

一种重要的副本是由客户机发起的副本。客户机发起的副本通常称为(客户机)缓存。从本质上讲，缓存是一种本地存储设施，客户机使用它来临时存储它刚刚请求的数据的副本。原则上，管理缓存完全留给客户机。从其中获取数据的数据存储与保持缓存数据的一致性无关。然而，在很多情况下，客户端可以依赖数据存储的参与来通知它缓存的数据何时过期。客户端缓存仅用于提高对数据的访问时间。通常，当客户机想要访问某些数据时，它会连接到最近的数据存储副本，从该副本获取它想要读取的数据，或者将数据存储到它刚刚修改过的数据。当大多数操作只涉及读取数据时，可以通过让客户机将请求的数据存储在附近的缓存中来提高性能。这样的缓存可以位于客户机的机器上，也可以位于与客户机相同的局域网中的另一台机器上。下一次需要读取相同的数据时，客户机只需从这个本地缓存中获取即可。只要所获取的数据没有被修改，这个方案就可以很好地工作。

数据通常保存在缓存中有限的时间，通常为了防止使用非常陈旧的数据，或者只是为其他数据腾出空间。每当可以从本地缓存中获取请求的数据时，就会发生**缓存命中**。为了提高缓存命中次数，可以在客户机之间共享缓存。基本假设是，来自客户机C1的数据请求对于来自附近客户机C2的请求也可能有用。

这个假设是否正确在很大程度上取决于数据存储的类型。例如，在传统的文件系统中，很少共享数据文件，这使得共享缓存毫无用处。同样，事实证明，使用Web缓存共享数据也在逐渐失势，部分原因是网络和服务器性能的改善。相反，服务器发起的复制方案正变得更加有效。

客户端缓存的放置相对简单：高速缓存通常放置在与其客户机相同的机器上，或者放在由客户机共享的同一局域网上。然而，在某些情况下，系统管理员通过在多个部门或组织之间放置共享缓存，甚至为整个区域(比如一个省或国家)放置共享缓存，引入了额外的缓存级别。

另一种方法是在广域网的特定位置放置(缓存)服务器，并让客户机找到最近的服务器。当服务器位于某个位置时，可以让这个服务器获取客户端曾经获得到的数据。

**内容分布 Content distribution**

复制管理还负责将更新的内容传播到相关的复制服务器。有各种各样的权衡。

**状态与操作**：

一个重要的设计问题涉及到实际要传播什么。基本上，有三种可能性：

•只传播更新通知。

•将数据从一个副本传输到另一个副本。

•将更新操作传播到其他副本

传播通知是**无效协议**的功能。在无效协议中，其他副本被告知发生了更新，并且它们包含的数据不再有效。无效声明可以指定更新了数据存储的哪一部分，因此只有一部分副本实际上是无效的。重要的问题是只传播通知。每当请求对无效副本执行操作时，通常需要首先更新该副本，这取决于要支持的特定一致性模型。

无效协议的主要优点是它们占用的网络带宽很少。与读操作相比，当有很多更新操作时，这类协议通常工作得最好，也就是说，读写比相对较小。

例如，考虑一个数据存储，其中通过将修改后的数据发送给所有副本来传播更新。如果修改数据的大小很大，并且与读取操作相比更新频繁发生，则可能出现这样的情况:两个更新相继发生，而它们之间没有执行任何读取操作。因此，将第一个更新传播到所有副本实际上是无用的，因为它将被第二个更新覆盖。相反，发送数据已被修改的通知会更有效。

第二种方法是在副本之间传输修改后的数据，当读写比相对较高时，这种方法非常有用。在这种情况下，更新有效的可能性很大，因为修改后的数据将在下一次更新之前读取。不传播修改后的数据，还可以记录更改并只传输这些日志以节省带宽。此外，传输通常是聚合在一起的，因为多个修改被打包到一个消息中，从而节省了通信开销。

第三种方法根本不传输任何数据修改，而是告诉每个副本应该执行哪个更新操作(并且只发送这些操作所需的参数值)。这种方法，也称为主动复制，假设每个副本都由一个进程表示，该进程能够通过执行操作“主动地”保持其关联数据的最新数据[Schneider, 1990]。主动复制的主要好处是，只要与操作相关的参数相对较小，更新通常可以以最小的带宽成本传播。此外，操作可以是任意复杂的，这可能允许在保持副本一致性方面进行进一步的改进。另一方面，每个副本可能需要更多的处理能力，特别是在操作相对复杂的情况下。

