James Madison University
Information Technology (IT) Program

Physical and Virtual DHCP NAT and Python Socket programming

Sunday 12th May, 2024

13:56:34

Author(s):
Jason Amaya

Jason Amaya

Signature

Date

Contents

List of Figures

1 Introduction

The seamless communication and interaction between devices in modern computer networks are facilitated by a complex system of protocols and technologies. Two fundamental components of this network infrastructure are the Dynamic Host Configuration Protocol (DHCP) and Network Address Translation (NAT). DHCP plays a critical role in automatically assigning IP addresses to devices on a network, ensuring efficient resource allocation and network management. NAT, on the other hand, enables multiple devices within a private network to share a single public IP address, enhancing security and optimizing the utilization of limited IP addresses.

In this lab report, we explore the workings of Physical (Access Point) and Virtual (VM) DHCP & NAT functions, and their roles in networking. Furthermore, we will explore the realm of Python socket programming, focusing on the development of TCP and UDP servers.

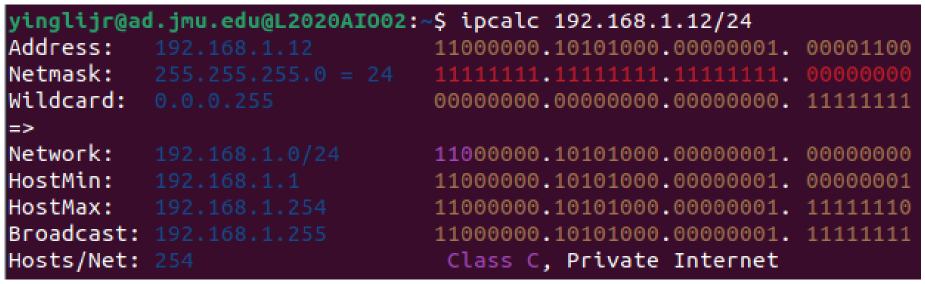
By the end of this, I will gain a deeper understanding of how DHCP, NAT, and socket programming are essential in enabling efficient networking solutions, both physically and virtually. This knowledge will empower to navigate the complex world of network management and communication, a vital skill in future careers and increasingly connected world. [1]

2 Exercises : Results & Analysis

2.1 Exercise 1 - Host & Network Devices

2.1.1 Step 1: Physical and Virtual DHCP & NAT function

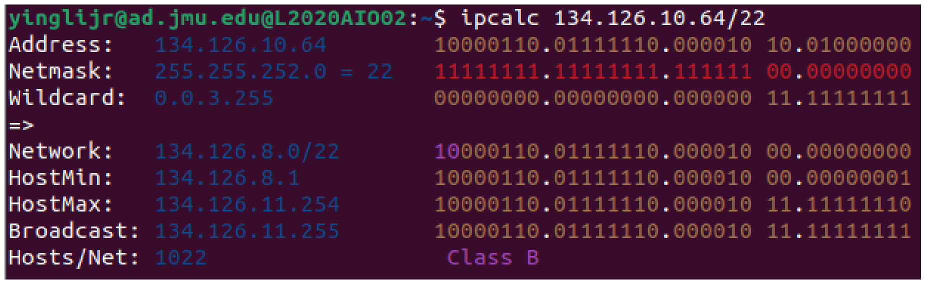
T1: ipcalc command:



```
yinglijr@ad.jmu.edu@L2020AI002:~$ ipcalc 192.168.1.12/24
Address:   192.168.1.12          11000000.10101000.00000001. 00001100
Netmask:   255.255.255.0 = 24    11111111.11111111.11111111. 00000000
Wildcard:  0.0.0.255             00000000.00000000.00000000. 11111111
=>
Network:   192.168.1.0/24        11000000.10101000.00000001. 00000000
HostMin:   192.168.1.1           11000000.10101000.00000001. 00000001
HostMax:   192.168.1.254         11000000.10101000.00000001. 11111110
Broadcast: 192.168.1.255         11000000.10101000.00000001. 11111111
Hosts/Net: 254                    Class C, Private Internet
```

Figure 1: ipcalc command

The ipcalc command on linux calculates IP information for a host. 1



```
yinglijr@ad.jmu.edu@L2020AI002:~$ ipcalc 134.126.10.64/22
Address:   134.126.10.64         10000110.01111110.000010 10.01000000
Netmask:   255.255.252.0 = 22    11111111.11111111.111111 00.00000000
Wildcard:  0.0.3.255             00000000.00000000.000000 11.11111111
=>
Network:   134.126.8.0/22        10000110.01111110.000010 00.00000000
HostMin:   134.126.8.1           10000110.01111110.000010 00.00000001
HostMax:   134.126.11.254        10000110.01111110.000010 11.11111110
Broadcast: 134.126.11.255        10000110.01111110.000010 11.11111111
Hosts/Net: 1022                   Class B
```

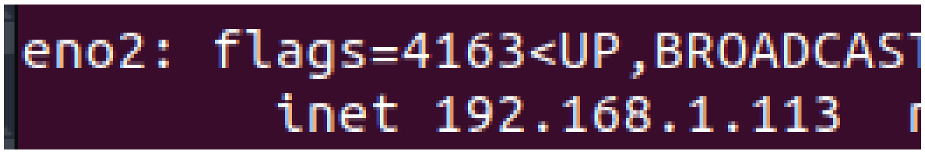
Figure 2: 134.126.10.64/22

Q1: Is 134.126.10.64/22 a private or public IP address?

A1: It is a public IP address, because upon running ipcalc, it is not specified that it is a private address, so we can assume it is public. 2

T2: Using an Ethernet cord/cable, connect the Ethernet port on the Ubuntu Desktop to one of the LAN ports (yellow) on the AP. You will need to disconnect and connect the Ethernet connection on Ubuntu Desktop to acquire an IP address from the AP.
Check that the Ethernet port on Dell 990/Ubuntu Desktop acquired an IP address using the command ifconfig.

Q1: State the IP address acquired by your Dell 990 and identify the server in the AP that allows the Ubuntu Desktop/Dell 990 Ethernet port to acquire an IP address.

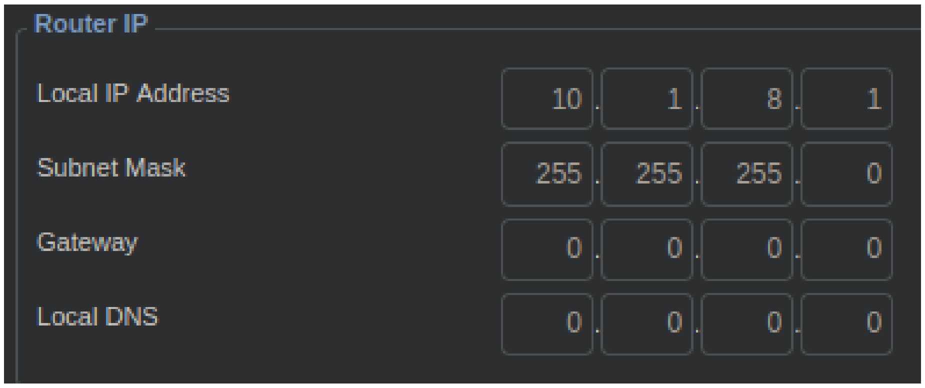


```
eno2: flags=4163<UP,BROADCAS
        inet 192.168.1.113  n
```

Figure 3: eno2 IP address via ifconfig command

A1: The IP address acquired by the Dell 990 is 192.168.1.133. The server in the AP that allows the Dell 990 to acquire an IP address is the DHCP server. 3

T3: We must log into the access point by opening a browser and entering: ”192.168.1.1”.

