说明

目前，计算机系统在迅猛发展。在1945 – 1985年，现代计算机时代刚刚开始，那时的计算机都是非常大而且昂贵的。除此之外，由于计算机之间连接的缺乏，使得每个计算机只能独立的运行。

80年代中期开始，两个技术改变了这一现状。第一个就是微处理器。期初，只有8位的处理器，发展到后来有 16位，32位，以及64位等。我们可以通过多核处理器来编写并行程序。现在我们的处理器悠着30-40年前大型机的处理能力，然而造价却只有之前的千分之一。

第二项技术是高速的计算机网络的发明。Local-area networks 或 LANS（局域网）可以使得一栋大楼内的成百上千的计算机在毫秒量级进行通信。并且很大的数据可以以每秒数十亿比特的速度进行传输（千兆网络）。Wide-area networks 或 WANS (广域网）可以让地球上数百万计算机以 数十K到数百兆的速度进行传播。

同样的，计算机系统的迷你化也是我们关注的焦点，手机就是最好的例子。用传感器，内存和cpu包装的这些设备已经和完备的计算机相差无异。当然，他们也有网络通信的能力。 沿着这条发展线路，那些插入式设备同样找到了市场。那些像电源适配器一样大小的设备，可以通过直接插入的方式，并产生和near-desktop 相近的性能。

这些成果不但证明这些技术可行，并能够容易构成一个由很多计算机组成的计算机网络系统。由于这些计算机是在地理层面分散的，故我们将此系统称为分布式系统。这个系统可以包含几个或者数百万台计算机。这些计算机的连接可以是通过有线的，无线的，以及混合这两种。由于分布式系统总会有新的计算机加入，或者老的计算机退出，所以网络的拓扑和性能是一直在改变的，所以我们称之为动态的系统。

在这一章，我们将通过几个知名的系统 对分布式系统进行初探并了解其设计目标。

## 分布式系统是个什么鬼东西

分布式系统的定义出现在很多的文献中，然而他们的定义都不那么尽如人意。我们这里给出一个松散的定义：

*分布式系统是由一些各自独立的计算机组成的集合，对使用者而言，它确是一个单一的系统。*

这个定义参考了分布式系统的两个特性。一个是分布式系统的计算机是能够彼此运行。这些独立的计算机我们称之为节点，可以是一个硬件，也可以是一个软件。第二个特性是从他的使用者（人 或者 应用）来说明的。这些独立的计算机必须要协作，如何构建这种协作关系是分布式系统的核心内容。这里强调下，我们并没有对节点（独立的计算机）的类型做任何假设。原则上，从高性能主机到小的传感器设备都是可以的。同样的，我们也没对节点之间的连接方式做任何假设。

特性1：独立计算机组成的集合

大部分现代操作系统是由许多高性能的计算机或者是可插拔式的(plug)计算机或者是一些小的设备组成的。一个基本的原则是这些计算机可以独立运行，但是如果他们只是独立运行，这个系统就不能称之为分布式系统了。实际上，这些节点彼此之间可以交换信息，并为了共同的目标而存在的。一个节点收到传入的信息后，反应并加工之后，将导致更深远层次的消息传递。

一个重要的现象是，每个独立的计算机都有他自己的一套时间。或者说，我们不能假设这里有一个统一的时钟概念。缺乏统一的全局时钟会导致基本的同步和协调问题，关于这一点，我们将在第六章阐述。事实上，我们必须要管理好这个这些节点的关系和关联。说白了，我们需要知道哪些节点在集群，哪些已经排除了，或者哪些节点可以直接相连等等。

管理节点集群的关系是很困难的，即使只涉及权限控制。为了方便阐述，我们先搞清楚开放集群和关闭集群。在开放集群中，每个节点都是可以加入到集群的，这表明所有节点都可以互相通信。相反是闭合集群，只有集群内的节点可以互相通信，并有另外一套机制控制其他节点的加入或离开。

很明显的，这种权限控制很困难。这种处理节点是否可以进入集群的机制很重要，如果设计不得当，将成为集群可扩展性的瓶颈，这点我们将在第九章阐述。第二点，每个节点都必须确认自己和其他的正确的集群，正确的节点进行通信，而不是那些恶意的搞破坏的节点。最后，由于集群成员可以和非集群成员通信，这里将会面临信任问题（是否相信那些非成员节点）。

考虑到集群的结构，经验表明，分布式集群很大程度都是一个覆盖网络（overlay network）。每个节点都是由一个进程组成，这个进行有其他的进程的列表，可以向其他进程发送小心。通常来讲，与节点最近的邻居节点是最先被通信的。小心是通过 TCP/UDP 协议发送的，高级别的设备也同样适用，这点我们将在第四章阐述。下面介绍两种 覆盖网络。

结构化覆盖网络： 每个节点都会有一套设定好的邻居节点用来通信

非结构化覆盖网络： 每个节点都随机选择其他节点用来通信

无论怎样，覆盖网络应该是连通的，即任意两个节点都可以通过某个路径进行通信。一个众所周知的覆盖网络是 P2P (peer-to-peer)网络。覆盖网络的例子我们将在第二章以及后续章节进行阐述。通过以上我们应该能认识到，集群的组织构建是需要下很大功夫的，同样也是分布式网络管理很重要的组成部分。

特性2：单一的系统

正如之前提到的，一个分布式系统，应该是一个单一的系统。一些情况下，研究者表示应该有一个单一系统的视角，即终端用户感知不到他们正在处理的数据，程序是分散在不同区域的。从某些方面来说，单一系统的视角的要求太高了，在我们定义的分布式系统里，我们称之为连贯的。粗略的讲，分布式系统的连贯性表示的是它表现的和使用者的预期一致。更特殊的来讲，单一系统的连贯性表示，集群中的节点作为一个整体，无论在什么时间，地点以及用户和系统是如何交互的情况下，都表现的非常一致。

提供单一连贯性视角是非常有挑战性的。例如，它要求终端用户应当无法分辨程序运行在哪台计算机上，或者无法感知到这个任务的一部分已经派发到其他地方的另外一个进程去执行。同样的，用户不用关心数据会存储到哪里，以及复制一些数据用来提升性能。这就叫做分布式系统的透明性，这是一个重要的分布式系统的设计目标，我们将在1.2节进行讨论。从某种意义来说，这类似于类unix系统中的文件系统的操作，它屏蔽了文件，设备，主存以及网络的不同。

然而，争取一个连贯性系统需要一个重要的权衡。由于我们不能忽略分布式系统有许多网络节点，这有可能导致集群的一部分有可能会失效。这表示有些应用将会执行成功，而有些则会失败，这种情况是需要处理的。部分失败在任何一个复杂系统中都是固有的，在分布式系统中更是难以掩藏的。就像某个图灵奖获得者说的“你将不会知道是计算机哪个部分的故障导致你计算机的故障”。

