位置，Netflix 仅推送最为流行的视频，视频的流行度是基于逐天的数据来决定的。在 YouTube 视频（［Netflix Video 1］和［Netflix Video 2］）中更为详细地描述了 Netflix CDN 设计。

描述了 Netflix 体系结构的组件后，我们来更为仔细地看一下客户与各台服务器之间的交互，这些服务器与电影交付有关。如前面指出的那样，浏览 Netflix 视频库的 Web 网页由亚马逊云中的服务器提供服务。当用户选择一个电影准备播放时，运行在亚马逊云中的 Netflix 软件首先确定它的哪个 CDN 服务器具有该电影的拷贝。在具有拷贝的服务器中，该软件决定客户请求的“最好的”服务器。如果该客户正在使用一个住宅 ISP，它具有安装在该 ISP 中 Netflix CDN 服务器机架并且该机架具有所请求电影的拷贝，则通常选择这个机架中的一台服务器。倘若不是，通常选择邻近 IXP 的一台服务器。

一旦 Netflix 确定了交付内容的 CDN 服务器，它向该客户发送特定服务器的 IP 地址以及资源配置文件，该文件具有所请求电影的不同版本的 URL。该客户和那台 CDN 服务器则使用专用版本的 DASH 进行交互。具体而言，如 2.6.2 节所述，该客户使用 HTTP GET 请求报文中的字节范围首部，以请求来自电影的不同版本的块。Netflix 使用大约 4 秒长的块［Adhikari 2012］。随着这些块的下载，客户测量收到的吞吐量并且运行一个速率确定算法来确定下一个要请求块的质量。

Netflix 包含了本节前面讨论的许多关键原则，包括适应性流和 CDN 分发。然而，因为 Netflix 使用自己专用的 CDN，而它仅分发视频（而非 Web 网页），所以 Netflix 已经能够简化并定制其 CDN 设计。特别是，Netflix 不需要用如 2.6.3 节中所讨论的 DNS 重定向来将特殊的客户连接到一台 CDN 服务器；相反，Netflix 软件（运行在亚马逊云中）直接告知该客户使用一台特定的 CDN 服务器。此外，Netflix CDN 使用推高速缓存而不是拉高速缓存（2.2.5 节）：内容在非高峰时段的预定时间被推入服务器，而不是在高速缓存未命中时动态地被推入。

2. YouTube

YouTube 具有每分钟 300 小时的视频上载和每天几十亿次观看［YouTube 2016］，毫无疑问 YouTube 是世界上最大的视频共享站点。YouTube 于 2005 年 4 月开始它的服务，并于 2006 年 11 月被谷歌公司收购。尽管谷歌/YouTube 设计和协议是专用的，但通过几个独立的测量结果，我们能够基本理解 YouTube 的工作原理［Zink 2009；Torres 2011；Adhikari 2011a］。与 Netflix 一样，YouTube 广泛地利用 CDN 技术来分发它的视频［Torres 2011］。类似于 Netflix，谷歌使用其专用 CDN 来分发 YouTube 视频，并且已经在几百个不同的 IXP 和 ISP 位置安装了服务器集群。从这些位置以及从它的巨大数据中心，谷歌分发 YouTube 视频［Adhikari 2011a］。然而，与 Netflix 不同，谷歌使用如 2.2.5 节中描述的拉高速缓存和如 2.6.3 节中描述的 DNS 重定向。在大部分时间，谷歌的集群选择策略将客户定向到某个集群，使得客户与集群之间的 RTT 是最低的。然而，为了平衡流经集群的负载，有时客户被定向（经 DNS）到一个更远的集群［Torres 2011］。

YouTube 应用 HTTP 流，经常使少量的不同版本为一个视频可用，每个具有不同的比特率和对应的质量等级。YouTube 没有应用适应性流（例如 DASH），而要求用户人工选择一个版本。为了节省那些将被重定位或提前终止而浪费的带宽和服务器资源，YouTube 在获取视频的目标量之后，使用 HTTP 字节范围请求来限制传输的数据流。

每天有几百万视频被上载到 YouTube。不仅 YouTube 视频经 HTTP 以流方式从服务器到客户，而且 YouTube 上载者也经 HTTP 从客户到服务器上载他们的视频。YouTube 处理它收到的每个视频，将它转换为 YouTube 视频格式并且创建具有不同比特率的多个版本。这种处理完全发生在谷歌数据中心。（参见 2.6.3 节中有关谷歌的网络基础设施的学习案例。）

3. 看看

我们刚讨论了由专用 CDN 运行的专用服务器以流的方式向客户发送 Netflix 和 YouTube 视频。Netflix 和 YouTube 不仅必须为服务器硬件付费，而且要为服务器分发视频时使用的带宽付费。考虑到这些服务的规模和它们消耗的带宽量，这种 CDN 部署的代价可能是很高的。

我们通过描述一种完全不同的方法来对本节进行总结，即经因特网大规模地按需提供视频。这是一种允许服务提供商极大地减少其基础设施和带宽成本的方法。如你可能猜测的那样，这种方法使用 P2P 交付而不是客户 - 服务器交付。自 2011 年以来，看看（由迅雷公司拥有并运行）部署的 P2P 视频交付取得了巨大的成功，每个月都有数千万用户[Zhang 2015]。

从高层面看，P2P 流式视频非常类似于 BitTorrent 文件下载。当一个对等方要看一个视频时，它联系一个跟踪器，以发现在系统中具有该视频副本的其他对等方。这个请求的对等方则并行地从具有该文件的其他对等方请求该视频的块。然而，不同于使用 BitTorrent 下载，请求被优先地给予那些即将播放的块，以确保连续播放［Dhungel 2012］。

看看近期已经向混合 CDN-P2P 流式系统迁移［Zhang 2015］。特别是，看看目前在中国部署了数以百计的服务器并且将视频内容推向这些服务器。这个看看 CDN 在流式视频的启动阶段起着主要作用。在大多数场合，客户请求来自 CDN 服务器的内容的开头部分，并且并行地从对等方请求内容。当 P2P 总流量满足视频播放时，该客户将从 CDN 停止流并仅从对等方获得流。但如果 P2P 流的流量不充分，该客户重新启动 CDN 连接并且返回到混合 CDN-P2P 流模式。以这种方式，看看能够确保短启动时延，与此同时最小地依赖成本高的基础设施服务器和带宽。

2.7　套接字编程：生成网络应用

我们已经看到了一些重要的网络应用，下面探讨一下网络应用程序是如何实际编写的。在 2.1 节讲过，典型的网络应用是由一对程序（即客户程序和服务器程序）组成的，它们位于两个不同的端系统中。当运行这两个程序时，创建了一个客户进程和一个服务器进程，同时它们通过从套接字读出和写入数据在彼此之间进行通信。开发者创建一个网络应用时，其主要任务就是编写客户程序和服务器程序的代码。

网络应用程序有两类。一类是由协议标准（如一个 RFC 或某种其他标准文档）中所定义的操作的实现；这样的应用程序有时称为“开放”的，因为定义其操作的这些规则为人们所共知。对于这样的实现，客户程序和服务器程序必须遵守由该 RFC 所规定的规则。例如，某客户程序可能是 HTTP 协议客户端的一种实现，如在 2.2 节所描述，该协议由 RFC 2616 明确定义；类似地，其服务器程序能够是 HTTP 服务器协议的一种实现，也由 RFC 2616 明确定义。如果一个开发者编写客户程序的代码，另一个开发者编写服务器程