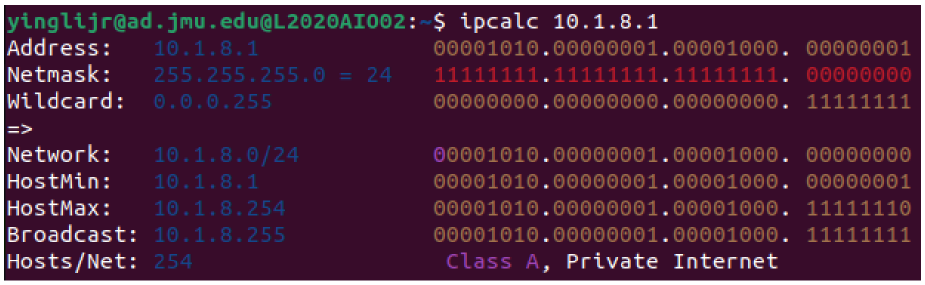


Router IP

Local IP Address	10	1	8	1
Subnet Mask	255	255	255	0
Gateway	0	0	0	0
Local DNS	0	0	0	0

Figure 4: 10.1.8.1

Since we are section 1 group 8, we must change the local address to ”10.1.8.1”. 4

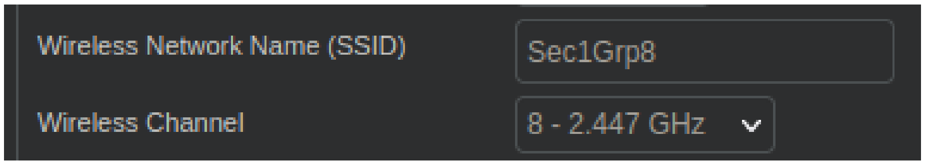


```
yinglijr@ad.jmu.edu@L2020AI002:~$ ipcalc 10.1.8.1
Address:   10.1.8.1             00001010.00000001.00001000. 00000001
Netmask:   255.255.255.0 = 24   11111111.11111111.11111111. 00000000
Wildcard:  0.0.0.255            00000000.00000000.00000000. 11111111
=>
Network:   10.1.8.0/24          00001010.00000001.00001000. 00000000
HostMin:   10.1.8.1             00001010.00000001.00001000. 00000001
HostMax:   10.1.8.254           00001010.00000001.00001000. 11111110
Broadcast: 10.1.8.255           00001010.00000001.00001000. 11111111
Hosts/Net: 254                   Class A, Private Internet
```

Figure 5: 10.1.8.1

Q1: Is 10.1.8.1 a public or private IP address?

A1: It is a private IP address, because upon running ipcalc, it is specified that it is a private address. 5

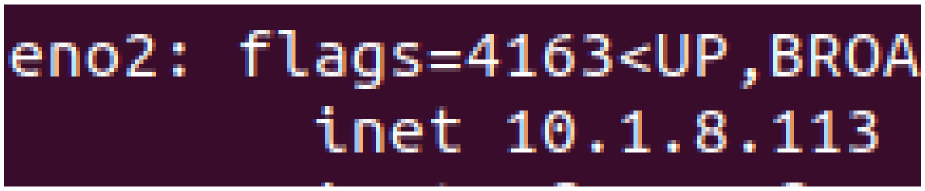


Wireless Network Name (SSID) Sec1Grp8

Wireless Channel 8 - 2.447 GHz

Figure 6: SSID & Channel

T4: We must change the Wireless Network Name (SSID) under the Wireless Physical Interface wl0 [2.4 GHz] to Sec1Grp8. We must also change the Wireless Channel to the channel number corresponding to our group number, which is 8. 6
To acquire a new IP address, we can physically disconnect the 990 from the AP, and then physically reconnect it.

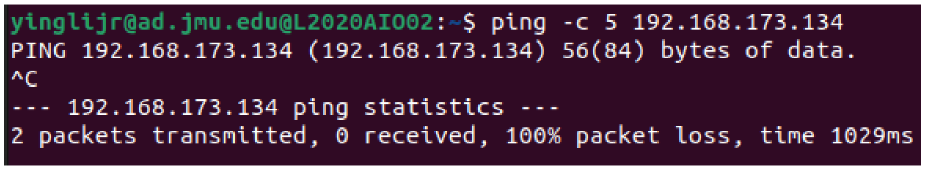


```
eno2: flags=4163<UP,BROA
        inet 10.1.8.113
```

Figure 7: eno2 IP address after editing via ifconfig command

Q1: What is the new IP acquired by Dell 990/Ubuntu?

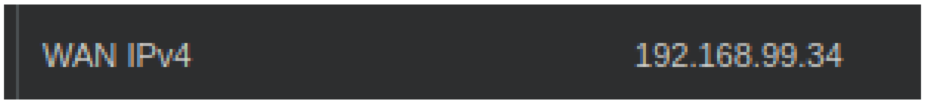
A1: The new IP address is 10.1.8.113. 7



```
yinglijr@ad.jmu.edu@L2020AI002:~$ ping -c 5 192.168.173.134
PING 192.168.173.134 (192.168.173.134) 56(84) bytes of data.
^C
--- 192.168.173.134 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1029ms
```

Figure 8: Ping Dell 9020

T5: We open a terminal on Dell 990/Ubuntu Desktop and ping the IP address of Dell 9020, type ping –c 5 ¡ip address of the Host Dell 9020 Desktop¿. We can see that the Dell 9020 is unreachable 8. The pings fail because the network adapter of the VM is bridged, which means that bridged connection, being a pure switching interface in the datalink layer, has no idea how to interpret ping. There is also no physical connection established between the two through the AP. [2]

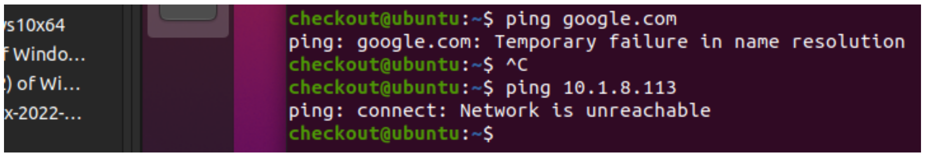


WAN IPv4 192.168.99.34

Figure 9: WAN IPv4

Q1: On the router webpage (control panel) find the IP address next to WAN IPv4.

A1: The WAN IPv4 address is 192.168.99.34 9



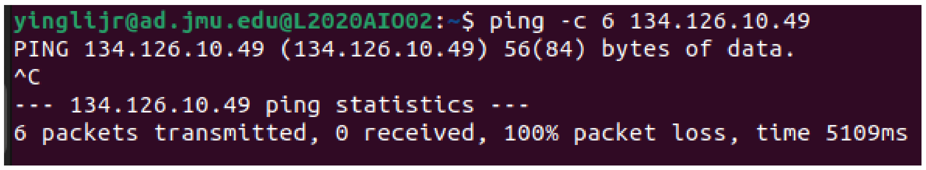
```
checkout@ubuntu:~$ ping google.com
ping: google.com: Temporary failure in name resolution
checkout@ubuntu:~$ ^C
checkout@ubuntu:~$ ping 10.1.8.113
ping: connect: Network is unreachable
checkout@ubuntu:~$
```

Figure 10: Ping Host from VM

T6: Now ping the Dell 9020 using the Host IP address, type ping –c 5 ¡ip address of the Host Dell 9020 Desktop¿.

