Network Lab Assignment

Name	Tusher Mondal
Roll	002210503039

Assignment 4

Problem Statement 1 & 2 & 3:
Using scapy do the following:

1. Capture 10000 packets
2. Count the number distinct host IP addresses and display them
3. For each distinct pair of source/destination host IP addresses determine the number of TCP/UDP segments exchanged and also the average payload length.

Desired Output:

Source IP	Destination IP	Protocol	Number of Segments	Average Payload Length
...	...	TCP	...	...
...	...	UDP	...	...

My Solution Approach and Use of Data Structure:

We are using the sniff function to capture packets. We are instructed to capture 10000 packets. Since We are using online capture, we could not use threading to make the process faster. Anyway, we are storing the captured packets and it is stored in PacketList Object of scapy. Captured packets needed to be stored as we are to process those packets further to meet the requirements of the given problem statement.

To count the number of Distinct Host IP addresses, We are using a Set() Data Structure. We are iterating through the PacketList and adding any unique Host IP to the Set() we could find. Finally we displayed the count (length of set) and displayed the IPs.

For constructing the table we need some additional processing. We are to extract the information - Source IP, Destination IP, Protocol, Number of Segments, Average Payload Length. Now, We propose to use a Dictionary() Data Structure to store the values of individual packets and the related information. In this Data Structure, We are to use the tuple of Source IP, Destination IP, and Protocol as key (Why tuple? Because it is immutable) and for value we are using a list of data that looks like this - [Number_of_Packets, [len(Packet1), len(Packer2), ...], Average_Payload_Length]

Example :
Distinct_Pair = {

(192.168.0.105, 192.168.0.108, 'TCP') : [5, [0, 0, 3, 3, 0], 1.20]

}

Then we displayed the Dictionary structured as given in the Desired Output.

Source Code :

```
# Import the necessary module from Scapy for packet sniffing
from scapy.all import sniff

# Ask the user for the number of packets to capture
NUMBER_COUNT = int(input('How many packets to capture?'))

# Use Scapy's sniff function to capture packets based on the specified count and filter
CAPTURED_Packets = sniff(count=NUMBER_COUNT, filter='tcp or udp or icmp')

# Create an empty set to store distinct source IP addresses
Distinct_Hosts = set()

# Iterate through captured packets and try to extract the source IP address because ARP packets
may exist.
for Packets in CAPTURED_Packets:
    try:
        if Packets['IP'].src not in Distinct_Hosts:
            Distinct_Hosts.add(Packets['IP'].src)
    except:
        pass

# Print the count of distinct source IP addresses and the IP addresses themselves
print('COUNT OF DISTINCT HOSTS : ', len(Distinct_Hosts))
print('THEY ARE ', Distinct_Hosts)



# Define a dictionary that maps IP protocol numbers to protocol names
Proto_Dictionary = {
```

```
    6: 'TCP',
    17: 'UDP',
    1: 'ICMP'
}

# Create a dictionary to store information about distinct pairs of source IP, destination IP, and
protocol
Distinct_Pair = {}

# Iterate through captured packets and try to extract relevant information
for Packet in CAPTURED_Packets:
    try:
        Current_IP_src = Packet['IP'].src
        Current_IP_dst = Packet['IP'].dst
        Current_IP_proto = Proto_Dictionary[Packet['IP'].proto]

        # Check if the pair is already in the dictionary; if not, add it
        if (Current_IP_src, Current_IP_dst, Current_IP_proto) not in Distinct_Pair:
            Distinct_Pair[(Current_IP_src, Current_IP_dst, Current_IP_proto)] = [1,
[len(Packet[Current_IP_proto].payload)], 0]
        else:
            # Update the count and payload length list for the existing pair
            Distinct_Pair[(Current_IP_src, Current_IP_dst, Current_IP_proto)][0] += 1
            Distinct_Pair[(Current_IP_src, Current_IP_dst,
Current_IP_proto)][1].append(len(Packet[Current_IP_proto].payload))
    except:
        pass

# Calculate the average payload length for each distinct pair
for Pair in Distinct_Pair:
    Distinct_Pair[Pair][2] = sum(Distinct_Pair[Pair][1]) / Distinct_Pair[Pair][0]

# Print a table header
print('The Table : ')
print('Source Ip | Destination Ip | Protocol | Number of Segments | Average Payload Length')

# Iterate through distinct pairs and print their information
for PAIR in Distinct_Pair:
    print(PAIR[0], ' | ', PAIR[1], ' | ', PAIR[2], ' | ', Distinct_Pair[PAIR][0], ' | ', Distinct_Pair[PAIR][2])
```