## 中间件和分布式系统

为了帮助分布式系统的发展，分布式系统通常会构建一个分离的软件层，这个软件层放置在各自计算机的操作系统的上层，并属于分布式系统的一部分。如下图1.1

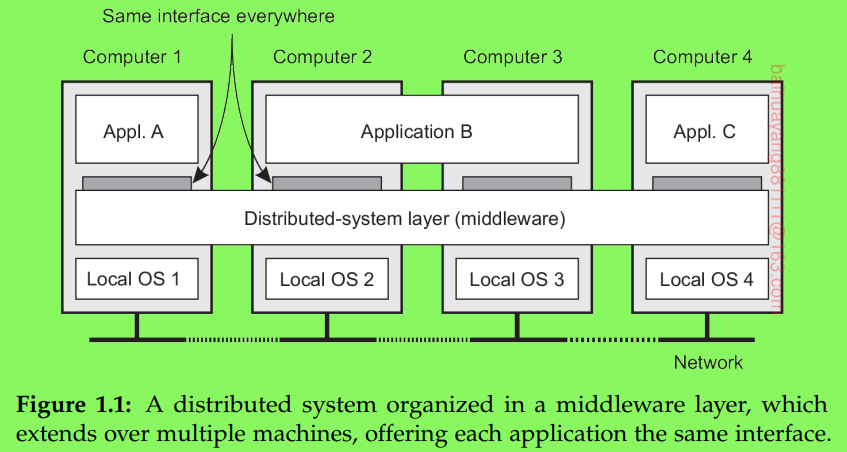


图1.1 中有四个网络计算机和三个应用，B应用分布在计算机2和3上。每个应用都提供相同的接口。分布式系统为单个分布式应用的不同组件提供了方法，也为不同的应用提供了通信的手段。同时，它还尽可能的隐藏了不同硬件和操作系统的不同。

某种意义来说，中间件对于分布式系统的意义，就相当于操作系统对于计算机的意义。即一个在网络中分配、提供和管理资源的一个组件。仅次于资源管理，它能提供类似操作系统的服务：

内部通信的设备

安全服务

Accounting services

掩饰并恢复错误

他们唯一的不同在于，中间件是存在于网络环境中的。事实表明，大部分服务是都是应用有所帮助的。基于这种原因，中间件可以当做是一个常用组件和函数的容器，已经不需要由应用程序单独实现。为了证明这些观点，让我们短暂的考虑几个典型中间件服务的例子。

**通信**：一个大众化的通信服务被称作RPC（远程过程调用）。一个RPC服务可以使得应用在远程计算机执行一个方法，但看起来就好像在本地调用一样。开发者无需在特殊的编程语言中特殊化函数的写法，RPC子系统会生成必要的代码来执行远程调用。

**事务**：许多应用使用分布在很多计算机上的各种各样的服务。中间件通常会利用原子化的事务操作，对这些服务的调用提供特殊的支持。在这种情况下，应用开发者只需要保证远程服务被调用，并且遵循了某种协议，剩下的中间件会保证服务被调用或者没有被调用。

**服务组成**：现在有一种趋势，将已经存在的程序组合而成为新的应用。这在Web应用中体现很广泛。基于Web的中间件可以帮助Web服务提供的服务调用标准化，是方法调用在一个特殊的顺序。一个关于Web服务混搭的例子是：Web 组合并聚合不同来源的数据。有名的混搭是哪些基于谷歌地图的旅行计划应用和天气预报应用。

**可靠性**：最后一个例子，这有一个有价值的研究，是关于如何提供建设可靠分布式系统的方法。Horus toolkit 让一个开发人员构建一组进程，可以使得任何一个进程发送的消息，要么被所有进程收到，要么所有进程都收不到。这个结果可以大大简化分布式系统应用，并且通常作为部分中间件的实现。

## 设计目标

这一节，我们讨论4个重要的目标，这些是分布式系统的建设应该满足的。分布式系统应当是的资源可以容易访问；应当隐藏资源是分布在一个网络中；应当对外开放（使得其他节点可以加入）；可扩展。

**支持资源分享**：

分布式系统的一个重要的设计目标是是的用户或者应用能够容易的访问远程资源。这里的资源可以使任何事物，但典型的是外围设备，存储设备，数据，文件，服务和网络等。分享资源的原因有很多。一个明显的原因是经济成本。例如：将数据存储在一个可共享的设备中要比为每一个用户单独存储一个资源的成本低。

用户和资源的连接是的信息的交换变得容易，这也要归功于简单的网络协议（这里的简单应该是只方便）。互联网的连接使得地理上分布很广泛的人可以在一起工作，如用于协作编辑、电话会议等。

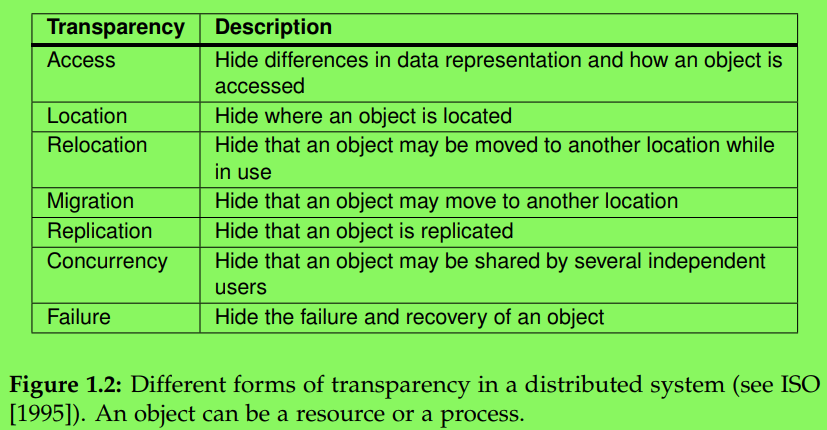
然而，分布式资源分享最好的体现还是在peer-to-peer的文件分享上，就像BitTorrent这个软件。这个分布式系统使得用户通过互联网分享文件变得容易。Peer-to-Peer 网络通常和分布式文件介质相关，如音频和视频。在其他的情况下，这个技术已经用在通过主机间的软件升级，服务备份，数据同步等大数据的分布式网络中。

