属性，并且你可以在文本编辑器中打开file1.txt。

当使用C编程语言时，你能看到操作系统API函数的细节，因为open、write和close都被定义为C函数。但是，在与操作系统交互时并不会只限于C。其他语言在API之上提供了自己的层，对软件开发人员隐藏了一些复杂性。为了说明这一点，我们用Python编写一个等价的程序。

使用文本编辑器在主文件夹根目录中创建一个名为newfile.py的文件。在文本编辑器中输入如下Python代码：

```
f = open('file2.txt', 'w')❶
f.write('Hello from Python!\n');❷
f.close()
```

让我们查看一下源代码。这个程序实际上执行了与前面程序一样的操作，只是其输出文件名有所不同（file2.txt）❶，写入该文件的文本也不同❷。在这个例子中，Python正好使用了与操作系统API相同的名称（open、write、close），但它们不会直接调用操作系统，而是对Python标准库进行调用。

保存这些代码后，你就可以运行它了。请记住，Python是一种解释型语言，所以与其编译Python代码，不如使用Python解释器运行它，如下所示：

```
$ python3 newfile.py
```

要确定程序是否成功运行，你需要查看是否已经用预期内容创建了file2.txt。你可以再次使用ls和cat进行验证，或者在桌面文件管理器中查看该文件。

```
$ ls
$ cat file2.txt
```

虽然看起来你只利用了Python的功能来操作文件，但请记住，Python无法独立做到这一点。Python解释器在运行代码时，代表你进行系统API调用。你将在设计24中观察到这一点。

设计24：观察系统调用

前提条件：完成设计23。

在本设计中，你将观察在设计23中编写的程序所做的系统调用。为此，你将使用名为strace的工具，该工具跟踪系统调用并把输出结果输出到终端。

在Raspberry Pi上打开一个终端，使用strace运行之前用C编写并编译的newfile程序：

```
$ strace ./newfile
```

strace工具启动一个程序（本例中为newfile）并展示该程序运行时进行的所有系统调用。在输出的开头，你可以看见许多系统调用，它们代表加载可执行文件以及所需库需要做的工作。这是在你编写的代码运行之前所进行的工作，你可以跳过这段文本。在输出的末尾，你应该看到与下面类似的文本：

```
openat(AT_FDCWD, "file1.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) = 3
write(3, "Hello, file!\n", 13)        = 13
close(3)                              = 0
```

这看起来应该很熟悉，它与你用来创建file1.txt并向其写入文本的3个API函数几乎是相同的。从程序中调用的C函数只是对同名系统调用的瘦包装器，open除外，它调用openat系统调用。等号后面的数值是3个系统调用的返回值，在我的系统上，openat函数返回3，这个数字被称为文件描述符，它指向打开的文件。你可以看到后面的write和close调用把文件描述符值当作参数使用。write函数返回13，即已写的字节数。close函数返回0，表示成功。

现在使用相同的方法来查看在设计23中编写的Python程序进行的系统调用。

```
$ strace python3 newfile.py
```

由于strace实际上监视着Python解释器，而Python解释器又必须加载并运行newfile.py，所以在这里我们期望看到更多的输出。如果你查看输出末尾附近的内容，你应该看到对openat、write和close的调用，就像在C程序中所做的那样。这表明，尽管C和Python的源代码之间存在差异，但最终会进行相同的系统调用来与文件交互。

strace工具可以用来快速了解程序如何与操作系统交互。例如，在本章的前面，我们使用ps实用程序得到进程列表。如果你想了解ps是如何工作的，你可以在strace下面运行ps，如下所示：

```
$ strace ps
```