Q1: Is the ping (ICMP request) successful?

A1: The ping still failed for us, we might have configured something wrong. We tried with VM network adapter set to NAT, and bridged, and tried multiple VMs. 10

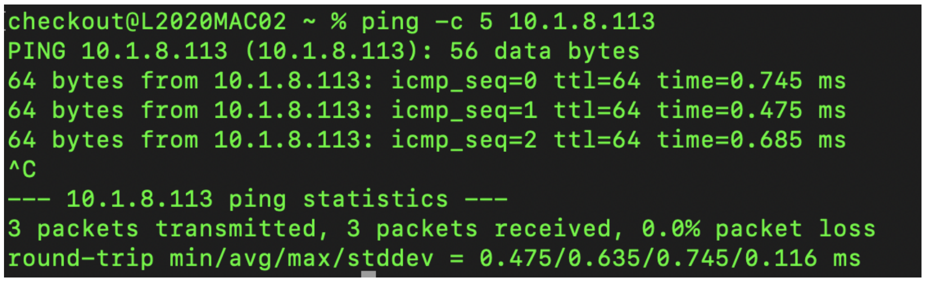


```
yinglijr@ad.jmu.edu@L2020AI002:~$ ping -c 6 134.126.10.49
PING 134.126.10.49 (134.126.10.49) 56(84) bytes of data.
^C
--- 134.126.10.49 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5109ms
```

Figure 11: Ping JMU mail server

T7: Ping JMU mail server for students account at 134.126.10.49 by typing: ping –c 6 134.126.10.49.

- **Q1:** Is the ping successful?
- **A1:** The ping is not successful, as the JMU mail server has been set up to block all incoming ICMP packets due to fears of attacks. 11



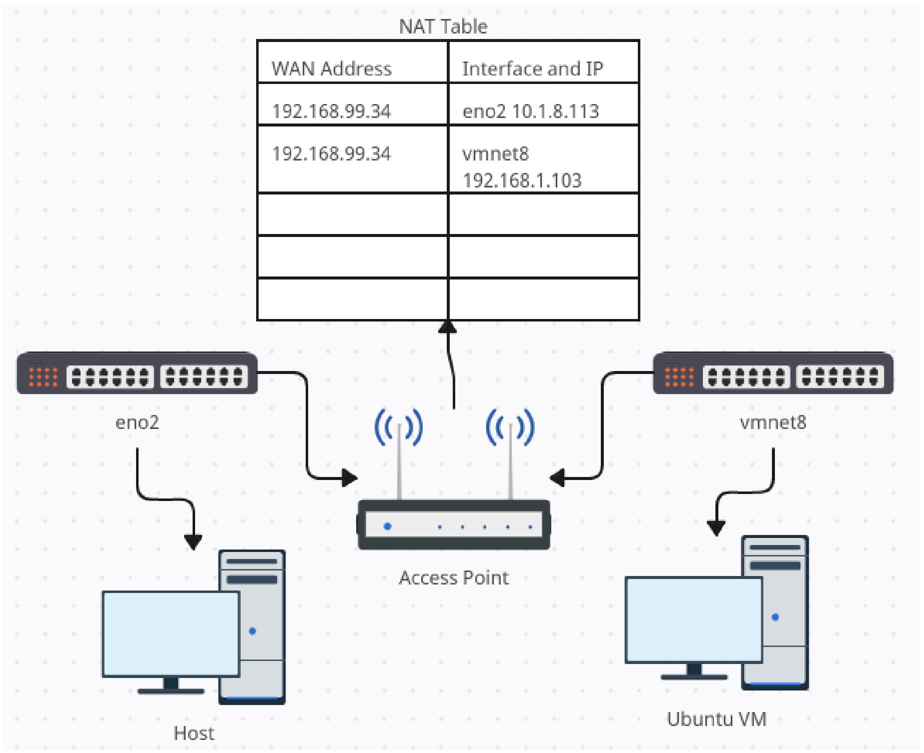
```
checkout@L2020MAC02 ~ % ping -c 5 10.1.8.113
PING 10.1.8.113 (10.1.8.113): 56 data bytes
64 bytes from 10.1.8.113: icmp_seq=0 ttl=64 time=0.745 ms
64 bytes from 10.1.8.113: icmp_seq=1 ttl=64 time=0.475 ms
64 bytes from 10.1.8.113: icmp_seq=2 ttl=64 time=0.685 ms
^C
--- 10.1.8.113 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.475/0.635/0.745/0.116 ms
```

Figure 12: Mac ping Dell 990

T8: Disconnect the iMac from the station switch and connect it to a LAN port (yellow) on the AP, identify the IP address acquired, and check that the iMac has connectivity to Dell 990, Dell 9020 and JMU students email server by pinging the IP addresses of each of these entities. 12

T9: Set up Ubuntu VM to be connected to vmnet8 by configuring its Network Adapter for NAT connection.

- **Q1:** Check out its connectivity to the Host IP address and the JMU email server
- **A1:** The ping failed for us, we might have configured something wrong. We tried with VM network adapter set to NAT, and bridged, and tried multiple VMs.
- **Q2:** Draw and submit a network diagram including Ubuntu VM, vmnet8 (virtual Private IP DHCP server, virtual NAT), and Host.



WAN Address	Interface and IP
192.168.99.34	eno2 10.1.8.113
192.168.99.34	vmnet8 192.168.1.103

