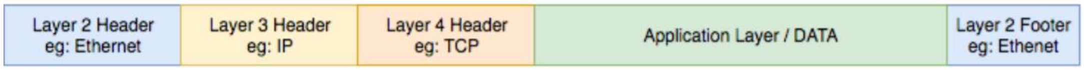
Python Packet Sniffer

How the packet is constructed

The first part of creating the packet sniffer was to fully understand what the packet is composed of in the figure below it shows the different layers/ parts of the packet.



- Layer 2 Ethernet Frame

  The first part of the packet is called an ethernet frame. It contains the source and destination MAC addresses along with the ether type (protocol). The source and destination mac address are both 6 bytes each and the protocol is 2 bytes.

- Layer 3 IP Packet Header

  The second part of the frame is the IP header. It includes the destination and source IP addresses along with the protocol number.

- Layer 4 IP Packet Segment

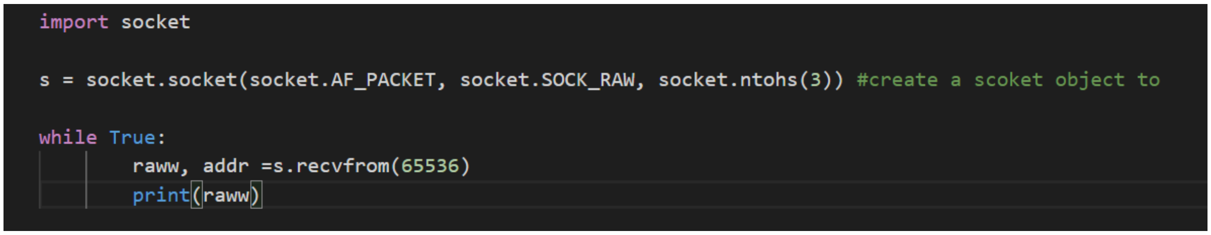
  The third part of the packet is called a segment and it includes all the information about the transport layer. Witch protocol it uses TCP or UDP.

- Application layer

  This includes all other data

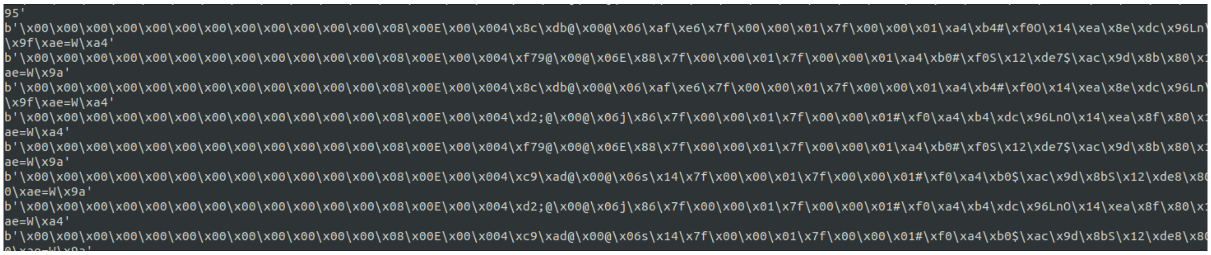
Packet sniffer 0 – Raw Data –

The first thing I did was create a packet sniffer that listens to all traffic and outputs it to the screen.



```
import socket

s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3)) #create a scoket object to

while True:
        raww, addr =s.recvfrom(65536)
        print(raww)
```

The output is as follows.

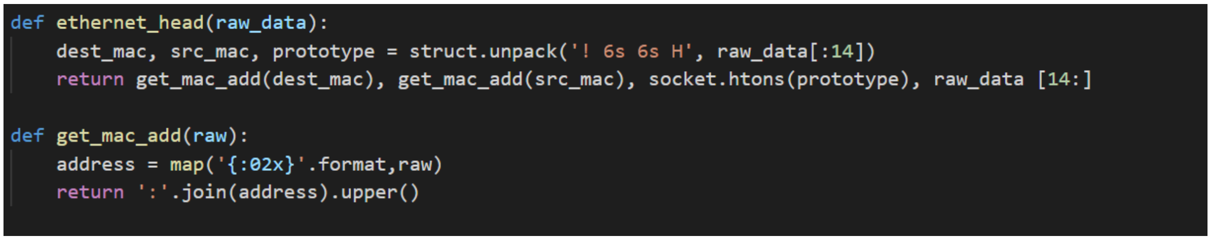


```
95'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\x8c\xdb@\x00@\x06\xaf\xe6\x7f\x00\x00\x01\x7f\x00\x00\x01\xa4\xb4#\xf0O\x14\xea\x8e\xdc\x96Ln
\x9f\xae=W\xa4'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\xf79@\x00@\x06E\x88\x7f\x00\x00\x01\x7f\x00\x00\x01\xa4\xb0#\xf0S\x12\xde7$\xac\x9d\x8b\x80\x1
ae=W\x9a'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\x8c\xdb@\x00@\x06\xaf\xe6\x7f\x00\x00\x01\x7f\x00\x00\x01\xa4\xb4#\xf0O\x14\xea\x8e\xdc\x96Ln
\x9f\xae=W\xa4'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\xd2;@\x00@\x06j\x86\x7f\x00\x00\x01\x7f\x00\x00\x01#\xf0\xa4\xb4\xdc\x96LnO\x14\xea\x8f\x80\x1
ae=W\xa4'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\xf79@\x00@\x06E\x88\x7f\x00\x00\x01\x7f\x00\x00\x01\xa4\xb0#\xf0S\x12\xde7$\xac\x9d\x8b\x80\x1
ae=W\x9a'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\xc9\xad@\x00@\x06s\x14\x7f\x00\x00\x01\x7f\x00\x00\x01#\xf0\xa4\xb0$\xac\x9d\x8bS\x12\xde8\x80
0\xae=W\x9a'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\xd2;@\x00@\x06j\x86\x7f\x00\x00\x01\x7f\x00\x00\x01#\xf0\xa4\xb4\xdc\x96LnO\x14\xea\x8f\x80\x1
ae=W\xa4'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00E\x00\x004\xc9\xad@\x00@\x06s\x14\x7f\x00\x00\x01\x7f\x00\x00\x01#\xf0\xa4\xb0$\xac\x9d\x8bS\x12\xde8\x80
0\xae=W\x9a'
```

As shown in the figure above the output is not understandable Because it is in raw form. To understand the output, I need to parse this data.

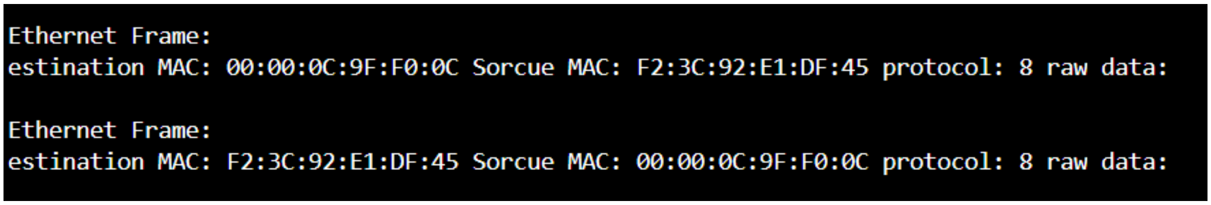
Packet sniffer 1 – Ethernet Header –

After being able to sniff the packets, I needed to parse the output. I parsed the layer 2 header (ethernet frame) by using 2 main functions. The first function (ethernet_head) is a function that takes in raw data. Unpacks the first 14 bytes from the raw data and returns the destination MAC address (6 bytes), source MAC address (6 bytes), and the protocol (2 bytes). The second function is used to format, capitalize, and add colons to the MAC address.



```
def ethernet_head(raw_data):
    dest_mac, src_mac, prototype = struct.unpack('! 6s 6s H', raw_data[:14])
    return get_mac_add(dest_mac), get_mac_add(src_mac), socket.htons(prototype), raw_data [14:]

def get_mac_add(raw):
    address = map('{:02x}'.format,raw)
    return ':'.join(address).upper()
```

