实验四 Socket 网络编程讲义

谢瑞桃

March 29, 2021

本讲义内容主要摘自《计算机网络：自顶向下方法》第 2 章第 7 节，略有修改。更多内容请参考原书。

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；传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，它几乎无法控制传输层一侧。

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将该分组路由到接收进程的套接字。当分组到达接收套接字时，接收进程将通过接收套接字取回分组，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，其包含目的主机的 IP 地址。通过在分组中包括目的 IP 地址。因特网中的路由器将能够通过因特网将分组路由到目的主机。但是因为一台主机可能运行很多网络应用程序，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是必要的。当生成一个套接字时，系统就为它分配一个端口号。因此，分组的目的地址也包括进程或套接字的端口号。此外，发送方的源地址也是由源 IP 地址和源套接字的端口号组成，附在分组上。然而，将源地址附在分组上不是由 UDP 应用程序代码实现，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。

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。

1.1 UDPClient.py

下面是该应用程序客户端的代码：

```
from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence:') #used in python3, use raw_input() in python2
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 = 'localhost'
serverPort = 12000
```

第一行将变量 serverName 置为字符串”localhost”，即本地主机。这里，可以为服务器的 IP 地址（如“128.138.32.126”）或者主机名（如”cis.poly.edu”）。如果我们使用主机名，则将自动执行 DNSlookup 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 = input('Input lowercase sentence:') #used in python3, use raw_input() in python2
```

input() 是 python 中的内置功能。当执行这条命令时，客户端将显示提示行"Input lowercase sentence:"，用户则使用她的键盘输入一行，该内容被放入变量 message 中。注意：该函数在 Python3 中使用；如果你使用的是 Python2，请使用 raw_input()。

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 取缓存长度 2048 作为输入。

```
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。

1.2 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 包含了客户端的 IP 地址和端口号。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() 将其转换成大写。

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

最后一行将该客户端的地址（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。）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端发送的）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两个程序。也可以在两台不同主机上运行，但要注意修改 UDPClient.py 中的 serverName 。在服务器主机上执行服务器程序 UDPServer.py。这在服务器上创建了一个进程，等待着某个客户端与之联系。然后，在客户端主机上执行客户端程序 UDPClient.py。这在客户上创建了一个进程。最后，在客户端上使用应用程序，键入一个句子并以回车结束。

可以通过稍加修改上述客户端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 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。客户端具有向服务器发起接触的任务。服务器为了能够对客户端的初始接触做出反应，服务器必须先准备好。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。当该客户端生成其 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 地址及其进程的端口号。生成其套接字后，该客户端发起了一个三次握手并创建与服务器的一个 TCP 连接。发生在传输层的三次握手，对于客户和服务器程序是完全透明的（应用程序看不到三次握手）。

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），它专门用于这次敲门的客户。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；新生成的专门连接某个客户的套接字，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和每个新生成的服务器侧的连接套接字。

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。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。

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：客户端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。

2.1 TCPClient.py

```
from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = input('Input lowercase sentence:')
```

```
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print('From Server: {}'.format(modifiedSentence.decode()))
clientSocket.close()
```

现在我们来看看上述与 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，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 = 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())
```

上一行通过该客户端的套接字进入 TCP 连接并发送字符串 sentence。值得注意的是，该程序并未显式地附上目的地址，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。客户端然后就等待接收来自服务器的字节。

```
modifiedSentence = clientSocket.recv(1024)
```

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，我们关闭客户端的套接字。

```
clientSocket.close()
```

最后一行关闭了套接字，因此关闭了客户端和服务器之间的 TCP 连接。它引起客户中的 TCP 向服务器中的 TCP 发送一条拆除连接的报文。

2.2 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 connect')
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.py 及 TCPClient.py 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。