# 进程

在这一章中，我们将研究各种不同类型的进程是如何影响分布式系统工作的。进程的概念起源于操作系统中，它是用来定义一个程序的运行的。在操作系统层面，进程的管理和调度可能是最重要的课题。然而，当在分布式系统中讨论时，进程的一些其他课题变得更为重要。

我们首先重点讨论线程和它在分布式系统的作用。事实证明，线程不但在多核多处理器环境中对性能有重要的作用，更有助于构造客户端和服务端。在很多情况下，我们看到线程被进程所替代，并且使用底层操作系统来保证保护和促进通信。然而，当性能受到威胁时，线程将扮演着重要的角色。

近些年，虚拟化的概念开始流行起来。虚拟化允许应用程序及其完整的环境(包括操作系统)与其他应用程序并发运行，但高度独立于底层硬件和平台，从而实现高度的可移植性。另外，虚拟化可以帮助隔离由故障和安全引起的错误。这对于分布式系统是很重要的概念，我们将单独讨论。

客户-服务器组织架构在分布式系统中很重要。在这一章节，我们将仔细研究客户端和服务端的典型组织。我们将注意到服务端的典型设计，包括那些在object-based分布式系统中使用的类型。一个应用很广泛的Web服务是Apache，我们将单独了解。服务端的集群组织也很重要，特别是他们需要合作得像是单一系统在工作一样。我们将介绍类似的例子，包括广域服务PlanetLab。

在广域分布式系统中，一个重要的课题是机器间的晋城移动。进程转移或者代码转移可以帮助实现可伸缩性，或者帮助动态配置客户端和服务端。关于代码转移还有其如何实现的，我们将在这一章节讨论。

## 3.1线程

虽然进程组成了分布式系统的基本单元，但实践表明，分布式系统以进程作为最小粒度并不足够充分（就是还有线程呗）。作为替代，有进程控制的多个线程组成了更小的粒度，这能更好的构建分布式应用并获得更好的性能。在这一节中，我们将关注在分布式系统中的线程是如何发挥作用的并解释它为什么这么重要。关于线程以及如何通过线程构建应用可以参考

[Lewis and Berg, 1998; Stevens, 1999; Robbins and Robbins, 2003]。这些很好的从概念上诠释了多线程并发程序。

**线程的介绍**

为了理解分布式系统中的线程，我们需要首先理解进程是什么，并且进程和线程是什么关系。为了运行一个程序，操作系统会创造一系列虚拟进程，每一个会运行不同的程序。为了跟踪这些进程，操作系统有一个进程对照表，包括存储CPU的寄存器只，内存映射，打开文件，计数器，privileges，等。这些条目构成了进程的上下文。

进程上下文可以看作是硬件处理器上下文的软件模拟。硬件处理器上下文包括硬件处理终端，以及存储CPU处理现场的最小信息。这个进程上下文包括程序计数器，或者其他的寄存器值比如栈指针。

进程通常定义为一个运行的程序，也即正在操作系统虚拟处理器上运行的程序。一个关键问题是，操作系统会很小心的保证独立的进程之间不会相互干扰的彼此。换句话说，多个进程共享同一个cpu以及其他硬件资源这件事，是透明的。通常，操作系统需要硬件的支持来完成这种分割操作。

这种透明性是有代价的。比如，当一个进程被创建时，操作系统必须创建一个完全独立的地址空间。这种分配可以意味着初始化内存段，例如，将数据段归零，将相关程序复制到文本段，并为临时数据建立堆栈。这样，在两个进程间切换CPU需要消耗性能。除了在寄存器临时存放数据外（包括程序寄存器和栈指针），操作系统同样会修改内存管理单元（MMU）的寄存器和使地址转换缓存无效，例如在转换后备缓冲区(TLB)中。另外，如果操作系统需要支持超过他内存大小的进程数量，那么需要在内存和磁盘间进行内存交换。

像进程一样，线程也会执行一段代码，并独立于其他线程。然而，不同于进程，如果实现并发透明性会导致性能下降，则不会尝试实现高度的并发透明性。因此，线程系统通常保持最小的信息来允许CPU可以被其他线程共享。特别是，一个**线程上下文**通常和进程上下文的组成没有区别，除了一些关于线程管理的信息。例如，线程系统可以跟踪一个线程当前在互斥量变量上被阻塞的事实，以便不选择它执行。关于其他线程的不必要的信息是不必理会的。因此，保护数据不受单个进程内线程的不适当访问完全留给应用程序开发人员。因为我们发现一个处理器的上下文包含在线程上下文中，而线程上下文包含在进程上下文中。

正如我们刚才概述的，部署线程有两个重要含义。首先，多线程应用的系统性能很难比同样的单线程系统性能差。实际上，多线程可以获得更多的性能。其次，由于线程不能自动的像进程一样保护自己不受其他线程干扰，多线程应用的独立需要更多的智慧和努力。合适的设计和事情的简单化是有帮助的选择。不幸的是，从目前的实践来看，这个原则并不好理解。

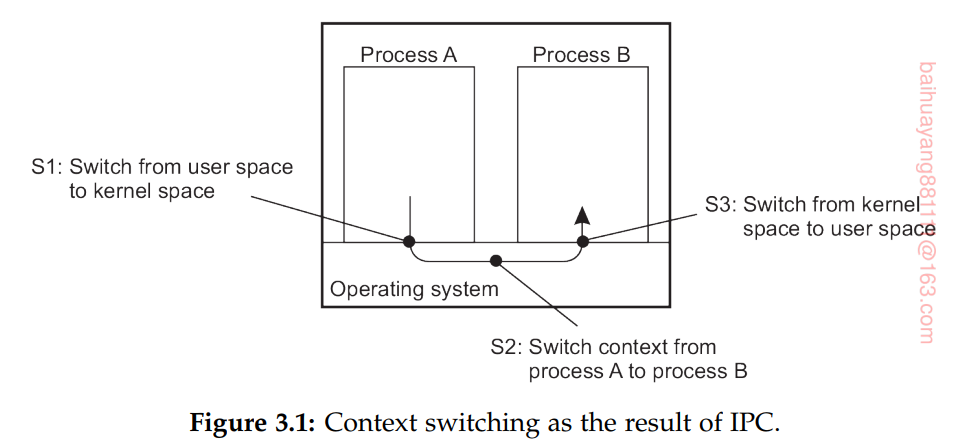
**在非分布式系统中的线程使用**

在讨论分布式系统中线程的功能之前，我们首先讨论在非分布式系统中线程的功能。使用多线程系统有很多好处。

最重要的一个优势是，作为但线程进程，当一个阻塞方法被调用时，进程将会被阻塞。为了说明这一点，考虑一个应用程序，例如电子表格程序，并假设用户持续地、交互地想要更改值（举个访问数据库的栗子更好，比如需要向数据库刷10分钟的数据，进程将阻塞10分钟）。电子表格程序的一个重要特性是，它维护不同单元格之间的功能依赖关系，这些单元格通常来自不同的电子表格。因此当一个单元更改时，其他依赖的单元会自动更新。当用户更改一个单元时，这种更改会出发一系列计算。如果只有一个进程进行控制，在程序等待输入之前，计算是不能够完成的。像这样，当依赖的计算完成之前，提供数据是不容易的。一个简单的处理方式是，获取两个线程：一个负责处理用户输入，另一个负责更新。同时，第三个线程将数据备份。

另一个多线程的优点是当在一个多处理器或者多核系统上运行程序是，可以进行并行计算。当数据都被存储在共享的内存时，每一个线程都被分配给了不同的CPU和处理器。如果涉及得当，这种并行计算可以是透明的：这个进程同样可以在单核系统中运行，尽管这样运行会比较慢。在多核和多处理器价格变得相对便宜的情况下，多线程的并行处理变得很重要了。这种计算机系统通常发生在客户-服务端系统的服务端，但目前也广泛应用于智能手机等设备。

多线程同样在大型应用的上下文中很有作用。这些应用通常由一些相互合作的程序，即独立执行的进程组成。这个方式是典型的Unix 环境的方式。程序之间的合作通过**interprocess communication（IPC）**来实现。对于Unix系统，这些机制包括，管道，消息队列和共享内存等。这种IPC机制的主要缺点是上下文切换很消耗资源，图3.1介绍了这些机制。



除了使用进程，一个应用还可以通过不同的独立线程来构成。他们之间的通信完全通过共享数据来完成。线程切换有时可以只在用户态完成，在一些其他的实现中，内核也会参与线程并进行调度。这样会极大的提高性能。最终，这里有一个使用线程的纯粹的软件工程的原因：许多由线程构成的应用会变得更简单。考虑需要执行多个(或多或少独立的)任务的应用程序，如前面讨论的电子表格示例。

**线程实现**

线程通常以线程包的形式提供。这种包包括线程的创建和摧毁操作，同样包括线程同步相关的锁和条件变量等。这里主要有两种方法来实现线程包。第一种方法是构成完全在用户控件执行的一个线程库。第二种方法是，使用内核感知这些线程并调度线程。

用户级别的线程库有一系列好处。第一，创建和摧毁线程都不是很好资源。因为所有的线程权限都在用户地址区域，创建线程唯一值得考虑的就是在栈空间分配内存。同样的，摧毁线程主要包括在栈空间释放内存，但现在已经不再使用这种方式了。这两种方式都很廉价。

第二个使用用户级别线程的优点是上下文切换可以在几个指令下完成。基本上，只有CPU寄存器的值需要保存并且接下来当线程切换时重新加载之前存储的值。这里不需要改变内存映射，冲刷TLB，或做CPU计数等。然而，如Note3.1讨论的，上下文切换的大部分开销是由扰乱内存缓存造成的。