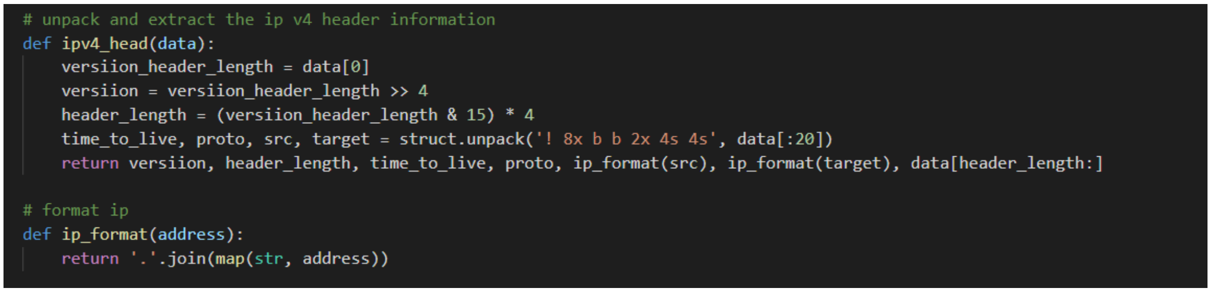
The output of this program is as follows:



```
Ethernet Frame:
estination MAC: 00:00:0C:9F:F0:0C Sorcue MAC: F2:3C:92:E1:DF:45 protocol: 8 raw data:

Ethernet Frame:
estination MAC: F2:3C:92:E1:DF:45 Sorcue MAC: 00:00:0C:9F:F0:0C protocol: 8 raw data:
```

Packet sniffer 2 – IP Header –

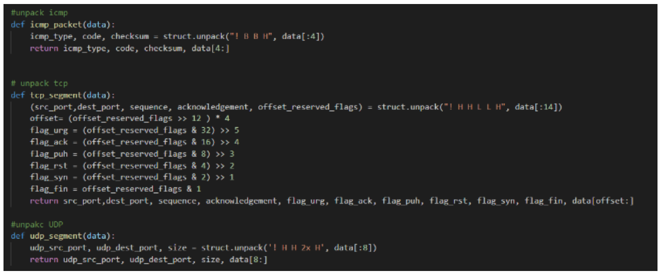
I then parsed the IP header to do that I created 2 new functions. The first function (ipv4_head), takes in data and extracts the version, header length, source IP, destination IP and returns the rest of the data. And the second function formats the Ip properly and joins it with “.”.



```
# unpack and extract the ip v4 header information
def ipv4_head(data):
    versiion_header_length = data[0]
    versiion = versiion_header_length >> 4
    header_length = (versiion_header_length & 15) * 4
    time_to_live, proto, src, target = struct.unpack('! 8x b b 2x 4s 4s', data[:20])
    return versiion, header_length, time_to_live, proto, ip_format(src), ip_format(target), data[header_length:]

# format ip
def ip_format(address):
    return '.'.join(map(str, address))
```

Packet sniffer 3 – IP Segment –

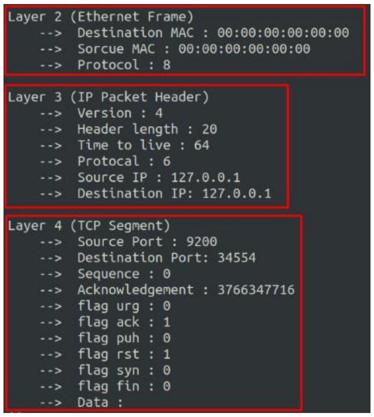
I was able to extract the segments from the packet by first knowing the protocol from the packet header. I created 3 functions for this TCP, UDP, and ICMP. And to finalize this program I also edited the main function to display the new unpacked data properly.



```
#unpack icmp
def icmp_packet(data):
    icmp_type, code, checksum = struct.unpack("! B B H", data[:4])
    return icmp_type, code, checksum, data[4:]


# unpack tcp
def tcp_segment(data):
    (src_port,dest_port, sequence, acknowledgement, offset_reserved_flags) = struct.unpack("! H H L L H", data[:14])
    offset= (offset_reserved_flags >> 12 ) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_puh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1
    return src_port,dest_port, sequence, acknowledgement, flag_urg, flag_ack, flag_puh, flag_rst, flag_syn, flag_fin, data[offset:]

#unpakc UDP
def udp_segment(data):
    udp_src_port, udp_dest_port, size = struct.unpack('! H H 2x H', data[:8])
    return udp_src_port, udp_dest_port, size, data[8:]
```

How To Run the Program

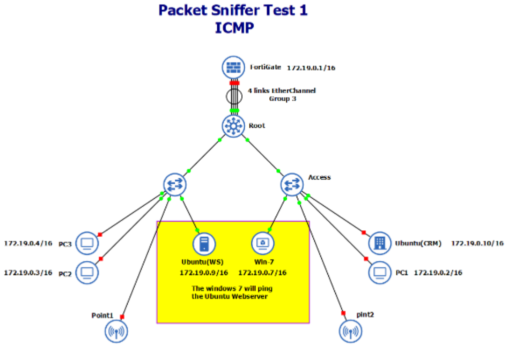
Note, this program is only available for Linux because windows does not support raw socket capturing. To run the program on Linux. Type # python3 Packet-sniffer-3.py the program will start capturing traffic and displaying it on the terminal. The figure below shows a sample of the output.



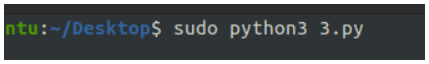
```
Layer 2 (Ethernet Frame)
    -->  Destination MAC : 00:00:00:00:00:00
    -->  Sorcue MAC : 00:00:00:00:00:00
    -->  Protocol : 8

Layer 3 (IP Packet Header)
    -->  Version : 4
    -->  Header length : 20
    -->  Time to live : 64
    -->  Protocal : 6
    -->  Source IP : 127.0.0.1
    -->  Destination IP: 127.0.0.1

Layer 4 (TCP Segment)
    -->  Source Port : 9200
    -->  Destination Port: 34554
    -->  Sequence : 0
    -->  Acknowledgement : 3766347716
    -->  flag urg : 0
    -->  flag ack : 1
    -->  flag puh : 0
    -->  flag rst : 1
    -->  flag syn : 0
    -->  flag fin : 0
    -->  Data :
```

Test 1 ICMP

Since I used GNS3 in part one of the course work, I had an actual Linux server, and a windows 7 machine ruing on VM-ware. The first test I conducted was an ICMP test. Pinging the server from the windows 7 machine.

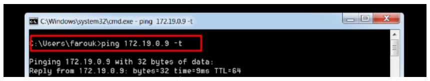


Step 1: I coped my python program to the ubuntu webhosting machine and started the program.



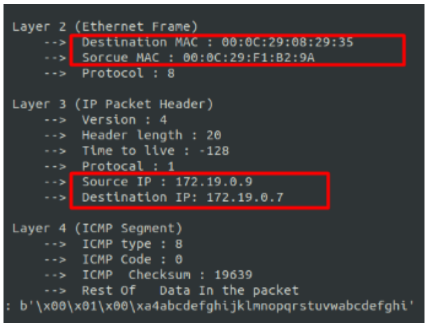
```
ntu:~/Desktop$ sudo python3 3.py
```

Step 2: started pinging the server. To do this I opened CMD on the windows 7 machine and wrote the command ping 172.19.0.9 (the Linux server IP)



```
:\Users\farouk>ping 172.19.0.9 -t

Pinging 172.19.0.9 with 32 bytes of data:
Reply from 172.19.0.9: bytes=32 time=9ms TTL=64
```