序的代码，并且两者都完全遵从该 RFC 的各种规则，那么这两个程序将能够交互操作。实际上，今天许多网络应用程序涉及客户和服务器程序间的通信，这些程序都是由独立的程序员开发的。例如，谷歌 Chrome 浏览器与 Apache Web 服务器通信，BitTorrent 客户与 BitTorrent 跟踪器通信。

另一类网络应用程序是专用的网络应用程序。在这种情况下，由客户和服务器程序应用的应用层协议没有公开发布在某 RFC 中或其他地方。某单独的开发者（或开发团队）产生了客户和服务器程序，并且该开发者用他的代码完全控制该代码的功能。但是因为这些代码并没有实现一个开放的协议，其他独立的开发者将不能开发出和该应用程序交互的代码。

在本节中，我们将考察研发一个客户 - 服务器应用程序中的关键问题，我们将“亲力亲为”来实现一个非常简单的客户 - 服务器应用程序代码。在研发阶段，开发者必须最先做的一个决定是，应用程序是运行在 TCP 上还是运行在 UDP 上。前面讲过 TCP 是面向连接的，并且为两个端系统之间的数据流动提供可靠的字节流通道。UDP 是无连接的，从一个端系统向另一个端系统发送独立的数据分组，不对交付提供任何保证。前面也讲过当客户或服务器程序实现了一个由某 RFC 定义的协议时，它应当使用与该协议关联的周知端口号；与之相反，当研发一个专用应用程序时，研发者必须注意避免使用这些周知端口号。（端口号已在 2.1 节简要讨论过。它们将在第 3 章中更为详细地涉及。）

我们通过一个简单的 UDP 应用程序和一个简单的 TCP 应用程序来介绍 UDP 和 TCP 套接字编程。我们用 Python 3 来呈现这些简单的 TCP 和 UDP 程序。也可以用 Java、C 或 C ++ 来编写这些程序，而我们选择用 Python 最主要原因是 Python 清楚地揭示了关键的套接字概念。使用 Python，代码的行数更少，并且向新编程人员解释每一行代码不会有困难。如果你不熟悉 Python，也用不着担心，只要你有过一些用 Java、C 或 C ++ 编程的经验，就应该很容易看懂下面的代码。

如果读者对用 Java 进行客户 - 服务器编程感兴趣，建议你去查看与本书配套的 Web 网站。事实上，能够在那里找到用 Java 编写的本节中的所有例子（和相关的实验）。如果读者对用 C 进行客户 - 服务器编程感兴趣，有一些优秀参考资料可供使用［Donahoo 2001；Stevens 1997；Frost 1994；Kurose 1996］。我们下面的 Python 例子具有类似于 C 的外观和感觉。

2.7.1 UDP 套接字编程

在本小节中，我们将编写使用 UDP 的简单客户 - 服务器程序；在下一小节中，我们将编写使用 TCP 的简单程序。

2.1 节讲过，运行在不同机器上的进程彼此通过向套接字发送报文来进行通信。我们说过每个进程好比是一座房子，该进程的套接字则好比是一扇门。应用程序位于房子中门的一侧；运输层位于该门朝外的另一侧。应用程序开发者在套接字的应用层一侧可以控制所有东西；然而，它几乎无法控制运输层一侧。

现在我们仔细观察使用 UDP 套接字的两个通信进程之间的交互。在发送进程能够将数据分组推出套接字之门之前，当使用 UDP 时，必须先将目的地址附在该分组之上。在该分组传过发送方的套接字之后，因特网将使用该目的地址通过因特网为该分组选路到接收进程的套接字。当分组到达接收套接字时，接收进程将通过该套接字取回分组，然后检

查分组的内容并采取适当的动作。

因此你可能现在想知道，附在分组上的目的地址包含了什么？如你所期待的那样，目的主机的 IP 地址是目的地址的一部分。通过在分组中包括目的地的 IP 地址，因特网中的路由器将能够通过因特网将分组选路到目的主机。但是因为一台主机可能运行许多网络应用进程，每个进程具有一个或多个套接字，所以在目的主机指定特定的套接字也是必要的。当生成一个套接字时，就为它分配一个称为**端口号**（port number）的标识符。因此，如你所期待的，分组的目的地址也包括该套接字的端口号。总的来说，发送进程为分组附上目的地址，该目的地址是由目的主机的 IP 地址和目的地套接字的端口号组成的。此外，如我们很快将看到的那样，发送方的源地址也是由源主机的 IP 地址和源套接字的端口号组成，该源地址也要附在分组之上。然而，将源地址附在分组之上通常并不是由 UDP 应用程序代码所为，而是由底层操作系统自动完成的。