用户级线程的一个主要缺点是使用了**多对一的线程模型**：即多个线程映射为一个调度实体。作为一个结果，调用阻塞系统调用将立即阻塞线程所属的整个进程，从而阻塞该进程中的所有其他线程。正如我们解释的，线程通常对于构建由许多在逻辑上同时执行的小组成部分构成的大型应用是很有帮助的。在这种情况下，在同一时刻，IO阻塞不应该阻止其他部分的执行。对于这种应用，用户级别的线程是不会起到作用的。（注意这里是用户级别的，我们平常用的是内核级别的）

这个问题可以通过内核级别的实现来解决，也即使用**一对一线程模型**，即一个线程为一个调度实体。当然，每个线程的操作都是由操作系统内核负责，每一次都需要系统调用。这里线程上下文切换看起来会比进程的上下文切换更耗资源。然而，考虑到上下文切换的性能通常是由内存缓存的无效使用决定的，而不是由多对一或一对一线程模型之间的区别决定的，许多操作系统现在提供了后者模型，即使只是为了简单。

最后一点需要注意的是，必须认识到使用线程是组织应用程序中同时执行和并发执行的一种方式。在实践中，我们经常看到应用程序被构造为一组并发进程，共同使用操作系统提供的进程的设施。一个很好的例子就是ApacheWeb服务器，它是通过一系列进程来处理请求的。每个进程都组成一个单一的线程实例，但是都能通过标注你的方法与其他实例通信。

正如Srinivasan讨论的，使用进程代替线程对于将数据空间分离有很大的好处：每个线程都在独立的自己的数据区工作，并在操作系统层面与其他的进程隔离。这种分离的好处不应低估：线程编程被认为是出了名的困难，因为开发人员完全负责管理对共享数据的并发访问。使用进程，数据空间最终受到硬件支持的保护。如果一个进程访问了它自己之外的内存，硬件将会抛出一个异常，这个异常将在之后被操作系统处理。在线程则没有这种待遇。

**分布式系统的线程**

线程的一个重要性质是，它能够提供方便的手段使得在系统中的阻塞调用，不影响该线程所属进程的运行。这个性质使得线程在分布式系统中的使用变得很有吸引力，因为它使得同时维持多个逻辑通信的连接变得容易。为了说明这个观点，我们将分别观察多线程的客户-服务端系统。

**多线程客户端**

为了建立高度透明的分布式系统，在广域网中运行的分布式系统可能需要隐藏较长的进程间消息传播时间。广域网中的往返延迟很容易达到几百毫秒，有时甚至是几秒钟。

隐藏通信延迟的通常方法是启动通信并立即处理其他事情。这在Web浏览器中很常见。在很多情况下，一个Web文档包含很多个HTML文件如图片，标签等。为了获取这些元素，浏览器需要建立TCP/IP连接，读取传入的数据，并将它发送到需要展示的区域。建立连接或者读入数据通常都是阻塞操作。在处理长途通信时，我们也有一个缺点，每个操作完成的时间可能相对较长。

一个Web浏览器通常都是抓取HTML页面然后展示它。为了尽量隐藏通信延迟，一些浏览器开始会一边接受数据，一边进行展示。当文本对用户可用，或者使用滚动工具等，浏览器将继续获取组成页面的其他文件，图像等。这些文件和图像将被展示。用户不用等到页面所有组件都被获取才能看到页面。

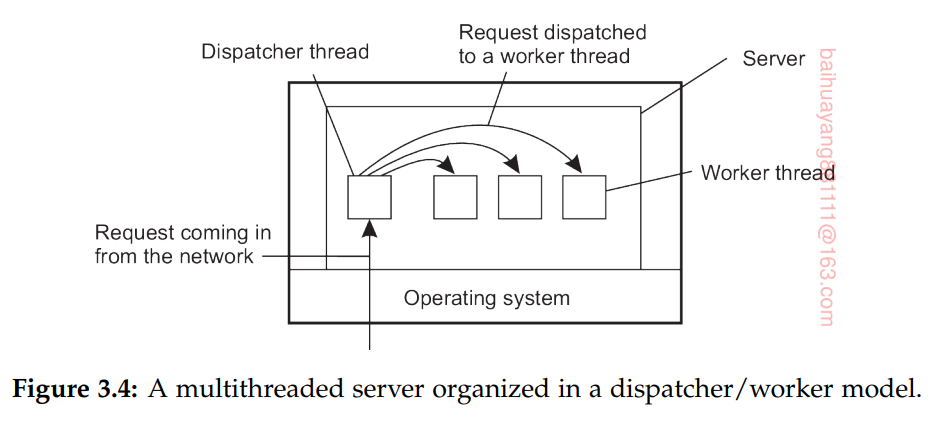
实际上，Web浏览器在同时做一系列任务。事实证明，将浏览器开发为多线程客户机大大简化了工作。当主要的HTML文件被获取时，其他的线程会被激活并获取其他部分。每个线程都单独和服务器建立连接并抓取数据。建立连接和读取数据可以使用标准系统调用，这里假定一个阻塞调用不会暂停所有的进程。同样Stevens提到，每个线程的代码都很简单。与此同时，用户只注意到图像显示的延迟等等，但是可以浏览文档。Web浏览器使用多线程的另一个好处是可以同时打开多个连接。在之前的例子中，和服务端建立了多个连接。如果这个服务器超负荷了，那么这种获取方式将不会提高性能。

然而，在许多例子中，Web服务器会复制到多个机器，每个服务器提供相同的Web文档。这些复制的服务器位于同一个站点，并且都叫同一个名字。当一个Web请求到来时，这个请求会来到其中的某一个服务器，这通常会使用轮询或者其他负载均衡的技术。当使用多线程客户端时，连接会在多个副本上建立，允许数据并行传输，这与不适用复制服务器相比，整个Web文档的传输要快很多。只有当客户机能够处理真正并行的传入数据流时，这种方法才有可能。线程是实现此目的的理想方法。

**多线程服务器**

虽然多线程在客户端具有很好的优势，但更主要的应用是放在了服务端。实践表明，多线程不仅大大简化了服务器代码，而且使开发利用并行性获得高性能的服务器变得更加容易，甚至在单处理器系统上也是如此。然而，随着现代多核处理器的发展，多线程并行成为了一种主要的实现方式。

为了了解服务端多线程的有点，考虑一个偶尔会因为硬盘等待儿阻塞的文件系统。这个文件服务器通常会等待一个输入请求，进而获取请求，然后返回一个应答。如图3.4所示。这里一个线程，或者说是**分配器**，读取进入的请求。紧接着请求被发送到服务器。检查这个请求后，服务器挑选一个空闲的**工作线程**来处理请求。



这个工作线程会执行一个本地读取操作，这是阻塞的，可能会使得获取数据的线程暂停。如果线程暂停，其他的线程将会继续执行。这样，分配线程可以执行更多的工作。作为替代，另一个工作线程会被选择来继续运行。

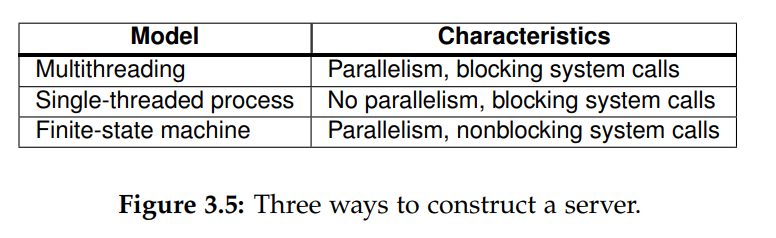
现在考虑服务没有使用线程的情况。一个可能就是讲服务器处理为单线程了。主要的循环即为获取请求，检测，处理返回，然后再进行下一个请求。当在硬盘上等待时，服务器是空闲的并且没有处理其他的请求。通常，其他客户端的请求不会被处理。此外，如果文件服务器运行在专用机器上(通常是这种情况)，那么在文件服务器等待磁盘时，CPU只是处于空闲状态。这样处理的结果是更少的请求会被执行。线程虽然获得了可观的性能，但是每个线程都是顺序执行的。

到目前为止，我们看到两种设计：多线程文件服务和单线程文件服务。第三种选择是将服务器作为大型单线程有限状态机运行。当请求到达时，唯一的线程将会检测它。如果可以从内存缓存中得到满足，那很好，但是如果不能，线程必须访问磁盘。然而，和磁盘阻塞不同的是，线程会调度一种异步磁盘操作，即之后会被操作系统中断。为了使它工作，线程会记录请求的状态，并持续观察之后是否会有其他的请求。（不是多个线程，而是单一线程接收请求，系统调用一个神奇的异步操作访问硬盘，之后发出中断来通知线程。）

当一个磁盘操作完成时，一个操作系统将会通知线程，并查找关联的请求状态并进一步执行它。最终，一个响应将会被一个非阻塞网络调用返回给客户端。

在这个设计中，我们在前两种情况下拥有的“顺序过程”模型丢失了。每次线程需要执行阻塞操作时，它都需要准确地记录它在处理请求时的位置，还可能存储其他状态。一旦完成这些工作，就可以开始操作并继续进行其他工作。其他工作意味着处理新到达的请求，或者处理先前启动的操作已经完成的请求。当然，如果没有其他工作，线程将会阻塞。实际上，我们正在艰难地模拟多线程及其各自堆栈的行为。该流程作为一个有限状态机进行操作，该有限状态机获取一个事件，然后根据其中的内容对其做出反应。

现在我们能明白线程提供了什么好处了。它可以同时实现如同进程一样的顺序执行或者执行并行。阻塞系统调用使编程变得更容易，因为它们看起来就像普通的过程调用一样。另外，多线程允许并发执行并提高性能。单线程服务器保留了阻塞系统调用的易用性和简洁性，但就每个时间单元可以处理的请求数量而言，它可能严重阻碍性能。有限状态机方法通过并行实现高性能，但使用非阻塞调用，这通常很难编程，因此很难维护。这些模型的总结如图3.5.



