4.5 Accessing Networks

Python is a good language that can manage networking very easily and efficiently. Network services can be managed from a low level to a high level. Services over networks are carried out between servers and clients. At a low level, sockets are a networking interface and are bound to a port of computers or network devices. On the other hand, at a high level, services on applications are also based on a network protocol, such as FTP, HTTP and so on.

In this section, we will discuss about client-server Python programming and a socket-interface based packing data analysis. In Python scripting of any cases, sockets need to be fully understood and properly configured.

There are various socket families to be considered: `AF_UNIX, AF_INET, AF_NETLINK AF_TIPC, AF_CAN, PF_NET, PF_SYSTEM` and `AF_BLUETOOTH`. AF_* refers to address format family, while PF_* packet format family. AF_INET uses a (host, port) pair and AF_INET6 uses a four-tuple (host, port, flowinfo, scopeid). As shown in https://docs.python.org/3/library/socket.html, the exceptions to be caught include `socket.error, socket.herror, socket.gaierror` and `socket.timeout`, and the constants to be used include `socket.AF_UNIX, socket.AF_INET, socket.AF_INET6, socket.SOCK_STREAM, socket.Sock_DGRAM, socket.SOCK_RAW, socket.SOCK_RDM, socket.SOCK_SEQPACKET, socket.SOCK_CLOEXEC` and `socket.SOCK_NONBLOCK`. Etc.

The socket-related functions to be used include **`socket.socket(family=AFINET, type=SOCK_STREAM, proto=0, filen=None)`**, `socket.socketpair(), socket.create_connection(), socket.fromfd(), socket.fromshare(), socket.SocketType, socketgetaddrinfo(), socket.getfqdn(), socket.gethostbyname(), socket.gethostbyname_ex, socket.gethostname(), socket.gethostbyaddr(), socket.getnameinfo(), socket.getprotobyname(), socket.getservbyname(), socket.getservbyport(), socket.sethostname(), socket.if_nameindex()`, etc, as shown in https://docs.python.org/3/library/socket.html.

The socket methods that socket objects can invoke include `socket.accept(), socket.bind(), socket.close(), socket.connect(), socket.detach(), socket.dup(), socket.getpeername(), socket.getsockname(), socket.getsockopt(), socket.listen(), socket.recv(), socket.recvfrom(), socket.recvmsg(), sock.send(), socket.sendall(), socket.sendfile(), socket.setblocking(), socket.shutdown(), socket.share(), socket.family, socket.type, socket.proto`, etc.

With this introductory to socket description, let us consider client and server communications.

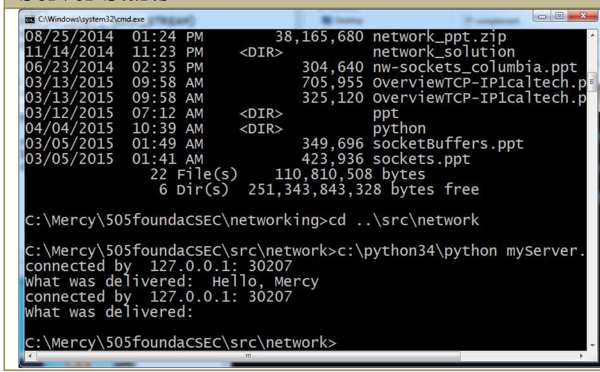
4.5.1 Client and Server Programming

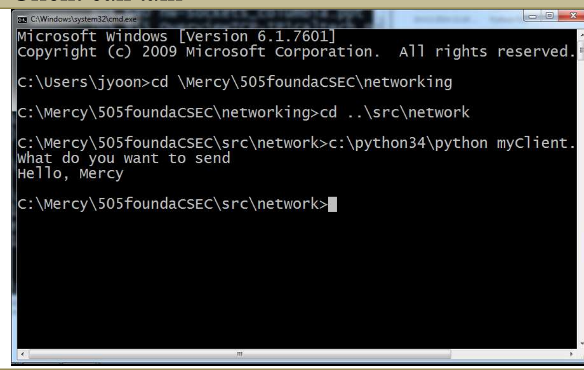
There are two basic IP-based network communication protocols. The protocol that connects two endpoints is called TCP (Transmission Control Protocol), and the one that does not need a connection is called UDP (User Datagram Protocol). TCP is used in the communications that need to confirm data transmission, while UDP is used for broadcasting which may not need data transmission confirmation.