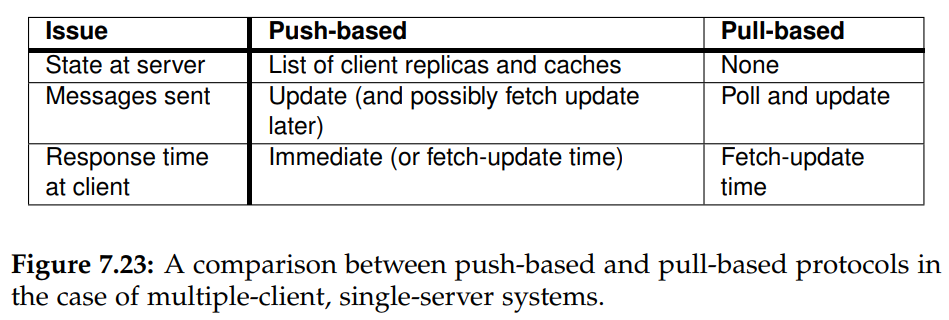
**获取与推送的比较**

另一个设计问题是更新是被拉入还是推入。在**基于推送**的方法中，也称为**基于服务器的协议**，更新被传播到其他副本，而这些副本甚至不需要更新。基于推送的方法通常在永久副本和服务器发起的副本之间使用，但也可以用于将更新推送到客户机缓存。当需要强一致性时，通常使用基于服务器的协议。这种对强一致性的需求与以下事实有关:许多客户机常常共享永久副本和服务器发起的副本，以及大型共享缓存，而这些客户机又主要执行读操作。因此，每个副本的读更新比率相对较高。在这些情况下，基于推送的协议是有效的，因为每个推送的更新至少可以被一个，但可能有更多的读者使用。此外，基于push的协议可以在需要时立即提供一致的数据。

相反，在**基于拉的**方法中，服务器或客户端请求另一个服务器向其发送此时的任何更新。基于拉的协议，也称为基于客户机的协议，通常由客户机缓存使用。例如，应用于Web缓存的一个常见策略是首先检查缓存的数据项是否仍然是最新的。当缓存接收到对仍然在本地可用的项的请求时，缓存将与原始Web服务器检查这些数据项在缓存之后是否已被修改。在修改的情况下，首先将修改后的数据传输到缓存，然后返回给请求的客户机。如果没有发生修改，则返回缓存的数据。换句话说，客户机轮询服务器以查看是否需要更新。

当读更新比相对较低时，基于拉的方法是有效的。当只有一个客户机时，这通常是(非共享)客户机缓存的情况。然而，即使缓存由许多客户机共享，当缓存的数据项很少共享时，基于拉的方法也可能被证明是有效的。与基于push的方法相比，基于pull的策略的主要缺点是，在缓存未命中的情况下，响应时间会增加。在比较基于push和基于pull的解决方案时，需要进行许多权衡，如图7.23所示。

为简单起见，考虑一个客户机-服务器系统，该系统由单个非分布式服务器和许多客户机进程组成，每个进程都有自己的缓存。



一个重要的问题是，在基于push的协议中，服务器需要跟踪所有客户机缓存。除了有状态服务器容错性较差这一事实外，跟踪所有客户机缓存可能会给服务器带来相当大的开销。例如，在基于push的方法中，Web服务器可能很容易需要跟踪成千上万的客户机缓存。每次更新Web页面时，服务器将需要遍历包含该页面副本的客户机缓存列表，然后传播更新。更糟的是，如果客户机由于空间不足而清除页面，它必须通知服务器，从而导致更多的通信。

需要在客户机和服务器之间发送的消息也有所不同。在基于push的方法中，唯一的通信是服务器向每个客户机发送更新。当更新实际上只是失效时，客户机需要额外的通信来获取修改后的数据。在基于拉的方法中，客户机必须轮询服务器，并在必要时获取修改后的数据。

最后，客户机的响应时间也不同。当服务器将修改后的数据推送到客户机缓存时，客户机端的响应时间显然为零。当推送失效时，响应时间与基于拉的方法相同，由从服务器获取修改数据所需的时间决定。