同样，我们也可以使用多进程来代替多线程。它的优势是操作系统可以提供进程维度的数据隔离。而，如果进程需要进行大量通信，那么与使用线程相比，我们可能会看到对性能的明显负面影响。

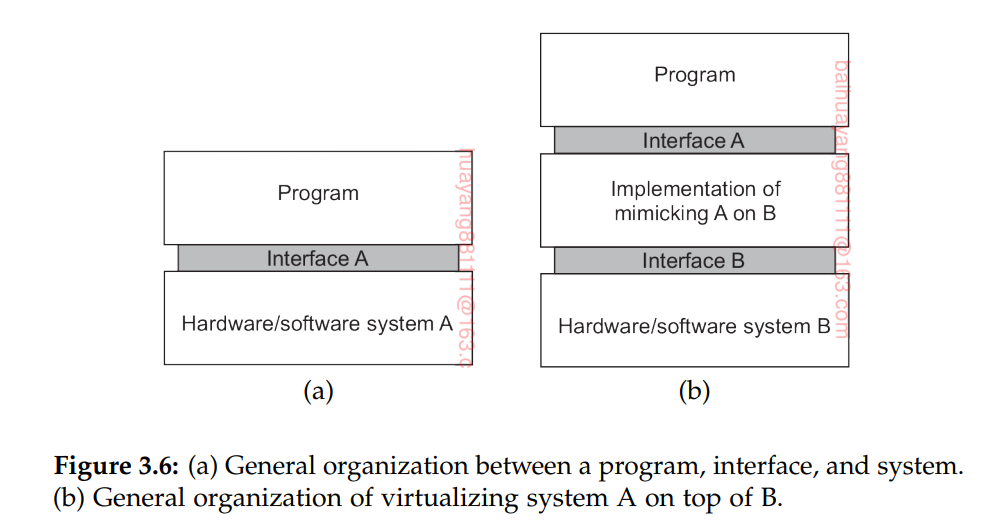
## 3.2虚拟化

线程和进程可以在同一时间做很多事情。实际上，他们允许我们构建同时执行的程序。在一个单进程的机器上，这种同时执行其实是一种错觉。因为只有一个单核cpu，所有只有一个单线程或者进程的设施会执行。通过快速的切换进程和线程，这种并发的错觉将创建。

这种使单个cpu分开运行，并加装可以同时运行多个资源的技术，称之为**资源虚拟化**。虚拟化已经被应用了许多年，但由于分布式计算机系统变得更加大众和复杂，使得人们对虚拟化又重新燃起了兴趣，这也使得应用软件可以在这种软件和硬件系统上存在更长的时间。

**虚拟化原则**

原则上，每个分布式系统都会提供一个给高级软件的编程接口，如图3.6(a)。这里有很多不同的接口类型，从CPU提供的基本指令集到许多当前中间件系统附带的大量应用程序编程接口集合。从本质上讲，虚拟化处理扩展或替换现有接口，以便模拟另一个系统的行为，如图3.6(b)。我们将简短的讨论虚拟化的技术细节，但让我们首先讨论虚拟化的重要性。



**虚拟化和分布式系统**

70年代有一个很重要的介绍虚拟化背景的原因，即是否允许遗留软件在昂贵的主机硬件上运行。这样的软件不仅仅包括各种各样应用，但实际上也包括他们依赖的操作系统。这种支持遗留软件的方法已经成功地应用于IBM 370大型机(及其继任者)上，这些大型机提供了一个虚拟机，将不同的操作系统移植到其中。

当硬件变得越来越便宜，计算机变得越来越强大，并且操作系统的种类越来越少，虚拟化的话题则越来越少。然而，在90年代，这个问题再次被提及。首先，虽然硬件和底层系统软件变化相当快，但是更高抽象级别的软件(例如，中间件和应用程序)通常要稳定得多。换句话说，我们面临的情况是，遗留软件不能与它所依赖的平台保持相同的步调。虚拟化可以将遗留接口移植到新平台，从而立即为现有程序的大型类开放新平台，从而在这方面提供帮助。

同样重要的是，现在网络已经是无处不在的了。很难想象一个现代的计算机没有连上网络。实际上，这种连接要求系统管理员维护大量异构的服务器计算机集合，每台服务器计算机运行非常不同的应用程序，客户机可以访问这些应用程序。同时有很多的资源可以同时被这些应用访问到。虚拟化可以帮助这一点：可以通过让每个应用程序在自己的虚拟机上运行(可能包括相关的库和操作系统，而这些库和操作系统又运行在一个公共平台上)来减少平台和机器的多样性。

最后一种虚拟化提供了高度的可移植性和灵活性。例如，为了实现能够轻松支持动态内容复制的内容交付网络，Awadallah和Rosenblum[2002]认为，如果边缘服务器支持虚拟化，允许动态复制整个站点(包括其环境)，管理就会变得容易得多。这些论点仍然有效，实际上，可移植性可能是虚拟化在许多分布式系统中扮演如此关键角色的最重要原因。

**虚拟化类型**

虚拟化可以通过很多方式来实现。Smith和Nair对这些不同的方法进行了概述。为了理解虚拟化的不同，重要的是理解计算机系统大体上为我们三个层级的四种不同类型的接口：

1. 硬件和软件之间的接口，称为指令集体系结构(ISA)，形成一组机器指令。这个集合被分成两个子集：

•特权指令，仅允许由操作系统执行。

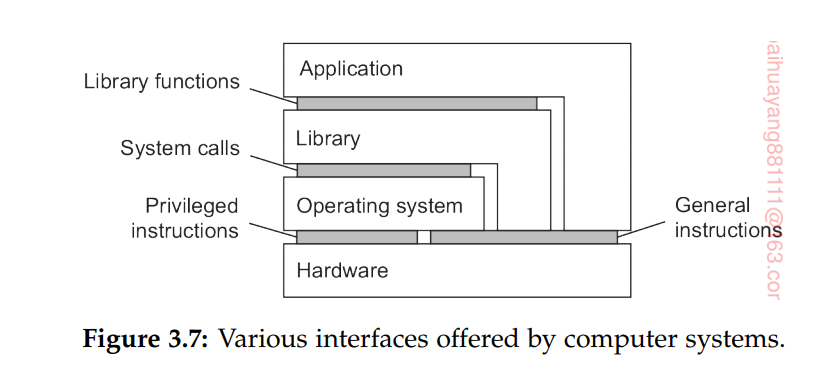
•通用指令，可由任何程序执行

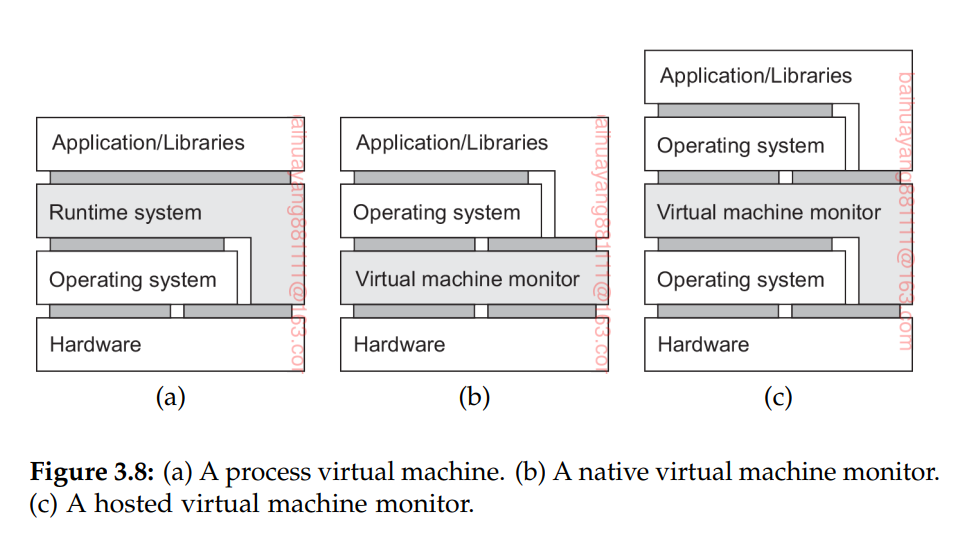
2. 由操作系统提供的系统调用组成的接口

3. 由库调组成的接口，并由此组成的**应用程序接口(API).**在许多情况下，前面提到的系统调用是由API隐藏的。

这些不同的类型如图3.7所示。虚拟化的本质是模拟这些接口的行为。

虚拟化可以发生在两种不同的情况。第一，我们可以构建一个运行时系统，这个系统本质上提供了一组抽象的指令集为执行应用调用。指令可以被解释(Java运行时环境就是这样)，但是也可以像在Unix平台上运行Windows应用程序那样进行模拟。请注意，在后一种情况下，仿真器还必须模拟系统调用的行为，这一点已被证明是非常重要的。这种虚拟化类型如图3.8(a)所示，Smith和Nair称之为**process virtual machine**，强调了虚拟化只是针对一个进程。