我们将使用下列简单的客户 - 服务器应用程序来演示对于 UDP 和 TCP 的套接字编程：

1）客户从其键盘读取一行字符（数据）并将该数据向服务器发送。

2）服务器接收该数据并将这些字符转换为大写。

3）服务器将修改的数据发送给客户。

4）客户接收修改的数据并在其监视器上将该行显示出来。

图 2-26 着重显示了客户和服务器的主要与套接字相关的活动，两者通过 UDP 运输服务进行通信。

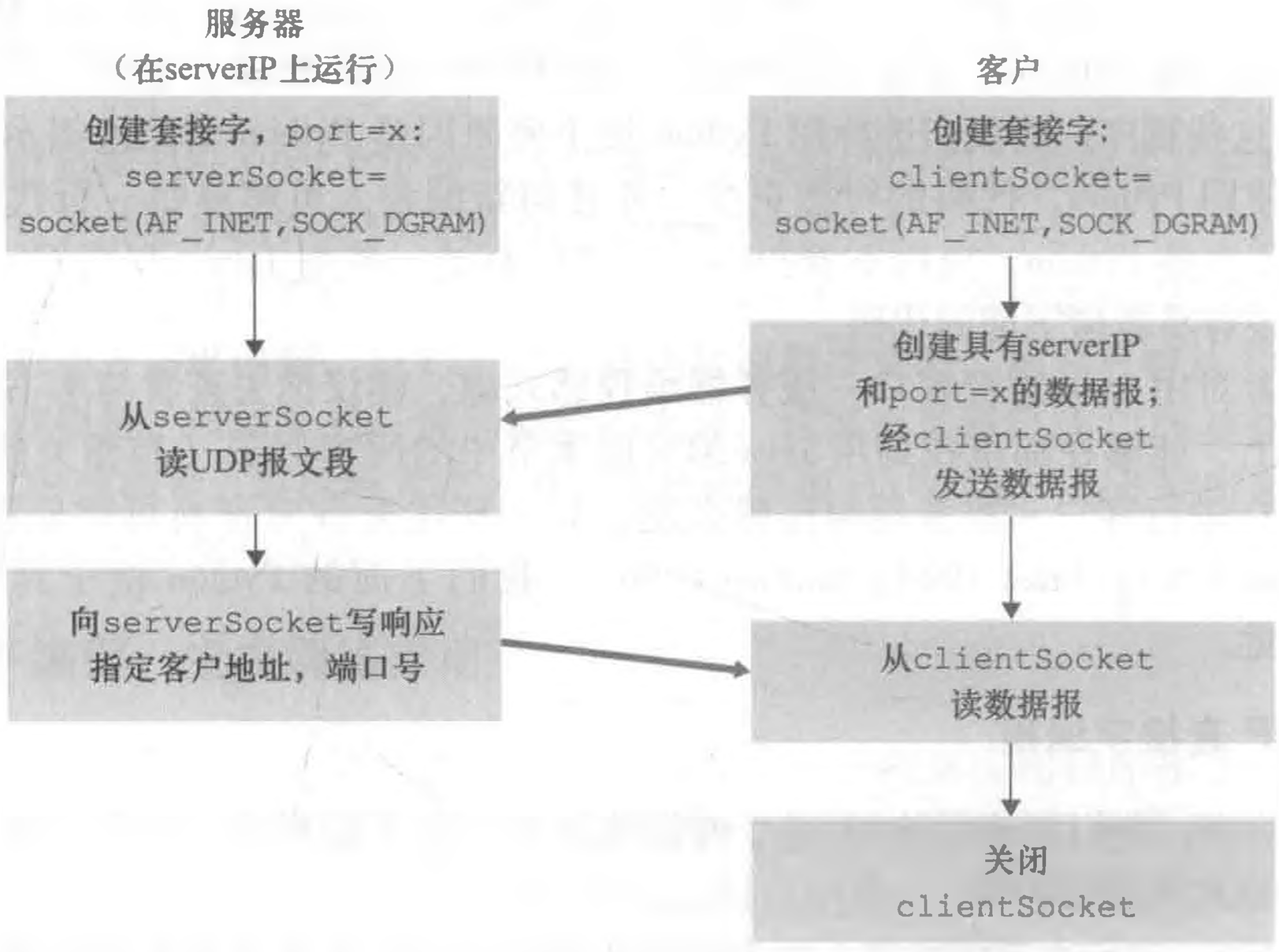

图 2-26　使用 UDP 的客户 - 服务器应用程序

现在我们自己动手来查看用 UDP 实现这个简单应用程序的一对客户 - 服务器程序。我们在每个程序后也提供一个详细、逐行的分析。我们将以 UDP 客户开始，该程序将向服务器发送一个简单的应用级报文。服务器为了能够接收并回答该客户的报文，它必须准备好并已经在运行，这就是说，在客户发送其报文之前，服务器必须作为一个进程正在运行。

客户程序被称为 UDPClient. py，服务器程序被称为 UDPServer. py。为了强调关键问题，我们有意提供最少的代码。“好代码”无疑将具有更多辅助性的代码行，特别是用于处理出现差错的情况。对于本应用程序，我们任意选择了 12000 作为服务器的端口号。

1. UDPClient. py

下面是该应用程序客户端的代码：

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode())
clientSocket.close()
```

现在我们看在 UDPClient. py 中的各行代码。

```
from socket import *
```

该 socket 模块形成了在 Python 中所有网络通信的基础。包括了这行，我们将能够在程序中创建套接字。

```
serverName = 'hostname'
serverPort = 12000
```

第一行将变量 serverName 置为字符串“hostname”。这里，我们提供了或者包含服务器的 IP 地址（如“128. 138. 32. 126”）或者包含服务器的主机名（如“cis. poly. edu”）的字符串。如果我们使用主机名，则将自动执行 DNS lookup 从而得到 IP 地址。第二行将整数变量 serverPort 置为 12000。

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

该行创建了客户的套接字，称为 clientSocket。第一个参数指示了地址簇；特别是，AF_INET 指示了底层网络使用了 IPv4。（此时不必担心，我们将在第 4 章中讨论 IPv4。）第二个参数指示了该套接字是 SOCK_DGRAM 类型的，这意味着它是一个 UDP 套接字（而不是一个 TCP 套接字）。值得注意的是，当创建套接字时，我们并没有指定客户套接字的端口号；相反，我们让操作系统为我们做这件事。既然已经创建了客户进程的门，我们将要生成通过该门发送的报文。

```
message = raw_input('Input lowercase sentence:')
```

raw_input() 是 Python 中的内置功能。当执行这条命令时，客户上的用户将以单词“Input lowercase sentence:”进行提示，用户则使用她的键盘输入一行，该内容被放入变量 message 中。既然我们有了一个套接字和一条报文，我们将要通过该套接字向目的主机发送报文。

```
clientSocket.sendto(message.encode(),(serverName, serverPort))
```

在上述这行中，我们首先将报文由字符串类型转换为字节类型，因为我们需要向套接字中发送字节；这将使用 encode() 方法完成。方法 sendto() 为报文附上目的地址（serverName，serverPort）并且向进程的套接字 clientSocket 发送结果分组。（如前面所述，源地址也附到分组上，尽管这是自动完成的，而不是显式地由代码完成的。）经一

个 UDP 套接字发送一个客户到服务器的报文非常简单！在发送分组之后，客户等待接收来自服务器的数据。

```
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
```

对于上述这行，当一个来自因特网的分组到达该客户套接字时，该分组的数据被放置到变量 modifiedMessage 中，其源地址被放置到变量 serverAddress 中。变量 serverAddress 包含了服务器的 IP 地址和服务器的端口号。程序 UDPClient 实际上并不需要服务器的地址信息，因为它从起始就已经知道了该服务器地址；而这行 Python 代码仍然提供了服务器的地址。方法 recvfrom 也取缓存长度 2048 作为输入。（该缓存长度用于多种目的。）

```
print(modifiedMessage.decode())
```

这行将报文从字节转化为字符串后，在用户显示器上打印出 modifiedMessage。它应当是用户键入的原始行，但现在变为大写的了。

```
clientSocket.close()
```

该行关闭了套接字。然后关闭了该进程。

2. UDPServer. py

现在来看看这个应用程序的服务器端：

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

注意到 UDPServer 的开始部分与 UDPClient 类似。它也是导入套接字模块，也将整数变量 serverPort 设置为 12000，并且也创建套接字类型 SOCK_DGRAM（一种 UDP 套接字）。与 UDPClient 有很大不同的第一行代码是：

```
serverSocket.bind(('', serverPort))
```

上面行将端口号 12000 与该服务器的套接字绑定（即分配）在一起。因此在 UDPServer 中，（由应用程序开发者编写的）代码显式地为该套接字分配一个端口号。以这种方式，当任何人向位于该服务器的 IP 地址的端口 12000 发送一个分组，该分组将导向该套接字。UDPServer 然后进入一个 while 循环；该 while 循环将允许 UDPServer 无限期地接收并处理来自客户的分组。在该 while 循环中，UDPServer 等待一个分组的到达。

```
message, clientAddress = serverSocket.recvfrom(2048)
```

这行代码类似于我们在 UDPClient 中看到的。当某分组到达该服务器的套接字时，该分组的数据被放置到变量 message 中，其源地址被放置到变量 clientAddress 中。变量 clientAddress 包含了客户的 IP 地址和客户的端口号。这里，UDPServer 将利用该地址信息，因为它提供了返回地址，类似于普通邮政邮件的返回地址。使用该源地址信息，服务器此时知道了它应当将回答发向何处。