这些权衡导致了基于租约的更新传播的混合形式。在副本管理的情况下，**租约**是服务器承诺在指定的时间内将更新推送到客户机。当租约到期时，客户端必须轮询服务器以获得更新，并在必要时拉入修改后的数据。另一种方法是，客户端请求一个新租约，以便在前一个租约到期时推送更新。

最初由Gray和Cheriton[1989]引入的契约提供了一种方便的机制，可以在基于推和基于拉的策略之间进行动态切换。考虑以下租赁系统，该系统允许根据不同的租赁标准动态调整到期时间，如[Duvvuri et al.， 2003]中所述。我们区分了以下三种租赁类型。

首先，基于**年龄的租约**根据数据项的最后一次修改给出。基本假设是，长时间未修改的数据在未来一段时间内有望保持不变。这种假设在基于web的数据和常规文件的情况下是合理的。通过将长期租约授予预计将保持不变的数据项，与所有租约具有相同到期时间的情况相比，更新消息的数量可以大大减少。

另一个租赁标准是特定客户机请求更新其缓存副本的频率。使用基于更新频率的租约，服务器将向经常需要刷新缓存的客户端提供长期租约。另一方面，偶尔要求特定数据项的客户端将获得该数据项的短期租约。这种策略的效果是，服务器基本上只跟踪其数据流行的客户端;此外，这些客户提供了高度的一致性。

最后一个标准是服务器的状态空间开销。当服务器意识到它正在逐渐超载时，它就会降低它分发给客户机的新租约的过期时间。这种基于状态的租约策略的效果是，随着租约的到期越来越快，服务器需要跟踪更少的客户机。换句话说，服务器动态切换到更无状态的操作模式，从而期望卸载自己，以便更有效地处理请求。一个明显的缺点是，当读更新比高时，可能需要做更多的工作。

**单播和多播**

与推或拉更新相关的是决定应该使用单播还是多播。在单播通信中，当作为数据存储的一部分的服务器将其更新发送给N个其他服务器时，它通过向每个服务器发送N条单独的消息来实现。通过多播，底层网络负责将消息有效地发送给多个接收器。

在许多情况下，使用可用的多播设备更便宜。一种极端的情况是，所有副本都位于相同的局域网中，并且硬件广播可用。在这种情况下，广播或多播消息并不比单点对点消息更昂贵。这样一来，单播更新的效率就会降低。

多播通常可以与基于推送的更新传播方法有效地结合使用。当这两者被仔细集成时，决定将其更新推送到多个其他服务器的服务器只需使用一个多播组来发送其更新。相反，与基于拉的方法相比，通常只有一个客户机或服务器请求更新其副本。在这种情况下，单播可能是最有效的解决方案。

**管理复制对象**

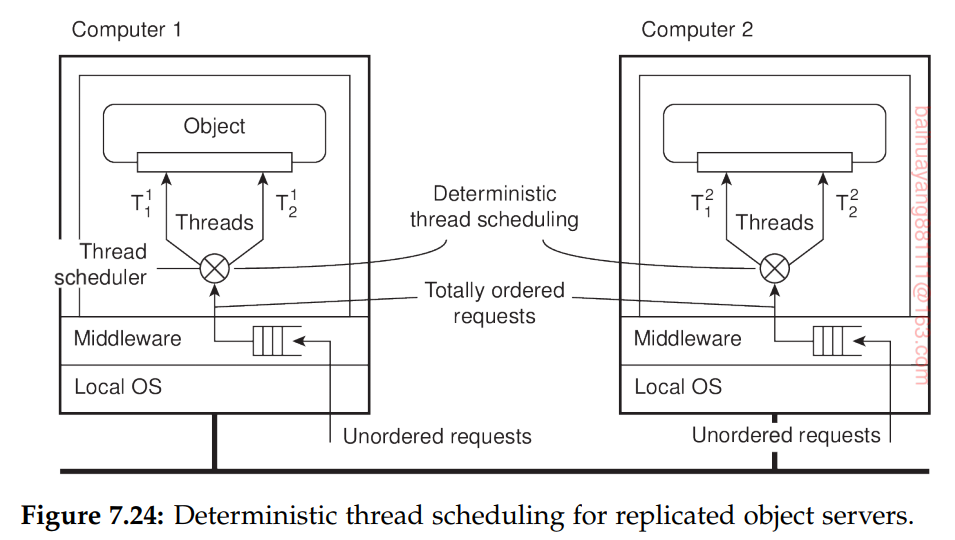
如前所述，分布式对象的以数据为中心的一致性很自然地以条目一致性的形式出现。回想一下，在本例中，目标是使用同步变量(例如，以锁的形式)对共享数据上的操作进行分组。由于对象会自然地组合数据和数据上的操作，所以在调用期间锁定对象会序列化访问并保持它们的一致性（say what？）。