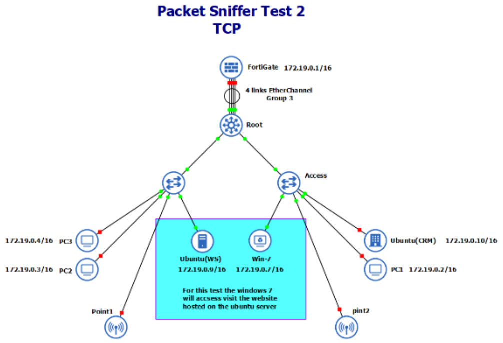
Step 3: Now I went back to the server's terminal and investigated the output. Every packet consisted of 3 main parts layer 2, layer 3, and layer 4 explained above. Both the source and the destination MAC addresses are visible in the layer 2 section. Both the source and destination IP addresses are visible in the layer 3 section. Extra information about the ICMP packet is visible in the layer 4 section

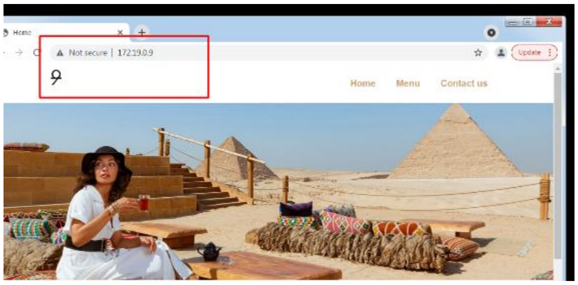


```
Layer 2 (Ethernet Frame)
    -->  Destination MAC : 00:0C:29:08:29:35
    -->  Sorcue MAC : 00:0C:29:F1:B2:9A
    -->  Protocol : 8

Layer 3 (IP Packet Header)
    -->  Version : 4
    -->  Header length : 20
    -->  Time to live : -128
    -->  Protocal : 1
    -->  Source IP : 172.19.0.9
    -->  Destination IP: 172.19.0.7

Layer 4 (ICMP Segment)
    -->  ICMP type : 8
    -->  ICMP Code : 0
    -->  ICMP  Checksum : 19639
    -->  Rest Of   Data In the packet
: b'\x00\x01\x00\xa4abcdefghijklmnopqrstuvwabcdefghi'
```

Test 2 TCP and HHTP

In this test I accessed the website hosted on the Linux webserver from the windows 7 machine and investigated the traffic.

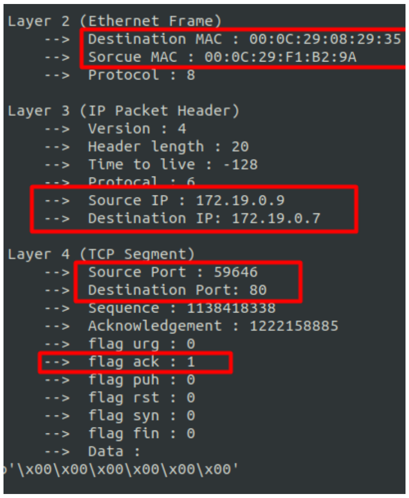




Step 1: I coped my python program to the ubuntu webhosting machine and started the program.

Step 2: Visited the website hosted on the webserver. To do this I opened chrome and went to the servers IP address (172.19.0.9) and generated some more traffic by looking through the website.

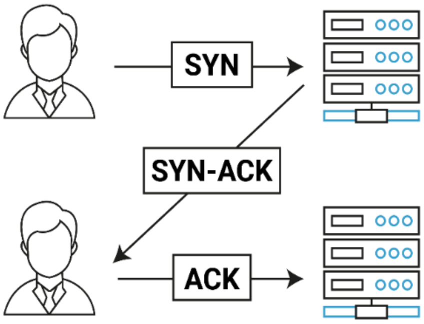
Step 3: Like the example above, I went back to the server's terminal to analyze the packets captured. Layer 2 both included the source, and the destination MAC addresses. Layer 3 included both the source and the destination IP addresses. Layer 4 included both the Source and the destination port along with all the flags for the 3-way handshake. And since this is regular http request the port had to be 80.



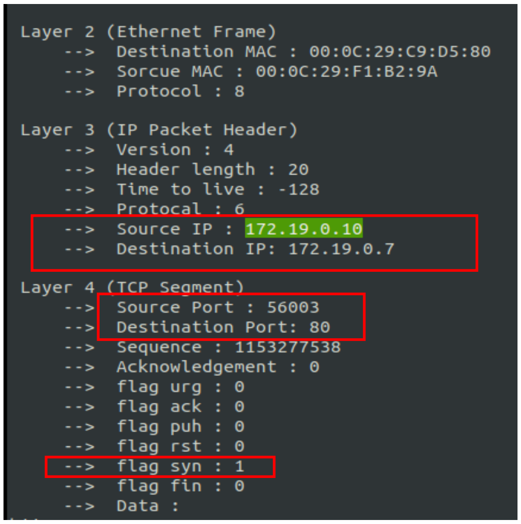
```
Layer 2 (Ethernet Frame)
    -->  Destination MAC : 00:0C:29:08:29:35
    -->  Sorcue MAC : 00:0C:29:F1:B2:9A
    -->  Protocol : 8

Layer 3 (IP Packet Header)
    -->  Version : 4
    -->  Header length : 20
    -->  Time to live : -128
    -->  Protocal : 6
    -->  Source IP : 172.19.0.9
    -->  Destination IP: 172.19.0.7

Layer 4 (TCP Segment)
    -->  Source Port : 59646
    -->  Destination Port: 80
    -->  Sequence : 1138418338
    -->  Acknowledgement : 1222158885
    -->  flag urg : 0
    -->  flag ack : 1
    -->  flag puh : 0
    -->  flag rst : 0
    -->  flag syn : 0
    -->  flag fin : 0
    -->  Data :
'\x00\x00\x00\x00\x00\x00'
```

Test 3 Wireshark VS My program

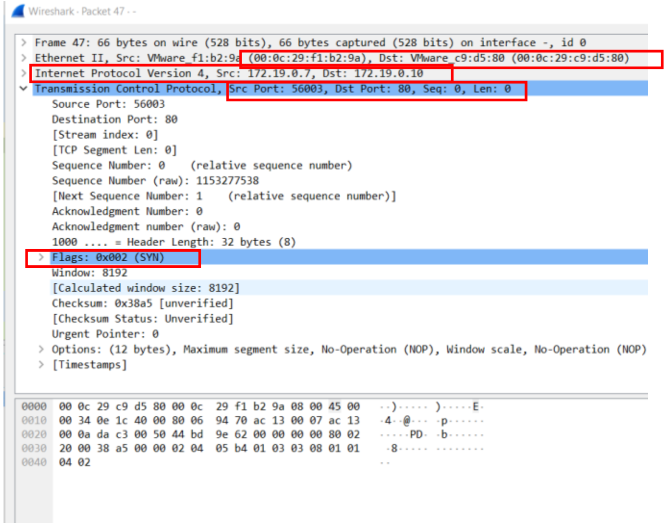
To test the accuracy of my program, I wanted to compare the packets captured from my program to the packets captured by wire shark. To do this I ran both Wireshark and my program at the same time to capture, analyze, and compare a three-way handshake. A three-way handshake is primarily used to create a TCP socket connection to reliably transmit data between devices. it contains 3 main steps.



Step 1: client sends a SYN message. As shown in the figures below I was able to capture the SYN packet and comparing my program with Wireshark I can see that everything is the same. All the ports, IP addresses, flags, and MAC addresses are correct.