查看这个命令的输出，了解ps进行了哪些系统调用。

设计25：使用glibc

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将编写代码来使用C库，并研究其工作细节。使用文本编辑器在主文件夹根目录中创建一个名为random.c的新文件。在文本编辑器中输入如下C代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
  srand(time(0));❶
  printf("%d\n", rand());❷
  return 0;
}
```

这个小程序只在终端输出一个随机整数值。程序做的第一件事是调用srand函数来设定随机数生成器的种子❶，这是确保生成唯一数字序列的必要步骤。time函数返回的当前时间用作种子值。下一行输出从rand函数返

回的随机值❷。为了实现所有这些操作，程序使用了C库的4个函数（time、srand、rand和printf）。

保存文件后，可以使用GNU C编译器（gcc）把代码编译成可执行文件。下面的命令将random.c作为输入，生成一个名为random的可执行文件：

```
$ gcc -o random random.c
```

现在尝试用下面的命令运行代码。程序应该输出一个随机值。多次运行该程序以确认它输出了不同的数。但是，快速运行程序两次可能产生相同的结果，因为time函数返回的种子值每秒只递增1。

```
$ ./random
```

确保程序正常工作后，你可以查看程序导入的库。一种实现方法是运行readelf实用程序，如下所示：

```
$ readelf -d random | grep NEEDED
```

查看输出中的NEEDED部分，如下所示：

```
0x00000001 (NEEDED)                     Shared library: [libc.so.6]
```

这告诉你库libc.so.6是运行该程序所必需的。这是意料之中的，因为这是GNU C库（也称为glibc）。换句话说，由于程序依赖于C标准库中的函数，因此操作系统必须加载libc.so.6库，以便库代码可用。这是一个好的开始，但是如果你想查看random程序所用的来自该库的特定函数列表，又该怎么办呢？你可以通过下面的命令来观察这一点：

```
$ objdump -TC random
```

它的输出如下：

```
random:     file format elf32-littlearm

DYNAMIC SYMBOL TABLE:
00000000  w   D  *UND*  00000000              __gmon_start__
00000000      DF *UND*  00000000  GLIBC_2.4   srand
00000000      DF *UND*  00000000  GLIBC_2.4   rand
00000000      DF *UND*  00000000  GLIBC_2.4   printf
00000000      DF *UND*  00000000  GLIBC_2.4   time
00000000      DF *UND*  00000000  GLIBC_2.4   abort
00000000      DF *UND*  00000000  GLIBC_2.4   __libc_start_main
```

在上面的输出中，你可以在最右边的列看到预期的4个函数（time、srand、rand和printf）以及一些其他函数。

现在你已确定了random程序导入了哪些glibc函数，你可能还希望看看glibc导出的所有函数。这些函数是库供给程序使用的。你可以用如下命令获取这个信息：

```
$ objdump -TC /lib/arm-linux-gnueabihf/libc.so.6
```

有时，在调试运行中的程序时，查看已加载库的信息是很有用的。让我们通过调试random程序来试一下。首先，输入如下命令：

```
$ gdb random
```

此时，gdb已经加载文件，但还未运行任何指令。在（gdb）提示符下输入下面的内容来启动程序。在到达main函数的起点时，调试器暂停执行。

```
(gdb) start
```

查看已加载的共享库：

```
(gdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0x76fcea30  0x76fea150  Yes         /lib/ld-linux-armhf.so.3❶
0x76fb93ac  0x76fbc300  Yes (*)     /usr/lib/arm-linux-gnueabihf/libarmmem-v71.so❷
0x76e6e050  0x76f702b4  Yes         /lib/arm-linux-gnueabihf/libc.so.6❸
(*): Shared library is missing debugging information.
```

第一个库ld-linux-armhf.so.3❶是Linux动态链接库。它负责加载其他库。Linux ELF二进制文件被编译成使用特定链接器库，这个信息在已编译程序的ELF头文件中。你可以从终端窗口使用如下命令找到random程序的链接器库（非gdb）：