Figure 13: Network diagram

A2: Network Diagram

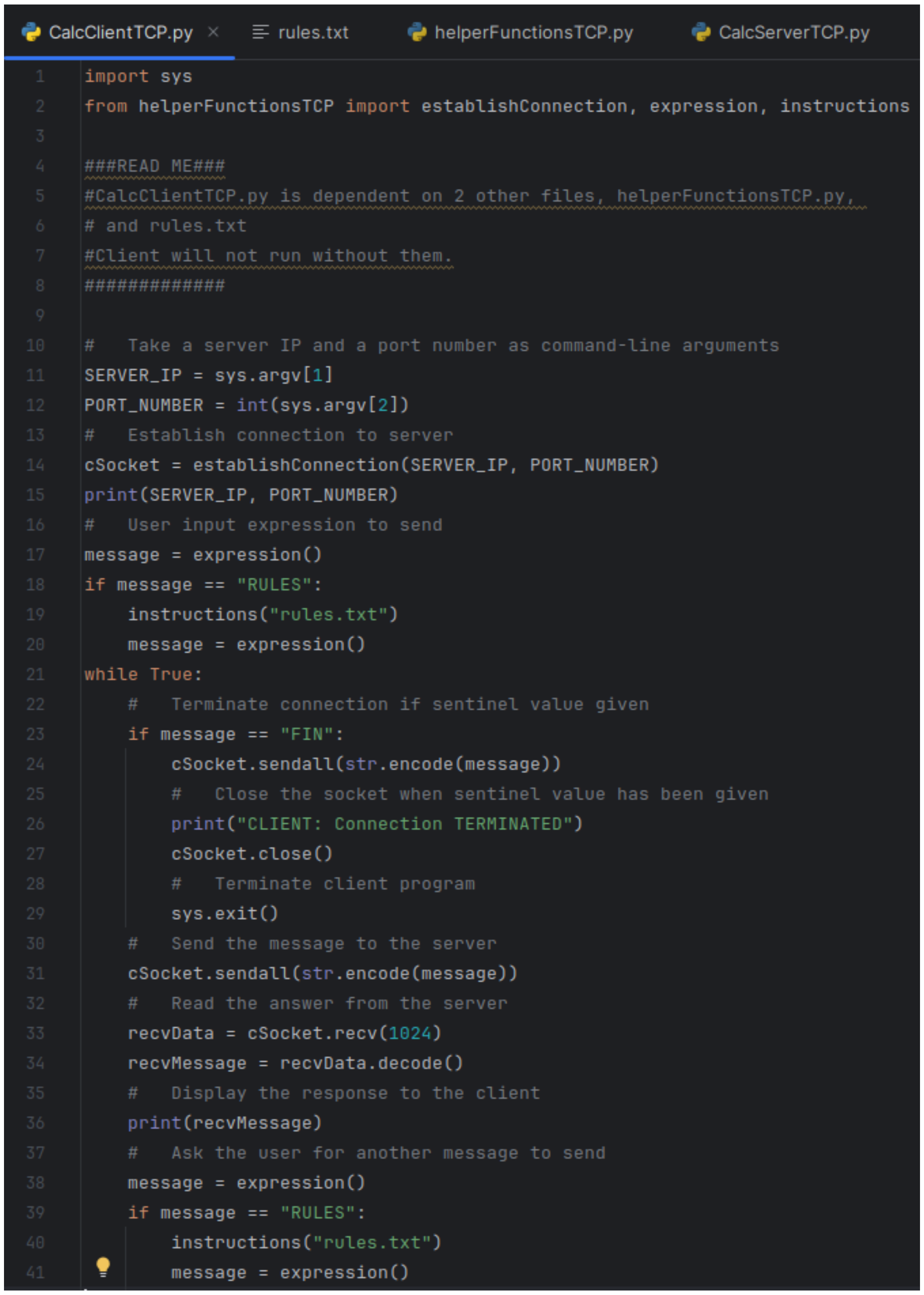
T10: Additional Questions:

Q1: What is the difference between Dynamic and Static IP addresses?

A1: When a device is assigned a static IP address, the address does not change. Most devices use dynamic IP addresses, which are assigned by the network when they connect and change over time.

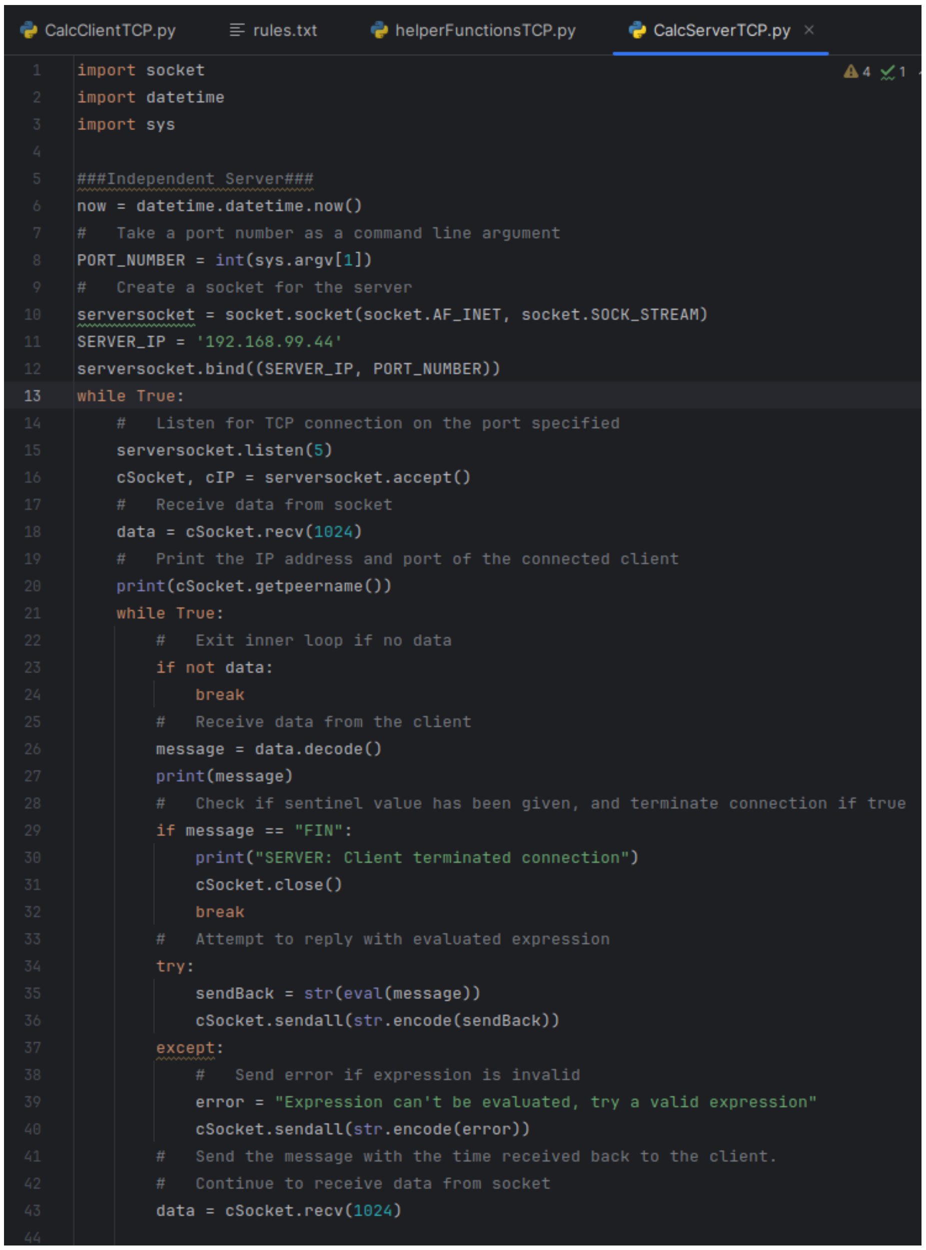
2.2 Exercise 2 - Socket Programming – TCP/UDP Client/Server Calculator