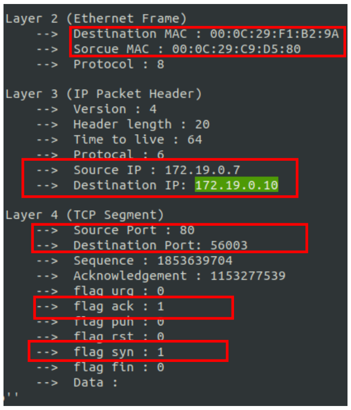
```
Layer 2 (Ethernet Frame)
    -->  Destination MAC : 00:0C:29:C9:D5:80
    -->  Sorcue MAC : 00:0C:29:F1:B2:9A
    -->  Protocol : 8

Layer 3 (IP Packet Header)
    -->  Version : 4
    -->  Header length : 20
    -->  Time to live : -128
    -->  Protocal : 6
    -->  Source IP : 172.19.0.10
    -->  Destination IP: 172.19.0.7

Layer 4 (TCP Segment)
    -->  Source Port : 56003
    -->  Destination Port: 80
    -->  Sequence : 1153277538
    -->  Acknowledgement : 0
    -->  flag urg : 0
    -->  flag ack : 0
    -->  flag puh : 0
    -->  flag rst : 0
    -->  flag syn : 1
    -->  flag fin : 0
    -->  Data :
```



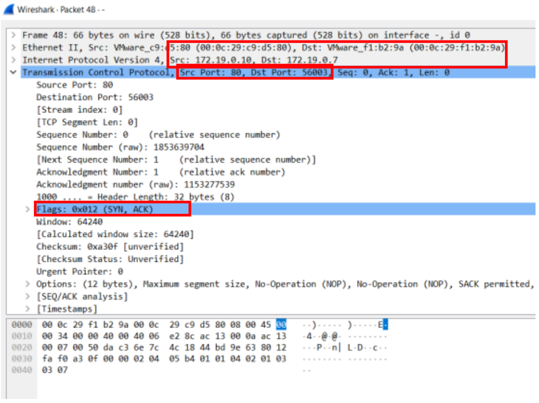
Wireshark · Packet 47 · -

```
> Frame 47: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface -, id 0
> Ethernet II, Src: VMware_f1:b2:9a (00:0c:29:f1:b2:9a), Dst: VMware_c9:d5:80 (00:0c:29:c9:d5:80)
> Internet Protocol Version 4, Src: 172.19.0.7, Dst: 172.19.0.10
v Transmission Control Protocol, Src Port: 56003, Dst Port: 80, Seq: 0, Len: 0
     Source Port: 56003
     Destination Port: 80
     [Stream index: 0]
     [TCP Segment Len: 0]
     Sequence Number: 0    (relative sequence number)
     Sequence Number (raw): 1153277538
     [Next Sequence Number: 1    (relative sequence number)]
     Acknowledgment Number: 0
     Acknowledgment number (raw): 0
     1000 .... = Header Length: 32 bytes (8)
   > Flags: 0x002 (SYN)
     Window: 8192
     [Calculated window size: 8192]
     Checksum: 0x38a5 [unverified]
     [Checksum Status: Unverified]
     Urgent Pointer: 0
   > Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP)
   > [Timestamps]

0000  00 0c 29 c9 d5 80 00 0c  29 f1 b2 9a 08 00 45 00   ··)····· )·····E·
0010  00 34 0e 1c 40 00 80 06  94 70 ac 13 00 07 ac 13   ·4··@··· ·p······
0020  00 0a da c3 00 50 44 bd  9e 62 00 00 00 00 80 02   ·····PD· ·b······
0030  20 00 38 a5 00 00 02 04  05 b4 01 03 03 08 01 01    ·8····· ········
0040  04 02                                               ··
```

Step 2: server responds with a SYN-ACK message. (Everything is correct).



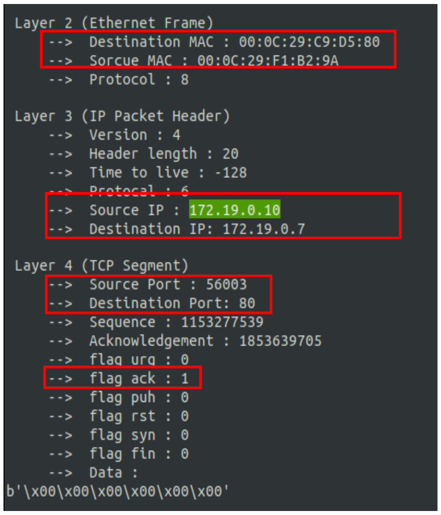
```
Layer 2 (Ethernet Frame)
    -->  Destination MAC : 00:0C:29:F1:B2:9A
    -->  Sorcue MAC : 00:0C:29:C9:D5:80
    -->  Protocol : 8

Layer 3 (IP Packet Header)
    -->  Version : 4
    -->  Header length : 20
    -->  Time to live : 64
    -->  Protocal : 6
    -->  Source IP : 172.19.0.7
    -->  Destination IP: 172.19.0.10

Layer 4 (TCP Segment)
    -->  Source Port : 80
    -->  Destination Port: 56003
    -->  Sequence : 1853639704
    -->  Acknowledgement : 1153277539
    -->  flag urg : 0
    -->  flag ack : 1
    -->  flag puh : 0
    -->  flag rst : 0
    -->  flag syn : 1
    -->  flag fin : 0
    -->  Data :
''
```



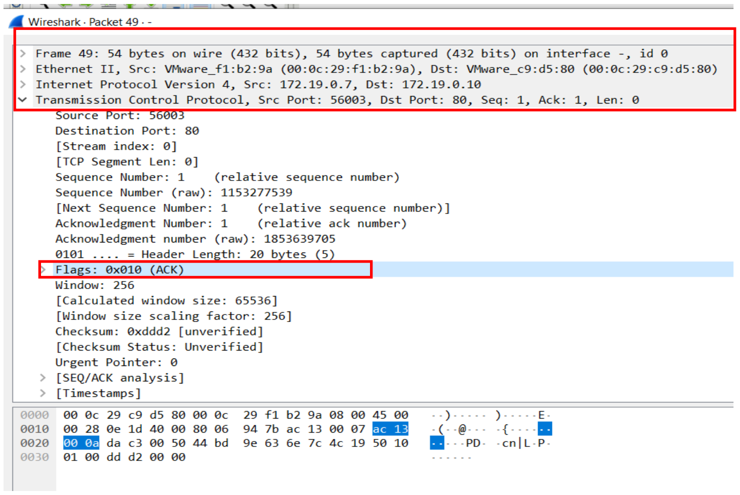
Wireshark · Packet 48 · -

```
> Frame 48: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface -, id 0
> Ethernet II, Src: VMware_c9:d5:80 (00:0c:29:c9:d5:80), Dst: VMware_f1:b2:9a (00:0c:29:f1:b2:9a)
> Internet Protocol Version 4, Src: 172.19.0.10, Dst: 172.19.0.7
v Transmission Control Protocol, Src Port: 80, Dst Port: 56003, Seq: 0, Ack: 1, Len: 0
     Source Port: 80
     Destination Port: 56003
     [Stream index: 0]
     [TCP Segment Len: 0]
     Sequence Number: 0    (relative sequence number)
     Sequence Number (raw): 1853639704
     [Next Sequence Number: 1    (relative sequence number)]
     Acknowledgment Number: 1    (relative ack number)
     Acknowledgment number (raw): 1153277539
     1000 .... = Header Length: 32 bytes (8)
   > Flags: 0x012 (SYN, ACK)
     Window: 64240
     [Calculated window size: 64240]
     Checksum: 0xa30f [unverified]
     [Checksum Status: Unverified]
     Urgent Pointer: 0
   > Options: (12 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted,
   > [SEQ/ACK analysis]
   > [Timestamps]

0000  00 0c 29 f1 b2 9a 00 0c  29 c9 d5 80 08 00 45 00   ··)····· )·····E·
0010  00 34 00 00 40 00 40 06  e2 8c ac 13 00 0a ac 13   ·4··@·@· ········
0020  00 07 00 50 da c3 6e 7c  4c 18 44 bd 9e 63 80 12   ···P··n| L·D··c··
0030  fa f0 a3 0f 00 00 02 04  05 b4 01 01 04 02 01 03   ········ ········
0040  03 07                                               ··
```

Step 3: client responds with ACK message. (Everything is correct).



```
Layer 2 (Ethernet Frame)
    -->  Destination MAC : 00:0C:29:C9:D5:80
    -->  Sorcue MAC : 00:0C:29:F1:B2:9A
    -->  Protocol : 8

Layer 3 (IP Packet Header)
    -->  Version : 4
    -->  Header length : 20
    -->  Time to live : -128
    -->  Protocal : 6
    -->  Source IP : 172.19.0.10
    -->  Destination IP: 172.19.0.7

Layer 4 (TCP Segment)
    -->  Source Port : 56003
    -->  Destination Port: 80
    -->  Sequence : 1153277539
    -->  Acknowledgement : 1853639705
    -->  flag urg : 0
    -->  flag ack : 1
    -->  flag puh : 0
    -->  flag rst : 0
    -->  flag syn : 0
    -->  flag fin : 0
    -->  Data :
b'\x00\x00\x00\x00\x00\x00'
```