然而在概念上将锁与对象关联很简单，但是在复制对象时，它不一定提供正确的解决方案。实现条目一致性需要解决两个问题。首先，我们需要一种方法来防止在同一个对象上并发执行多个调用。换句话说，当执行对象的任何方法时，不能执行任何其他方法。这一要求确保对对象内部数据的访问确实是序列化的。只要使用本地锁定机制就可以确保这种序列化。

第二个问题是，对于复制对象，我们需要确保对对象复制状态的所有更改都是相同的。换句话说，我们需要确保在不同的副本上不会同时发生两个独立的方法调用。这一要求意味着我们需要对调用进行排序，以便每个副本都能以相同的顺序看到所有调用。我们在7.5节中描述了一些通用的解决方案。

在许多情况下，设计复制的对象首先要设计一个对象，可能通过本地锁定保护它免受并发访问，然后复制它。中间件的角色是确保如果客户机调用一个复制的对象，那么调用将传递给副本，并以相同的顺序传递给它们各自的对象服务器。但是，我们还需要确保这些服务器中的所有线程也以正确的顺序处理这些请求。这个问题在图7.24中进行了概述。

多线程(对象)服务器只是接收一个传入的请求，将其传递给一个可用的线程，然后等待下一个请求。服务器的线程调度程序随后将CPU分配给可运行线程。当然，如果中间件已经尽力为请求交付提供了总的顺序，那么线程调度程序应该以确定的方式运行，以避免混合对同一对象的方法调用的顺序，如图7.24。



当然，不需要确定地调度所有线程。原则上，如果我们已经有了完全有序的请求交付，我们只需要确保对相同复制对象的所有请求都按照交付的顺序进行处理。这种方法允许并发地处理不同对象的调用，并且不受线程调度程序的进一步限制。

Basile等人[2002]描述的一种方法确保在每个副本上以相同的顺序调度共享相同(本地)锁的线程。最基本的是一个基于主模式的方案，其中一个副本服务器将带头确定哪个线程将首先获得一个特定的锁。在[Basile et al.， 2003]中描述了一种避免服务器之间频繁通信的改进。注意，不共享锁的线程因此可以在每个服务器上并发操作。

该方案的一个缺点是，它在底层操作系统的级别上运行，这意味着需要管理每个锁。通过提供应用程序级的信息，只确定对复制对象进行序列化访问所需的锁，可以极大地提高性能。

7.5一致性协议

现在，通过研究几个一致性协议，我们将重点放在一致性模型的实际实现上。**一致性协议**描述特定一致性模型的实现。我们遵循对一致性模型的讨论的组织方式，首先查看以数据为中心的模型，然后查看以客户端为中心的模型的协议。

**连续的一致性**

作为持续一致性工作的一部分，Yu和Vahdat[2000]开发了一些协议来处理三种形式的一致性。在下面的文章中，我们简要考虑了一些解决方案，为了清晰起见省略了一些细节。

**边界数值偏差**

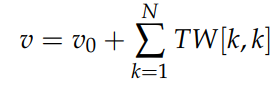
我们首先关注一个保持数值偏差在一定范围内的解。同样，我们的目的并不是深入讨论每个协议的所有细节，而是给出一个大致的概念。边界数值偏差的详细信息可以在[Yu and Vahdat, 2000]中找到。

我们主要关注对单个数据项x的写操作。每个写W(x)都有一个关联值，该值表示更新x的数值，表示为val(W(x))或简单的val(W)。简单起见，我们假设val(W) > 0。每个写W最初提交给N个可用副本服务器中的一个，在这种情况下，该服务器成为写的原点，记作原点(W)。如果我们在特定的时间点考虑系统，我们将看到几个提交的写仍然需要传播到所有服务器。为此，每个服务器Si将跟踪它在自己的本地x副本上执行的写操作的日志Li。