2.2.1 Part 1 - TCP



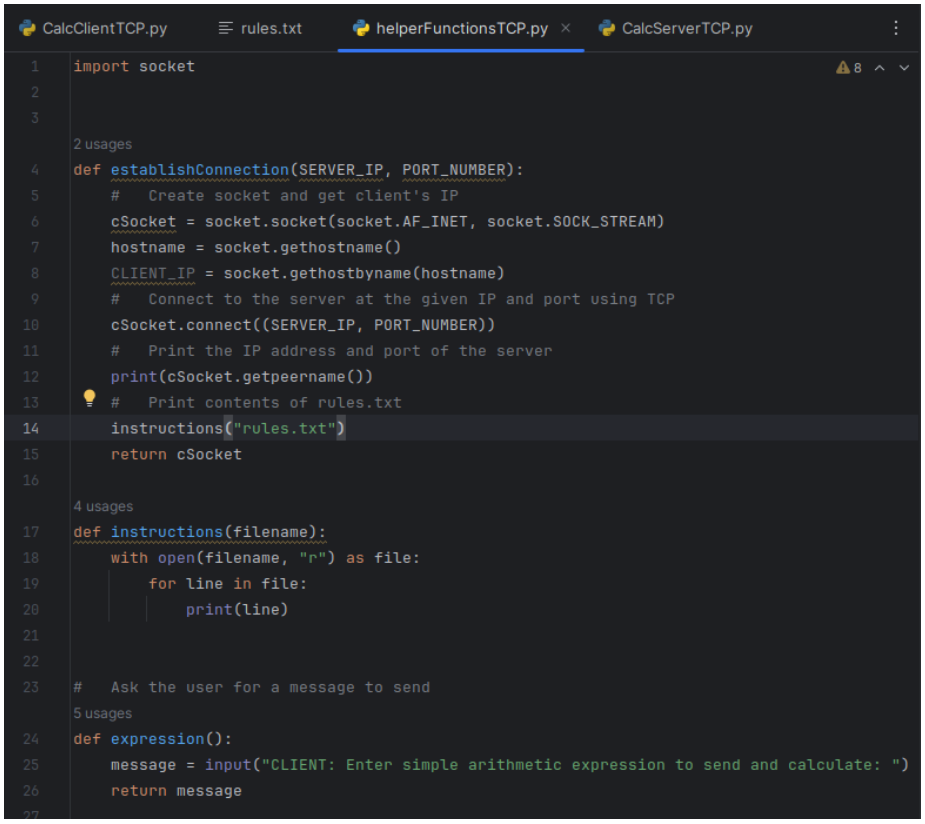
```python
import sys
from helperFunctionsTCP import establishConnection, expression, instructions

###READ ME###
#CalcClientTCP.py is dependent on 2 other files, helperFunctionsTCP.py,
# and rules.txt
#Client will not run without them.
#############

#   Take a server IP and a port number as command-line arguments
SERVER_IP = sys.argv[1]
PORT_NUMBER = int(sys.argv[2])
#   Establish connection to server
cSocket = establishConnection(SERVER_IP, PORT_NUMBER)
print(SERVER_IP, PORT_NUMBER)
#   User input expression to send
message = expression()
if message == "RULES":
    instructions("rules.txt")
    message = expression()
while True:
    #   Terminate connection if sentinel value given
    if message == "FIN":
        cSocket.sendall(str.encode(message))
        #   Close the socket when sentinel value has been given
        print("CLIENT: Connection TERMINATED")
        cSocket.close()
        #   Terminate client program
        sys.exit()
    #   Send the message to the server
    cSocket.sendall(str.encode(message))
    #   Read the answer from the server
    recvData = cSocket.recv(1024)
    recvMessage = recvData.decode()
    #   Display the response to the client
    print(recvMessage)
    #   Ask the user for another message to send
    message = expression()
    if message == "RULES":
        instructions("rules.txt")
        message = expression()
```

Figure 14: TCP client



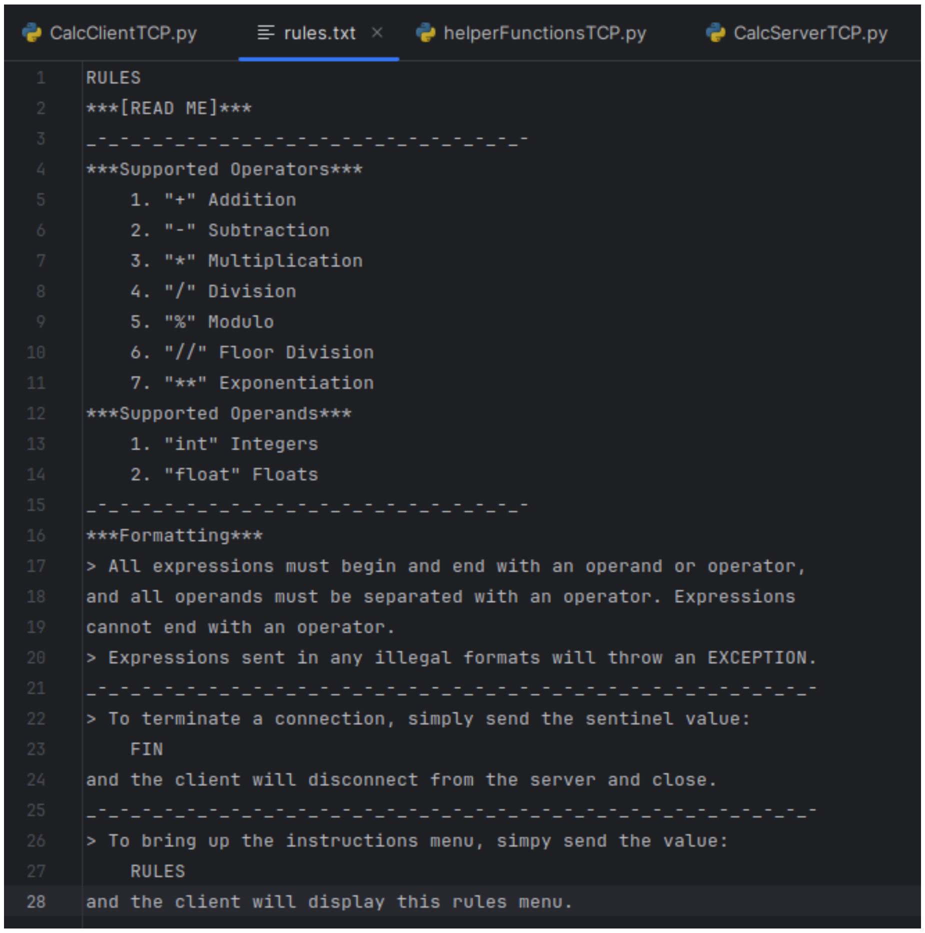
```python
import socket
import datetime
import sys

###Independent Server###
now = datetime.datetime.now()
#   Take a port number as a command line argument
PORT_NUMBER = int(sys.argv[1])
#   Create a socket for the server
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
SERVER_IP = '192.168.99.44'
serversocket.bind((SERVER_IP, PORT_NUMBER))
while True:
    #   Listen for TCP connection on the port specified
    serversocket.listen(5)
    cSocket, cIP = serversocket.accept()
    #   Receive data from socket
    data = cSocket.recv(1024)
    #   Print the IP address and port of the connected client
    print(cSocket.getpeername())
    while True:
        #   Exit inner loop if no data
        if not data:
            break
        #   Receive data from the client
        message = data.decode()
        print(message)
        #   Check if sentinel value has been given, and terminate connection if true
        if message == "FIN":
            print("SERVER: Client terminated connection")
            cSocket.close()
            break
        #   Attempt to reply with evaluated expression
        try:
            sendBack = str(eval(message))
            cSocket.sendall(str.encode(sendBack))
        except:
            #   Send error if expression is invalid
            error = "Expression can't be evaluated, try a valid expression"
            cSocket.sendall(str.encode(error))
        #   Send the message with the time received back to the client.
        #   Continue to receive data from socket
        data = cSocket.recv(1024)