Take a look at a sample run below. First of all, socket programs instantiate `socket()` by assigning the socket family, `AF_INET` for windows and specifying the content type, `SOCK_STREAM`. A server socket program, `myServer.py`, should start running and listening to a designated port.

EXAMPLE 4.5.1: Write Python scripts, `myServer.py` that can start to open a socket and listen to any possible client request, and `myClient.py`, that can connect the server and send a data message. When the designated port detects a data being transmitted, the server socket object accepts it. It can receive any data that may be transmitted to the server port. While the server is up and running, a client socket program, `myClient.py`, can connect the designated port of the server.

Server Starts	Client can talk
(see below)	(see below)





The sample code that enables the above client/server data transmission is below:

Script `myServer.py` for a TCP server

```
(1)  from socket import *

(2)  def main():
(3)    s=socket(AF_INET, SOCK_STREAM)
(4)    s.bind((' ',10530))
(5)    s.listen(1)
(6)    conn, (rmip, rmpt) = s.accept()

(7)    while 1:
(8)      print ("connected by ",
     str(rmip)+": " + str(rmpt))
(9)      data = conn.recv(1024)
(10)     print ("What was delivered: ",
     data.decode())

(11)     if not data:
(12)        break
(13)   conn.close()
(14) main()
```

Script `myClient.py` for a TCP client

```
(1)   from socket import *

(2)   def main():
(3)     s = socket(AF_INET, SOCK_STREAM)
(4)     s.connect(('localhost',10530))
(5)     sendme = input("What do you want
      to send\n")
(6)     s.send(sendme.encode())
(7)   main()
```

Note that the code address family used by the sample Python script above is only for IPv4. If you want to extend this script for both IPv4 and IPv6 together, the server side socket should listen to the first address family available. Once IPv6 takes precedence, then the server may not accept IPv4 traffic. The detailed will be left for the reader's assignment.

Server socket instance is bound to the local host's port 10530 in line (4) and listen for connection through the port in line (5) on the left. A port number can be assigned in the coding lists. Meanwhile, a client socket tries to connect the port 10530 of the server in line (4) on the right, which is local host. At this point, the server socket accepts the data signal in line (6). The acceptance of a socket returns a pair of (conn, address), where conn is a socket object which can send and receive data, and address in both ip and port is bound to the socket on the other end of the connection. In line (8) on the left, the remote IP and remote port, `rmip` and `rmpt`, can be recognized by the server socket as shown in line (8) on the left. For each client socket, there will be a separate connection with the address to the client.

The protocol implemented above is TCP. Lines (5)-(6) in the server code and line (4) are the Python statements that connect two endpoints of communication. Implementation of UDP protocol does not need such statements for connection. Please note that each implementation for TCP or UDP starts from the construction of a specific socket object. A TCP socket object is constructed with the parameter of content type called `SOCK_STREAM`, and a UDP with the content type called `SOCK_DGRAM`. See the following sample script for both UPD server and client.

Script `myServer.py` for a UDP server

```
(1)     from socket import *
(2)
(3)     def main():
(4)       s = socket(AF_INET, SOCK_DGRAM)
(5)       s.bind(('', 54321)) # port number
(6)
(7)       while 1:
(8)         message, address =
        s.recvfrom(1024)
(9)         print( 'Server recieved ',
        message)
(10)        message = message.upper()
(11)        s.sendto(message, address)
(12)
(13)    main()
```

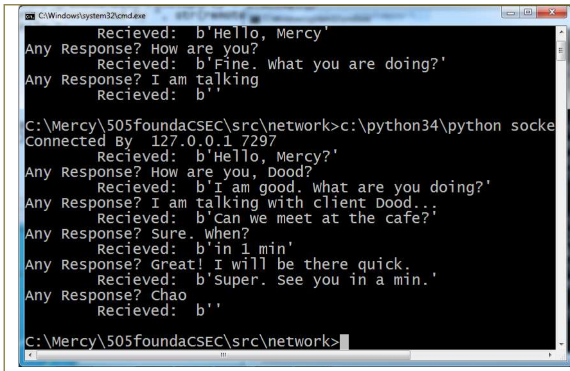
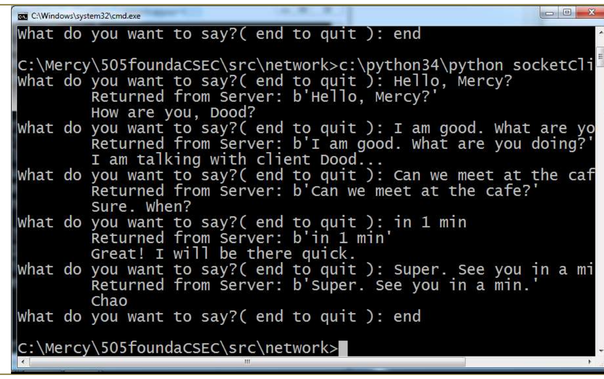
Script `myClient.py` for a UDP client

```
(1)     from socket import *
(2)
(3)     for ping in range(10):
(4)       c = socket(AF_INET, SOCK_DGRAM)
(5)       c.settimeout(1)
(6)       message = b"hello, mercy"
(7)       address = ('localhost', 54321)
        # port number
(8)
(9)       c.sendto(message, address)
(10)
(11)      try:
(12)        data, server =
        c.recvfrom(1024)
(13)      except timeout:
(14)        print ("Request timed out")
```