```
modifiedMessage = message.decode().upper()
```

此行是这个简单应用程序的关键部分。它在将报文转化为字符串后，获取由客户发送的行并使用方法 upper()将其转换为大写。

```
serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

最后一行将该客户的地址（IP 地址和端口号）附到大写的报文上（在将字符串转化为字节后），并将所得的分组发送到服务器的套接字中。（如前面所述，服务器地址也附在分组上，尽管这是自动而不是显式地由代码完成的。）然后因特网将分组交付到该客户地址。在服务器发送该分组后，它仍维持在 while 循环中，等待（从运行在任一台主机上的任何客户发送的）另一个 UDP 分组到达。

为了测试这对程序，可在一台主机上运行 UDPClient. py，并在另一台主机上运行 UDPServer. py。保证在 UDPClient. py 中包括适当的服务器主机名或 IP 地址。接下来，在服务器主机上执行编译的服务器程序 UDPServer. py。这在服务器上创建了一个进程，等待着某个客户与之联系。然后，在客户主机上执行编译的客户器程序 UDPClient. py。这在客户上创建了一个进程。最后，在客户上使用应用程序，键入一个句子并以回车结束。

可以通过稍加修改上述客户和服务器程序来研制自己的 UDP 客户 - 服务器程序。例如，不必将所有字母转换为大写，服务器可以计算字母 s 出现的次数并返回该数字。或者能够修改客户程序，使其在收到一个大写的句子后，用户能够向服务器继续发送更多的句子。

2.7.2 TCP 套接字编程

与 UDP 不同，TCP 是一个面向连接的协议。这意味着在客户和服务器能够开始互相发送数据之前，它们先要握手和创建一个 TCP 连接。TCP 连接的一端与客户套接字相联系，另一端与服务器套接字相联系。当创建该 TCP 连接时，我们将其与客户套接字地址（IP 地址和端口号）和服务器套接字地址（IP 地址和端口号）关联起来。使用创建的 TCP 连接，当一侧要向另一侧发送数据时，它只需经过其套接字将数据丢进 TCP 连接。这与 UDP 不同，UDP 服务器在将分组丢进套接字之前必须为其附上一个目的地地址。

现在我们仔细观察一下 TCP 中客户程序和服务器程序的交互。客户具有向服务器发起接触的任务。服务器为了能够对客户的初始接触做出反应，服务器必须已经准备好。这意味着两件事。第一，与在 UDP 中的情况一样，TCP 服务器在客户试图发起接触前必须作为进程运行起来。第二，服务器程序必须具有一扇特殊的门，更精确地说是一个特殊的套接字，该门欢迎来自运行在任意主机上的客户进程的某种初始接触。使用房子与门来比喻进程与套接字，有时我们将客户的初始接触称为“敲欢迎之门”。

随着服务器进程的运行，客户进程能够向服务器发起一个 TCP 连接。这是由客户程序通过创建一个 TCP 套接字完成的。当该客户生成其 TCP 套接字时，它指定了服务器中的欢迎套接字的地址，即服务器主机的 IP 地址及其套接字的端口号。生成其套接字后，该客户发起了一个三次握手并创建与服务器的一个 TCP 连接。发生在运输层的三次握手，对于客户和服务器程序是完全透明的。

在三次握手期间，客户进程敲服务器进程的欢迎之门。当该服务器“听”到敲门声时，它将生成一扇新门（更精确地讲是一个新套接字），它专门用于特定的客户。在我们下面的例子中，欢迎之门是一个我们称为 serverSocket 的 TCP 套接字对象；它是专门对客

户进行连接的新生成的套接字，称为连接套接字（connectionSocket）。初次遇到 TCP 套接字的学生有时会混淆欢迎套接字（这是所有要与服务器通信的客户的起始接触点）和每个新生成的服务器侧的连接套接字（这是随后为与每个客户通信而生成的套接字）。

从应用程序的观点来看，客户套接字和服务器连接套接字直接通过一根管道连接。如图 2-27 所示，客户进程可以向它的套接字发送任意字节，并且 TCP 保证服务器进程能够按发送的顺序接收（通过连接套接字）每个字节。TCP 因此在客户和服务器进程之间提供了可靠服务。此外，就像人们可以从同一扇门进和出一样，客户进程不仅能向它的套接字发送字节，也能从中接收字节；类似地，服务器进程不仅从它的连接套接字接收字节，也能向其发送字节。