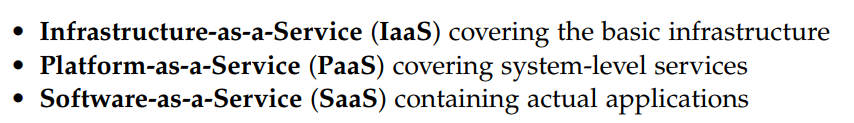


一个替代的方法如图3.8b所示，在原始硬件上提供一层，并提供相同的指令集。这就是**native virtual machine monitor**。它被叫做本地的是因为它直接在硬件上层实现。注意，虚拟机监视器提供的接口可以同时提供给不同的程序。因此，现在可以在同一个平台上独立并发地运行多个不同的客户机操作系统。本地虚拟机监视器会提供不同资源的正常访问，比如外部存储和网络。像任何一个操作系统一样，它需要实现这些资源的设备驱动。除了上面的方法，还有一种 建立在受信任的**主机操作系统**上的**主机虚拟化机器监视器**，如图3.8(c)所示。在这种情况下，虚拟机监视器可以使用该主机操作系统提供的现有设施。它通常必须被赋予特权，而不是作为用户级应用程序运行。使用托管虚拟机监视器在现代分布式系统(如数据中心和云)中非常流行。

正如Rosenblum和Garfinkel讨论的，虚拟机器对于分布式系统的稳定性和安全性都十分重要。因为他允许应用和环境保持独立，一个错误的发生或者一个安全攻击都不会对一整个计算机产生影响。另外，正如我们之前提到的，由于虚拟机提供了一个硬件和软件的解耦，可移植性被大大提高了。我们将在3.5节讨论。

**分布式系统虚拟机的应用**

从分布式系统的特性来看，虚拟化应用的最重要的部分就是云计算。正如我们再1.3节提到的，云计算提供了三种不同类型的服务



虚拟化在IaaS中很重要。不需要出租一个物理机，云计算只需要出租物理机的一部分-即虚拟机给用户。这种虚拟化的优雅就在于使得各个用户之间保持独立性，而用户却好像有独占一个物理机的错觉。独立性不是完整性。如果实际的物力资源被共享，就会导致性能的下降。

为了使问题更具体化，我们考虑亚马逊的**Elastic Compute Cloud**或者**EC2**。EC2允许创建在环境上连续的几个联网的虚拟机，这样组成一个分布式系统。为了简化工作，有大量预配置的机器映像可用，称为Amazon机器映像，或者简称为**AMI**。AMI是一个可安装的软件包，由一个操作系统内核和一些服务组成。一个简单的例子就是**LAMP**，包括Linux 内核，Apache Web服务，Mysql数据库系统，还有PHP。更多的镜像包括其他的软件也都是可用的，比如Unix和Windows软件。从这个意义上说，AMI本质上与引导磁盘是相同的(尽管我们稍后将返回一些重要的差异)。

一个EC2客户炫耀选择一个配置好的AMI。一个AMI可以启动，从而产生所谓的**EC2实例**：即可用于承载客户应用程序的实际虚拟机。一个重要的话题是客户很难知道一个实例确切是在哪里被运行的。明显的，他是在一个物理机上，但是是哪里的物理机却不得而知。客户能够知道的最确切的机器定位是由亚马逊提供的在哪个区域（美国，南美，欧洲，亚洲等）。

为了方便通信，每个实例都有两个IP地址：一个私有IP支持了内网访问，一个公网IP支持互联网客户的访问。这个共有IP和私有IP的映射使用的是**Network Address Translation（NAT）技术**。一个管理实例的简单方法是使用SSH连接，Amazon负责提供合适的私钥的生成。

提供实例运行环境的EC2提供了几种不同的服务：

**CPU**：允许选择不同数量个种类的CPU，包括GPU

**内存**：决定实例分配多少主存

**存储**：决定分配多少存储空间

**平台**：32位还是64位

**网络**：分配带宽

另外，其他的资源比如网络接口也可以申请。本地存储不是持久的：当实例终止时，所有的数据就会丢失。为了阻止数据丢失，用户需要将数据持久化，比如，使用亚马逊的Simple Storage Service（S3）。另一种方法是附加一个映射到Amazon **Elastic Block Store (Amazon EBS)** 的存储设备。同样，这是另一种服务，但它可以以虚拟块设备的形式使用，而虚拟块设备只是像挂载额外的硬盘一样挂载。当实例停止时，一个EBS设备也可以被挂载到任何其他的实例上。

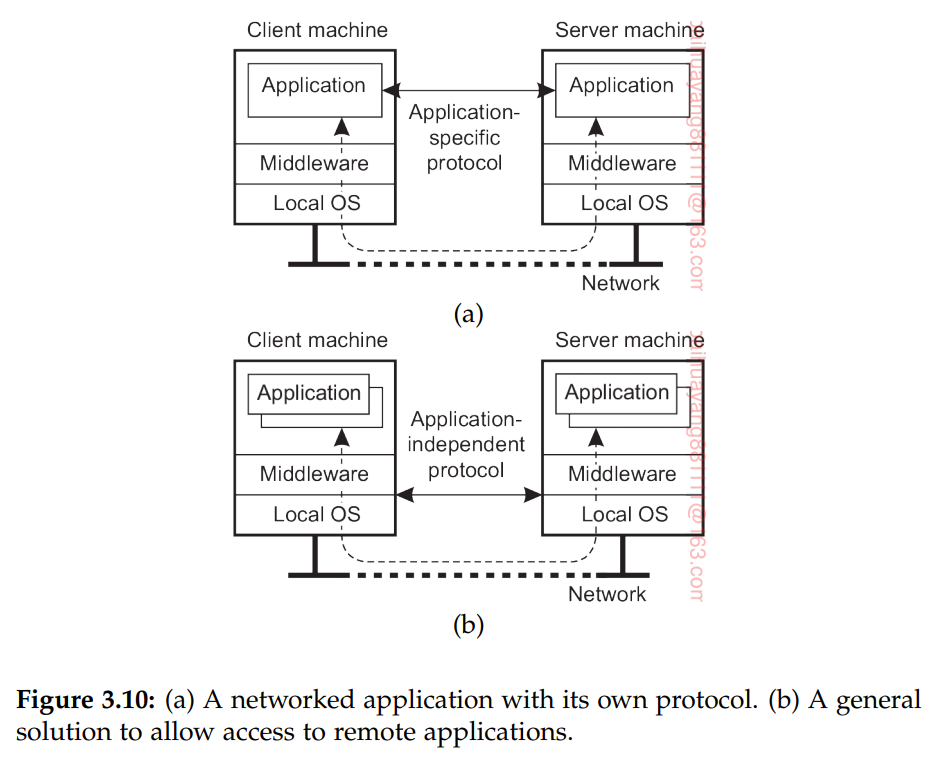
现在应该很清楚了，在没有深入任何重要细节的情况下，EC2提供的IaaS允许客户创建(可能很大)数量的虚拟机，每个虚拟机都根据需要配置了资源，并且能够通过IP网络交换消息。另外，这些虚拟机可以被网络上的可信用户访问。因此，Amazon EC2和许多其他IaaS提供者一样，提供了配置完整分布式系统的方法，该系统由网络虚拟服务器和运行客户提供的分布式应用程序组成。与此同时，这些客户将不需要维护任何物理机器，而这通常已经是一个巨大的收益，我们将在本文中多次遇到这种情况。实际上可以说虚拟化在云计算中起到了核心的作用。

## 3.3客户端

在前几章中，我们讨论了客户机-服务器模型、客户机和服务器的角色以及它们之间的交互方式。现在我们来近距离分别解剖这种客户-服务端的模型。我们这章讨论客户端，下一章讨论服务端。

**网络用户接口**

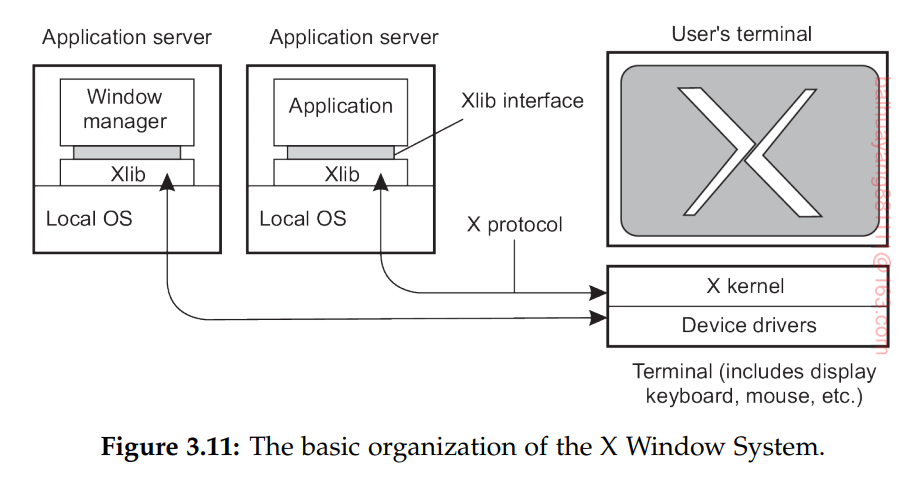
对于客户端一个重要的任务是提供访问远程服务器的手段。这里有两种方式可以支持。第一，首先，对于每个远程服务，客户机机器都有一个单独的对等方，可以通过网络与服务联系。一个典型的栗子是，用户智能手机的日历系统需要同服务端的同步。在这种情况下，应用级别的协议将会处理这种同步，如图3.10(a)所示。