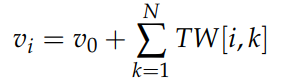
令TW[i, j]为服务器Si执行来自服务器Sj的写操作。



注意，TW[i, i]表示提交给Si的聚合写入。我们的目标是在任意时刻t，让服务器Si上的当前值vi (x)偏离实际值v (x)在标准范围之内。这个实际值完全由所有提交的写决定。也就是说，如果v0是x的初值，



和



即vi<=v。让我们只关注绝对偏差。特别地，对于每个服务器Si，我们需要保证：



提交到服务器Si的写操作需要传播到所有其他服务器。有许多不同的方法可以做到这一点，但通常流行协议将允许快速传播更新。无论如何，当服务器Si将来自Sj的写传播到Sk时，Sk将能够在发送写时了解到TW[i, j]的值。换句话说，Sk可以维护TWk[i, j]的视图，。很明显，



整个想法是，当服务器Sk注意到Si没有跟上提交给Sk的更新的速度时（Si没有及时给Sk发送更新信息），它将写日志转发给Si。这种转发有效地提高了Sk对TW[i, k]的视图，使偏差TW[i, k] - TWk[i, k]变小。目的就是TW[k,k]-TW[i,k]小于 derta/(N-1)。

**边界过时的偏差**

有许多方法可以将副本的过时性保持在指定的范围内。一个简单的方法是让服务器Sk保持一个实时的矢量时钟RVCk，其中RVCk[i] = ti表示Sk已经看到了所有提交给Si的写入，直到ti。在本例中，我们假设每个提交的写都由其原始服务器加盖时间戳，ti表示Si的本地时间。

如果复制服务器之间的时钟是松散同步的，那么可以接受的限制过期的协议如下。每当服务器Sk注意到tk - RVCk[i]即将超过指定的限制时，它就会开始拉入来自Si的写，这些写的时间戳比RVCk[i]晚。

请注意，在这种情况下，副本服务器负责将其x副本保持为最新的关于已在其他地方发出的写操作的副本。相反，在维护数值边界时，我们采用了push方法，通过转发写操作，让原始服务器保持副本的最新状态。推写的问题是，在过时的情况下，当事先不知道最大传播时间时，无法保证一致性。通过引入更新，这种情况有所改善，因为多台服务器可以帮助保持服务器的x副本是新鲜的(即，最新的)。

**边界点的偏差**

回想一下，连续一致性中的顺序偏差是由于副本服务器暂时应用已提交给它的更新而造成的。因此，每个服务器将有一个包含暂定写的本地队列，它们应用于x的本地副本的实际顺序仍然需要确定。通过指定暂定写队列的最大长度来限制排序偏差。

因此，检测什么时候需要强制执行一致性是很简单的: 当此本地队列的长度超过指定的最大长度时。此时，服务器将不再接受任何新提交的写，而是尝试通过与其他服务器协商其写的执行顺序来提交暂定的写。换句话说，我们需要强制执行一个全局一致的临时写入顺序。

（矩阵懵逼啊）

**Primary-based协议**

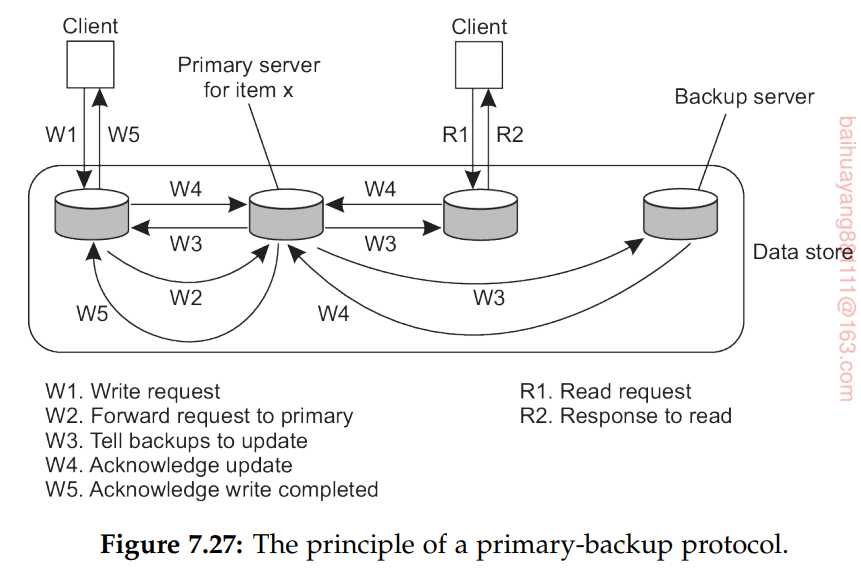
在实践中，我们看到分布式应用程序通常遵循相对容易理解的一致性模型。这些模型包括边界过时偏差模型和边界数值偏差模型。当涉及到处理操作的一致顺序的模型时，顺序一致性，尤其是那些可以通过锁定或事务对操作进行分组的模型是很受欢迎的。