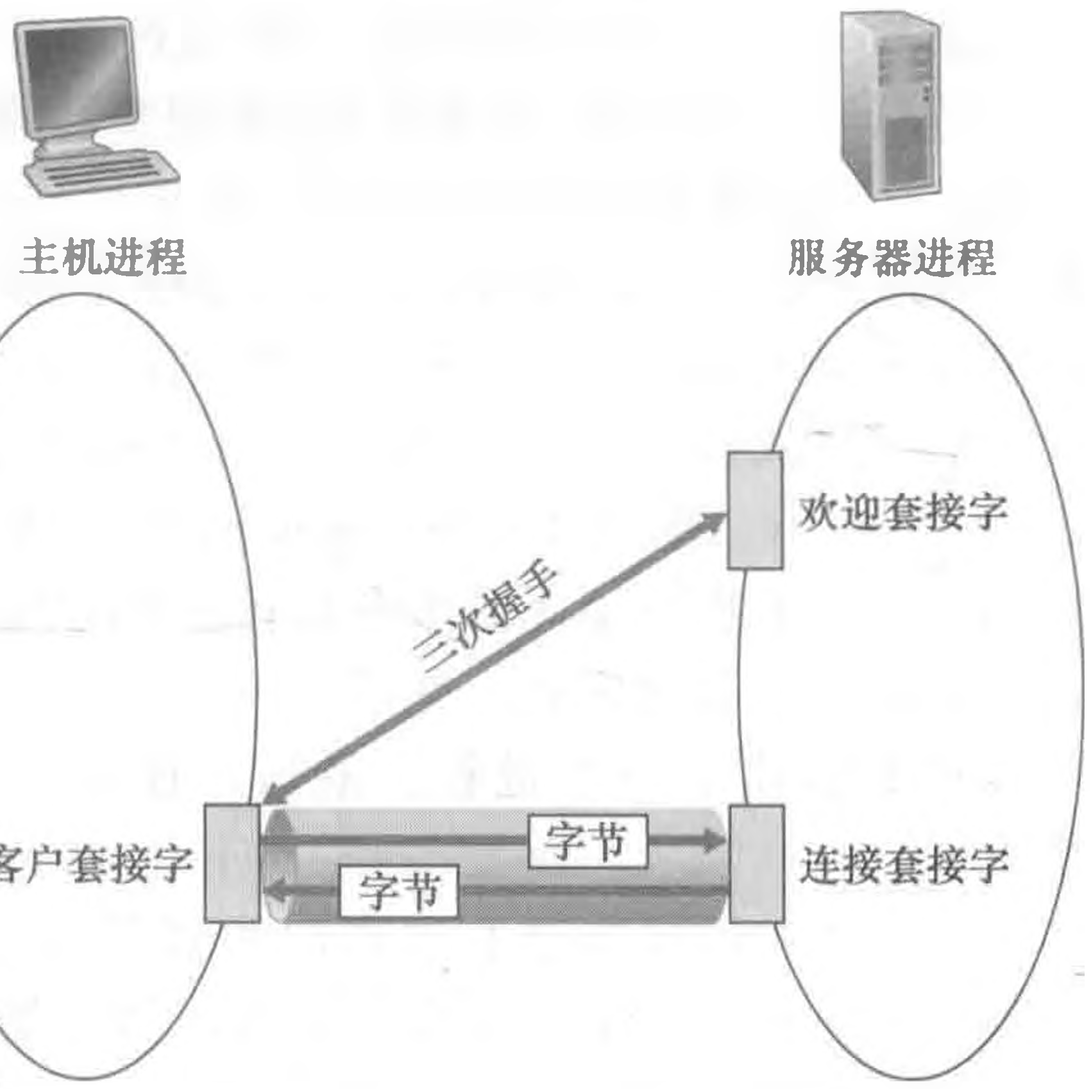

图 2-27　TCPServer 进程有两个套接字

我们使用同样简单的客户 - 服务器应用程序来展示 TCP 套接字编程：客户向服务器发送一行数据，服务器将这行改为大写并回送给客户。图 2-28 着重显示了客户和服务器的主要与套接字相关的活动，两者通过 TCP 运输服务进行通信。

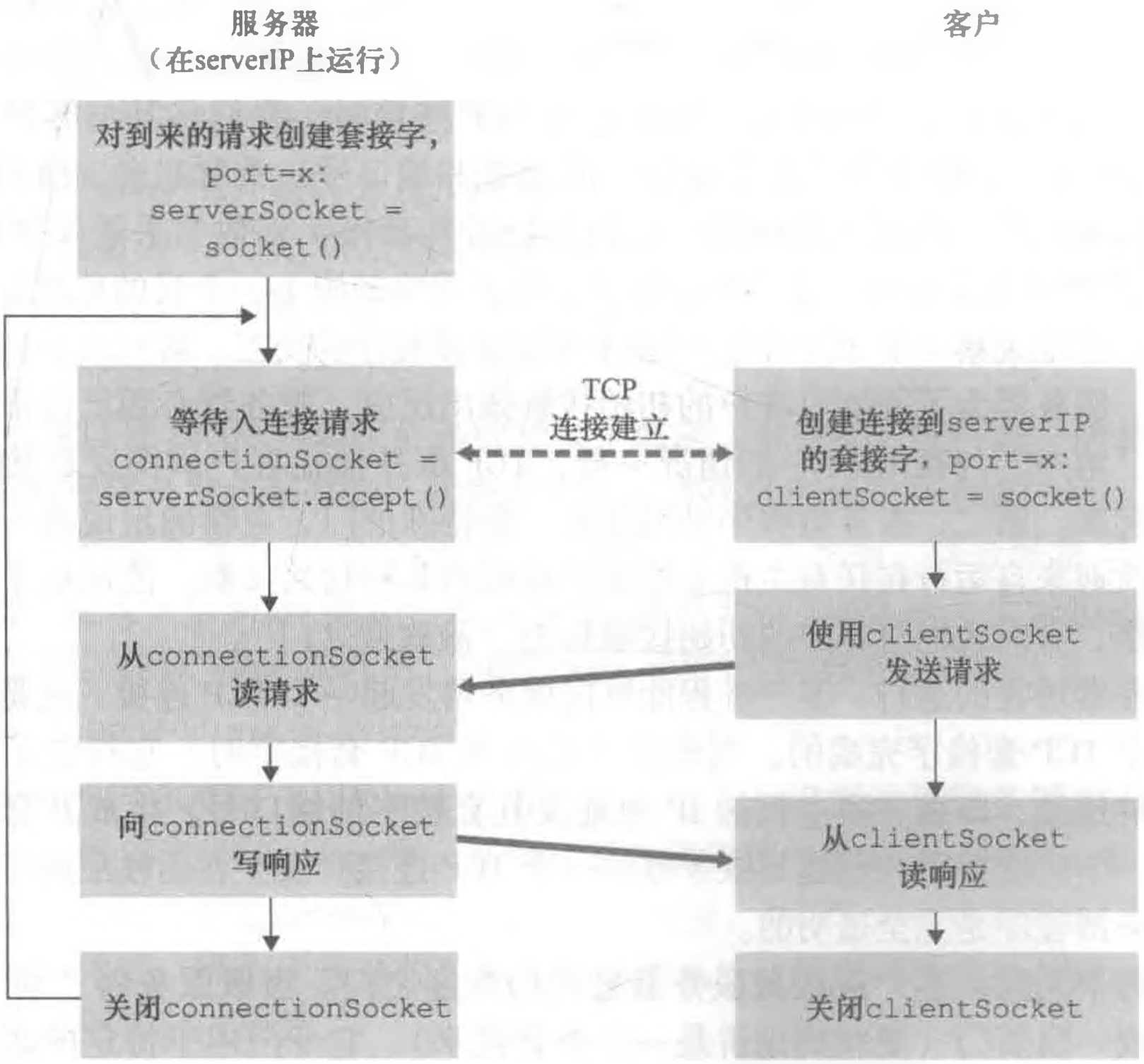

图 2-28　使用 TCP 的客户 - 服务器应用程序

1. TCPClient. py

这里给出了应用程序客户端的代码：

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print('From Server: ', modifiedSentence.decode())
clientSocket.close()
```

现在我们查看这些代码中与 UDP 实现有很大差别的各行。首当其冲的行是客户套接字的创建。

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

该行创建了客户的套接字，称为 clientSocket。第一个参数仍指示底层网络使用的是 IPv4。第二个参数指示该套接字是 SOCK_STREAM 类型。这表明它是一个 TCP 套接字（而不是一个 UDP 套接字）。值得注意的是当我们创建该客户套接字时仍未指定其端口号；相反，我们让操作系统为我们做此事。此时的下一行代码与我们在 UDPClient 中看到的极为不同：

```
clientSocket.connect((serverName,serverPort))
```

前面讲过在客户能够使用一个 TCP 套接字向服务器发送数据之前（反之亦然），必须在客户与服务器之间创建一个 TCP 连接。上面这行就发起了客户和服务器之间的这条 TCP 连接。connect()方法的参数是这条连接中服务器端的地址。这行代码执行完后，执行三次握手，并在客户和服务器之间创建起一条 TCP 连接。

```
sentence = raw_input('Input lowercase sentence:')
```

如同 UDPClient 一样，上一行从用户获得了一个句子。字符串 sentence 连续收集字符直到用户键入回车以终止该行为止。代码的下一行也与 UDPClient 极为不同：

```
clientSocket.send(sentence.encode())
```

上一行通过该客户的套接字并进入 TCP 连接发送字符串 sentence。值得注意的是，该程序并未显式地创建一个分组并为该分组附上目的地址，而使用 UDP 套接字却要那样做。相反，该客户程序只是将字符串 sentence 中的字节放入该 TCP 连接中去。客户然后就等待接收来自服务器的字节。

```
modifiedSentence = clientSocket.recv(2048)
```

当字符到达服务器时，它们被放置在字符串 modifiedSentence 中。字符继续积累在 modifiedSentence 中，直到该行以回车符结束为止。在打印大写句子后，我们关闭客户的套接字。