第二个解决方案是通过只提供方便的用户界面来提供对远程服务的直接访问。实际上，这意味着客户机仅作为终端使用，不需要本地存储，从而得到与应用程序无关的解决方案，如图3.10(b)所示。在这种用户网络接口中，所有事物都是存储在服务端的。随着网络连接的增加和移动设备的使用，这种 thin-client approach 已经受到了更多的关注。Thin-client 解决方法同样因为使管理系统任务变得容易而有名。

**例子：the X window system**

比如一个很老但使用很广泛的网络用户接口的例子是**X window system**。X窗口系统，通常简称为X，用于控制位图终端，包括显示器、键盘和鼠标等指向设备。除了支持台式电脑和工作站等传统终端，X还支持平板电脑和智能手机上的触摸屏等现代设备。在某种意义上，X可以看作是控制终端的操作系统的一部分。系统的核心是我们提到的**X kernel**。它包含所有特定于终端的设备驱动程序，因此通常高度依赖于硬件。X内核提供了一个相对底层的接口来控制屏幕，还可以从键盘和鼠标捕获事件。这个接口作为一个名为Xlib的库提供给应用程序。这个通用的组织在图3.11中显示。注意，应用程序很少直接使用Xlib，相反，应用程序更容易部署使用在Xlib上实现的工具包。



X的有趣之处在于，X内核和X应用程序不一定要驻留在同一台机器上。特别是，X提供了X协议，这是一个应用程序级的通信协议，通过它，Xlib的实例可以与X内核交换数据和事件。例如，Xlib可以向X内核发送创建或关闭窗口、设置颜色和定义要显示的光标类型的请求，以及其他许多请求。反过来，X内核将通过向Xlib发送事件包来响应本地事件，如键盘和鼠标输入。

若干个应用可以同时和X kernel 通信。这里有一个特殊的应用有着特殊的权限，我们称之为**window manager**。这个应用程序可以在显示给用户时指定显示的“外观和感觉”。例如，窗口管理器可以规定如何用额外的按钮装饰每个窗口，如何将窗口放置在显示器上，等等。其他应用程序必须遵守这些规则。实际上，这意味着应用程序和X终端之间的大部分交互都是通过窗口管理器重定向的。

值得注意的是，X window系统实际上是如何适应client-server计算的。根据我们到目前为止所描述的，应该很清楚，X内核接收到操作显示的请求。它从(可能是远程)应用程序获取这些请求。从这个意义上说，X内核充当服务器，而应用程序扮演客户机的角色。这个术语已经被X采用，虽然严格来说是正确的，但它很容易导致混淆。

**Thin-client network computing**

显然，应用程序使用X提供的特定显示命令来操作显示。这些命令通常通过网络发送，然后由X内核执行。从本质上讲，为X编写的应用程序最好将应用程序逻辑与用户界面命令分开。不幸的是，这并不容易。如Lai和Nieh研究指出很多应用逻辑和用户交互都是紧耦合的，意味着应用将发送很多请求到X内核，并在下一步执行前等待响应。当在具有长延时的广域网上运行时，这种同步行为可能会对性能产生负面影响。

这里将会有若干个解决办法。一个是重新实现X协议，比如NX。这个工作的重点是通过降低消息数量来降低带宽。为此，消息被认为由一个固定的部分(作为标识符)和一个可变的部分组成。在许多情况下，多个消息将具有相同的标识符，在这种情况下，它们通常包含类似的数据。这样对于相同的标识符，紧发送不同的数据即可。通过让发送方和接收方维护标识符，可以很容易地应用接收方的解码。据报道，带宽减少高达1000倍，这使得X也可以通过只有9600kbps的低带宽链接运行。

作为使用X的另一种选择，研究人员和实践者还试图让应用程序完全控制远程显示，即像素级的显示。然后，位图中的更改通过网络发送到显示器，并立即传输到本地帧缓冲区。这种方法的一个著名例子是**虚拟网络计算(VNC)** [Richardson et al.， 1998]，它从20世纪90年代末就已经存在了。显然，让应用程序控制显示需要复杂的编码技术，以防止带宽可用性成为一个问题。例如，考虑在一个简单的320×240屏幕上以每秒30帧的速度显示视频流。如果每个像素由24位编码，那么如果没有有效的编码方案，我们将需要大约53 Mbps的带宽。在实践中，使用了各种编码技术，但是选择最佳编码技术通常取决于应用程序。

与高级协议(如X)相比，发送原始像素数据的缺点是不可能使用任何应用程序语义，因为这些语义在该级别是失效的。Baratt发明了另外一种技术。在他们的解决方案(称为THINC)中，它们提供了一些高级显示命令，这些命令在视频设备驱动程序级别上操作。因此，这些命令依赖于设备，比原始像素操作功能更强大，但与X之类的协议相比功能较弱。是，显示服务器可以简单得多，这有利于CPU的使用，同时可以使用依赖于应用程序的优化来减少带宽和同步。

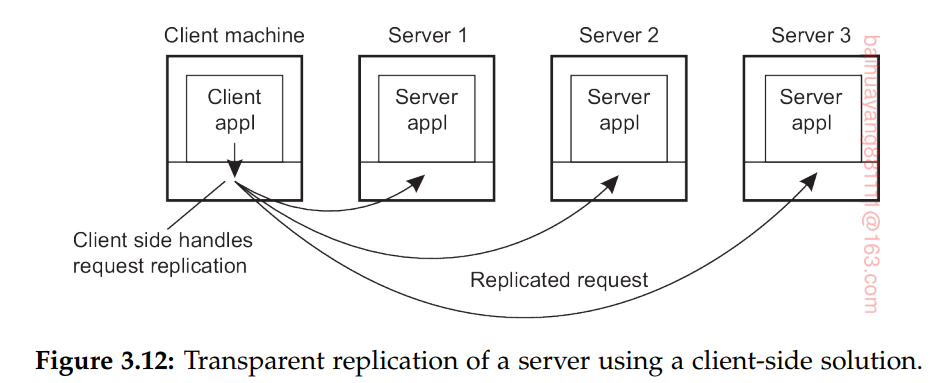
**分布式客户端的透明性**

客户端软件不仅仅包括用户界面。在许多情况下，客户机-服务器应用程序中的处理和数据级别的部分也是在客户机端执行的。嵌入式客户端软件，如自动柜员机(atm)、收银机、条形码阅读器、电视机顶盒等，构成了一个特殊的类别。在这些情况下，与本地处理和通信设施相比，用户界面是客户机软件中相对较小的一部分。

除了用户界面和其他与应用程序相关的软件外，客户端软件还包括实现分发透明性的组件。理想情况下，客户机不应该知道它正在与远程进程通信。相反，由于性能和正确性的原因，分发对于服务器来说通常不那么透明。

访问的透明性通常通过服务端提供的接口定义来生成的**client stub** 来处理。这个 stub 提供与服务端相同的接口，但是隐藏了机器架构的不同，同样隐藏了通信。这个 client stub 将本地调用信息传送给服务端，然后将来自服务器的消息作为返回值传回。处理位置、迁移和重新定位透明性有不同的方法。使用方便的命名系统是至关重要的。很多情况下，和客户端软件的合作也是十分重要的。例如，当客户机已经绑定到服务器时，可以在服务器更改位置时直接通知客户机。这种情况下，客户端的中间件可以对用户隐藏服务端现在的网络地址，然后透明的重新绑定服务端。最糟糕的情况是，客户端的应用会发现一段短暂的性能丢失。

类似的，许多分布式系统通过复制实现这种客户端的解决方式。例如，假设一个具有复制服务器的分布式系统，可以通过将请求转发到每个副本来实现这种复制，如图3.12所示。客户端软件可以透明地收集所有响应并将单个响应传递给客户端应用程序。



关于故障透明性，屏蔽与服务器的通信故障通常是通过客户机中间件完成的。例如，可以将客户机中间件配置为重复尝试连接到服务器，或者在多次尝试之后尝试另一台服务器。甚至在某些情况下，客户端中间件返回它在前一个会话期间缓存的数据，有时是由无法连接到服务器的Web浏览器完成的。

最后，并发透明性可以通过特殊的中间服务器来处理，尤其是事务监视器，并且不需要客户机软件的支持。

## 3.4服务端

接下来讨论服务端的构建。在下面几页，我们将集中讨论一些服务端通用的设计，然后讨论服务器集群。

**通用设计课题**

服务端是相对于客户端的另一边的实现了一些特殊服务的进程。本质上每个服务器都是由相同的方式构建：他们等待客户端发出的请求并确保请求被处理，然后等待另一个请求的进入。