```

Figure 15: TCP Server



```python
import socket


2 usages
def establishConnection(SERVER_IP, PORT_NUMBER):
    #   Create socket and get client's IP
    cSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    hostname = socket.gethostname()
    CLIENT_IP = socket.gethostbyname(hostname)
    #   Connect to the server at the given IP and port using TCP
    cSocket.connect((SERVER_IP, PORT_NUMBER))
    #   Print the IP address and port of the server
    print(cSocket.getpeername())
    #   Print contents of rules.txt
    instructions("rules.txt")
    return cSocket

4 usages
def instructions(filename):
    with open(filename, "r") as file:
        for line in file:
            print(line)


#   Ask the user for a message to send
5 usages
def expression():
    message = input("CLIENT: Enter simple arithmetic expression to send and calculate: ")
    return message
```

Figure 16: TCP helper functions



```
RULES
***[READ ME]***
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-
***Supported Operators***
    1. "+" Addition
    2. "-" Subtraction
    3. "*" Multiplication
    4. "/" Division
    5. "%" Modulo
    6. "//" Floor Division
    7. "**" Exponentiation
***Supported Operands***
    1. "int" Integers
    2. "float" Floats
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-
***Formatting***
> All expressions must begin and end with an operand or operator,
and all operands must be separated with an operator. Expressions
cannot end with an operator.
> Expressions sent in any illegal formats will throw an EXCEPTION.
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-
> To terminate a connection, simply send the sentinel value:
    FIN
and the client will disconnect from the server and close.
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-
> To bring up the instructions menu, simpy send the value:
    RULES
and the client will display this rules menu.
```

Figure 17: TCP expression rules

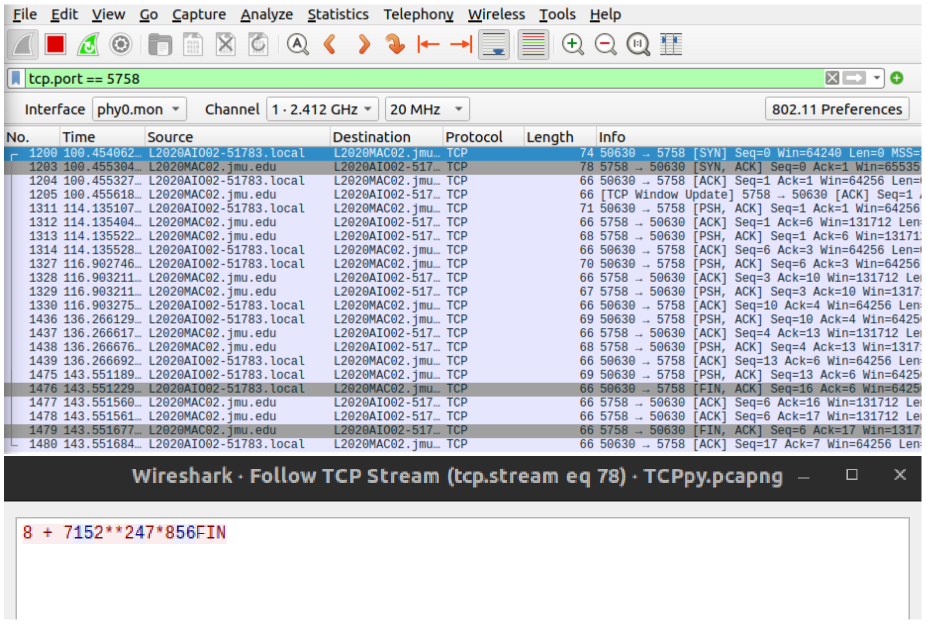
Q1: Start your TCP client application without the TCP server running. What happens? Why?

A1: A ConnectionRefusedError exception is thrown, and the program is terminated. The Client can only operate if it can establish a connection with the server.

Q2: Try different examples and explain the output!

A2: Server only - when only the TCP server application is running, no connection is established, but the server will stay up, waiting for a connection to be made.
Server and client - when both the TCP server and TCP client are running, but the TCP client has entered the wrong IP address, an OSError exception gets thrown, saying there is no route to the host. If the Client specifies only the incorrect port, a Con-

nectionRefusedError exception will be thrown. All exceptions thrown result in the client program terminating.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
802.11 Preferences

tcp.port == 5758

Interface phy0.mon Channel 1 · 2.412 GHz 20 MHz

No.	Time	Source	Destination	Protocol	Length	Info
1200	100.454062…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	74	50630 → 5758 [SYN] Seq=0 Win=64240 Len=0 MSS=
1203	100.455304…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	78	5758 → 50630 [SYN, ACK] Seq=0 Ack=1 Win=65535
1204	100.455327…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	66	50630 → 5758 [ACK] Seq=1 Ack=1 Win=64256 Len=
1205	100.455618…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	66	[TCP Window Update] 5758 → 50630 [ACK] Seq=1
1311	114.135107…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	71	50630 → 5758 [PSH, ACK] Seq=1 Ack=1 Win=64256
1312	114.135404…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	66	5758 → 50630 [ACK] Seq=1 Ack=6 Win=131712 Len
1313	114.135522…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	68	5758 → 50630 [PSH, ACK] Seq=1 Ack=6 Win=13171
1314	114.135528…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	66	50630 → 5758 [ACK] Seq=6 Ack=3 Win=64256 Len=
1327	116.902746…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	70	50630 → 5758 [PSH, ACK] Seq=6 Ack=3 Win=64256
1328	116.903211…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	66	5758 → 50630 [ACK] Seq=3 Ack=10 Win=131712 Le
1329	116.903211…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	67	5758 → 50630 [PSH, ACK] Seq=3 Ack=10 Win=1317
1330	116.903275…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	66	50630 → 5758 [ACK] Seq=10 Ack=4 Win=64256 Len
1436	136.266129…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	69	50630 → 5758 [PSH, ACK] Seq=10 Ack=4 Win=6425
1437	136.266617…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	66	5758 → 50630 [ACK] Seq=4 Ack=13 Win=131712 Le
1438	136.266676…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	68	5758 → 50630 [PSH, ACK] Seq=4 Ack=13 Win=1317
1439	136.266692…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	66	50630 → 5758 [ACK] Seq=13 Ack=6 Win=64256 Len
1475	143.551189…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	69	50630 → 5758 [PSH, ACK] Seq=13 Ack=6 Win=6425
1476	143.551229…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	66	50630 → 5758 [FIN, ACK] Seq=16 Ack=6 Win=6425
1477	143.551560…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	66	5758 → 50630 [ACK] Seq=6 Ack=16 Win=131712 Le
1478	143.551561…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	66	5758 → 50630 [ACK] Seq=6 Ack=17 Win=131712 Le
1479	143.551677…	L2020MAC02.jmu.edu	L2020AI002-517…	TCP	66	5758 → 50630 [FIN, ACK] Seq=6 Ack=17 Win=1317
1480	143.551684…	L2020AI002-51783.local	L2020MAC02.jmu…	TCP	66	50630 → 5758 [ACK] Seq=17 Ack=7 Win=64256 Len