```
$ readelf -l random | grep interpreter
      [Requesting program interpreter: /lib/ld-linux-armhf.so.3]
```

正如你在前面的输出中看到的，为random程序指定的链接器库是ld-linux-armhf.so.3，与我们刚才讨论的动态链接库相同。

回忆一下gdb中的info sharedlibrary输出，你可以看到列出来的第二个库是libarmmem-v71.so❷。这个库在/etc/ld.so.preload文件中指定，这个文件是一个文本文件，它列出了为这个系统上执行的每个程序所加载的库。

现在来看第三个库，也是我们最感兴趣的一个库libc.so.6❸，即GNU、C库（glibc）。在之前的readelf和objdump输出中，你可以看到这个库是由可执行文件导入的，这里，你可以看到它确实在运行时成功加载了。你还可以看出加载它的特定地址范围（0x76e6e050到0x76f702b4），以及加载它的特定目录路径。

你可以通过在gdb中输入quit随时退出调试器。

设计26：查看加载的内核模块

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将查看Raspberry Pi操作系统上加载的内核模块，包括设备驱动程序。设备驱动程序通常在Linux上作为内核模块来实现，虽然不是所有的内核模块都是设备驱动程序。要列出加载的模块，你可以检查/proc/modules文件的内容，或者如下所示使用lsmod工具：

```
$ lsmod
```

要查看特定模块的更多细节，可以使用modinfo实用程序（以snd模块为例），如下所示：

```
$ modinfo snd
```

设计27：了解存储设备和文件系统

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将了解存储设备和文件系统。我们先列出块设备，这是Linux描述存储设备的方式。

```
$ lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
mmcblk0     179:0    0 29.8G  0 disk❶
|—mmcblk0p1 179:1    0  256M  0 part /boot❷
|_mmcblk0p2 179:2    0 29.6G  0 part /❸
```

这里，我们看到名为mmcblk0的单个“盘”❶，它是Raspberry Pi中的MicroSD卡。你可以看到它被分成了两个不同大小的分区。分区1映射到统一目录结构中的/boot目录❷，而分区2则映射到根目录（/）❸。

现在用df命令来看看存储设备的总体使用情况：

```
$ df -h -T
Filesystem     Type      Size  Used Avail Use% Mounted on
/dev/root      ext4       30G  3.0G   25G  11% /❶
devtmpfs       devtmpfs  459M     0  459M   0% /dev
tmpfs          tmpfs     464M     0  464M   0% /dev/shm
tmpfs          tmpfs     464M  6.3M  457M   2% /run
tmpfs          tmpfs     5.0M  4.0K  5.0M   1% /run/lock
tmpfs          tmpfs     464M     0  464M   0% /sys/fs/cgroup
/dev/mmcblk0p1 vfat      253M   52M  202M  21% /boot❷
tmpfs          tmpfs      93M     0   93M   0% /run/user/1000
```

这个命令能让你查看各种已挂载的文件系统、它们的大小以及满载程度。只有根目录❶和/boot❷目录映射到存储设备。其他的是驻留在内存中的临时文件系统，而不是持久存储设备。

通过运行tree命令，你可以查看系统中的目录。这里使用的参数把输出限制为只有目录，并且仅限三层：

```
$ tree -d -L 3 /
```

你可以使用文件管理器应用程序从桌面环境查看类似的视图。

设计28：查看服务

前提条件：一个运行Raspberry Pi操作系统的Raspberry Pi。

在本设计中，你将了解服务/守护进程。Raspberry Pi操作系统使用systemd init系统，它包括一个名为systemctl的实用程序，你可以使用它来查看服务的状态：

```
$ systemctl list-units --type=service --state=running
```

这个命令应该会产生与下面类似的输出：