```
clientSocket.close()
```

最后一行关闭了套接字，因此关闭了客户和服务器之间的 TCP 连接。它引起客户中的 TCP 向服务器中的 TCP 发送一条 TCP 报文（参见 3.5 节）。

2. TCPServer. py

现在我们看一下服务器程序。

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

现在我们来看看上述与 UDPServer 及 TCPClient 有显著不同的代码行。与 TCPClient 相同的是，服务器创建一个 TCP 套接字，执行：

```
serverSocket=socket(AF_INET,SOCK_STREAM)
```

与 UDPServer 类似，我们将服务器的端口号 serverPort 与该套接字关联起来：

```
serverSocket.bind(('',serverPort))
```

但对 TCP 而言，serverSocket 将是我们的欢迎套接字。在创建这扇欢迎之门后，我们将等待并聆听某个客户敲门：

```
serverSocket.listen(1)
```

该行让服务器聆听来自客户的 TCP 连接请求。其中参数定义了请求连接的最大数（至少为 1）。

```
connectionSocket, addr = serverSocket.accept()
```

当客户敲该门时，程序为 serverSocket 调用 accept()方法，这在服务器中创建了一个称为 connectionSocket 的新套接字，由这个特定的客户专用。客户和服务器则完成了握手，在客户的 clientSocket 和服务器的 serverSocket 之间创建了一个 TCP 连接。借助于创建的 TCP 连接，客户与服务器现在能够通过该连接相互发送字节。使用 TCP，从一侧发送的所有字节不仅确保到达另一侧，而且确保按序到达。

```
connectionSocket.close()
```

在此程序中，在向客户发送修改的句子后，我们关闭了该连接套接字。但由于 serverSocket 保持打开，所以另一个客户此时能够敲门并向该服务器发送一个句子要求修改。

我们现在完成了 TCP 套接字编程的讨论。建议你在两台单独的主机上运行这两个程序，也可以修改它们以达到稍微不同的目的。你应当将前面两个 UDP 程序与这两个 TCP 程序进行比较，观察它们的不同之处。你也应当做在第 2、4 和 9 章后面描述的套接字编程作业。最后，我们希望在掌握了这些和更先进的套接字程序后的某天，你将能够编写你自己的流行网络应用程序，变得非常富有和声名卓著，并记得本书的作者！

2.8　小结

在本章中，我们学习了网络应用的概念和实现两个方面。我们学习了被因特网应用普遍采用的客户 - 服务器模式，并且看到了该模式在 HTTP、SMTP、POP3 和 DNS 等协议中的使用。我们已经更为详细地学习了这些重要的应用层协议以及与之对应的相关应用（Web、文件传输、电子邮件和 DNS）。我们也已学习了 P2P 体系结构以及它如何应用在许多应用程序中。我们也学习了流式视频，以及现代视频分发系统是如何利用 CDN 的。对

于面向连接的（TCP）和无连接的（UDP）端到端传输服务，我们走马观花般地学习了套接字的使用。至此，我们在分层的网络体系结构中的向下之旅已经完成了第一步。

在本书一开始的1.1节中，我们对协议给出了一个相当含糊的框架性定义："在两个或多个通信实体之间交换报文的格式和次序，以及对某报文或其他事件传输和/或接收所采取的动作。"本章中的内容，特别是我们对HTTP、SMTP、POP3和DNS协议进行的细致研究，已经为这个定义加入了相当可观的实质性的内容。协议是网络连接中的核心概念；对应用层协议的学习，为我们提供了有关协议内涵的更为直觉的认识。

在2.1节中，我们描述了TCP和UDP为调用它们的应用提供的服务模型。当我们在2.7节中开发运行在TCP和UDP之上的简单应用程序时，我们对这些服务模型进行了更加深入的观察。然而，我们几乎没有介绍TCP和UDP是如何提供这种服务模型的。例如，我们知道TCP提供了一种可靠数据服务，但我们未说它是如何做到这一点的。在下一章中我们将不仅关注运输协议是什么，而且还关注它如何工作以及为什么要这么做。

有了因特网应用程序结构和应用层协议的知识之后，我们现在准备继续沿该协议栈向下，在第3章中探讨运输层。

课后习题和问题

复习题

2.1节

R1. 列出5种非专用的因特网应用及它们所使用的应用层协议。

R2. 网络体系结构与应用程序体系结构之间有什么区别？

R3. 对两进程之间的通信会话而言，哪个进程是客户，哪个进程是服务器？

R4. 对一个P2P文件共享应用，你同意"一个通信会话不存在客户端和服务器端的概念"的说法吗？为什么？

R5. 运行在一台主机上的一个进程，使用什么信息来标识运行在另一台主机上的进程？

R6. 假定你想尽快地处理从远程客户到服务器的事务，你将使用UDP还是TCP？为什么？

R7. 参见图2-4，我们看到在该图中所列出的应用程序没有一个同时既要求无数据丢失又要求定时的。你能设想一个既要求无数据丢失又高度时间敏感的应用程序吗？

R8. 列出一个运输协议能够提供的4种宽泛类型的服务。对于每种服务类型，指出是UDP还是TCP（或这两种协议）提供这样的服务？

R9. 前面讲过TCP能用SSL来强化，以提供进程到进程的安全性服务，包括加密。SSL运行在运输层还是应用层？如果某应用程序研制者想要用SSL来强化UDP，该研制者应当做些什么工作？

2.2～2.4节

R10. 握手协议的作用是什么？

R11. 为什么HTTP、SMTP及POP3都运行在TCP，而不是UDP上？

R12. 考虑一个电子商务网站需要保留每一个客户的购买记录。描述如何使用cookie来完成该功能？

R13. 描述Web缓存器是如何减少接收被请求对象的时延的。Web缓存器将减少一个用户请求的所有对象或只是其中的某些对象的时延吗？为什么？

R14. Telnet到一台Web服务器并发送一个多行的请求报文。在该请求报文中包含If-modified-since：首部行，迫使响应报文中出现"304 Not Modified"状态代码。

R15. 列出几种流行的即时通信应用。它们使用相同的协议作为SMS吗？

R16. 假定Alice使用一个基于Web的电子邮件账户（例如Hotmail或Gmail）向Bob发报文，而Bob使用

POP3 从他的邮件服务器访问自己的邮件。讨论该报文是如何从 Alice 主机到 Bob 主机的。要列出在两台主机间移动该报文时所使用的各种应用层协议。

R17. 将你最近收到的报文首部打印出来。其中有多少 Received：首部行？分析该报文的首部行中的每一行。

R18. 从用户的观点看，POP3 协议中下载并删除模式和下载并保留模式有什么区别吗？

R19. 一个机构的 Web 服务器和邮件服务器可以有完全相同的主机名别名（例如，foo. com）吗？包含邮件服务器主机名的 RR 有什么样的类型？

R20. 仔细检查收到的电子邮件，查找由使用 . edu 电子邮件地址的用户发送的报文首部。从其首部，能够确定发送该报文的主机的 IP 地址吗？对于由 Gmail 账号发送的报文做相同的事。

2. 5 节

R21. 在 BitTorrent 中，假定 Alice 向 Bob 提供一个 30 秒间隔的文件块吞吐量。Bob 将必须进行回报，在相同的间隔中向 Alice 提供文件块吗？为什么？

R22. 考虑一个新对等方 Alice 加入 BitTorrent 而不拥有任何文件块。没有任何块，因此她没有任何东西可上载，她无法成为任何其他对等方的前 4 位上载者。那么 Alice 将怎样得到她的第一个文件块呢？

R23. 覆盖网络是什么？它包括路由器吗？在覆盖网络中边是什么？

2. 6 节

R24. CDN 通常采用两种不同的服务器放置方法之一。列举并简单描述它们。

R25. 除了如时延、丢包和带宽性能等网络相关的考虑外，设计一种 CDN 服务器选择策略时还有其他重要因素。它们是什么？

2. 7 节

R26. 2. 7 节中所描述的 UDP 服务器仅需要一个套接字，而 TCP 服务器需要两个套接字。为什么？如果 TCP 服务器支持 n 个并行连接，每条连接来自不同的客户主机，那么 TCP 服务器将需要多少个套接字？

R27. 对于 2. 7 节所描述的运行在 TCP 之上的客户 - 服务器应用程序，服务器程序为什么必须先于客户程序运行？对于运行在 UDP 之上的客户 - 服务器应用程序，客户程序为什么可以先于服务器程序运行？

习题

P1. 是非判断题。

a. 假设用户请求由一些文本和 3 幅图像组成的 Web 页面。对于这个页面，客户将发送一个请求报文并接收 4 个响应报文。

b. 两个不同的 Web 页面（例如，www. mit. edu/research. html 及 www. mit. edu/students. html）可以通过同一个持续连接发送。

c. 在浏览器和初始服务器之间使用非持续连接的话，一个 TCP 报文段是可能携带两个不同的 HTTP 服务请求报文的。

d. 在 HTTP 响应报文中的 Date：首部指出了该响应中对象最后一次修改的时间。

e. HTTP 响应报文决不会具有空的报文体。

P2. SMS、iMessage 和 WhatsApp 都是智能手机即时通信系统。在因特网上进行一些研究后，为这些系统分别写一段它们所使用协议的文字。然后撰文解释它们的差异所在。

P3. 考虑一个要获取给定 URL 的 Web 文档的 HTTP 客户。该 HTTP 服务器的 IP 地址开始时并不知道。在这种情况下，除了 HTTP 外，还需要什么运输层和应用层协议？

P4. 考虑当浏览器发送一个 HTTP GET 报文时，通过 Wireshark 俘获到下列 ASCII 字符串（即这是一个 HTTP GET 报文的实际内容）。字符 <*cr*> <*lf*> 是回车和换行符（即下面文本中的斜体字符串 <*cr*> 表示了单个回车符，该回车符包含在 HTTP 首部中的相应位置）。回答下列问题，指出你在下面 HTTP GET 报文中找到答案的地方。