Wireshark · Follow TCP Stream (tcp.stream eq 78) · TCPpy.pcapng

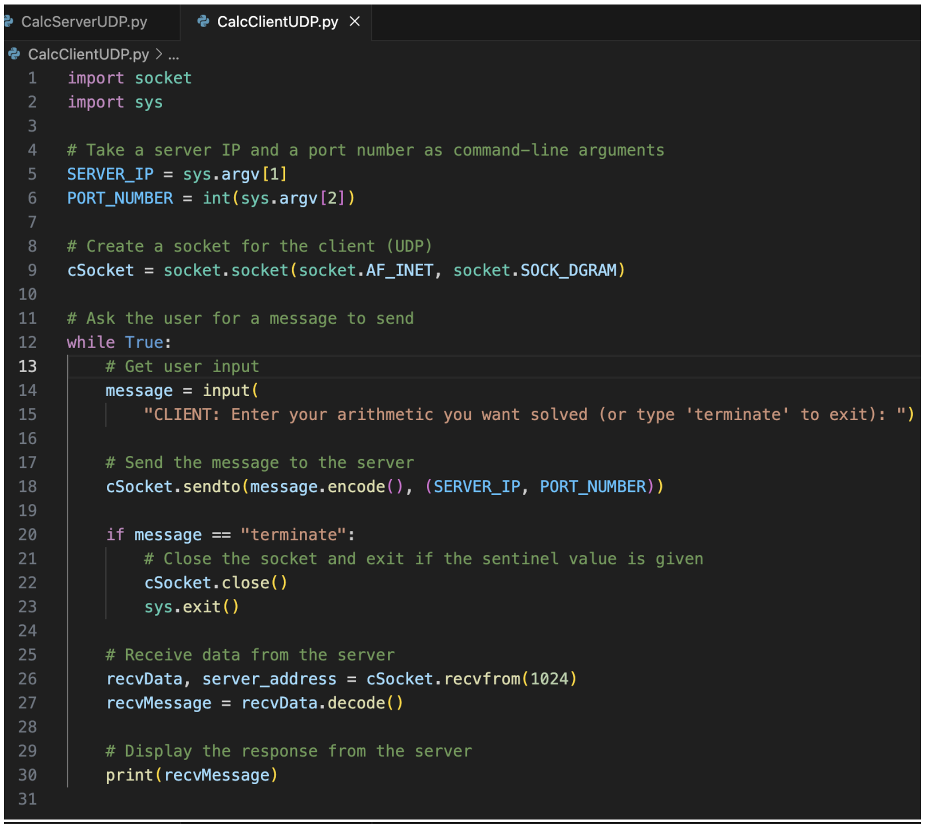
```
8 + 7152**247*856FIN
```

Figure 18: TCP Client/Server Capture & TCP Stream

Q3: Capture using Wireshark the data transferred between Client and Server, and explain it!

A3: We can see the TCP stream that took place. The text highlighted in red is expressions sent from the client, and the text highlighted in blue is the server calculation of the expression. The stream is closed when the client sends the sentinel value, FIN, and the server no longer can reply.

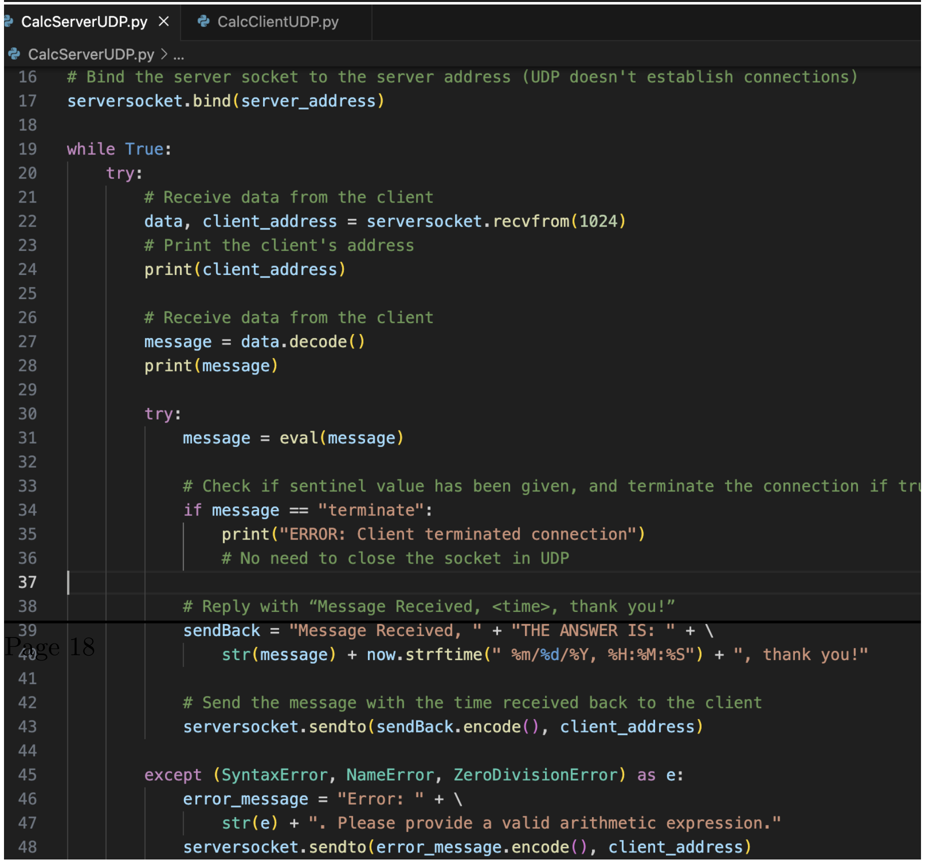
2.2.2 Part 2 - UDP



CalcServerUDP.py | CalcClientUDP.py

CalcClientUDP.py > ...

```python
import socket
import sys

# Take a server IP and a port number as command-line arguments
SERVER_IP = sys.argv[1]
PORT_NUMBER = int(sys.argv[2])

# Create a socket for the client (UDP)
cSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Ask the user for a message to send
while True:
    # Get user input
    message = input(
        "CLIENT: Enter your arithmetic you want solved (or type 'terminate' to exit): ")

    # Send the message to the server
    cSocket.sendto(message.encode(), (SERVER_IP, PORT_NUMBER))

    if message == "terminate":
        # Close the socket and exit if the sentinel value is given
        cSocket.close()
        sys.exit()

    # Receive data from the server
    recvData, server_address = cSocket.recvfrom(1024)
    recvMessage = recvData.decode()

    # Display the response from the server
    print(recvMessage)

```



CalcServerUDP.py | CalcClientUDP.py

CalcServerUDP.py > ...

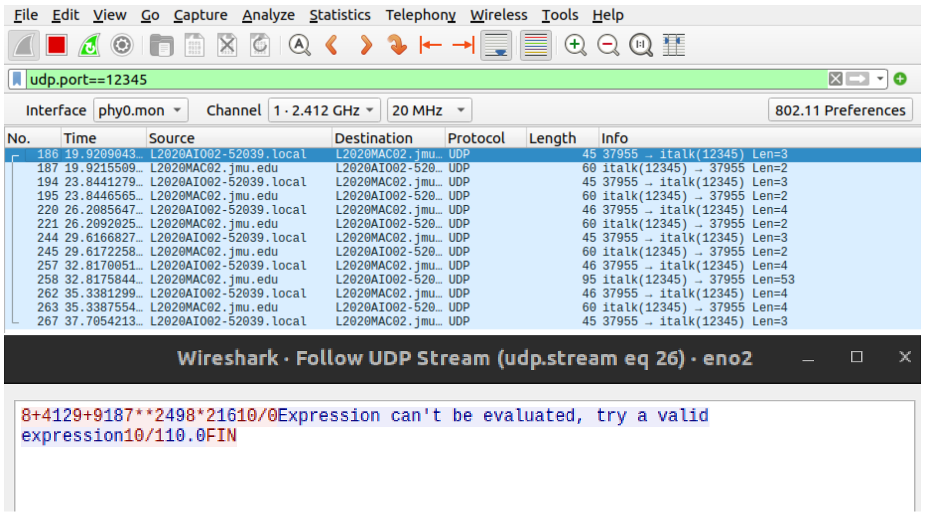
```python
# Bind the server socket to the server address (UDP doesn't establish connections)
serversocket.bind(server_address)

while True:
    try:
        # Receive data from the client
        data, client_address = serversocket.recvfrom(1024)
        # Print the client's address
        print(client_address)

        # Receive data from the client
        message = data.decode()
        print(message)

        try:
            message = eval(message)

            # Check if sentinel value has been given, and terminate the connection if tru
            if message == "terminate":
                print("ERROR: Client terminated connection")
                # No need to close the socket in UDP

            # Reply with “Message Received, <time>, thank you!”
            sendBack = "Message Received, " + "THE ANSWER IS: " + \
                str(message) + now.strftime(" %m/%d/%Y, %H:%M:%S") + ", thank you!"

            # Send the message with the time received back to the client
            serversocket.sendto(sendBack.encode(), client_address)

        except (SyntaxError, NameError, ZeroDivisionError) as e:
            error_message = "Error: " + \
                str(e) + ". Please provide a valid arithmetic expression."
            serversocket.sendto(error_message.encode(), client_address)
```