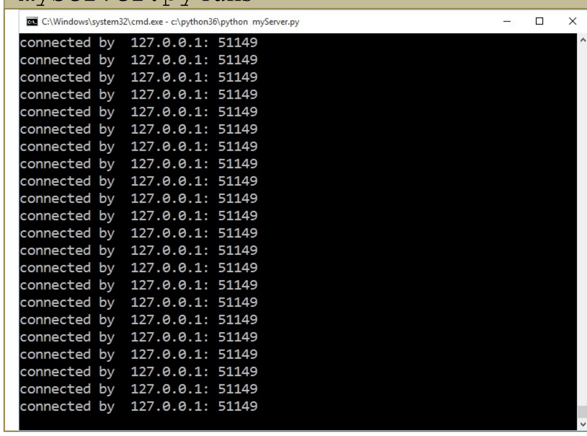
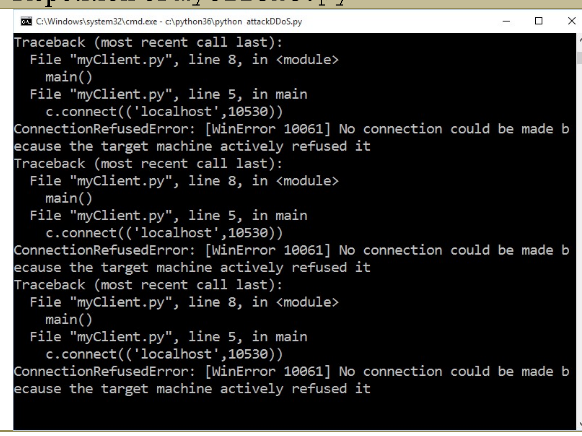
In the Python scripts above, both server and client can communicate without having to set up a either virtual or physical communication.

EXERCISE 4.5.1: Write Python scripts, `socketServer.py` and `socketClient.py`, that can talk continually. Connection will end only the client ends. See a sample run below:

Server Starts	Client can talk

EXERCISE 4.5.2: Write Python scripts, `attackDDoS.py` that can invoke the client script `myClient.py` automatically more than 100 times to the same server, and see how your socket server denies the requests from the clients. A sample run is illustrated below:

`myServer.py` runs	Repetition of `myClient.py`
	

Note that although the server is running but due to overloading, a few requests from the client are denied.

Hint: The Python script required is a simple Python that can run the client socket program 100 times in a loop.

4.5.2 Network Sniffer: Network Packet Capturing

In this section, we describe how Python scripts can access network interface cards and extract raw socket data to interpret. Recall a socket object is constructed by `socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)`, where the address family for the Internet, raw socket data and IP protocol are used. Please note that the accesses to socket data require administrator privileges. A simple network sniffer needs the following steps:

1) Acquire a network interface
2) Create an object of a raw socket (as discussed above)
3) Bind the socket object to the network interface obtained above
4) Include IP headers

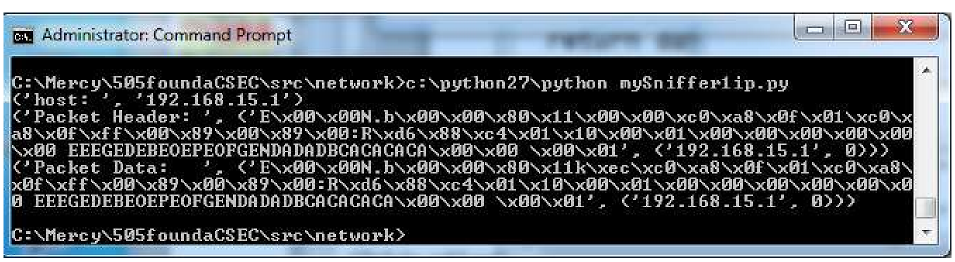
5) Receive all packages
6) Receive a packet header
7) Receive a packet data
8) Interpret the packet data

With this simple procedure, steps 1-7, the following Python script runs to display the packet data.