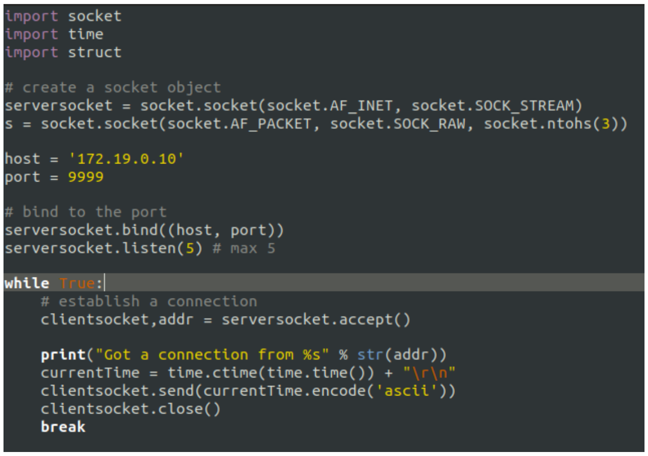
Wireshark · Packet 49 · -

```
> Frame 49: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface -, id 0
> Ethernet II, Src: VMware_f1:b2:9a (00:0c:29:f1:b2:9a), Dst: VMware_c9:d5:80 (00:0c:29:c9:d5:80)
> Internet Protocol Version 4, Src: 172.19.0.7, Dst: 172.19.0.10
v Transmission Control Protocol, Src Port: 56003, Dst Port: 80, Seq: 1, Ack: 1, Len: 0
     Source Port: 56003
     Destination Port: 80
     [Stream index: 0]
     [TCP Segment Len: 0]
     Sequence Number: 1    (relative sequence number)
     Sequence Number (raw): 1153277539
     [Next Sequence Number: 1    (relative sequence number)]
     Acknowledgment Number: 1    (relative ack number)
     Acknowledgment number (raw): 1853639705
     0101 .... = Header Length: 20 bytes (5)
   > Flags: 0x010 (ACK)
     Window: 256
     [Calculated window size: 65536]
     [Window size scaling factor: 256]
     Checksum: 0xddd2 [unverified]
     [Checksum Status: Unverified]
     Urgent Pointer: 0
   > [SEQ/ACK analysis]
   > [Timestamps]

0000  00 0c 29 c9 d5 80 00 0c  29 f1 b2 9a 08 00 45 00   ··)····· )·····E·
0010  00 28 0e 1d 40 00 80 06  94 7b ac 13 00 07 ac 13   ·(··@··· ·{······
0020  00 0a da c3 00 50 44 bd  9e 63 6e 7c 4c 19 50 10   ·····PD· ·cn|L·P·
0030  01 00 dd d2 00 00                                   ······
```

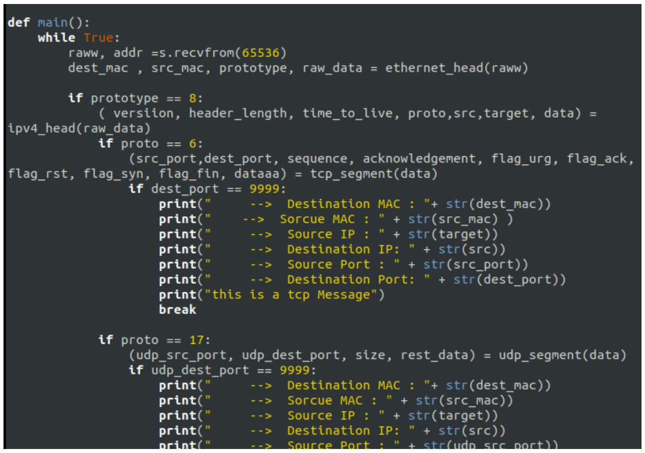
Client and server packet sniffer

Besides the normal packet sniffer, we were asked to create a client and server with socket programming in python. The client connects to the server then the server replies to the client with a message and on the server's terminal it displays the clients MAC address, IP address, and ports. To do this I had to create 2 files one for the client and another for the server.

Server program: The server program contains 2 main parts part one which is responsible for sending the server's current time to the client, and part two which is responsible for sniffing the client's information. (Explained above)

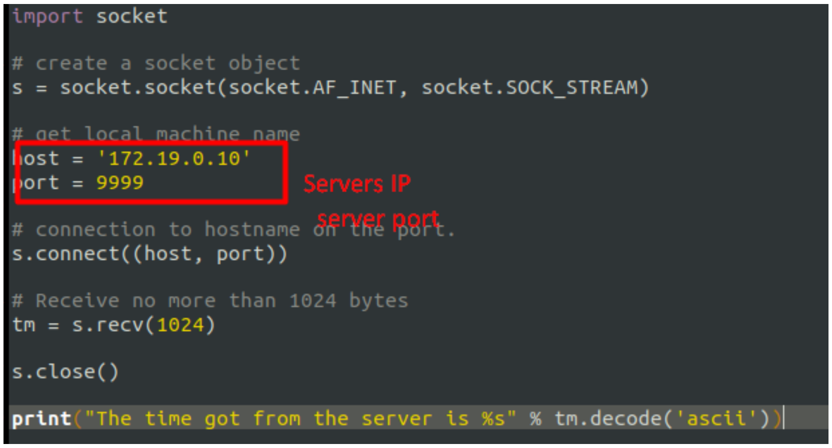


```python
import socket
import time
import struct

# create a socket object
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

host = '172.19.0.10'
port = 9999

# bind to the port
serversocket.bind((host, port))
serversocket.listen(5) # max 5

while True:
    # establish a connection
    clientsocket,addr = serversocket.accept()

    print("Got a connection from %s" % str(addr))
    currentTime = time.ctime(time.time()) + "\r\n"
    clientsocket.send(currentTime.encode('ascii'))
    clientsocket.close()
    break
```



```python
def main():
    while True:
        raww, addr =s.recvfrom(65536)
        dest_mac , src_mac, prototype, raw_data = ethernet_head(raww)

        if prototype == 8:
            ( versiion, header_length, time_to_live, proto,src,target, data) =
ipv4_head(raw_data)
            if proto == 6:
                (src_port,dest_port, sequence, acknowledgement, flag_urg, flag_ack,
flag_rst, flag_syn, flag_fin, dataaa) = tcp_segment(data)
                if dest_port == 9999:
                    print("      -->  Destination MAC : "+ str(dest_mac))
                    print("     -->  Sorcue MAC : " + str(src_mac) )
                    print("      -->  Source IP : " + str(target))
                    print("      -->  Destination IP: " + str(src))
                    print("      -->  Source Port : " + str(src_port))
                    print("      -->  Destination Port: " + str(dest_port))
                    print("this is a tcp Message")
                    break

            if proto == 17:
                (udp_src_port, udp_dest_port, size, rest_data) = udp_segment(data)
                if udp_dest_port == 9999:
                    print("      -->  Destination MAC : "+ str(dest_mac))
                    print("      -->  Sorcue MAC : " + str(src_mac))
                    print("      -->  Source IP : " + str(target))
                    print("      -->  Destination IP: " + str(src))
                    print("      -->  Source Port : " + str(udp_src_port))
```

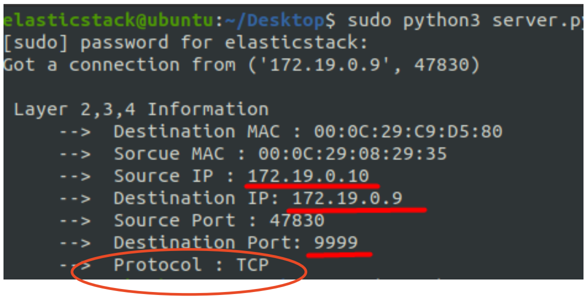
Client program: The client file is very simple. First, I imported the sockets library created a socket object and connected it to the servers IP and port.