一旦一致性模型对应用程序开发人员来说变得稍微难以理解，我们就会发现即使性能可以提高，也会忽略它们。底线是，如果一致性模型的语义不直观地清晰，应用程序开发人员将很难构建正确的应用程序。简单是值得赞赏的(也许是合情合理的)。

在顺序一致性的情况下，事实证明基于主协议的协议占优势。在这些协议中，数据存储中的每个数据项x都有一个关联的主节点，主节点负责协调对x的写操作。可以区分主服务器是否固定在远程服务器上，或者在将主服务器移动到启动写操作的进程之后，是否可以在本地执行写操作。

**Remote-write协议**

支持复制的最简单的基于主协议是所有写操作都需要转发到固定的单个服务器。读取操作可以在本地执行。这种方案也称为**主备份协议。**主备份协议的工作原理如图7.27所示。希望对数据项x执行写操作的进程，将该操作转发给x的主服务器。主服务器对其本地副本x执行更新，然后将更新转发给备份服务器。每个备份服务器也执行更新，并向主服务器发送确认。当所有备份都更新了它们的本地副本时，主进程向初始进程发送一个确认，然后通知客户机。



此方案的一个潜在性能问题是，启动更新的进程可能需要相当长的时间才能继续。实际上，更新是作为阻塞操作实现的。另一种方法是使用非阻塞方法。一旦主节点更新了它的本地x副本，它就返回一个确认。然后，它告诉备份服务器也执行更新。在[Budhiraja和Marzullo, 1992]中讨论了非阻塞主备份协议。

非阻塞主备份协议的主要问题与容错有关。在阻塞方案中，客户机进程确定更新操作由其他几个服务器备份。这与非阻塞解决方案不同。当然，这样做的好处是写操作可以大大加快速度。主备份协议提供了顺序一致性的直接实现，因为主备份可以按照全局惟一的时间顺序对所有传入的写进行排序。显然，无论使用哪个备份服务器执行读操作，所有进程都以相同的顺序查看所有写操作。而且，对于阻塞协议，进程总是会看到它们最近的写操作的效果(注意，如果不采取特殊措施，非阻塞协议就不能保证这一点)。

**本地写协议**

主备份协议的一种变体是主副本在希望执行写操作的进程之间迁移。与之前一样，每当一个进程想要更新数据项x时，它都会定位x的主副本，然后将其移动到自己的位置，如图7.28所示。这种方法的主要优点是可以在本地执行多个连续的写操作，而读进程仍然可以访问它们的本地副本。但是，只有在遵循非阻塞协议的情况下才能实现这种改进，在主服务器完成本地执行更新之后，更新才会传播到副本。

这种主备份本地写协议也可以应用于能够在断开连接模式下运行的移动计算机。在断开连接之前，移动计算机将成为它希望更新的每个数据项的主服务器。在断开连接时，所有更新操作都在本地执行，而其他进程仍然可以执行读操作(但没有更新)。稍后，当再次连接时，更新将从主服务器传播到备份服务器，使数据存储再次处于一致的状态。

作为该方案的最后一个变体，genera中的分布式文件系统也使用了基于非阻塞本地写的主协议。在这种情况下，可能有一个固定的中央服务器，通常通过它执行所有写操作，就像远程写主备份一样。但是，服务器暂时允许其中一个副本执行一系列本地更新，因为这可能大大提高性能。复制服务器完成后，更新被传播到中央服务器，然后从中央服务器分发到其他复制服务器。

**复制的写入协议（**Replicated-write protocols**）**

在复制写协议中，写操作可以在多个副本上执行，而不是像在基于主副本的副本中那样只能执行一个副本。可以对活动复制(将操作转发到所有副本)和基于多数表决的一致性协议进行区分。