```
UNIT                      LOAD   ACTIVE SUB     DESCRIPTION
avahi-daemon.service      loaded active running Avahi mDNS/DNS-SD Stack
bluealsa.service          loaded active running BluezALSA proxy
bluetooth.service         loaded active running Bluetooth service
cron.service              loaded active running Regular background ...
dbus.service              loaded active running D-Bus System Message Bus
dhcpcd.service            loaded active running dhcpcd on all interfaces
getty@tty1.service        loaded active running Getty on tty1
hciuart.service           loaded active running Configure Bluetooth Modems ...
rsyslog.service           loaded active running System Logging Service
ssh.service               loaded active running OpenBSD Secure Shell server
systemd-journald.service  loaded active running Journal Service
systemd-logind.service    loaded active running Login Service
systemd-timesyncd.service loaded active running Network Time Synchronization
systemd-udevd.service     loaded active running udev Kernel Device Manager
triggerhappy.service      loaded active running triggerhappy global hotkey daemon
user@1000.service         loaded active running User Manager for UID 1000
```

如果没有自动返回到终端提示，请在终端中按<Q>以退出服务视图。要查看特定服务的详细信息，请尝试以下命令，以cron.service为例：

```
$ systemctl status cron.service
```

该命令的输出包含了与服务关联的进程的路径和PID。以cron.service为例，在我的系统上，其路径是/usr/sbin/cron，它恰好是PID 367。

另一种查看守护进程的方式是查看systemd的所有子进程，即PID 1。这是相关的，因为服务是由systemd启动的，并且显示为PID 1的子进程。请注意，这个输出可能不仅仅包含服务/守护进程，因为PID 1采用了孤儿进程。