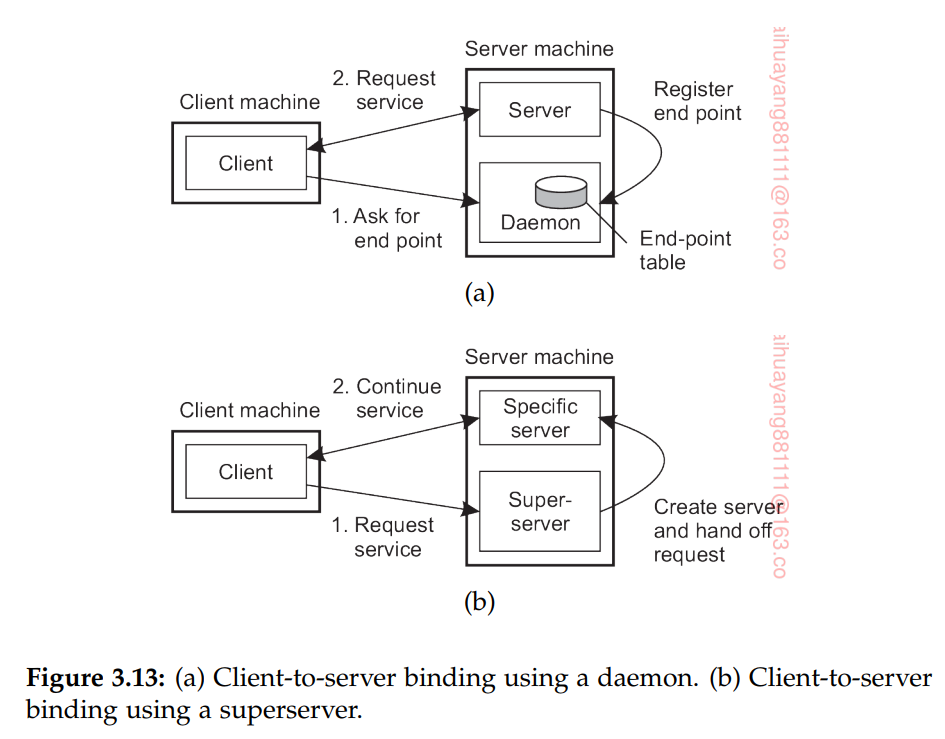
**并发服务器和迭代服务器**

这里有若干种方式组织服务器。在**迭代服务器中**，服务器处理请求然后返回响应给客户端。而**并发服务器**自己并不处理请求，它会将请求交给一个单独的进程或者线程处理，然后自己继续等待请求的到来。一个多线程服务器就是并发服务器最好的例子。另一种实现方式是新创建一个进程来处理。这种方式被许多Unix系统使用。这些进程和线程处理请求之后会将结果返回给客户端。

**Contacting a server: end points**

另一个课题室客户端从哪里连接服务端。在所有情况下，客户端发送请求到一个**end point**，或者叫做 **port（端口）**，也即服务端运行的机器。每个服务端监听一个特殊的end point。客户端怎么知道end point 对应的服务呢？一个方法是为知名的服务分配全局的端口。例如，FTP对应TCP端口21.Http对应80.这些end point被Internet Assigned Numbers Authority(IANA)设计。由于设计了这些end point，客户端只需要找到服务器机器的网络地址。命名服务帮助你找到这些地址。

有许多服务不需要提前设置end point。例如，一天中的某个时间服务器可能使用由其本地操作系统动态分配给它的端点。在这种情况下，客户端会首先查找end point。一个解决方法是在每个机器上运行一个daemon。这个daemon会对当前相关服务器提供的服务的end point 进行跟踪。这个daemon自己监听一个知名的end point。客户端会首先连接这个daemon，请求end point，然后连接服务器如图 3.13(a)所示。



将特定的服务和end poind 关联起来是很大众的。然而，通过分离的服务器去实现服务是对资源的一种浪费。例如，在典型的Unix系统中，通常同时会有很多个服务器在运行，然而大多数都是被动的等待客户端的请求。与其保持着这些用来被动监听的进程，更有效的方法是选择一个**超级服务器**用来监听相关服务的end point，如图3.13(b)。例如，Unix 系统中的inetd daemon 监听了一系列知名的网络服务端口。当一个请求到来时，daemon会fork一个进程来处理它。当结束时这个进程会退出。

**Interrupting a server**

设计服务器的另一个课题是服务器是否以及如何被终端。例如，考虑一个用户想要上传一个巨大的文件到FTP服务器。然而，突然发现这个文件搞错了(似不似撒)，它像like终端服务器来取消数据的传输。这里有许多种方式来实现。有一种方法在当前Internet中非常有效(有时是惟一的替代方法)，即用户突然退出客户机应用程序(客户机应用程序将自动断开与服务器的连接)，立即重启它，然后假装什么都没有发生。服务器最终会断开旧的连接，认为客户机可能已经崩溃。

一个更好地方法是，开发客户端和服务器使之支持发送**out-of-band**数据，这些数据能够被服务端优先处理。解决方法是，是服务器监听一个可控制的end point ，使得客户端发送 out-of-band 数据，同时监听正常的数据访问。另一个方法是通过相同的连接发送out-of-band数据。例如，在TCP中，可以传送紧急数据。当服务器获取到紧急数据时，服务器将会被终端，然后可以检查数据并对其做相应的处理。

**有状态和无状态服务器**

最后一个关键的设计课题是，是否服务器应该保存状态。一个**无状态服务器**不会保存客户端的状态信息，并且可以改变自己的状态且无需通知客户端。例如一个Web服务器，是无状态的。它仅仅是响应一个HTTP请求，无论是上传还是从服务器获取数据。当请求运行完毕时，Web服务器就完全忘记了客户端了。这样，服务端的文件集合的改变无需告知客户端。

请注意在许多无状态设计上，服务端实际上保持了客户端的信息，可关键是，如果丢失了这些信息，不会到时服务器提供的服务中断。例如，一个Web服务器通常会存取客户端请求的日志。例如，这些信息对于决定是否应该复制某些文档以及它们应该复制到哪里非常有用。显然，如果日志丢失，除了优化性能的可能丢失外，没有其他影响。

无状态设计的一种特殊形式是服务器维护所谓的软状态。在这种情况下，服务端保证保持一部分客户端的信息，但只限于一段有限的时间。当时间过期时，服务器回到默认状态，并丢弃关联的客户端的信息。一个这种类型的例子是，服务端承诺在一段时间内通知客户端服务端的更新，当超过时间限制时，客户端将不会获取这些更新。软状态方法起源于计算机网络中的协议设计，但同样适用于服务器设计。

相反，一个**状态服务器**通常持久化这些客户端的信息。这意味着这些信息需要在服务端删除。一个典型的例子是文件服务器，它允许客户机保存文件的本地副本，甚至用于执行更新操作。这样的服务端需要维护一个包含(client,file)实体的表。这样一个表允许服务端跟踪哪些客户端拥有哪些文件的更新权限，或者跟踪这些文件的最新版本。这种方法可以提高客户端感知的读写操作的性能。这种性能的提升对于无状态的服务器来说是一种很大的收益。然而，这里例子也说明了状态服务器的一个主要的缺点。如果服务器挂掉，它需要回复(client,file)实体的表，否则它将不能保证已经处理了最新的文件更新（就是用户没法更新了呗）。通常，有状态服务器需要恢复崩溃前的整个状态。正如我们在第8章中讨论的，启用恢复可能会带来相当大的复杂性。在无状态设计中，根本不需要采取任何特殊措施来恢复崩溃的服务器。它只是简单的将机器重新运行，然后等待客户端请求的到来。

Ling在[2004]提到我们应当分辨**session state** 和 **permanent state**。上面的例子是典型的会话状态:它与单个用户的一系列操作相关联，应该维护一段时间，但不是无限期的。事实证明，会话状态通常在三层客户机-服务器体系结构中维护，其中应用程序服务器实际上需要通过一系列查询访问数据库服务器，然后才能响应请求的客户机。这里的问题是，如果会话状态丢失，并没有造成真正的损害，前提是客户机可以简单地重新发出原始请求。这种观察允许更简单和更不可靠的状态存储。永久状态的信息通常保存在数据库中，比如用户信息，购买软件的秘钥等。然而，对于大多数分布式系统，维护会话状态已经意味着在发生故障时需要特殊措施的有状态设计，并对存储在服务器上的状态的持久性做出明确的假设。在讨论容错时，我们将广泛地回到这些问题。

在设计服务器时，对无状态或有状态设计的选择不应影响服务器提供的服务。例如，如果文件必须在读取或写入之前打开，那么无状态服务器应该以某种方式模仿这种行为。一种常见的解决方案是，服务器响应读或写请求时，首先打开所引用的文件，然后执行实际的读或写操作，然后立即再次关闭文件。（没理解想表达什么~~）

在其他情况下，服务器可能希望记录客户机的行为，以便更有效地响应其请求。例如，Web服务器有时提供立即将客户端定向到他喜欢的页面的可能性。这种方法只可能服务器保存了客户端的历史信息才能够实现。当服务器没有保存状态时，一个解决方法是让客户端发送之前访问的相关信息。在Web这种情况下，信息通常是被透明的存储在客户端的浏览器中，我们称之为**cookie**，即一小段保存了特定客户端对服务器偏好的信息。Cookie不会在浏览器执行，只是简单的保存。

当客户端第一次访问服务器时，服务器会发送一个cookie给浏览器，之后浏览器就会安全地把cookie收起来。之后每当浏览器访问服务器时，它的cookie将会随着请求一起发送到服务器。

**Object servers （What is a object ???）**

让我们看看分布式object所需的object服务器的一般组织。一个重要的object服务器和其他服务器的区别是一个object服务器不会提供一个特定的服务。特定的服务由驻留在服务器中的object实现。本质上，服务器只提供基于来自远程客户机的请求来调用本地object的方法。所以，我们可以通过增加或减少object 来简单的改变服务。

这样一个object服务器就是object存在的地方。一个object由两部分组成：表示其状态的数据和执行其方法的代码。这些部分是否分离，或者方法实现是否由多个对象共享，都取决于object服务器。此外，object服务器调用其object 的方式也有所不同。例如，在多线程服务器中，每一个object都会被分配一个线程，或者每个线程负责一个单独的请求。接下来将讨论这些问题。