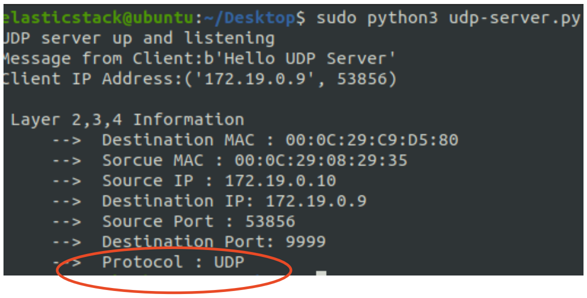
```python
import socket

# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = '172.19.0.10'
port = 9999

# connection to hostname on the port.
s.connect((host, port))

# Receive no more than 1024 bytes
tm = s.recv(1024)

s.close()

print("The time got from the server is %s" % tm.decode('ascii'))
```

Note: The server and client files must be on different machines to display MAC addresses. If the server and the client are on the machine (localhost) the MAC addresses will be 00:00:00:000. This is because sending and receiving from the local host does not need a MAC address.

Server output: TCP/UDP

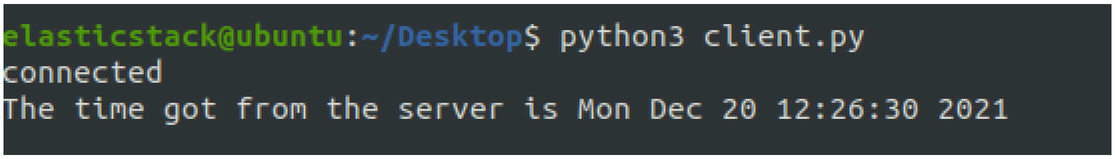


```
elasticstack@ubuntu:~/Desktop$ sudo python3 server.p
[sudo] password for elasticstack:
Got a connection from ('172.19.0.9', 47830)

 Layer 2,3,4 Information
     -->  Destination MAC : 00:0C:29:C9:D5:80
     -->  Sorcue MAC : 00:0C:29:08:29:35
     -->  Source IP : 172.19.0.10
     -->  Destination IP: 172.19.0.9
     -->  Source Port : 47830
     -->  Destination Port: 9999
     -->  Protocol : TCP
```

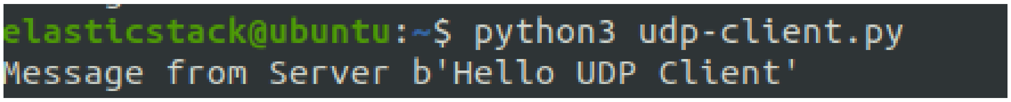


```
elasticstack@ubuntu:~/Desktop$ sudo python3 udp-server.py
UDP server up and listening
Message from Client:b'Hello UDP Server'
Client IP Address:('172.19.0.9', 53856)

 Layer 2,3,4 Information
     -->  Destination MAC : 00:0C:29:C9:D5:80
     -->  Sorcue MAC : 00:0C:29:08:29:35
     -->  Source IP : 172.19.0.10
     -->  Destination IP: 172.19.0.9
     -->  Source Port : 53856
     -->  Destination Port: 9999
     -->  Protocol : UDP
```

Client output: TCP/UDP



```
elasticstack@ubuntu:~/Desktop$ python3 client.py
connected
The time got from the server is Mon Dec 20 12:26:30 2021
```



```
elasticstack@ubuntu:~$ python3 udp-client.py
Message from Server b'Hello UDP Client'
```

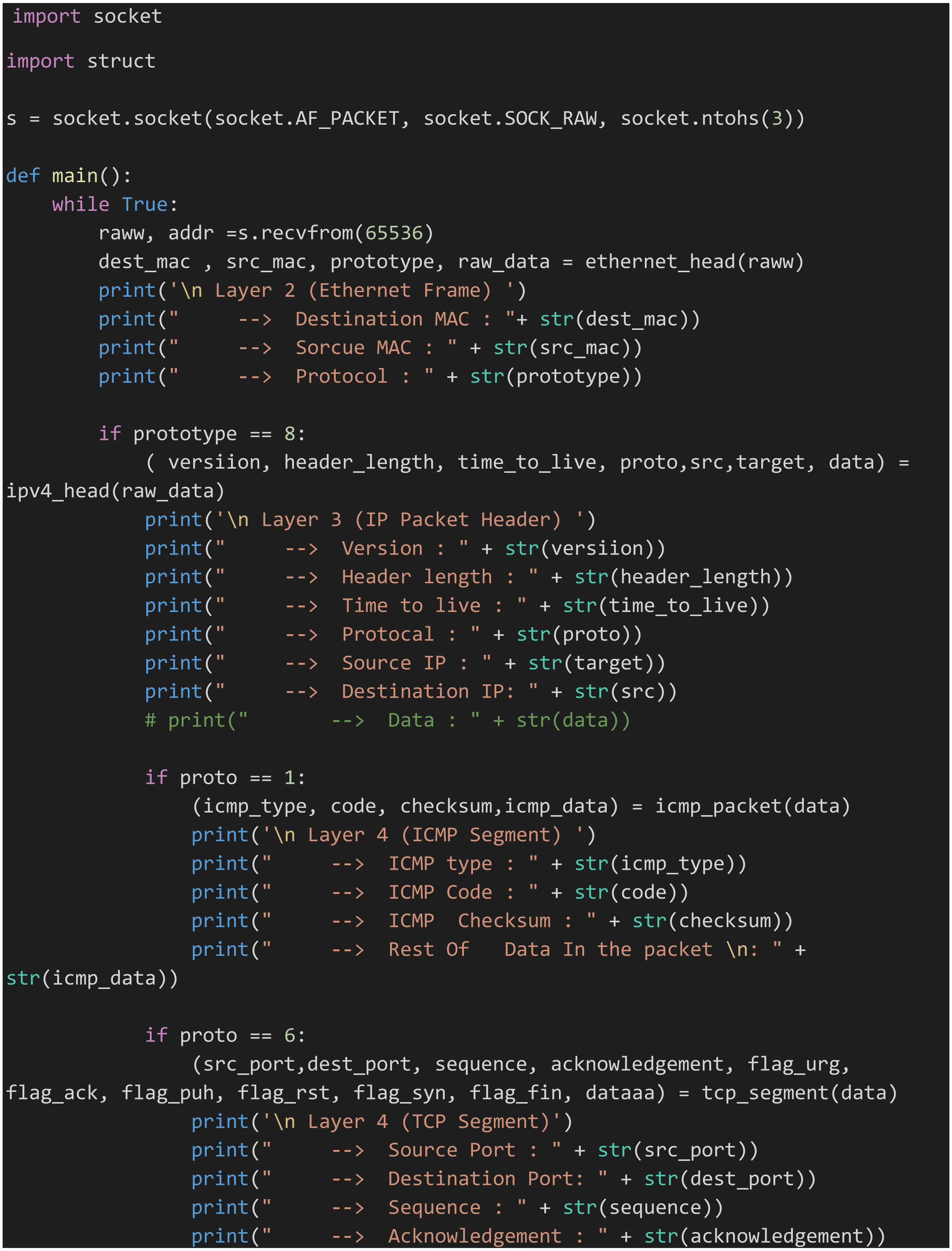
Git-Hub Repo

Inside the git hub repository there are a total of 6 files.