EXAMPLE 4.5.2: Write Python scripts, `mySniffer1ip.py` to capture IP packets one time when the socket is bound to a network interface.

Script `mySniffer1ip.py`

```
import socket

# acquire network interface
HOST = socket.gethostbyname(socket.gethostname())
print ("host: ", HOST)

# create an object of a raw socket
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)

# bind the socket object to the network interface obtained above
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a packet header
ipHeader = s.recvfrom(65565)
print ("Packet Header: ", ipHeader)

# receive a packet data
tcpData = s.recvfrom(65565)
print ("Packet Data:   ", tcpData)
```



Sample Run

The above example uses SOCK_RAW type for IP protocol. Can we extend this basic technique to TCP and UDP protocols? Let's first consider TCP on Linux and leave UDP protocol for exercise.

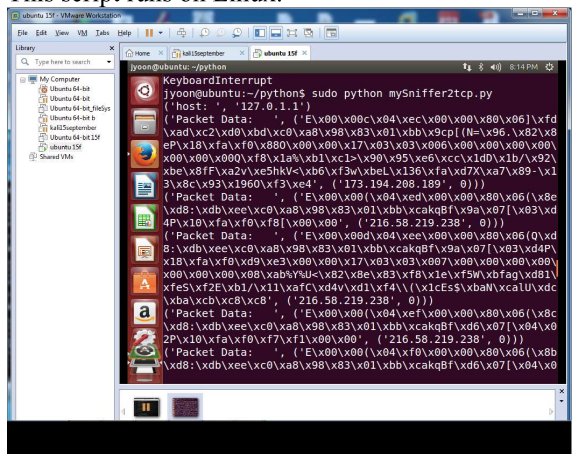
EXAMPLE 4.5.3: Write Python script, `mySniff2tcpp.py` to capture TCP packets. The Python script should bind an interface and a port. How do you know the logical interface ID for an interface card that you want to bind? In Windows, the command `netsh interface show interface` may give you some hints. Or, a graphical user interface is available by typing, `msinfo32`. In Linux, run the command, `sudo ethtool eth0` with options.

Script `mySniff2tcp.py` on Linux

```
(1) import socket, sys

(2) try:
(3)   # create an object of a raw socket
(4)   s = socket.socket(socket.AF_INET,
      socket.SOCK_RAW, socket.IPPROTO_TCP)

      except socket.error as e:
(5)   print ("Socket creation failed: Code
        {} Message
        {}".format(str(e[0]),str(e[1])))
(6)   sys.exit()

(7)while True:
(8)   # receive a packet data
(9)   tcpData = s.recvfrom(65565)
10)   print ("Packet Data:   ", tcpData)
```

Sample Run `mySniff2tcp.py` on Linux

This script runs on Linux.



Running on Ubuntu

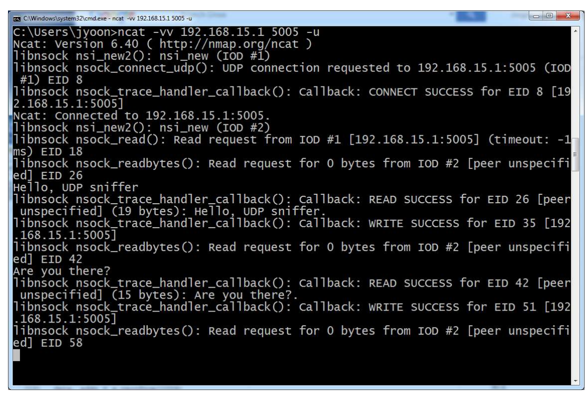
EXERCISE 4.5.3: Write Python scripts, `mySniff2udp.py` to capture UDP. This program can run on Windows.