**使得分布式透明：**

分布式系统的一个重要的目标是掩藏进程和资源是分布在很远距离的事实。换句话说，对用户和应用来说，分布式系统使得资源和进程的分布变得透明。

**分布式透明的类型：**

分布式系统在很多方面有透明这个概念，几个比较重要的如下图1.2所示。我们用Object表示一个进程或者一个资源。



**访问透明** 对数据展现方式的不同以及数据访问的隐藏。基本上，我们想要隐藏机器架构的不同，但更重要的是我们将对不同机器和操作系统的展现方式加以隐藏。举个栗子，一个分布式系统会有多个操作系统，每个操作系统都有自己的文件命名特性。不同的命名特性，不同的文件操作，不同的底层进程通信都是应该对用户和应用透明的（隐藏）。

一组重要的透明类型涉及到进程和资源的位置信息。位置信息透明的意思是，用户或应用无法分辨一个进程或资源的具体物理位置。命名在位置透明的功能里起到了重要的作用。特别是，位置透明通常可以通过设置资源的逻辑路径来实现，资源的位置命名并不是秘密的编码的。举个例子就是 URL ，<http://www.prenhall.com/index.html>，这个url并没有具体给出Prentice Hall 的Web服务地址。这个URL也没有给出index.html是否一直在现在的地址或者最近才移动过来的。就是说，可能整个网址是从一个数据中心搬移过来的，然而用户却没有发现。第二个例子是重定向透明，这一点在云计算中十分重要，稍后我们将讲解这部分。

重定向透明指的是分布式系统，而迁移透明更多的是指由人产生的迁移，并且不影响通信和操作。例如移动中打电话，移动中开电话会议等。

同样的，复制在分布式系统中也扮演者重要的作用。例如，复制的资源可以增加可靠性，并且把资源复制到访问路径可以增加性能。复制透明需要将存在多份相同资源的事实隐藏起来，并且多份资源可以保证一份出问题时，其他的复制资源可以进行代替。为了对用户隐藏复制，我们需要将所有的复制资源进行相同的命名。因此，一个支持复制透明的系统应当也支持位置透明，否则不太容易实现。

我们提到分布式系统的一个重要目标是资源分享。在通信层次上，资源分享体现在一种合作层面上。例如，两个独立的用户需要访问服务器的同一个资源或数据库的同一张表。在这种情况下，让用户感觉不到另外有人在访问同样的资源是十分重要的。这种叫做**并发透明**。一个重要的课题是这种并发访问需要让资源保持一致性。这可以通过加锁的机制，例如通过给需要的资源加入排它锁。一个更完善的方案是使用事务，但这在分布式系统是是很难实现的，这会导致可扩展性的问题。

其次，提供**错误透明**也是十分重要的。这表示，用户或应用将感知不到系统的某台机器发生错误，而系统将会自然而然的进行修复。错误掩盖在分布式系统中是一个很艰难的课题并且经常是不可实现的，我们将在第八章阐述。错误掩盖最主要的困难就是无法判断一个进程究竟是死亡了，或者只是相应太慢。正如我们访问一个运行很繁忙的服务器时，浏览器最终超时并报告服务不可用。而我们根本无法盘点这个服务就是挂掉了还是网络拥塞的原因。

**分布式系统的透明度**

虽然对每一个分布式系统，透明性都是很重要的，但也不是所有的分布式系统都要盲目的掩盖系统的每一个特性。一个简单的例子就是，如果你想要当地7点的早上新闻，但是分布式系统的主机在另外一个时区，给你推送了一个晚间的新闻报道。

同样的，一个覆盖很广的分布式系统在两地想要通信，比如在旧金山和阿姆斯特丹，那么访问延迟肯定不能小于35ms，至少需要几百毫秒，这不仅仅是由于光速的限制，童谣受限于其他一些介质。

同样在性能和透明度上也有一些冲突。比如，我们会反复的访问一个节点知道它失败。而分布式的透明性告诉我们，它会去寻找其他方法帮助访问，而不会立即返回失败，这就会降低系统的响应性。这种情况下，我们应当更早的选择失败，或者让用户取消这次访问。

同样的，我们了解备份数据的一致性必须要保持。当一份数据发生变动时，所有的备份数据都需要更新。在更新完成之前，是不允许其他操作的。而我们知道一次更新大概需要几秒钟，这对于用户来说是不可隐藏的。

综上，我们发现隐藏分布式系统所有特性并不是最好的方式。随着分布式涉及到人们随身携带的设备，并且地理位置和消息推送变得越来越重要，适当的暴露分布式的信息可能是更好的选择。一个明显的例子就是手机的定位服务，比如查找最近的外卖和附近的朋友。

还有一些其他的关于分布式透明性的争论。考虑到完全的分布式透明是不可能的，我们应当对一些可以实现的方面进行透明控制。将分布式系统暴露出来而不是隐藏，可能对于用户和开发者来说是更好的选择，这样会让用户和应用开发者不会考虑到系统中会有透明性这样的东西存在。这样用户将会更好的理解分布式系统的行为，并更好的处理分布式系统的一些行为。

结论就是，对于分布式系统的透明控制是一个很好的目标，但同样应该考虑到系统的性能和可理解性。做到完全透明的系统所付出的成本是非常高的。

**保持开放**

分布式系统另一个重要的目标是开放。一个开放的分布式系统能够提供一些容易被使用的组件，并集成到其他系统中。同时，也能够使用一切其他地方的组件。

**互操作性、可组合性、可扩展性**

开放意味着，组件需要遵循一些标准，并准确的描述该组件的一些语法和语义。一个普遍的方法是使用服务接口语言(IDL)来描述。接口定义使用IDL来表示服务的语法。也就是说，他准确的描述了方法名，参数名和类型，返回值，以及可能抛出的异常。最难的部分是语义解释，解释该接口的功能。实际上，这种描述已经在很多自然语言中使用。

合适的描述是完整和中性的。完整意味着接口的每一个必须的属性都要进行描述。然而，很多接口定义并不完整，所以需要开发者增加实现细节的描述。所以描述应当是中性的。

别人提出的观点，完整性和中性对于可操作性和可移植性是很重要的。互操作性的特征是系统的不同实现和厂商的不同组件可以共存并共同工作且互不依赖。可移植性表示在分布式系统A开发的应用，可以不用修改的在分布式系统B应用。

另一个重要的目标是，分布式系统可以容易的由各个不同的组件组成。同样的，在不影响组件的前提下，可以很容易的增加组件也是必要的。也就是说，系统是可扩展的。可扩展系统是可以很容易的在不同的操作系统上更换组件，或者更换文件系统的。

**方案机制分离**

为了使开放的分布式系统更有弹性，关键点是系统需要一些相对较小并且容易替换的合适的组件。这表明我们应该不止提供最高级别的面对用户和应用的接口，同样需要定义一些针对系统内部组成并描述他们之间如何运作的接口。这个提议相对来说比较新。很多老的甚至是比较现代的系统还是使用的是逻辑上分离但是却实现为一个的巨大的程序。这样想要更换其中的组件就会比较困难。这样的系统不是开放的。