```
GET /cs453/index.html HTTP/1.1<cr><lf>Host: gai
a.cs.umass.edu<cr><lf>User-Agent: Mozilla/5.0 (
Windows;U; Windows NT 5.1; en-US; rv:1.7.2) Gec
ko/20040804 Netscape/7.2 (ax) <cr><lf>Accept:ex
t/xml, application/xml, application/xhtml+xml, text
/html;q=0.9, text/plain;q=0.8,image/png,*/*;q=0.5
<cr><lf>Accept-Language: en-us,en;q=0.5<cr><lf>Accept-
Encoding: zip,deflate<cr><lf>Accept-Charset: ISO
-8859-1,utf-8;q=0.7,*;q=0.7<cr><lf>Keep-Alive: 300<cr>
<lf>Connection:keep-alive<cr><lf><cr><lf>
```

a. 由浏览器请求的文档的 URL 是什么?

b. 该浏览器运行的是 HTTP 的何种版本?

c. 该浏览器请求的是一条非持续连接还是一条持续连接?

d. 该浏览器所运行的主机的 IP 地址是什么?

e. 发起该报文的浏览器的类型是什么? 在一个 HTTP 请求报文中，为什么需要浏览器类型?

P5. 下面文本中显示的是来自服务器的回答，以响应上述问题中 HTTP GET 报文。回答下列问题，指出你在下面报文中找到答案的地方。

```
HTTP/1.1 200 OK<cr><lf>Date: Tue, 07 Mar 2008
12:39:45GMT<cr><lf>Server: Apache/2.0.52 (Fedora)
<cr><lf>Last-Modified: Sat, 10 Dec2005 18:27:46
GMT<cr><lf>ETag: "526c3-f22-a88a4c80"<cr><lf>Accept-
Ranges: bytes<cr><lf>Content-Length: 3874<cr><lf>
Keep-Alive: timeout=max=100<cr><lf>Connection:
Keep-Alive<cr><lf>Content-Type: text/html; charset=
ISO-8859-1<cr><lf><cr><lf><!doctype html public "-
//w3c//dtd html 4.0transitional//en"><lf><html><lf>
<head><lf> <meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1"><lf> <meta
name="GENERATOR" content="Mozilla/4.79 [en] (Windows NT
5.0; U) Netscape]"><lf> <title>CMPSCI 453 / 591 /
NTU-ST550ASpring 2005 homepage</title><lf></head><lf>
<much more document text following here (not shown)>
```

a. 服务器能否成功地找到那个文档? 该文档提供回答是什么时间?

b. 该文档最后修改是什么时间?

c. 文档中被返回的字节有多少?

d. 文档被返回的前 5 个字节是什么? 该服务器同意一条持续连接吗?

P6. 获取 HTTP/1.1 规范（RFC 2616）。回答下面问题：

a. 解释在客户和服务器之间用于指示关闭持续连接的信令机制。客户、服务器或两者都能发送信令通知连接关闭吗?

b. HTTP 提供了什么加密服务?

c. 一个客户能够与一个给定的服务器打开 3 条或更多条并发连接吗?

d. 如果一个服务器或一个客户检测到连接已经空闲一段时间，该服务器或客户可以关闭两者之间的传输连接。一侧开始关闭连接而另一侧通过该连接传输数据是可能的吗? 请解释。

P7. 假定你在浏览器中点击一条超链接获得 Web 页面。相关联的 URL 的 IP 地址没有缓存在本地主机上，因此必须使用 DNS lookup 以获得该 IP 地址。如果主机从 DNS 得到 IP 地址之前已经访问了 n 个 DNS 服务器；相继产生的 RTT 依次为 RTT_1、…、RTT_n。进一步假定与链路相关的 Web 页面只包含一个对象，即由少量的 HTML 文本组成。令 RTT_0 表示本地主机和包含对象的服务器之间的 RTT 值。假定该对象传输时间为零，则从该客户点击该超链接到它接收到该对象需要多长时间?

P8. 参照习题 P7，假定在同一服务器上某 HTML 文件引用了 8 个非常小的对象。忽略发送时间，在下列情况下需要多长时间：

a. 没有并行 TCP 连接的非持续 HTTP。

b. 配置有 5 个并行连接的非持续 HTTP。

c. 持续 HTTP。

P9. 考虑图 2-12，其中有一个机构的网络和因特网相连。假定对象的平均长度为 850 000 比特，从这个机构网的浏览器到初始服务器的平均请求率是每秒 16 个请求。还假定从接入链路的因特网一侧的路由器转发一个 HTTP 请求开始，到接收到其响应的平均时间是 3 秒（参见 2.2.5 节）。将总的平均响应时间建模为平均接入时延（即从因特网路由器到机构路由器的时延）和平均因特网时延之和。对于平均接入时延，使用 $\Delta/(1-\Delta\beta)$，式中 Δ 是跨越接入链路发送一个对象的平均时间，β 是对象对该接入链路的平均到达率。

a. 求出总的平均响应时间。

b. 现在假定在这个机构 LAN 中安装了一个缓存器。假定命中率为 0.4，求出总的响应时间。

P10. 考虑一条 10 米短链路，某发送方经过它能够以 150bps 速率双向传输。假定包含数据的分组是 100 000比特长，仅包含控制（如 ACK 或握手）的分组是 200 比特长。假定 N 个并行连接每个都获得 $1/N$ 的链路带宽。现在考虑 HTTP 协议，并且假定每个下载对象是 100Kb 长，这些初始下载对象包含 10 个来自相同发送方的引用对象。在这种情况下，经非持续 HTTP 的并行实例的并行下载有意义吗？现在考虑持续 HTTP。你期待这比非持续的情况有很大增益吗？评价并解释你的答案。

P11. 考虑在前一个习题中引出的情况。现在假定该链路由 Bob 和 4 个其他用户所共享。Bob 使用非持续 HTTP 的并行实例，而其他 4 个用户使用无并行下载的非持续 HTTP。

a. Bob 的并行连接能够帮助他更快地得到 Web 页面吗？

b. 如果所有 5 个用户打开 5 个非持续 HTTP 并行实例，那么 Bob 的并行连接仍将是有好处的吗？为什么？

P12. 写一个简单的 TCP 程序，使服务器接收来自客户的行并将其打印在服务器的标准输出上。（可以通过修改本书中的 TCPServer. py 程序实现上述任务。）编译并执行你的程序。在另一台有浏览器的机器上，设置浏览器的代理服务器为你正在运行服务器程序的机器，同时适当地配置端口号。这时你的浏览器向服务器发送 GET 请求报文，你的服务器应当在其标准输出上显示该报文。使用这个平台来确定你的浏览器是否对本地缓存的对象产生了条件 GET 报文。

P13. SMTP 中的 MAIL FROM 与该邮件报文自身中的 From：之间有什么不同？

P14. SMTP 是怎样标识一个报文体结束的？HTTP 是怎样做的呢？HTTP 能够使用与 SMTP 标识一个报文体结束相同的方法吗？试解释。

P15. 阅读用于 SMTP 的 RFC 5321。MTA 代表什么？考虑下面收到的垃圾邮件（从一份真实垃圾邮件修改得到）。假定这封垃圾邮件的唯一始作俑者是恶意的，而其他主机是诚实的，指出产生了这封垃圾邮件的恶意主机。