Sample Run :

```
devfrost@penguin:~/Workspace/JU_SUBMISSIONS/Semester 3/Computer_Network/Assignment4$ sudo python3 1_2_3.py
How many packets to capture?10000
COUNT OF DISTINCT HOSTS :  18
THEY ARE  {'20.189.172.33', '13.107.42.18', '100.115.92.193', '20.189.173.5', '13.107.246.58', '140.82.113.22', '20.189.172.32', '100.1
15.92.25', '13.107.5.93', '20.250.58.93', '52.182.143.210', '140.82.112.21', '52.182.143.209', '13.78.111.199', '140.82.114.22', '20.20
7.73.85', '100.115.92.199', '20.207.73.82'}
The Table :
Source Ip | Destination Ip | Protocol | Number of Segments | Average Payload Length
100.115.92.193  |  224.0.0.251  |  UDP  |  6920  |  59.58034682080925
100.115.92.193  |  239.255.255.250  |  UDP  |  884  |  289.3608597285068
100.115.92.199  |  100.115.92.193  |  UDP  |  81  |  45.370370370370374
100.115.92.193  |  100.115.92.199  |  UDP  |  81  |  162.14814814814815
100.115.92.199  |  20.207.73.85  |  TCP  |  51  |  88.80392156862744
20.207.73.85  |  100.115.92.199  |  TCP  |  45  |  437.55555555555554
100.115.92.199  |  20.250.58.93  |  TCP  |  66  |  291.8939393939394
20.250.58.93  |  100.115.92.199  |  TCP  |  58  |  205.08620689655172
100.115.92.199  |  140.82.112.21  |  TCP  |  22  |  412.45454545454544
140.82.112.21  |  100.115.92.199  |  TCP  |  22  |  413.27272727272725
100.115.92.199  |  13.107.5.93  |  TCP  |  116  |  96.31896551724138
13.107.5.93  |  100.115.92.199  |  TCP  |  116  |  507.8189655172414
100.115.92.199  |  169.254.169.254  |  TCP  |  8  |  0.0
100.115.92.199  |  20.189.172.33  |  TCP  |  20  |  181.25
20.189.172.33  |  100.115.92.199  |  TCP  |  13  |  777.6923076923077
100.115.92.199  |  20.189.173.4  |  TCP  |  8  |  0.0
100.115.92.199  |  13.107.246.58  |  TCP  |  17  |  63.529411764705884
13.107.246.58  |  100.115.92.199  |  TCP  |  15  |  528.6666666666666
100.115.92.199  |  13.107.42.18  |  TCP  |  66  |  42.696969696969695
13.107.42.18  |  100.115.92.199  |  TCP  |  67  |  3220.0298507462685
100.115.92.199  |  52.182.143.210  |  TCP  |  336  |  1308.1339285714287
52.182.143.210  |  100.115.92.199  |  TCP  |  409  |  172.64058679706602
100.115.92.199  |  20.207.73.82  |  TCP  |  64  |  57.1875
20.207.73.82  |  100.115.92.199  |  TCP  |  53  |  235.03773584905662
100.115.92.199  |  140.82.113.22  |  TCP  |  21  |  135.0952380952381
140.82.113.22  |  100.115.92.199  |  TCP  |  19  |  593.6315789473684
100.115.92.199  |  20.189.172.32  |  TCP  |  20  |  171.1
20.189.172.32  |  100.115.92.199  |  TCP  |  14  |  717.7142857142857
100.115.92.199  |  52.182.143.209  |  TCP  |  8  |  439.625
52.182.143.209  |  100.115.92.199  |  TCP  |  6  |  838.1666666666666
100.115.92.199  |  140.82.114.22  |  TCP  |  25  |  290.72
140.82.114.22  |  100.115.92.199  |  TCP  |  23  |  394.95652173913044
100.115.92.25  |  100.115.92.199  |  ICMP  |  6  |  40.0
100.115.92.199  |  13.78.111.199  |  TCP  |  15  |  236.73333333333332
13.78.111.199  |  100.115.92.199  |  TCP  |  14  |  489.0
100.115.92.199  |  20.189.173.5  |  TCP  |  8  |  312.125
20.189.173.5  |  100.115.92.199  |  TCP  |  6  |  838.1666666666666
```