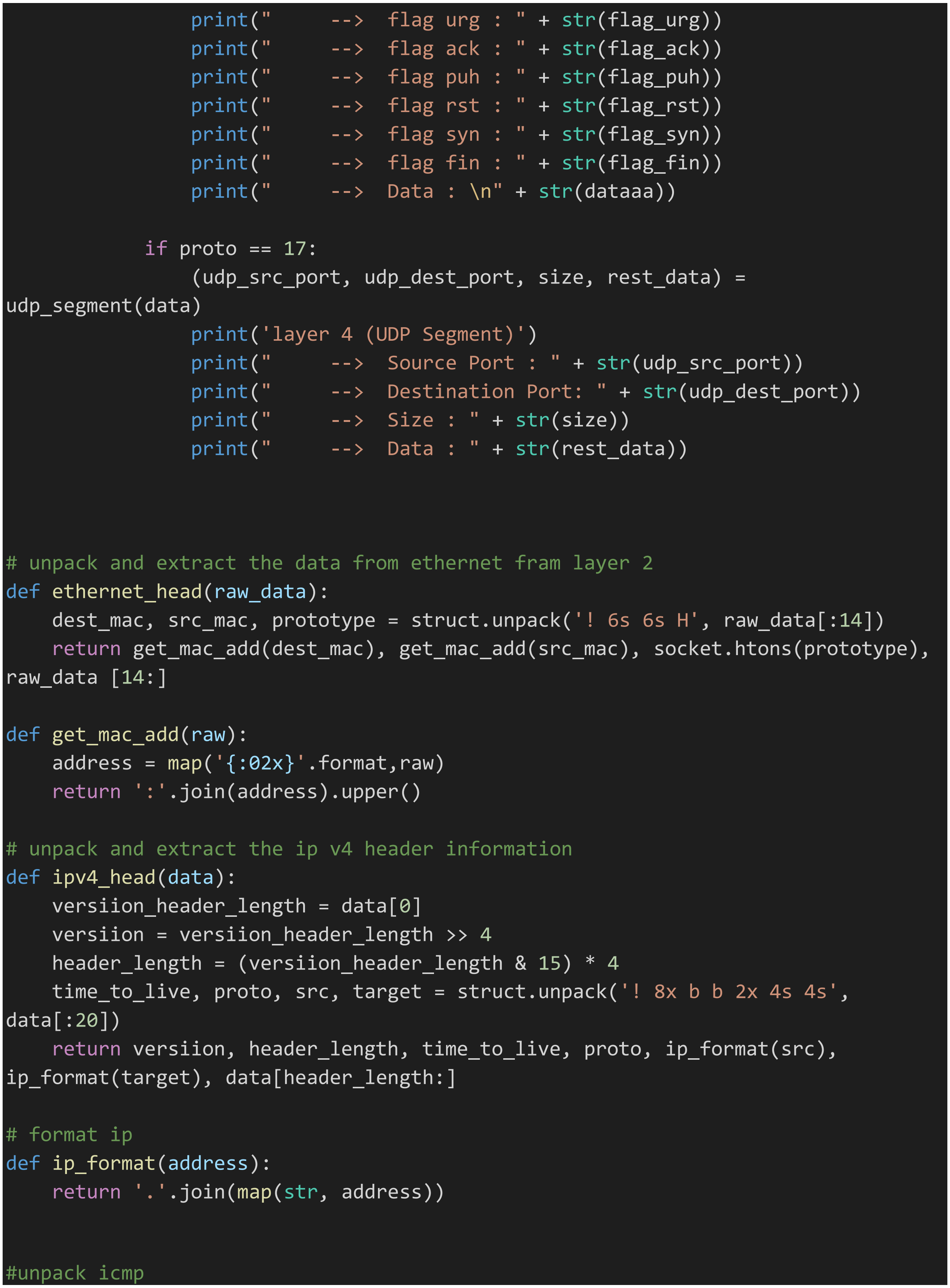
1. packet-sniffer-0-Raw-Data.py → packet sniffer that sniffs raw data
2. packet-sniffer-1-Ethernet-Header.py → packet sniffer that sniffs the layer 2 information only
3. packet-sniffer-2-IP-Header.py → packet sniffer that sniffs the layer 2 and 3 information only
4. packet-sniffer-3-IP-Segment.py → packet sniffer that sniffs the layer 2, 3, and 4 information
5. client.py → the client file
6. Server that sniffs client information.py → the server file

Git-hub link: https://github.com/ahmedfarou22/Packet-Sniffer

Packet-sniffer-3 source code



```
 import socket

import struct

s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

def main():
    while True:
        raww, addr =s.recvfrom(65536)
        dest_mac , src_mac, prototype, raw_data = ethernet_head(raww)
        print('\n Layer 2 (Ethernet Frame) ')
        print("     -->  Destination MAC : "+ str(dest_mac))
        print("     -->  Sorcue MAC : " + str(src_mac))
        print("     -->  Protocol : " + str(prototype))

        if prototype == 8:
            ( versiion, header_length, time_to_live, proto,src,target, data) =
ipv4_head(raw_data)
            print('\n Layer 3 (IP Packet Header) ')
            print("     -->  Version : " + str(versiion))
            print("     -->  Header length : " + str(header_length))
            print("     -->  Time to live : " + str(time_to_live))
            print("     -->  Protocal : " + str(proto))
            print("     -->  Source IP : " + str(target))
            print("     -->  Destination IP: " + str(src))
            # print("       -->  Data : " + str(data))

            if proto == 1:
                (icmp_type, code, checksum,icmp_data) = icmp_packet(data)
                print('\n Layer 4 (ICMP Segment) ')
                print("     -->  ICMP type : " + str(icmp_type))
                print("     -->  ICMP Code : " + str(code))
                print("     -->  ICMP  Checksum : " + str(checksum))
                print("     -->  Rest Of   Data In the packet \n: " +
str(icmp_data))

            if proto == 6:
                (src_port,dest_port, sequence, acknowledgement, flag_urg,
flag_ack, flag_puh, flag_rst, flag_syn, flag_fin, dataaa) = tcp_segment(data)
                print('\n Layer 4 (TCP Segment)')
                print("     -->  Source Port : " + str(src_port))
                print("     -->  Destination Port: " + str(dest_port))
                print("     -->  Sequence : " + str(sequence))
                print("     -->  Acknowledgement : " + str(acknowledgement))
```