改变分布式系统的需要起源于一些不适用于特殊用户和应用的组件。比如，浏览器缓存。这里有许多不同的参数需要考虑：

**存储**：数据缓存在哪里？通常是在内存中，其次是硬盘。如果是后者，具体在文件系统中的位置就需要考虑到。

**删除**：当缓存满了，如何删除呢？

**共享**：每个浏览器一个缓存还是每个用户一个缓存呢？

**刷新**：什么时候取刷新缓存呢？如果不用通过连接站点就能够返回页面，那么缓存通常是有效的。然而，这样会有一些风险，比如返回不是最新的数据。刷新的频率通常取决于存储的数据。如果数据不经常改变，则没有必要经常刷新，但如果是像股票这样的数据，则需要经常刷新。

这里我们需要的是方案与机制的分离。在页面缓存这个例子中，一个浏览器需要提供存储的方案，同事需要用户决定哪些文件需要存储，以及存储多久。实际上，这些可以通过给定一些可选的参数给用户来解决。更进一步说，浏览器可以提供插入工具作为用户单独组建实现的方案。

**可伸缩的**

对我们来说，世界范围的互联网连接就和发送明信片一样普遍。我们过去一直在使用相对强大的桌面电脑和办公软件，现在，我们瞄准的是哪些称作是云的服务，这种云的服务是由许多小的网络设备组成的。处于这方面考虑，可伸缩性成为一个重要的分布式系统的设计目标。

**可伸缩维度**

系统的可伸缩维度可以从3方面考虑

**规模可伸缩性：**

系统在规模上是可伸缩的表明，我们可以容易的增加用户和资源且不会担心性能会有所损失。

**地理可伸缩性**：

系统在地理位上可伸缩表明，我们的用户和资源即使相距很远，通信延迟也是可以忽略的。

**管理可伸缩性：**

系统在管理上的可伸缩性表明，我们可以容易的管理系统，即使系统处于很多独立的系统结构中。

细看一下这三种维度的伸缩性。

**规模可伸缩性** 当一个系统需要伸缩性时，会出现很多不同的待解决的问题。首先考虑规模，当更多的用户和资源注入时，出于多种原因，我们的中心服务会遇到一些限制。例如，在分布式系统中，许多服务都是跑在一个单点机器上。在现代的配置中，我们会在一个地方有一组紧密联系的机器。问题是明显的，这样的单点或者一组机器，在遇到请求量急剧增大的情况下，会成为分布式系统的性能瓶颈。为了说明这是如何发生的，我们拿单点举个栗子，这主要体现在三个方面：

计算容量手CPU的限制

存储容量，包括IO效率

用户和中心化的服务

让我们首先考虑计算容量。如果我们遇到的是一个一次请求就要花费好多秒的请求，如果这个请求量爆发，那么这个单点是会遇到问题的。

同样的问题发生在IO上。在一个设计糟糕的中心化的搜索引擎上。当我们对一个内容进行搜索时，即使是最先进的索引技术，我们仍然会面对大量的超出我们内存的数据的处理。因此，大部分性能消耗在硬盘的访问和硬盘与内存之间的数据转移上。而仅仅的增加更多的高速硬盘并不是一个好的解决高访问量的方法。

最后，用户和服务之间的网络也是一种障碍。假设有一个为许多用户提供高质量的视频的服务。一个是频率可以轻松达到8-10兆的带宽，意味着如果客户是点对点的连接，将很容易达到带宽的上限。

这里有几种解决方法，在我们讨论完地理和管理伸缩性之后讨论。

**地理可伸缩性**： 地理可伸缩也有自己的问题。其中一个就是使一个在本地的分布式系统规模化是很困难的，因为他们需要同步通信。在这种通信中，请求服务的一方，需要阻塞直到一个服务端的回答到来。更特殊的是，我们经常看到一种通信模型，是有许多服务端和客户端组成的，例如数据库的事务处理。这种方式在局域网的情况下是工作的很好的，因为网络延迟最多几百毫秒。然而，在一个广域的系统中，这种内部的通信是需要几百毫秒的，慢了三个数量级。搭建使用同步通信的关于系统需要特别的小心，尤其需要注意客户端与服务端的内部连接。

地理伸缩的另一个问题是，广域的网络是不可靠的。而且我们需要处理带宽的限制。所以用来解决本地区域网络的方法对于广域的系统并不适用。一个典型的例子是视频流。在本地网络中，即使只有稳定的无线网络连接，实现高质量的视频流也是很轻松的。同样在广域网络中就会失败，带宽限制和网络的不可靠性都将发生。

另一个问题是，广域系统很缺乏多点的通行设施。相反，本地网络可以有效的支持广播机制。这种广播机制对于发现组件和服务是很有用的，是管理分布式系统的重要方法。在广域系统中，我们需要开发分离的服务，比如命名和文件路径的服务，这样我们就知道查询的位置在哪里了。这些服务同样需要规模化，而且目前没有明显的解决方法，这些我们将在之后的章节中遇到。

**权限可伸缩性**：最后，也是最困难的，发生在很多开放式系统中的问题是，如何在多个独立的权限系统中实现规模化。这里的主要问题指的是，资源使用，管理，和安全。

为了说明这种情况，科学家们一直在研究如何在一个计算网格中分享他们那些昂贵的设备。在这些网格中，一个整体的分布式系统是有每个本地的分布式系统组成的，这些本地的分布式系统允许一个运行在A系统的程序访问B系统的资源。

例如，一个单一区域的分布式系统的组件是会被同一个区域的用户所信任的。在这种情况下，系统管理员也许会采取测试和认证等方式，确保这些组件不会被篡改。一般情况下，用户是会信任系统管理员的。然而这种信任并不会扩散出区域。

一个分布式系统如果要扩展到其他的区域，两个安全措施必须实行。第一，分布式系统必须防止新区域的恶意攻击。例如，新区域的用户应该只有原始区域系统的只读权限。同样的，一些昂贵的设施和高性能的机器不能提供给未授权的用户。其次，新区域必须防止自己受到恶意的攻击。比如web的下载程序。实际上，新区域并不知道外来的代码应该是什么样子的。这类问题和解决方法我们将在第九章详述。