对于要调用的object，对象服务器需要知道要执行哪些代码，应该对哪些数据进行操作，是否应该启动单独的线程来处理调用，等等。一个简单的方法是假设所有对象看起来都一样，并且调用对象的方法只有一种。不幸的是，这种方法通常是不灵活的，并且常常不必要地限制分布式object的开发人员。

更好的方法是让服务器支持不同的策略。例如，考虑一个**transient object**：对象的存在时间仅与其服务器的存在时间一样长，但可能持续的时间更短。文件的内存只读副本通常可以实现为transient object。同样，计算器也可以实现为一个transient object。一个合理的策略是在第一次调用请求时创建一个transient object，并在没有客户端绑定这个object时销毁它。

这种方法的优点是，只有在真正需要transient object时，transient object才需要服务器的资源。缺点是创建并初始化这个transient object 需要耗费一定资源。这样，一个替代的方案是，在服务器初始化的时候，创建好所有的transient object，这是以消耗资源为代价的。

以类似的方式，服务器可以遵循这样的策略:将其每个object都放在自己的内存段中：即每个object不共享代码和数据。当object实现不分离代码和数据时，或者出于安全原因需要分离object时，可能需要使用这种策略。在后一种情况下，服务器将需要提供特殊措施，或者需要底层操作系统的支持，以确保没有违反段边界。

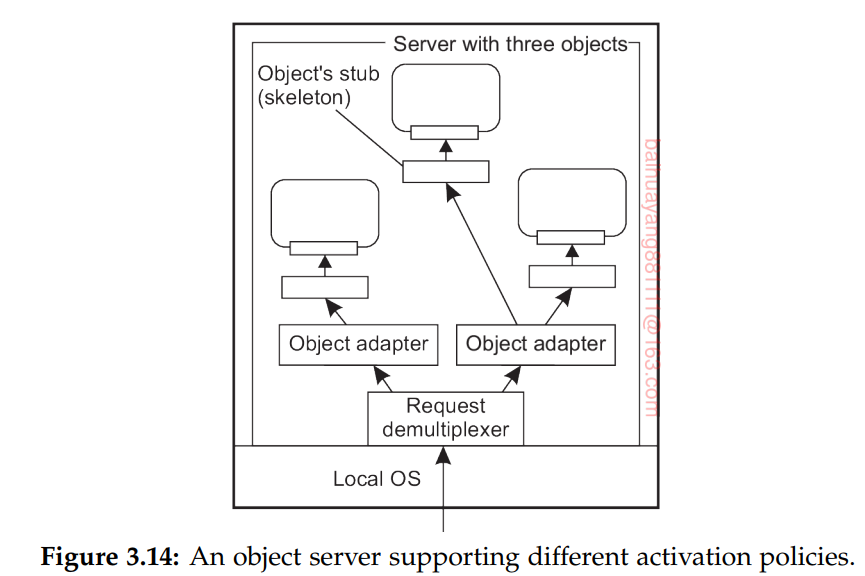
另一种替代的方法是，让object至少分享代码。例如，通过只将类实现加载到服务器一次，可以有效地实现包含属于同一个类的对象的数据库。当出现对象调用请求时，服务器只需要获取该对象的状态并执行所请求的方法。

同样，关于线程也有许多不同的策略。最简单的方法是通过一个单一的线程控制来实现服务器。或者，一个服务器有多个线程，每个 object拥有单一的一个线程。当一个请求进入一个object时，服务器将请求传送给object对应的线程来处理。如果线程都处于繁忙状态，这个请求会暂时排队。

这个方法的优点是object可以自助的处理并发的情形：所有调用都通过与对象关联的单个线程进行序列化。这么做简单而且整洁。当然，也可以为每个调用请求使用单独的线程，这要求对象应该已经受到保护，防止并发访问。独立于每个对象使用一个线程或每个方法使用一个线程，可以选择是按需创建线程，还是服务器维护一个线程池。通常这都没有一个最好的策略。选择哪一种通常由线程是否可以，会影响多少性能等简单的因素决定。

关于如何调用对象的决策通常称为**激活策略**，以强调在许多情况下，在实际调用之前，object本身必须首先进入服务器的地址空间(即(已激活)。那么，需要的是一种机制来对每个策略的object进行分组。这种机制有时称为**object适配器**，或者**object包装器。**object适配器可以被认为是实现特定激活策略的软件。然而，主要问题是object适配器作为通用组件来帮助分布式object的开发人员，并且只需要为特定的策略配置这些组件。

一个object适配器控制着多个objects。因为一个服务器需要同时支持多个object的不同策略，这些object适配器应该位于同一个服务器。当请求到达时，首先被分配到object适配器，如图3.14所示。



重要的是，对象适配器不知道它们控制的对象的特定接口。否则，它们永远不会是通用的。对对象适配器来说唯一重要的问题是，它可以从调用请求中提取对象引用，然后将请求分派给被引用的对象，但现在要遵循特定的激活策略。如图3.14所示，适配器不是直接将请求传递给对象，而是将调用请求传递给该对象的服务器端stub。stub(也称为骨架)通常从对象的接口定义生成，解析请求并调用适当的方法。

只要在运行时配置对象适配器，就可以支持不同的激活策略。例如，在符合corba的系统中[OMG, 2001]，指定一个对象在其关联的适配器停止后是否应该继续存在变得可能。同样，可以配置适配器来生成对象标识符，或者让应用程序提供一个。最后一个示例是，可以将适配器配置为在单线程或多线程模式下运行，如上所述。

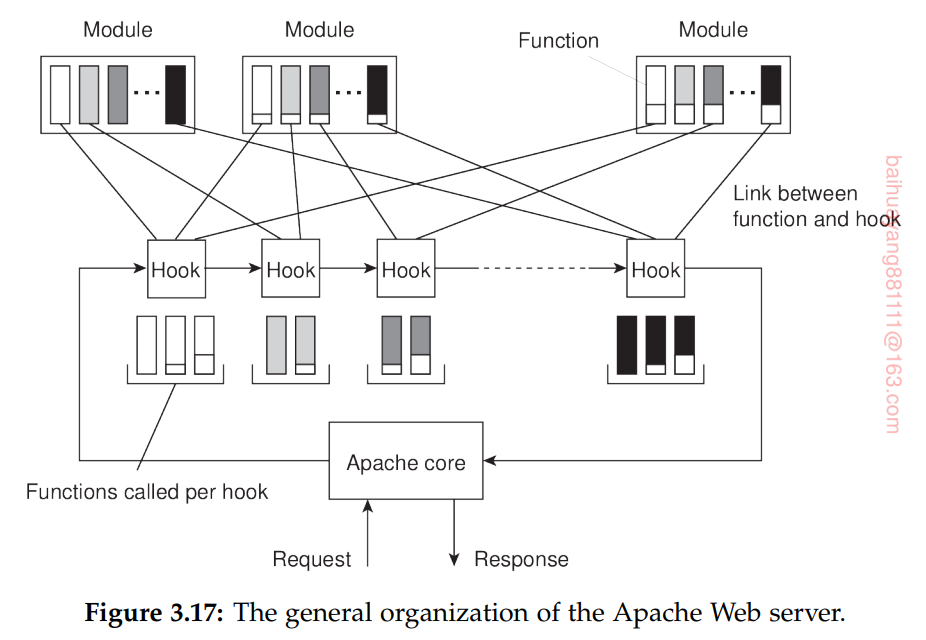
请注意，尽管在图3.14中我们已经讨论了对象，但是我们没有讨论这些对象实际上是什么（卧槽，你特么终于提到了）。特别需要说明的是，数据库的访问和库函数的调用都可以作为实现此类object 的一部分。只与骨架通信的对象适配器隐藏了实现细节。这样，在实现上会不同于我们在编程语言层面上的对象。出于这个原因，一般采用不同的术语。**Servant** 是构成对象实现的一段代码的通称。

**例子：Apache web服务**

一个有趣的平衡策略和机制分离的web服务的例子是Apache Web服务**。**它也是一个非常受欢迎的服务器，估计用于承载大约50%的所有Web站点。Apache是一种复杂的软件，并且随着Web提供的文档类型的增强，使得服务器的配置和扩展很重要，并且同时可以独立于特定的平台。

使得服务器平台独立的本质方式是，提供通过不同操作系统实现的自己的运行环境来实现的（就是windows一个安装包，linux一个安装包）。这个运行时环境称为Apache Portable runtime (APR)，它是一个库，为文件处理、网络、锁定、线程等提供了一个与平台依赖的接口。在扩展Apache时，因为是只调用APR，而不去调用特定于平台的库，可移植性在很大程度上得到了保证。从某种角度来看，Apache可以被看作是一个完全通用的服务器，专门针对传入的请求生成响应。当然，这里有很多种隐藏的依赖和假设，通过这些依赖和假设，Apache才很适合处理Web文档的请求。举个栗子，正如我们之前提到的，Web浏览器和服务器使用HTTP作为通信协议。由于HTTP是TCP的上层实现，Apache的内核假定所有的请求都是以TCP为基础的连接方式。如果不修改Apache内核，UDP为主的请求不能够被处理。

然而，Apache内核做了一些关于请求如何被处理的假定。如图3.17所示。这种组织的接触是一个叫做**hook**的概念，这个hook是一组特定方法的占位符。Apache内核假定请求处理分几个阶段，每个阶段由几个hook组成。每个hook都会展现一组相似的行为，这些行为都会作为处理请求的一部分来执行。