**活跃的复制**

在活跃复制中，每个副本都有一个执行更新操作的关联进程。与其他协议相比，更新通常通过引起更新的写操作传播。换句话说，操作被发送到每个副本。不过，也可以发送更新。（其他协议不也是发送更新？）

活动复制的一个问题是，在任何地方都需要按照相同的顺序执行操作。因此，需要的是一种全序组播机制。实现总排序的一种实用方法是使用中央协调器，也称为**排序器**。一种方法是首先将每个操作转发给排序器，排序器分配一个惟一的序列号，然后将操作转发给所有副本。操作按序号的顺序进行。

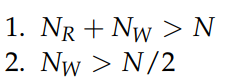
**Quorum-based协议**

支持复制写的另一种方法是使用投票，这是Thomas[1979]最初提出并由Gifford[1979]推广的。基本思想是要求客户机在读取或写入复制的数据项之前请求并获得多个服务器的权限。

作为算法工作原理的一个简单示例，考虑一个分布式文件系统，并假设一个文件在N个服务器上复制。我们可以制定一条规则，规定要更新一个文件，客户端必须首先联系至少一半的服务器加上一个(大多数)，并让它们同意进行更新。一旦他们达成一致，文件将被更改，并与新文件关联一个新版本号。版本号用于标识文件的版本，并且对于所有新更新的文件都是相同的。要读取复制的文件，客户机还必须联系至少一半的服务器和一台服务器，并要求它们发送与该文件关联的版本号。如果所有版本号都相同，那么这一定是最新的版本，因为只更新其余服务器的尝试将失败，因为没有足够的服务器。

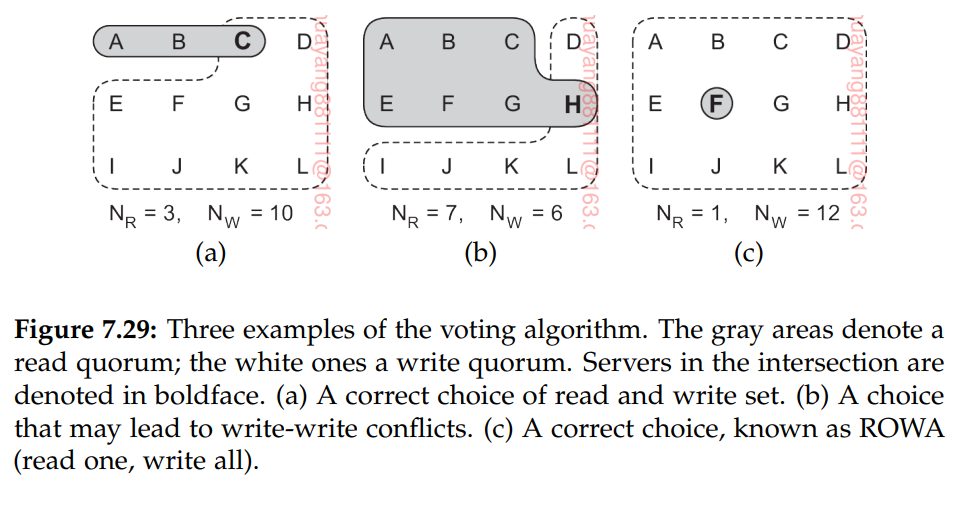
例如，如果有5台服务器，而客户机确定其中3台服务器有版本8，那么其他两台服务器不可能有版本9。毕竟，任何从版本8到版本9的成功更新都需要获得三个服务器的同意，而不仅仅是两个。

最初引入基于quorum的复制时，提出了一种更为通用的方案。在其中，要读取一个包含N个副本的文件，客户端需要装配一个读仲裁，一个任意NR服务器集合，或者更多。类似地，要修改文件，至少需要NW服务器的写仲裁。NR和NW的值受以下两个约束:



第一个约束用于防止读写冲突，而第二个约束用于防止写写冲突。只有在适当数量的服务器同意参与之后，才能读取或写入文件。（如何防止这两个冲突）

要查看此算法如何工作，请考虑图7.29(a)，其中NR = 3, NW = 10。假设最近的写仲裁由10台服务器C到L组成。所有这些都得到新版本和新版本号。三个读仲裁则必须包含写仲裁的其中一个。当客户端查看版本号时，它将知道哪个是最近的版本号并接受该版本号。