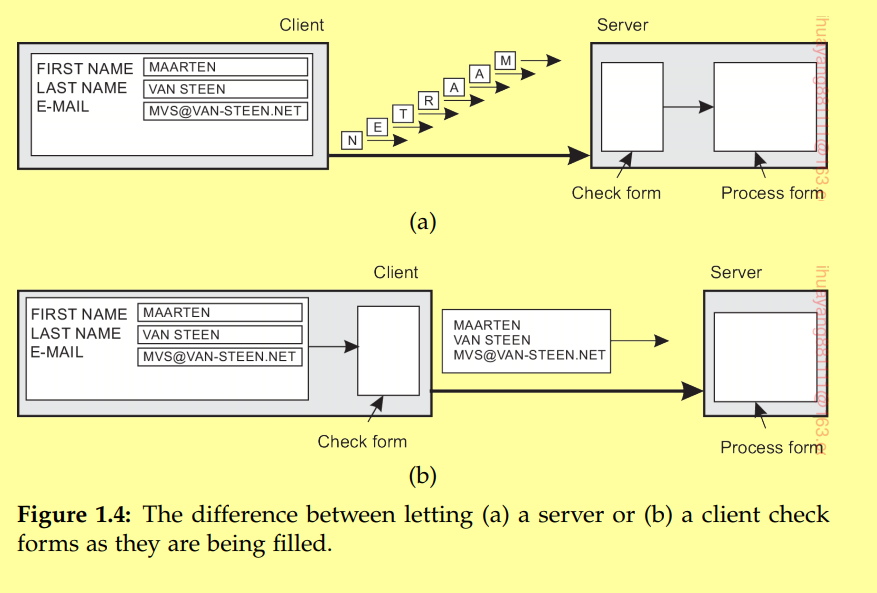
现代peer-to-oeer的网络文件分享系统，是一个分布式系统跨区域问题的一个反例。这个系统显然不收权限可伸缩性的困扰。在这些例子中，端用户只是简单的安装分布式系统下载程序，并在几分钟之内下载文件。另一个例子是skype的peer-to-peer的电话应用，Spotify的视频应用等。这些分布式系统的共同点是终端用户保持着整个系统的运行。如果可以，一些机构如ISPs可以对分布式系统进行管理，但现在还并不是有效的方法。

**伸缩技术**

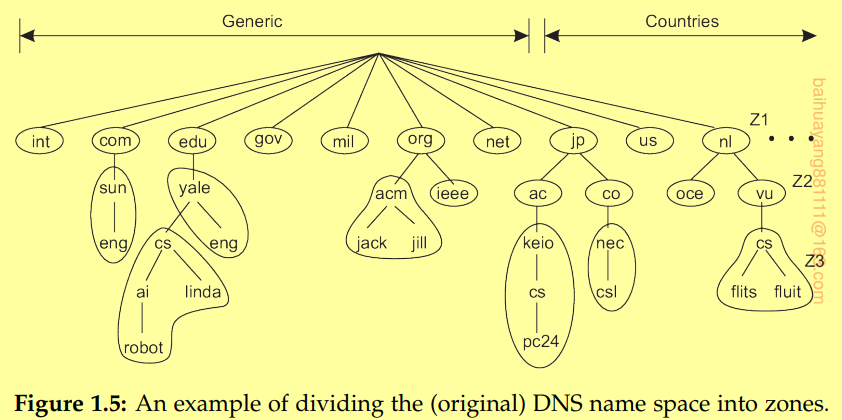
以上我们讨论了可伸缩性的问题，接下来我要们讨论如何解决。通常情况下，分布式系统的可伸缩问题都表现在服务器和网络的性能问题上。简单的扩容对于scaling up 通常是一个不错的选择。但当遇到 scaling out时，我们通常需要三种技术手段：隐藏通信延迟，分布式工作，复制。

**隐藏通信延迟** 隐藏通信延迟通常是对地理伸缩使用的。简单的思路是，尽量防止远程服务的响应等待。例如，当我们求情一个远程主机时，除了等待服务器的相应之外，还可以执行其他有用的工作。这表示请求者使用的是异步通信。 当一个回复来到时，应用将会被终端，并且一个特殊的句柄会被调用来完成之前的请求。异步通信在批量执行系统和并行应用中经常使用。这些系统和应用会有独立的任务执行，同时其他的任务等待通信的完成。另一种选择是，使用一个新的线程来完成这次请求，这样就不会影响其他的线程执行其他的任务了。

可是，许多应用并不能有效的使用异步通信。例如，在互动应用中，一个用户发送请求之后，并没有其他的事情好做，只有等待回复。这种情况下，最好的方法就是降低通信的次数，比如，使用一些完成任务的资源去执行这些通信。一种应用就是使用表单。使用表单可以将分别发送的消息作为一个整体并等待服务的相应。服务端会先校验表单的输入是否满足一定的语法。如下图所示：



**分区分布** 另一个重要的伸缩技术是分区分布，它会将一个成分拆分成小的部分，并将它们分散到系统的各个部分。一个很好的例子是DNS系统。DNS的命名空间是一个树形的结构，分割成不重叠的区域，如下图所示。每个区域的名字都被一个单独的服务所使用。暂时不去深入讲太多（第五章会介绍），我们想象每一条路径都是互联网上的一个主机名，这就是一个与网络地址相关的主机。所以，解析一个名字就以为止解析这个网络地址的主机。例如flits.cs.vu.nl。如果要解析这个域名，我们首先来到Z1区域，这个区域返回Z2区域，剩余的域名 flits.cs.vu 将会被处理。Z2区域将会返回Z3的地址，Z3会处理域名的最后一部分，并将返回对应的主机。



这个例子解释了像DNS这样的命名服务，是分布在若干个机器上的，这样防止了一个机器必须应付所有请求的情形。

另一个例子是，世界万维网。对大部分用户来说，这个网络是一个巨大的文件资源网，在那里的每一个文件都通过一个URL来独享一个名字。概念上来讲，也可以将它看成是一个服务。然而，这个网络在物理上分布在几百万个主机上，每一个都有大量的文件。每一个主机所处理的文件被编码成URL。这个万维网是能够使用分布式来完成规模的伸缩的。

**复制** 考虑到可伸缩性经常会产生性能下降，一个好的办法是在分布式系统中进行复制。复制并不能增加可靠性，但可以帮助均衡负载来协助保持性能。同样的，在地理伸缩分布式系统中，复制可以隐藏通信延迟的问题。

缓存是一种特殊形式的复制，虽然这两者有时很难分辨。在复制的情形下，缓存相当于是资源的一个副本，相当于客户端访问了这个资源。不同的是，缓存是客户端发起的一种资源复制。

缓存和复制会导致一些缺点。因为我们复制了很多资源作为副本，修改一个副本将导致副本的不一致。这样会导致一致性问题。

资源的不同使用对于不一致性的容忍度是不同的。如果用户上网冲浪的话，是不在意浏览器返回一个缓存的结构。如果用户看的是股票的信息，则必须返回实时而不是缓存的数据。强一致性的问题是，一个点的更新必须迅速的传播到其他的副本上。如果两个更新并发发生，如果我们需要这两个更新按顺序发生的话，将会导致分布式系统的更新顺序问题。更严重的是，如果考虑到要保证一些其他的性质，比如可靠性，一致性是不能够实现的。关于这一点，我们将在第8章讨论。