例如，一个hook会将URL转换为本地文件名字。在处理请求时，几乎肯定需要进行这样的转换。就像这样，一个hook负责将信息写入日志，一个hook负责检查客户身份，一个hook负责检查访问权限，一个hook检查请求与哪个MIME类型相关。正如图3.17所示，这些hooks会以一个预先确定好的顺序执行。在这里，我们明确地看到Apache强制执行与处理请求有关的特定控制流。

Hook关联的方法是由各个分隔的**modules**提供的。尽管原则上，开发人员可以更改Apache将处理的一组hook，但是编写由即将进行处理的标准hook提供的函数组成的module是更为常见的做法。基本原则是很简单的。每个hook都可以包含一组函数，每个函数都应该匹配一个特定的函数原型(即，参数列表和返回类型)。Module开发人员姜维特定的hook编写函数。当编译Apache是，开发者将执行哪些函数会被加到哪些hook中。图3.17展示了函数和hook的连接。

由于这里有十好几个modules，每个hook会包含若干个函数。通常，modules被认为是独立的，通常，模块被认为是相互独立的，因此同一个hook中的函数将以任意顺序执行

。然而，Apache同样可以通过让开发人员执行一个不同modules应该被执行的顺序。总的来说，其结果是一个功能极其丰富的Web服务器。有关配置Apache的详细信息，以及如何扩展Apache的介绍可以在[Laurie and Laurie, 2002]中找到。

**服务器集群**

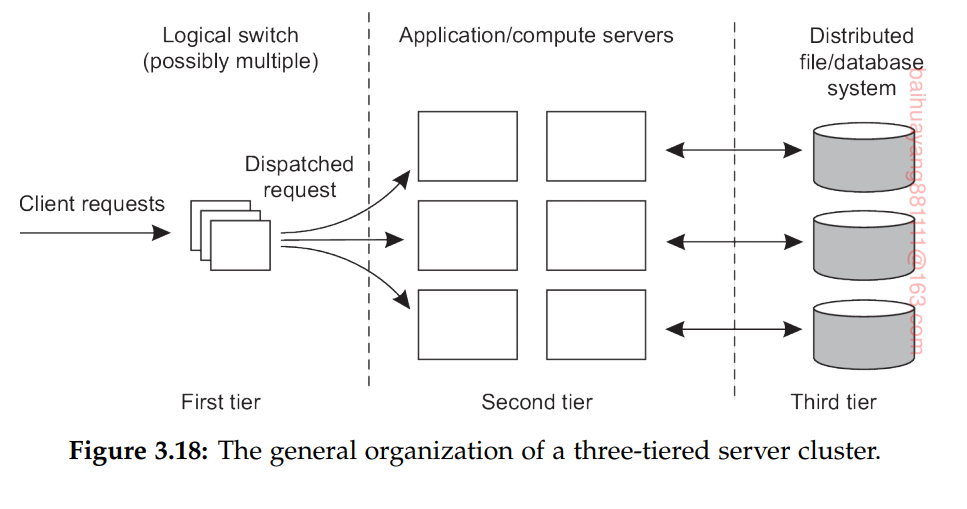
在第1章中，我们简要地讨论了作为分布式系统众多表现形式之一的集群计算。现在，我们将进一步研究服务器集群的组织，以及突出的设计问题。我们首先考虑在局域网中组织的服务器集群。广域网之后再讨论。

**本地集群**

简单来说，一个服务器集群是由一群机器通过网络连接组成的集合，其中每个机器都运行一个到多个服务。我们这里讨论的集群，是基于本地网络，有着高带宽和低延迟。

**通常的组织**

在许多情况下，服务器集群在逻辑上被组织成三层，如图3.18所示。第一层由一个(逻辑)交换机组成，客户机请求通过该交换机的路由。这个交换机方式可以多种多样。比如，传输层的交换机可以接受TCP的连接请求并发送请求到集群的某一台服务器。另一个完全不同的示例是Web服务器，它接受传入的HTTP请求，但将请求部分地传递给应用程序服务器进行进一步处理，以便稍后从这些服务器收集结果并返回HTTP响应。



任何多层客户机-服务器体系结构一样，许多服务器集群也包含专门用于应用程序处理的服务器。在集群计算中，这些服务器通常运行在高性能硬件上，用于交付计算能力。然而，在企业服务器集群的情况下，应用程序可能只需要在相对低端的机器上运行，因为所需的计算能力不是瓶颈，而是对存储的访问。

这里说到第三层，包括数据处理服务器，比如文件和数据库服务器。同样的，一句集群的使用，这些服务器需要运行在特定的机器上，这些机器需配备高速访问硬盘和高容量数据缓存。

当然，并不是所有的服务集群都严格遵循这样的分离。很多情况是每个机器都配备自己的本地存储，将应用和数据处理集成为一层成为一个两层架构。比如，当通过服务集群处理流媒体时，通常都部署一个两层系统架构，每一个机器都是一个流媒体服务器。

当一个服务集群提供多个服务时，会发生不同的服务器运行不同的应用的情况。作为一个结果，交换机需要辨别不同的服务否则将不能将特定的请求发送个特定的机器。所以，我们会发现一些服务有时是空闲的，而另一些服务却接受过载的请求。通常把服务迁移到空闲的机器是一个有效的办法。解决方案是使用虚拟机，允许相对容易地将代码迁移到真实的机器。

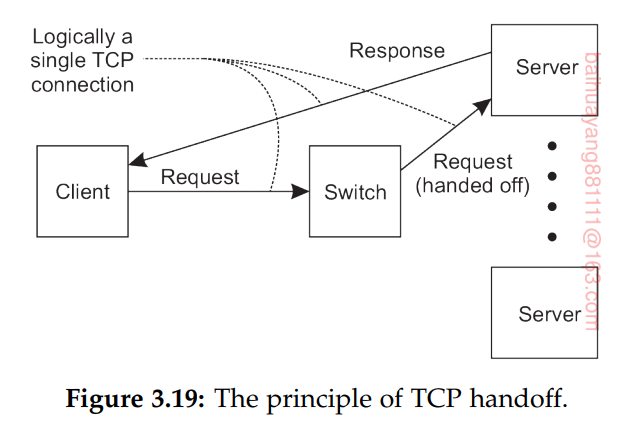
**请求分发**

我们首先来看第一层，包括交换机，也就是我们所谓的前端。分布式系统一个重要的设计目标是隐藏多个主机这个事实。换句话说，在远端运行的客户应用应当不需要知道任何关于集群内部结构的事情。这种访问透明性总是通过单个访问点提供的，然后通过某种硬件开关(如专用机器)实现。

这个交换机组成了服务集群的进入点，并提供了网络地址。为了可扩展和可使用性，一个服务集群需要许多接入点，其中每个接入点需要由不同的专用机器实现。这里我们只考虑单一接入点的情况。访问服务器集群的标准方法是建立一个TCP连接，然后通过该连接发送作为会话一部分的应用程序级请求。连接结束时代表了会话的结束。在传输层交换机的情况下，交换机接受进入的TCP连接请求，并将这些连接传递给其中的一台服务器。实际上，这里有两种交换机工作的方法。

第一，客户端建立一个TCP连接，之后所有的请求和响应都通过这个交换机。这个交换机同样会和服务器建立TCP连接，并将客户端的请求发送给服务器，并接受服务器的响应返回给客户端。实际上，这个交换机位于客户和服务端之间，重定向了TCP分片的源端和目标端。这种方法**network address translation（NAT）**的一种转换方式之一。

或者，交换机实际上可以将连接传递给所选的服务器，这样所有响应都可以直接与客户机通信，而无需经过服务器。图3.19显示了**TCP切换**的工作原理。



当交换机接收到TCP连接请求时，它首先确定处理该请求的最佳服务器，然后将请求包转发给该服务器。反过来，服务器将向发出请求的客户机发送一个确认，但是将交换机的IP地址作为携带TCP段的IP包的头的源字段（就是返回给客户端的ip包中，header中的源ip写的是交换机的，即对于客户端来说，这个请求时交换机发过来的）。注意，这个地址重写对于客户机继续执行TCP协议是必要的：它期望从交换机得到一个答复，而不是从某个它从未听说过的任意服务器得到。显然，TCP切换实现需要操作系统级别的修改。特别是在Web服务器中，当响应比请求大很多时，TCP切换就变得很重要。

可以看出，交换机可以在各个服务器分配负载方面发挥重要作用。通过决定将请求转发到何处，交换机还决定将处理请求的进一步处理的服务器。交换机可以遵循的最简单的负载平衡策略是循环操作：每次从列表中选择下一个服务器将请求转发给它时（轮询）。当然，交换机必须跟踪传递TCP连接的服务器，至少在该连接断开之前是这样（TCP handoff）。事实证明，维护这种TCP 切换的状态，会使得交换机的处理速度变慢。

还可以部署更高级的服务器选择标准。例如，假设服务器集群提供多个服务。如果交换机能够在请求传入时区分这些服务，那么它就可以对将请求转发到何处做出明智的决定。这种服务器的选择可以通过端口在传输层来进行区分。在传输层交互机这种情况下，关于将传入请求转发到何处的决策仅基于传输级别的信息。进一步的步骤是让交换机实际检查传入请求的有效负载。只有在知道有效负载是什么样子的情况下，才可以应用此**content-aware request istribution**的请求分发。例如，在Web服务器的情况下，交换机可以期待HTTP请求，然后根据HTTP请求决定由谁来处理它（note3.8）。