```
$ ps --ppid 1
```

第11章

互联网

到目前为止，我们专注于单个设备上发生的计算。本章和第12章将介绍分布在多个设备上的计算。我们将研究计算领域中的两个重大创新：互联网和万维网。本章重点介绍互联网，首先定义关键术语，然后研究网络层次模型，并深入探讨互联网使用的一些基础协议。

11.1 网络术语

要从整体上讨论互联网和网络，首先需要熟悉一些概念和术语，我们将在这里对它们进行介绍。计算机网络（network）是一个允许计算设备相互通信的系统，如图11-1所示。使用诸如Wi-Fi之类的技术，网络可以进行无线连接，Wi-Fi技术用无线电波传输数据。网络也可以用电缆（比如铜线或光纤）进行连接。网络上的计算设备必须使用通用通信协议，即描述信息如何交换的一组规则。

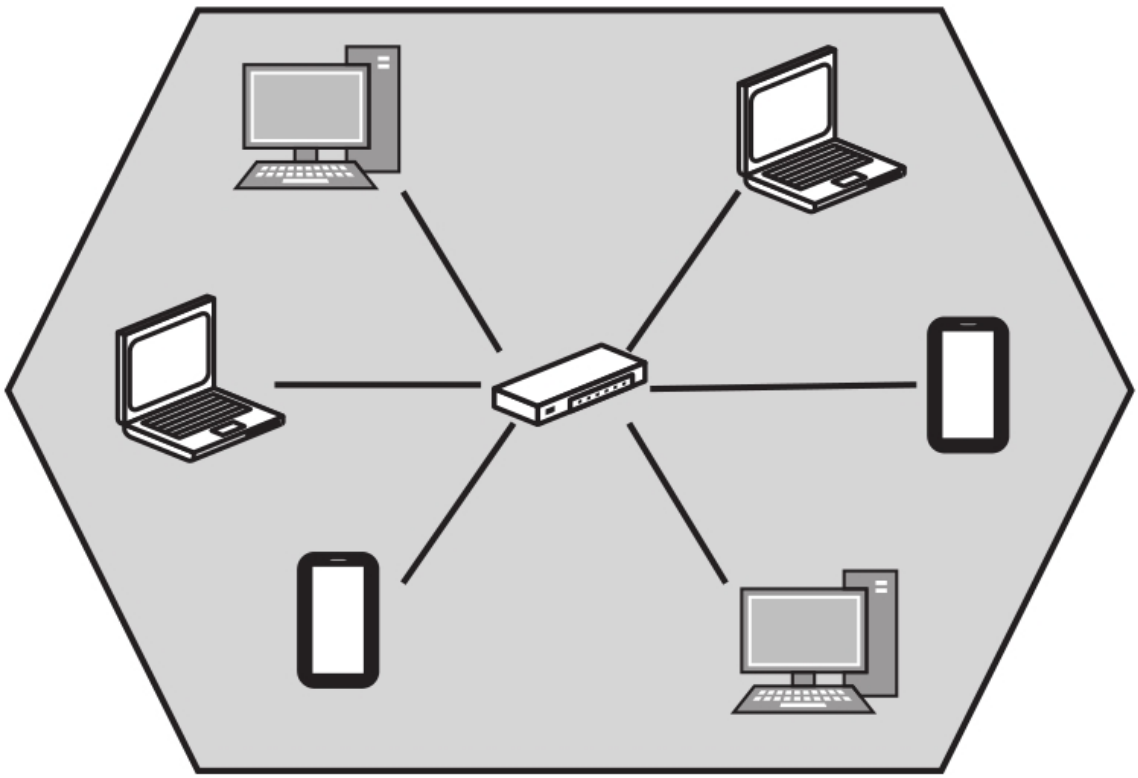

图11-1 计算机网络

互联网（internet）是全球连接的一组计算机网络，它们都使用一套通用协议。互联网是网络的网络，连接世界各地各种组织的网络，如图11-2所示。

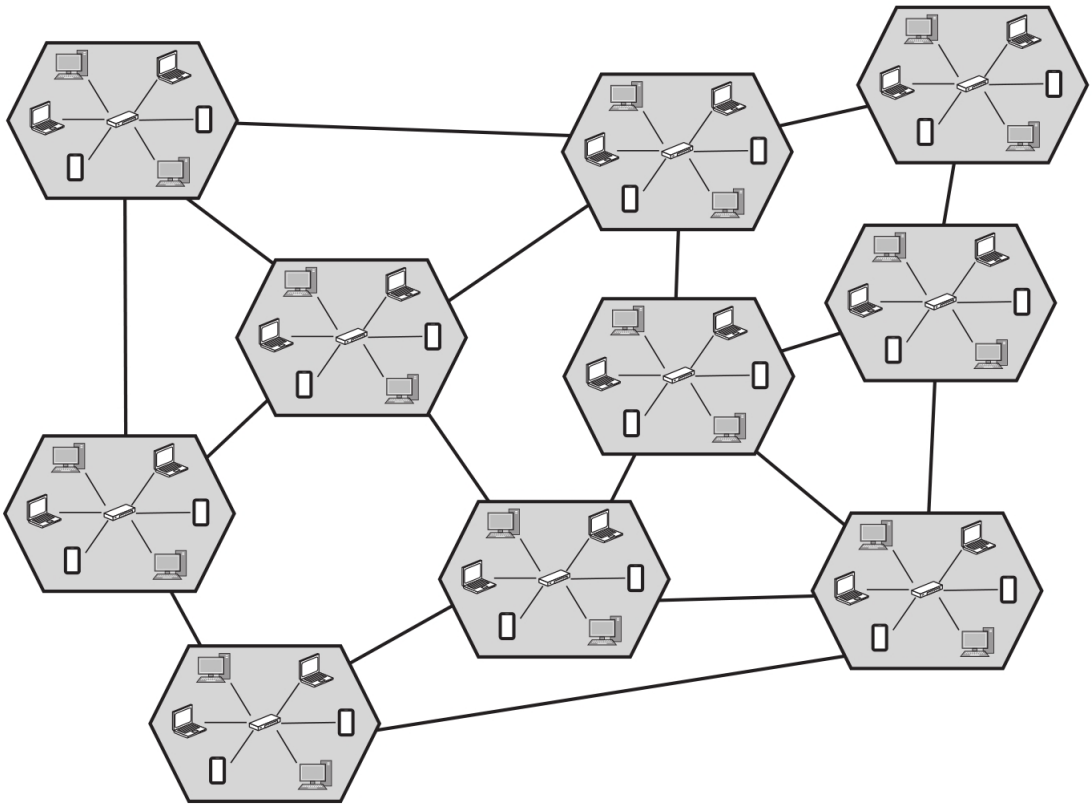

图11-2 互联网：网络的网络

主机或节点是连接到网络的单个计算设备。主机可以充当网络上的服务器或客户端，有时两者兼而有之。网络服务器是监听入站网络连接并向其他主机提供服务的主机，例如Web服务器和电子邮件服务器。网络客户端是从网络服务器建立出站连接并请求服务的主机，例如运行Web浏览器或电子邮件应用程序的智能手机和笔记本计算机。客户端向服务器发出请求，服务器进行响应，如图11-3所示。

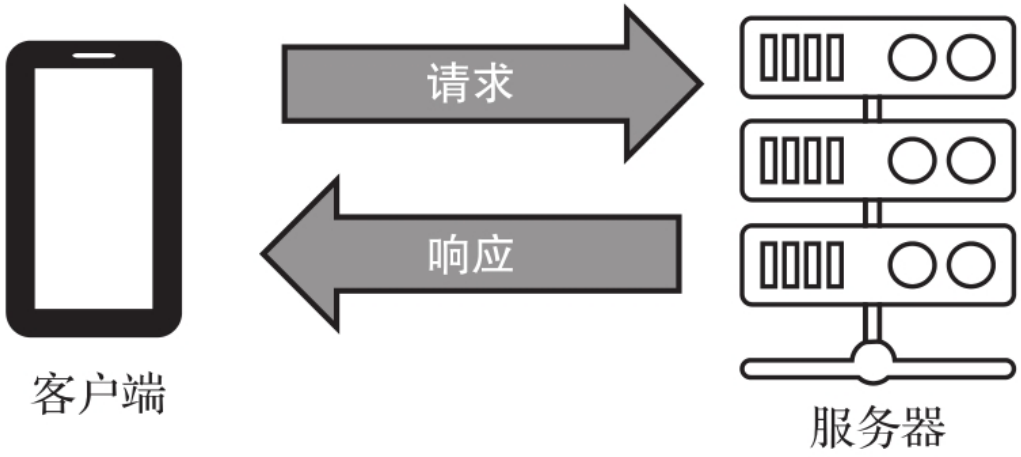

图11-3 客户端向服务器发出请求，服务器进行响应