复制需要很多的全局同步机制。不幸的是，由于网络延迟的原因，这些机制在规模伸缩的情况下是很难实现的。因此复制可伸缩性会产生一些其他的非伸缩的问题。我们将在第7章阐述。

**总结讨论** 当我们讨论伸缩性技术时，规模可伸缩性被认为是最容易解决的。通常通过增加机器的容量可以解决，虽然这会有一些成本。地理可伸缩性受制于网络延迟。因此，我们应当在客户端附近去复制资源，然而这就会产生数据一致性的问题。实践证明总和分布式，复制和缓存技术并总和各种不同性质的一致性，可通常解决问题。最后，权限可伸缩性是一个最难解决的问题，因为我们将面对非技术的问题，比如一些人为因素。现在的peer-to-peer技术可以解决这个问题，然而peer-to-peer并不能解决所有的问题。

**陷阱**

现在我们已经了解开发一个分布式系统是一项艰巨的任务。在这本书中，我们多次看到最终的结局方式都是很复杂的。然而，如果能遵循一系列的设计原则，分布式系统的开发工作还是能够向着目标前进的。

分布式系统不同于普通的软件，是因为他的各个组成部分分布在网络中。正式这种网络上的分散，导致了许多系统设计的不必要的复杂。有个叫Peter Deutsch 的人，总结了许多人们在构建分布式系统时设定的错误的假设：

网络是可靠的

网络是安全的

网络是同构的

网络拓扑不会改变

延迟=0

带宽无线

传播成本是0

总会有一个管理员

这些大部分在非分布式系统中都不是问题。

这本书中讨论的大部分原则都和这些假设有关。在任何情况下，我们都将讨论在这些假设不成立的情况下的一些解决方法。比如，不存在可靠的网络并且不能掩盖错误。我们将会用一整个章节讨论这种不安全的网络通信。我们同样讨论到分布式系统应当对外开放，并且将异质性考虑进去。永阳的，当讨论用复制来解决伸缩性问题是，我们有必要解决延迟和带宽问题。我们同样在这本书中会看到不同的管理的观点。

## 分布式系统的类型

在讨论分布式系统的原则之前，我们先看几个不同类型的分布式系统。我们先来区别下分布式计算系统，分布式信息系统，和其他的流行分布式系统。

**高性能分布式计算系统**

分布式系统的一个重要功能室高性能计算。这可以分两方面来说。在**集群计算**方面，分布式系统由一些相似的工作站和PC机器组成，这些机器在本地网络近距离的连接。另外，每一个节点都运行在同一个操作系统。

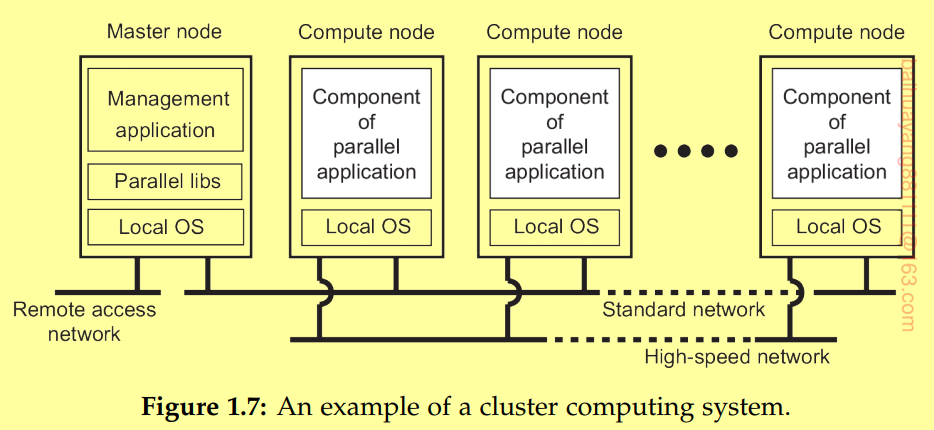
在**网格计算**中则变得有所不同。参考Wikipedia网格计算的定义。这种类型的分布式系统通常是有小的系统联合组成一个计算机系统，这里每一个小的系统都有不同的管理域，同样在硬件，软件和网络拓扑上都是不同的。

对于网格计算，下一个逻辑步骤是将密集型的应用的所有设施外包出去。实际上，这正是云计算所有考虑的事情，对于那些需要的服务，提供小的设施并动态的组成一个基础设施。不同于网格计算所要求的高性能，云计算更多的是提供一些资源。我们这里简单的介绍一下，之后的内容将在这本书中贯穿下来。

**集群计算**

自从个人计算机和工作站的性价比大大提升之后，集群计算就变得很时髦。在某一时刻，无论从财务还是技术上考虑，通过高速网络连接简单的计算机来构造成一个超级计算机变得很有吸引力。几乎在所有情况下，集群计算被用作并行计算的程序。

一个应用很广泛的集群计算的例子是贝奥武夫集群，这几集群的配置如下图所示。每个集群由一系列计算机节点组成，这些节点有一个主节点控制。主节点负责特定并行程序的节点分配，维持着一个队列的任务，并为系统用户提供接口。主几点通常去运行集群管理和程序运行所必须的中间件。同时，配置了标准操作系统的计算节点通过中间件的同喜，存储，和错误。除了主节点，其余的计算节点看起来都是相同的。



一个类似的系统是MOSIX系统。MOSIX尝试提供集群的单一系统镜像，使得运行在集群上的程序就好像运行在单一的机器上一样。像我们提到的，在满足所有情况下提供这样一个镜像是不现实的。在MOSIX这个例子中，通过使进程动态的在节点间迁移来造就这个高度透明的系统。进程迁移使得用户可以在任何一个机器上启动应用，之后为了有效的利用资源，这个进程可能转移到其他的节点。我们将在第三节讨论进程迁移。