Q1: Start your UDP client application without the UDP server running. What happens? Why?

A1: When running a UDP application on the client side, it won't immediately crash or raise errors in contrast to TCP. UDP is designed to send data without requiring an established connection, it will send the data and move on.

Q2: Try different examples and explain the output!

A2: When starting the server side of a UDP connection, it won't stall like TCP, as UDP is connectionless. The server will simply listen for incoming datagrams, and it doesn't wait for a client to establish a connection. If there's an error in the IP address or port specified on the client side in a UDP connection, the client will proceed without raising an error. It will send the data to the address provided, even if it's incorrect. However, since UDP doesn't guarantee delivery or provide acknowledgment, the data might be sent to the wrong address and remain unanswered. Unlike TCP, which would raise an error when there's an issue with the connection setup, UDP doesn't provide the same level of error handling for connection-related issues.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

udp.port==12345

Interface phy0.mon   Channel 1 · 2.412 GHz   20 MHz   802.11 Preferences

No.	Time	Source	Destination	Protocol	Length	Info
186	19.9209043…	L2020AI002-52039.local	L2020MAC02.jmu…	UDP	45	37955 → italk(12345) Len=3
187	19.9215509…	L2020MAC02.jmu.edu	L2020AI002-520…	UDP	60	italk(12345) → 37955 Len=2
194	23.8441279…	L2020AI002-52039.local	L2020MAC02.jmu…	UDP	45	37955 → italk(12345) Len=3
195	23.8446565…	L2020MAC02.jmu.edu	L2020AI002-520…	UDP	60	italk(12345) → 37955 Len=2
220	26.2085647…	L2020AI002-52039.local	L2020MAC02.jmu…	UDP	46	37955 → italk(12345) Len=4
221	26.2092025…	L2020MAC02.jmu.edu	L2020AI002-520…	UDP	60	italk(12345) → 37955 Len=2
244	29.6166827…	L2020AI002-52039.local	L2020MAC02.jmu…	UDP	45	37955 → italk(12345) Len=3
245	29.6172258…	L2020MAC02.jmu.edu	L2020AI002-520…	UDP	60	italk(12345) → 37955 Len=2
257	32.8170051…	L2020AI002-52039.local	L2020MAC02.jmu…	UDP	46	37955 → italk(12345) Len=4
258	32.8175844…	L2020MAC02.jmu.edu	L2020AI002-520…	UDP	95	italk(12345) → 37955 Len=53
262	35.3381299…	L2020AI002-52039.local	L2020MAC02.jmu…	UDP	46	37955 → italk(12345) Len=4
263	35.3387554…	L2020MAC02.jmu.edu	L2020AI002-520…	UDP	60	italk(12345) → 37955 Len=4
267	37.7054213…	L2020AI002-52039.local	L2020MAC02.jmu…	UDP	45	37955 → italk(12345) Len=3

Wireshark · Follow UDP Stream (udp.stream eq 26) · eno2

```
8+4129+9187**2498*21610/0Expression can't be evaluated, try a valid
expression10/110.0FIN
```

Figure 20: UDP Client/Server Capture & UDP Stream

Q3: Capture using Wireshark the data transferred between Client and Server and explain it!

A3: We can see the UDP stream that took place. The text highlighted in red is expressions sent from the client, and the text highlighted in blue is the server calculation of the expression. The stream is closed when the client sends the sentinel value, FIN, and the server no longer can reply.

Q4: Compare the difference between TCP and UDP programs!

A4: The primary distinction between these two protocols lies in their error detection and handling mechanisms. TCP provides users with robust error detection, which means that if there are issues with the IP address or port, it will result in a connection failure. In contrast, UDP simply sends data without verifying whether it reaches the correct server, making it important to exercise caution when configuring connections.

This goes to show the reliability of TCP, this stems from its error detection and correction features. With the lack of error handling means that with UDP, it's crucial for the application or the user to implement their own error-checking mechanisms if data integrity is essential. In summary, while TCP provides a robust safety net for data communication, UDP prioritizes speed and efficiency but necessitates a higher level of caution and responsibility when it comes to error detection and correction at the application level.

2.3 Key Learning & Takeaways

In these exercises, I learned about the critical differences between private and public IP addresses, which are fundamental to configuring networks effectively. I had the opportunity to set up physical and virtual network environments, including DHCP and NAT functionalities, and understand their roles in network management. Moreover, I gained hands-on experience in socket programming by creating both TCP and UDP client-server applications for a simple calculator, which deepened my understanding of networked applications. These exercises reinforced the importance of robust error handling and network traffic analysis using tools like Wireshark, enhancing my troubleshooting skills. Overall, these activities have provided valuable insights into network configurations, socket programming, and networking principles, crucial for anyone working in the field.

3 Conclusion

3.1 Summary

In this report, I explored two pivotal aspects of networking technology: Physical and Virtual DHCP & NAT functions, along with Python socket programming featuring TCP and UDP servers. Our exploration into these topics provided valuable insights into the core elements of modern network infrastructure and the tools that enable efficient communication between networked devices.

The report began by delving into the world of DHCP and NAT, I learned how to identify private and public IP addresses using Debian's "ipcalc" command in the terminal. I then used an access point that utilized NAT to create a private network with multiple hosts where all hosts on the network share the same public IP address. I then learned how to configure the private network through a browser, and tested connectivity between computers before and after connected them through the access point. I then explored Virtual NAT functionality using VMWare, specifically exploring the functionality of the "vmnet8" virtual switch.

In the second part of the report, we ventured into the realm of Python socket programming. With a focus on TCP and UDP servers, we gained a practical understanding of how these servers facilitate communication between networked devices. The implementation of socket programming in Python allowed us send mathematical expressions to a server, and have the server calculate the value and send it back to a client.

By combining our knowledge of Physical and Virtual DHCP NAT functions with Python socket programming, this lab report equips us with the tools to navigate the complex landscape of networking. It underscores the importance of these technologies in building and maintaining efficient, secure, and scalable network infrastructures, whether in physical or virtual environments. In an interconnected world, this knowledge is invaluable for anyone seeking to understand and master the principles that drive modern networking solutions.