刚才使用的术语“服务器”是指任何接受入站请求并向客户端提供服务的设备。但是，服务器也可以指专门用作网络服务器的一类计算机硬件。这些专用计算机在物理上设计成安装在数据中心的机架上，通常包含典型PC所没有的硬件冗余和管理功能。不过，任何具有适当软件的设备都可以充当网络的服务器。

11.2 互联网协议套件

物理连接世界网络不足以让这些网络上的设备相互通信。所有参与的计算机都需要用相同的方式通信。互联网协议套件规范了互联网上的通信方式，确保网络上的所有设备“说”同一种语言。互联网协议套件中的两个基本协议是传输控制协议（Transmission Control Protocol，TCP）和互联网协议（Internet Protocol，IP），统称为TCP/IP。

网络协议采用层次模型，这种模型的一个实现被称为网络栈（不要与内存中的栈混淆，第9章介绍了内存中的栈）。底层的协议与底层网络硬件交互，而应用程序与上层的协议交互。中间层的协议提供服务，比如寻址和数据的可靠传送。某一层的协议不必关注整个网络栈，只需关注与之交互的层，这样可以简化整体设计。这是封装的另一个例子。

图11-4 网络的互联网协议套件模型

互联网协议套件是围绕4层模型设计的，有时也被称为TCP/IP模型。TCP/IP模型的4层从下往上分别是：链路层、网络层、传输层和应用层，如图11-4所示。

OSI——另一个网络模型

另一个常用的网络协议模型是开放系统互连（Open Systems Interconnection，OSI）模型。OSI模型把协议分为7层，而不是4层。这个模型经常在技术文献中被提及，但互联网是以互联网协议套件为基础的，所以本书重点介绍TCP/IP模型。

这些网络层代表了一种抽象，一种在讨论互联网操作时使用的模型。在实践中，每一层都是用特定网络协议实现的。每个网络层都代表一个职责范围，协议必须履行相应层的职责。表11-1提供了对每个层的描述。