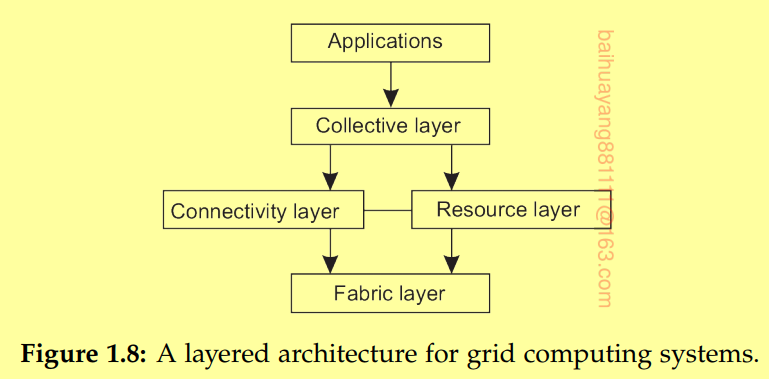
然而，现在的集群系统并没有使用这种架构，而是使用了混合策略，将中间件按功能分割成不同的节点。这么做的优点是，让计算节点使用更轻量级的和更专一化的系统将提供更好的性能。比如，存储方面可以更好的被那些专门设置了文件和文件系统的节点优化。另外一些中间件服务，比如任务管理，数据库服务等也由对应的节点提供。

**网格计算**

传统集群计算的一个明显特质是同质化。大部分情况下，集群的计算机大部分都一样，同样的操作系统，同样的网络中。然而如刚才讨论，现在流行一种混合架构是的特别的节点去适配特别的任务。这在网格计算中更能得到体现。在网格计算中，没有关于硬件，操作系统，网络，管理权限等需要相同相似的假设。

网格计算的一个主要课题是不同组织的资源为不同的用户服务，并组成一个多样化的系统。这样一个联合组织成为虚拟组织。通常，资源包括主机，存储设施，数据库和网络设施等。

基于网格计算的特性，很多软件都在不断发展来提供对来自不同管理域的资源的访问，以及特定虚拟组织的用户和应用。所以之后工作的焦点都聚焦在架构课题上。最初的架构如下图所示，这个架构组成了网格计算的基础部分。



这个架构分为4层。最低的fabric层提供本地和特定站点的接口。这些定制的接口可以使资源在一个虚拟结构中进行分享。通常，他们可以提供函数来查询资源的状态和容量，以及特性资源的管理。

连接层由许多用来调转多种资源的通信协议来支持。例如，通过通信协议我们可以将资源进行转换，或者从远端访问资源。另外，连接层会有许多安全协议用来认证用户和资源。在许多情况下，操作人时不会被验证的，而被执行的程序会被验证。在这种情况下，在连接层将权限从一个用户委托给一个一个程序是很重要的。我们将在第九章讨论。

资源层的责任是管理资源。它将使用连接层的方法以及fabric层的接口。例如，这层会提供访问特殊资源的配置信息的接口，或者用来创造一个进程或者读取数据。这个资源层看起来需要对控制权限负责，所以将会作为连接层验证的一部分。

接下来是汇聚层。它用来处理多种资源，并有服务发现，分配，执行任务计划等服务。不同于连接层和资源层由小的标准的协议结合组成，汇聚层由许多不同的协议组成。

最后，应用层有许多虚拟组织的应用组成并构成了网格计算环境。

通常，汇聚，连接和资源层构成了网格中间件的核心。这些层通过资源和管理连接在一起为多个站点提供服务。

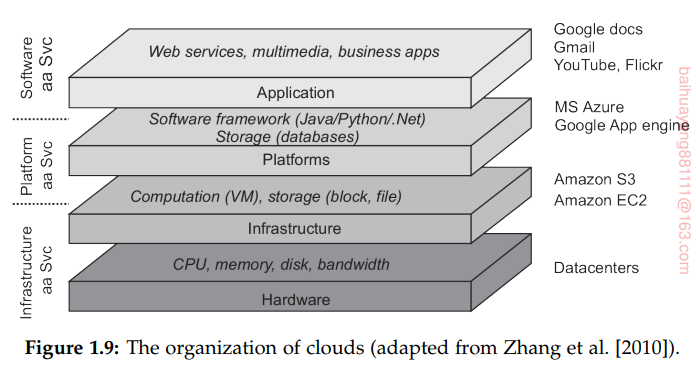
从中间件的角度来看，对于网格计算，站点的概念是相同的。这种面向服务的体系结构是很普遍的，这些站点通过互联网服务集合来访问这些层次结构。这就引出了一种称为开放网格服务的体系结构OGSA。OGSA是基于Foster等人提出的原始思想，通过标准化过程使其变得更复杂。OGSA通常遵循web服务的标准。

**云计算**

当研究者正在思考如何组织容易访问的计算网格，以此来负责面向客户开放的资源。最终，这就引出了效用计算的概念。用户可以通过上传任务到数据中心并按照资源进行收费。效能计算就构成了现在所谓的云计算。

云计算是以简单可用和可访问的虚拟资源池构成。资源的使用可以动态分配，提供基础的可伸缩性：如果用户需要更多的资源来完成任务，那么就申请更多的资源。与效能计算有联系的是，云计算是基于实用模型付费的，这样保证了可以提供用户定制的服务。

通常，云计算由4层构建，如下图所示。



**硬件**：最底层由所必须的硬件组成，处理器，路由器，电源和制冷系统。这些都是由数据中心实现并且用户通常都是不可见的。

**基础设施**：这个是形成云平台骨干重要的一层。这里开发出了虚拟技术为用户提供虚拟存储和计算资源。实际上，云平台就是用来对虚拟存储和虚拟主机进行分配儿存在的。

**平台**：一般认为平台层提供给用户的就像是操作系统提供应用给开发者一样，也就是容易的在云服务上开发应用。实际上，云平台给应用开发者提供了特定的接口，包括程序的上传和程序的执行。在某种意义上，这类似于Unix系统的系统调用接口的功能。

类似于操作系统，平台层提供对存储的更高级别的抽象。比如，亚马逊S3存储系统为开发者提供了API本地创建的接口使得文件可以被组织和存储在buckets中。Bucket可以比作一个文件夹。当存储一个文件在buckets时，文件将自动上传至亚马逊云端。

**应用**：通常在这一层提供给用户的应用都是定制化的。很有名的例子办公室需要的东东。同样重要的是这种应用是反复在供应商的云中执行的。这可以类比以前在操作系统中的应用。

云计算通过各种各样的接口为用户提供这些层服务(命令工具，程序接口和网络接口)，这些导向了当中服务：

**Infrastructure-as-a-Service(Iaas)** 包含硬件和基础设施层

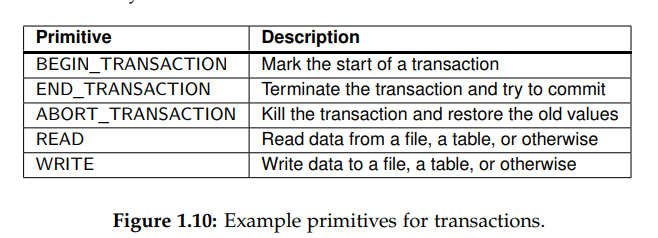
**Platform-as-a-Service(Pass)** 包含平台层

**Software-as-a-Service(Saas)** 包含应用层

到目前为止，使用云变得相对容易了，我们将在后续章节通过具体例子来讨论。作为一个结果，云计算作为外包本地计算的一种手段已经被许多企业所认可。然而，云计算仍会遇到许多困难包括安全性，服务可靠性等。同样的，因为云计算的执行细节通常是被隐藏的，甚至是未知和不可预测的，所以要提前安排满足性能需要的资源变得可能性很小。更重要的是，li等人已经在[2010]中证明了这一点，不同的云计算提供者容易显示出不同的性能。从此，云计算不再是炒作，而是可以代替本地巨大设施的一种手段，但云计算仍有很大的提升空间。