```
From - Fri Nov 07 13:41:30 2008
Return-Path: <tennis5@pp33head.com>
Received: from barmail.cs.umass.edu (barmail.cs.umass.
edu
[128.119.240.3]) by cs.umass.edu (8.13.1/8.12.6) for
<hg@cs.umass.edu>; Fri, 7 Nov 2008 13:27:10 -0500
Received: from asusus-4b96 (localhost [127.0.0.1]) by
barmail.cs.umass.edu (Spam Firewall) for <hg@cs.umass.
edu>; Fri, 7
Nov 2008 13:27:07 -0500 (EST)
Received: from asusus-4b96 ([58.88.21.177]) by barmail.
cs.umass.edu
for <hg@cs.umass.edu>; Fri, 07 Nov 2008 13:27:07 -0500
(EST)
Received: from [58.88.21.177] by inbnd55.exchangeddd.
com; Sat, 8
Nov 2008 01:27:07 +0700
From: "Jonny" <tennis5@pp33head.com>
To: <hg@cs.umass.edu>

Subject: How to secure your savings
```

P16. 阅读 POP3 的 RFC，即 RFC 1939。UIDL POP3 命令的目的是什么？

P17. 考虑用 POP3 访问你的电子邮件。

a. 假定你已经配置以下载并删除模式运行的 POP 邮件客户。完成下列事务：

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: blah blah ...
S: .....blah
S: .
?

b. 假定你已经配置以下载并保持模式运行的 POP 邮件客户。完成下列事务：

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: blah blah ...
S: .....blah
S: .
?
?

c. 假定你已经配置以下载并保持模式运行的 POP 邮件客户。使用（b）中的记录，假定你检索报文 1 和 2，退出 POP，5 分钟以后，你再访问 POP 以检索新电子邮件。假定在这 5 分钟间隔内，没有新报文发送给你。给出第二种 POP 会话的记录。

P18. 如题：

a. 什么是 whois 数据库？
b. 使用因特网上的各种 whois 数据库，获得两台 DNS 服务器的名字。指出你使用的是哪个 whois 数据库。
c. 你本地机器上使用 nslookup 向 3 台 DNS 服务器发送 DNS 查询：你的本地 DNS 服务器和两台你在（b）中发现的 DNS 服务器。尝试对类型 A、NS 和 MX 报告进行查询。总结你的发现。
d. 使用 nslookup 找出一台具有多个 IP 地址的 Web 服务器。你所在的机构（学校或公司）的 Web 服务器具有多个 IP 地址吗？
e. 使用 ARIN whois 数据库，确定你所在大学使用的 IP 地址范围。
f. 描述一个攻击者在发动攻击前，能够怎样利用 whois 数据库和 nslookup 工具来执行对一个机构的侦察。
g. 讨论为什么 whois 数据库应当为公众所用。

P19. 在本习题中，我们使用在 Unix 和 Linux 主机上可用的 dig 工具来探索 DNS 服务器的等级结构。图 2-18讲过，在 DNS 等级结构中较高的 DNS 服务器授权对该等级结构中较低 DNS 服务器的 DNS 请求，这是通过向 DNS 客户发送回那台较低层次的 DNS 服务器的名字来实现的。先阅读 dig 的帮助页，再回答下列问题。

a. 从一台根 DNS 服务器（从根服务器 [a-m].root-server.net 之一）开始，通过使用 dig 得到你所在系的 Web 服务器的 IP 地址，发起一系列查询。显示回答你的查询的授权链中的 DNS 服务器的名字列表。
b. 对几个流行 Web 站点如 google.com、yahoo.com 或 amazon.com，重复上一小题。

P20. 假定你能够访问所在系的本地 DNS 服务器中的缓存。你能够提出一种方法来粗略地确定在你所在系的用户中最为流行的 Web 服务器（你所在系以外）吗？解释原因。

P21. 假设你所在系具有一台用于系里所有计算机的本地 DNS 服务器。你是普通用户（即你不是网络/系统管理员）。你能够确定是否在几秒前从你系里的一台计算机可能访问过一台外部 Web 站点吗？解释原因。

P22. 考虑向 N 个对等方分发 $F = 15\text{Gb}$ 的一个文件。该服务器具有 $u_s = 30\text{Mbps}$ 的上载速率，每个对等方具有 $d_i = 2\text{Mbps}$ 的下载速率和上载速率 u。对于 $N = 10$、100 和 1000 并且 $u = 300\text{kbps}$、700kbps 和