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。

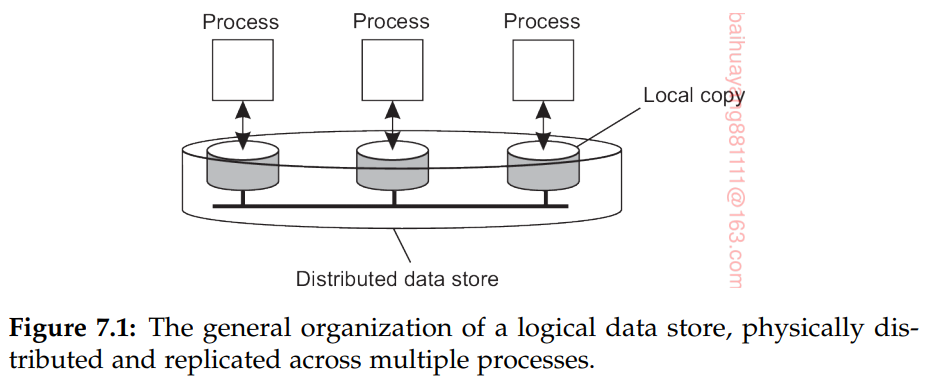
我们现在面临着进退两难的局面。一方面，通过应用复制和缓存可以缓解可伸缩性问题，从而提高性能。另一方面，为了保持所有副本的一致性，通常需要全局同步，这在性能方面有内在的代价。

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。