**分布式事务**

为了方便阐述，我们集中讨论数据库应用。实际上，数据库操作就是一种实务操作。程序使用事务需要分布式系统以及系统运行语言提供特殊的原语。通常的事务原语如下图所示。这些确定的原语列表是基于在事务中使用的数据类型。在一个邮件系统中，会有发送，接收，转发等原语。在账户系统中，原语将会不同，比如 读，写。过程调用，原始陈述等同样会包含在事务中。特别是，远程调用，通常被封装到一个事务中，称作事务RPC。我们将在4.2节讨论RPC。



BEGIN\_TRANSACTION 和 END\_TRANSACTION 用来划清事务的边界。在边界内的操作构成事务的执行体。事务的特性是，要么这些操作都被执行，要么一个都不被执行。这次操作可能是系统调用，库的过程调用 或者bracketing statements，通常由实现决定。

这种原子性是事务四个特性之一。事务本身有ACID属性：

**Atomic** ： 原子性

Consistent: 一致性

Isolated: 独立性

Durable: 永久性

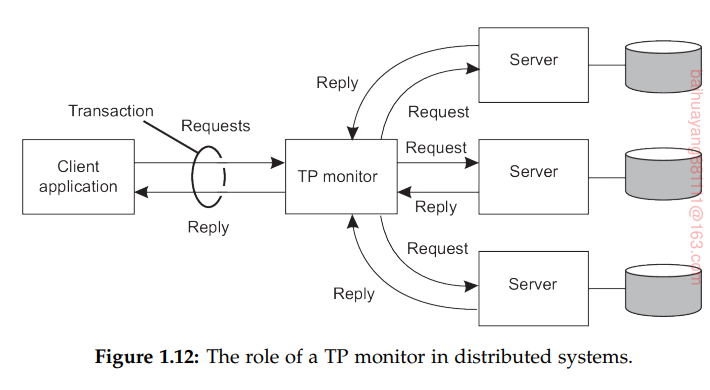
如下图所示，在分布式系统中，事务通常由一系列子事务交织而成。最顶层的事务会衍生出子事务并行执行在不同的机器上，用于获取更好的性能。这些子事务也有可能再衍生出子子事务。

子事务很精致，但也会出现问题。想象一个事务会并发出若干个子事务。当其中一个提交时，父事务将会看到这个子事务的提交结果。在之后的计算中，如果父事务终端，整个系统将会恢复到事务开始时的状态。这样，子事务提交的结果需要被撤销。所以只有最高级别的事务才能拥有执行的永久性。

由于事务可以被嵌套到很深的程度，因此需要大量的管理才能保证一切正常。当任何事务或子事务启动时，都会给它一个整个系统的私有副本，以便能按照自己的意愿行事。如果事务终止，他的私人空间被摧毁，就好像什么都没发生过一样。如果他提交了，他的私人空间代替了之前的空间。这样，如果一个事务提交了，然后另一个事务开启了，则第二个事务将会看到第一个事务的结果。这样，如果一个高级别的事务终止了，所有子事务必须终止。如果若干个事务同事开始，结果就像是他们以某种不定的顺序执行一样。

事务交织在分布式系统中是很重要的，因为他们需要提供一种分散在过个机器中的事务。例如，比如让一个事务计划翻转三个flights，这就需要将其分为三个子系统。这三个子系统可以独立的执行。

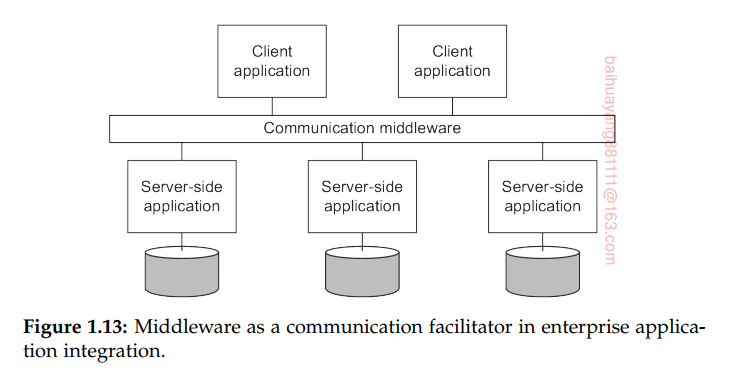
在早期的企业中间件系统中，分布式事务的组件构成了主机和数据库层级集成应用的核心。这个组件叫做事务进程监视器，简称TP。它的主要任务是允许一个应用通过一个事务模型来访问多重主机和数据库，如下图所示。特别是，TP会基于一种协议来协调子事务的提交过程，我们将在8.5节讨论。



一个重要的发现时这些应用更愿意协调这些子事务成为一个单一的事务，而不是协调这些子系统本身。基于简简单单的使用TP，他们之间的协调就完成了。这就是中间件系统的作用：它实现的服务是对许多应用有用的，而不是需要开发者去反复实现这些服务。

**企业应用集成**

正如之前提到的，应用越是和数据库解耦，集成应用的设备就越需要和数据库独立。特别是，应用组件应该可以和其他组件通信，而不仅仅通过request/reply等方式来传播。这种内部应用的通信需求导致了许多不同的通信模型，主要思想是存在的应用应该能直接交换信息，如下图所示。



现在存在几种通信中渐渐。通过远程调用(RPC)，一个应用组件可以有效地对其他应用组件发送请求。

随着对象技术的日益普及，现在技术允许远程访问对象，RMI。RMI和RPC类似，只不过他是操作在对象而不是方法上。

RPC和RMI有个缺点就是他们的调用者和被调用者都必须在通信时在线。另外，他们需要知道如何引用对方。紧耦合是一个缺点，这导致了面向消息的中间件的产生，简称MOM。在这种情况下，应用将消息发送到逻辑上的关系点，这种点通常用**subject** 来描述。同样的，应用可以使用一种特殊的消息指出他们的兴趣点，之后消息中间件会将他们的消息发送到指定的应用。这种消息发布订阅系统组成一个重要的扩展的分布式系统。