Script `mySniff2udp.py`

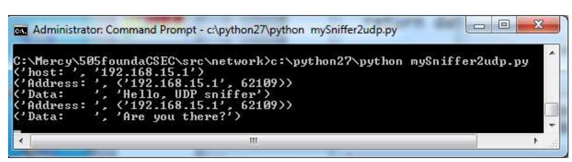
```
(1)  import socket

(2)  UDP_PORT = 5005

(3)  # acquire network interface
(4)  ______________________

(5)  # create an object of a raw socket
(6)  ______________________

(7)  # bind the socket object to the
     network interface obtained above
(8)  ______________________

(9)  # receive a UDP packet
(10) while True:
(11)   data, addr = ____________(1024)
(12)   print ("Address: ", addr)
(13)   print ("Data:    ", data.strip())
```

Sample Run `mySniff2udp.py`

Note that udp can be generated by executing netcat or ncat program on a separate cmd windows.



UDP packets generated by ncat



UDP packet captured by this Python script

Note that the Python script will never stop. It is obvious because the script runs a while with no termination statement. In this case, the Python process should be killed. How? The theory and Python hands-on exercises will be discussed in another section. Luckily, Windows provides the Task Manager, which lets you kill processes manually.

4.5.3 Network Sniffer: Network Packet Interpretation

First understand IP and TCP header structures as follows:

IP Header [http://www.ietf.org/rfc/rfc791.txt]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

TCP Header [http://www.ietf.org/rfc/rfc793.txt]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

UDP Header	[https://tools.ietf.org/html/rfc3828]

```
 0              15 16             31
+--------+--------+--------+--------+
|      Source     |   Destination   |
|       Port      |      Port       |
+--------+--------+--------+--------+
|    Checksum     |                 |
|    Coverage     |    Checksum     |
+--------+--------+--------+--------+
|                                   |
:              Payload              :
|                                   |
+-----------------------------------+
```

As you can see above, the first 4 bits of IP indicate the version which specifies the format of the IP packet header. There are 16 types of IP version: if those 4 bits are 0100, it indicate IP, Internet Protocol for example. For detailed explanation, please visit the website www.networksorcery.com. The next four bits indicate IHL (Internet Header Length). The minimum value for a valid header is 5.

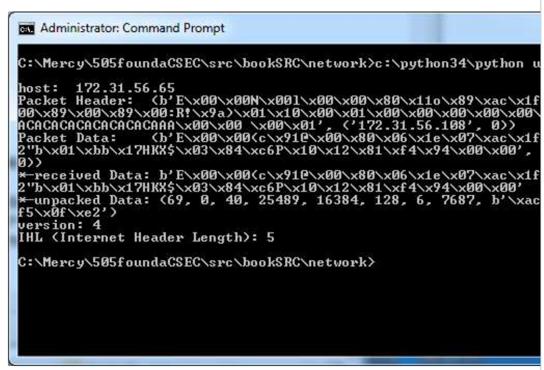
EXAMPLE 4.5.4: Extend EXAMPLE 4.5.2 to write Python script, `unpackPacket1ip.py` to capture IP packets and unpack to read first one byte.

Script `unpackPacket1ip.py`

```
(1)  import socket
(2)  import struct

(3)  # acquire network interface
(4)  HOST =
     socket.gethostbyname(socket.gethostname())
(5)  print ("host: ", HOST)

(6)  # create an object of a raw socket
(7)  s = socket.socket(socket.AF_INET,
     socket.SOCK_RAW, socket.IPPROTO_IP)

(8)  # bind the socket object to the network
     interface obtained above
(9)  s.bind((HOST, 0))

(10) # Include IP headers
(11) s.setsockopt(socket.IPPROTO_IP,
     socket.IP_HDRINCL, 1)

(12) # receive all packages
(13) s.ioctl(socket.SIO_RCVALL,
     socket.RCVALL_ON)

(14) # receive a packet header
```

Sample Run `mySniff2udp.py`

Note that the right end of the screenshot was intended to omit to show the left end display.



```
ipHeader = s.recvfrom(65565)
print ("Packet Header: ", ipHeader)

data = ipHeader[0]
print ("*-received Data:", data)

unpackedData =
    struct.unpack('!BBHHHBBH4s4s', data[:20])
print ("*-unpacked Data:", unpackedData)

version_IHL = unpackedData[0]
version = version_IHL >> 4
IHL = version_IHL & 0xF
print ("version:",version)
print ("IHL (Internet Header Length):",
    IHL)
```

EXERCISE 4.5.4: Write Python scripts, `sniff3.py` to capture the packets and interpret all the IP and TCP header data as shown in the tables above.

EXERCISE 4.5.5: Write Python scripts, `sniff4.py` to continue the script `sniff3.py` in EXERCISE 4.5.3 until stopped manually.

[Hint] multiprocessing or multithreading is needed.

4.6 Accessing Webpages