表11-1 互联网协议套件的4层描述

层	描述	示例协议
应用层	应用层协议提供针对应用程序的功能，比如发送电子邮件或者检索网页。这些协议完成最终用户（或后端服务）想要完成的任务。应用层协议构造了网络上进程对进程通信所使用的数据。所有下层协议都作为“管道”来支持应用层	HTTP、SSH
传输层	传输层协议为应用程序在主机之间发送和接收数据提供通信通道。应用程序根据应用层协议构造数据，然后把数据交给传输层协议，以便将其传输给远程主机	TCP、UDP
网络层	网络层协议提供网络通信机制。这一层识别有地址的主机，并让数据通过互联网从网络路由到网络。传输层依靠网络层进行寻址和路由	IP
链路层	链路层协议提供本地网络上的通信方式。这一层的协议与本地网络上的组网硬件类型密切相关，比如 Wi-Fi。网络层协议依靠链路层协议进行本地网络的通信	Wi-Fi、Ethernet

每层协议都与其相邻层的协议进行通信。从主机出来的传输向下穿过网络的各层，从应用层协议到传输层协议，再到网络层协议，最后到链路层协议。到主机的传输向上穿过网络的各层，其顺序与刚才的描述相反。

尽管网络主机（如客户端或服务器）使用全部4层的协议，但其他网络硬件（如交换机和路由器）只使用较低层的协议。这类设备不用费心检查网络传输所包含的较高层协议数据就可以执行其工作。

从客户端发出的到服务器的请求及其与网络各层的关系如图11-5所示。

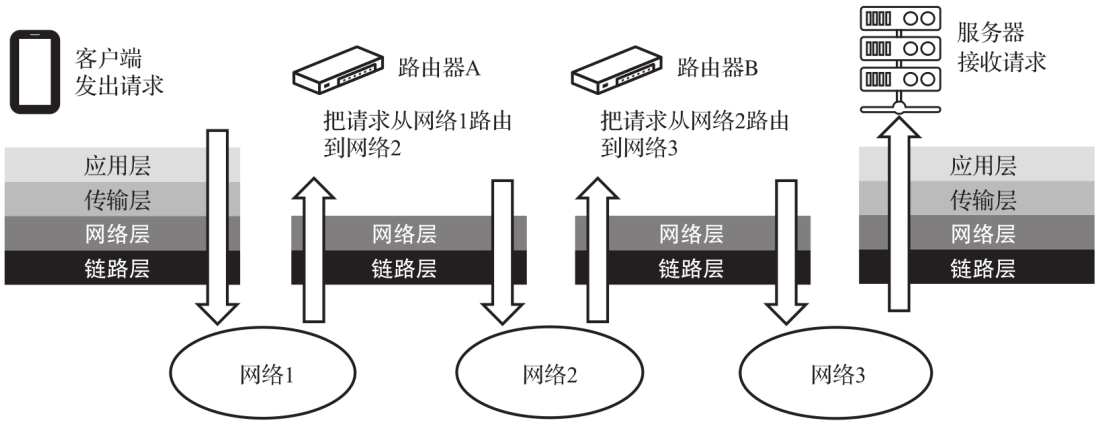

图11-5 一个网络请求通过不同的网络层

让我们来看看图11-5的流程。一个客户端上的应用程序使用应用层协议形成了一个请求。这个请求被传递给传输层协议，然后传递给网络层协