在图7.29中，我们看到了另外两个例子。在图7.29(b)中，由于NW≤N/2，可能会发生写写冲突。特别是,如果一个客户选择{A, B, C, E, F, G}为其写组和另一个客户端选择{D, H, I, J, K, L}的写组,显然我们将陷入困境作为两个更新都将被接受检测,他们实际上没有冲突。

图7.29(c)中所示的情况特别有趣，因为它将NR设置为1，从而可以通过查找任何副本并使用它来读取复制的文件。然而，这种良好的读取性能所付出的代价是写更新需要获得所有副本。这种方案通常称为Read-One、Write-All (ROWA)。基于quorum的复制协议有几种变体。Jalote[1994]提供了一个很好的概述。

**Cache-coherence协议**

缓存是复制的一种特殊情况，因为它们通常由客户机而不是服务器控制。然而，确保缓存与服务器发起的副本一致的cache-coherence协议在原则上与目前讨论的一致性协议没有太大的不同。

在高速缓存的设计和实现方面，特别是在共享内存多处理器系统中，已经有了很多研究。许多解决方案都基于底层硬件的支持，例如，假设可以进行监视或有效的广播。在构建于通用操作系统之上的基于中间件的分布式系统的上下文中，基于软件的缓存解决方案更有趣。在这种情况下，通常维护两个独立的标准来对缓存协议进行分类。

首先，缓存解决方案的**一致性检测策略**可能不同，也就是说，当实际检测到不一致时。在静态解决方案中，假定编译器在执行之前执行必要的分析，并确定哪些数据实际上可能导致不一致，因为它们可能被缓存。编译器只是插入避免不一致的指令。动态解决方案通常应用于本书研究的分布式系统。在这些解决方案中，在运行时检测到不一致。例如，将对服务器进行检查，以查看缓存后的数据是否已被修改。

在分布式数据库的情况下，基于动态检测的协议可以进一步分类，具体考虑在事务中何时进行检测。Franklin等人[1997]区分了以下三种情况。首先，当在事务期间访问缓存的数据项时，客户机需要验证该数据项是否仍然与存储在(可能复制的)服务器上的版本一致。事务不能继续使用缓存的版本，直到其一致性得到确定的验证。

第二种乐观的方法是在进行验证时让事务继续进行。在本例中，假定缓存的数据在事务启动时是最新的。如果后来证明这个假设是错误的，事务将不得不中止。

第三种方法是仅在事务提交时验证缓存的数据是否是最新的。实际上，事务只是开始对缓存的数据进行操作，并希望得到最好的结果。在完成所有工作之后，对访问的数据进行一致性验证。当使用陈旧数据时，事务将中止。

一致性协议的另一个设计问题是**一致性实施策略**，它决定了缓存如何与服务器上存储的副本保持一致。最简单的解决方案是完全不允许缓存共享数据。相反，共享数据只保存在服务器上，服务器使用上面讨论的基于主协议或复制-写协议之一来维护一致性。客户端只允许缓存私有数据。显然，这种解决方案只能提供有限的性能改进。

当共享数据可以缓存时，有两种方法可以增强缓存一致性。第一种方法是让服务器向所有缓存发送失效声明。第一种方法是让服务器在修改数据项时向所有缓存发送无效声明。第二种方法是简单地传播更新。大多数缓存系统使用这两种方案中的一种。客户机-服务器数据库有时支持在发送失效或更新之间进行动态选择。

最后，我们还需要考虑当进程修改缓存的数据时会发生什么。当使用只读缓存时，更新操作只能由服务器执行，服务器随后遵循某种分发协议，以确保更新被传播到缓存。在许多情况下，遵循的是基于拉的方法。在这种情况下，客户端检测到缓存过期，并请求服务器进行更新。

另一种方法是允许客户机直接修改缓存的数据，并将更新转发给服务器。这种方法在write-through缓存中使用，而write-through缓存通常用于分布式文件系统。实际上，直接写缓存类似于基于主体的本地写协议，其中客户机的缓存已成为临时主体。为了保证(顺序的)一致性，必须授予客户端独占的写权限，否则可能会发生写-写冲突。

由于所有操作都可以在本地执行，与其他方案相比，Write-through缓存可能提供更好的性能。如果我们允许在通知服务器之前进行多次写操作，从而延迟更新的传播，那么还可以进行进一步的改进。这就导致了所谓的**回写缓存**，它主要应用于分布式文件系统