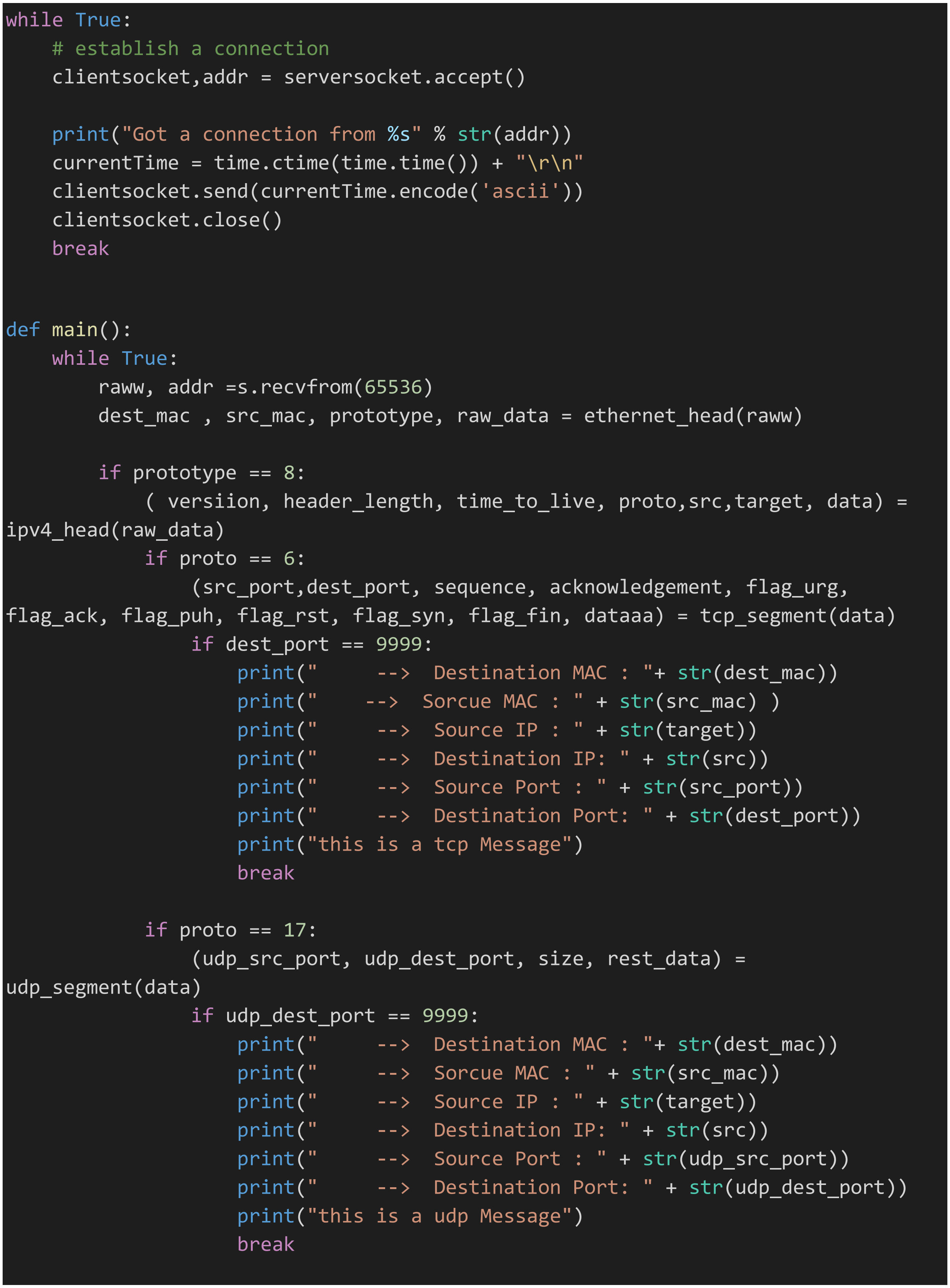


```
                print("     -->  flag urg : " + str(flag_urg))
                print("     -->  flag ack : " + str(flag_ack))
                print("     -->  flag puh : " + str(flag_puh))
                print("     -->  flag rst : " + str(flag_rst))
                print("     -->  flag syn : " + str(flag_syn))
                print("     -->  flag fin : " + str(flag_fin))
                print("     -->  Data : \n" + str(dataaa))

            if proto == 17:
                (udp_src_port, udp_dest_port, size, rest_data) =
udp_segment(data)
                print('layer 4 (UDP Segment)')
                print("     -->  Source Port : " + str(udp_src_port))
                print("     -->  Destination Port: " + str(udp_dest_port))
                print("     -->  Size : " + str(size))
                print("     -->  Data : " + str(rest_data))



# unpack and extract the data from ethernet fram layer 2
def ethernet_head(raw_data):
    dest_mac, src_mac, prototype = struct.unpack('! 6s 6s H', raw_data[:14])
    return get_mac_add(dest_mac), get_mac_add(src_mac), socket.htons(prototype),
raw_data [14:]

def get_mac_add(raw):
    address = map('{:02x}'.format,raw)
    return ':'.join(address).upper()

# unpack and extract the ip v4 header information
def ipv4_head(data):
    versiion_header_length = data[0]
    versiion = versiion_header_length >> 4
    header_length = (versiion_header_length & 15) * 4
    time_to_live, proto, src, target = struct.unpack('! 8x b b 2x 4s 4s',
data[:20])
    return versiion, header_length, time_to_live, proto, ip_format(src),
ip_format(target), data[header_length:]

# format ip
def ip_format(address):
    return '.'.join(map(str, address))


#unpack icmp
```

```
def icmp_packet(data):
    icmp_type, code, checksum = struct.unpack("! B B H", data[:4])
    return icmp_type, code, checksum, data[4:]


# unpack tcp
def tcp_segment(data):
    (src_port,dest_port, sequence, acknowledgement, offset_reserved_flags) =
struct.unpack("! H H L L H", data[:14])
    offset= (offset_reserved_flags >> 12 ) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_puh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1
    return src_port,dest_port, sequence, acknowledgement, flag_urg, flag_ack,
flag_puh, flag_rst, flag_syn, flag_fin, data[offset:]

#unpakc UDP
def udp_segment(data):
    udp_src_port, udp_dest_port, size = struct.unpack('! H H 2x H', data[:8])
    return udp_src_port, udp_dest_port, size, data[8:]
main()
```

Server File Source Code

```
import socket
import time
import struct

# create a socket object
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

# get local machine name
host = '172.19.0.10'
port = 9999

# bind to the port
serversocket.bind((host, port))
serversocket.listen(5) # max 5
```



```
while True:
    # establish a connection
    clientsocket,addr = serversocket.accept()

    print("Got a connection from %s" % str(addr))
    currentTime = time.ctime(time.time()) + "\r\n"
    clientsocket.send(currentTime.encode('ascii'))
    clientsocket.close()
    break


def main():
    while True:
        raww, addr =s.recvfrom(65536)
        dest_mac , src_mac, prototype, raw_data = ethernet_head(raww)

        if prototype == 8:
            ( versiion, header_length, time_to_live, proto,src,target, data) =
ipv4_head(raw_data)
            if proto == 6:
                (src_port,dest_port, sequence, acknowledgement, flag_urg,
flag_ack, flag_puh, flag_rst, flag_syn, flag_fin, dataaa) = tcp_segment(data)
                if dest_port == 9999:
                    print("     -->  Destination MAC : "+ str(dest_mac))
                    print("    -->  Sorcue MAC : " + str(src_mac) )
                    print("     -->  Source IP : " + str(target))
                    print("     -->  Destination IP: " + str(src))
                    print("     -->  Source Port : " + str(src_port))
                    print("     -->  Destination Port: " + str(dest_port))
                    print("this is a tcp Message")
                    break

            if proto == 17:
                (udp_src_port, udp_dest_port, size, rest_data) =
udp_segment(data)
                if udp_dest_port == 9999:
                    print("     -->  Destination MAC : "+ str(dest_mac))
                    print("     -->  Sorcue MAC : " + str(src_mac))
                    print("     -->  Source IP : " + str(target))
                    print("     -->  Destination IP: " + str(src))
                    print("     -->  Source Port : " + str(udp_src_port))
                    print("     -->  Destination Port: " + str(udp_dest_port))
                    print("this is a udp Message")
                    break
```

```
# unpack and extract the data from ethernet fram layer 2
def ethernet_head(raw_data):
    dest_mac, src_mac, prototype = struct.unpack('! 6s 6s H', raw_data[:14])
    return get_mac_add(dest_mac), get_mac_add(src_mac), socket.htons(prototype),
raw_data [14:]

def get_mac_add(raw):
    address = map('{:02x}'.format,raw)
    return ':'.join(address).upper()

# unpack and extract the ip v4 header information
def ipv4_head(data):
    versiion_header_length = data[0]
    versiion = versiion_header_length >> 4
    header_length = (versiion_header_length & 15) * 4
    time_to_live, proto, src, target = struct.unpack('! 8x b b 2x 4s 4s',
data[:20])
    return versiion, header_length, time_to_live, proto, ip_format(src),
ip_format(target), data[header_length:]

# format ip
def ip_format(address):
    return '.'.join(map(str, address))


# unpack tcp
def tcp_segment(data):
    (src_port,dest_port, sequence, acknowledgement, offset_reserved_flags) =
struct.unpack("! H H L L H", data[:14])
    offset= (offset_reserved_flags >> 12 ) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_puh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1
    return src_port,dest_port, sequence, acknowledgement, flag_urg, flag_ack,
flag_puh, flag_rst, flag_syn, flag_fin, data[offset:]

#unpakc UDP
def udp_segment(data):
    udp_src_port, udp_dest_port, size = struct.unpack('! H H 2x H', data[:8])
    return udp_src_port, udp_dest_port, size, data[8:]




main()
```

Client File Source Code

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # create a socket object

host = '172.19.0.10'
port = 9999

s.connect((host, port))      # connection to hostname on the port.
tm = s.recv(1024)     # Receive no more than 1024
bytes

s.close()
print("The time got from the server is %s" % tm.decode('ascii'))
```