Problem Statement 4: For each distinct quadruple of source/destination host IP addresses and source/destination port numbers determine the number of TCP/UDP segments exchanged and also the average payload length.

Desired Output:

Source IP	Destination IP	Protocol	Source Port	Destination Port	Number of Segments	Average Payload Length
...	...	TCP	...	...	...	...
...	...	UDP	...	...	...	...

My Solution Approach and Use of Data Structure: We have handled a similar problem statement (Question 3), Now we are to just modify this one and add additional Source Port and Destination Port to the Table and then we will get our solution. The tweaks we made to the previous code was that the tuple which we considered as key to the Dictionary Data Structure, was modified to encapsulate the existing data along with the two new fields (i.e Source Port and Destination Port). That is all we did to achieve the desired output. And yes a little bit of formatting was done.

Source Code :

```
# Import the necessary module from Scapy for packet sniffing
from scapy.all import sniff

# Ask the user for the number of packets to capture
NUMBER_COUNT = int(input('How many packets to capture?'))

# Use Scapy's sniff function to capture packets based on the specified count and filter
CAPTURED_Packets = sniff(count=NUMBER_COUNT, filter='tcp or udp or icmp')

# Define a dictionary that maps IP protocol numbers to protocol names
Proto_Dictionary = {
    6: 'TCP',
    17: 'UDP',
    1: 'ICMP'
}

# Print a message for Question 4
print('\n\nQuestion 4 : ')
```

```
# Create a dictionary to store information about distinct pairs of source IP, source port,
destination IP, destination port, and protocol
Distinct_Pair = {}

# Iterate through captured packets and try to extract relevant information
for Packet in CAPTURED_Packets:
    try:
        Current_IP_src = Packet['IP'].src
        Current_IP_dst = Packet['IP'].dst
        Current_IP_proto = Proto_Dictionary[Packet['IP'].proto]
        Current_IP_sport = Packet[Current_IP_proto].sport
        Current_IP_dport = Packet[Current_IP_proto].dport

        # Check if the 5-tuple (source IP, source port, destination IP, destination port, protocol) is
already in the dictionary; if not, add it
        if (Current_IP_src, Current_IP_sport, Current_IP_dst, Current_IP_dport, Current_IP_proto) not
in Distinct_Pair:
            Distinct_Pair[(Current_IP_src, Current_IP_sport, Current_IP_dst, Current_IP_dport,
Current_IP_proto)] = [1, [len(Packet[Current_IP_proto].payload)], 0]
        else:
            # Update the count and payload length list for the existing 5-tuple
            Distinct_Pair[(Current_IP_src, Current_IP_sport, Current_IP_dst, Current_IP_dport,
Current_IP_proto)][0] += 1
            Distinct_Pair[(Current_IP_src, Current_IP_sport, Current_IP_dst, Current_IP_dport,
Current_IP_proto)][1].append(len(Packet[Current_IP_proto].payload))
    except:
        pass

# Calculate the average payload length for each distinct 5-tuple
for Pair in Distinct_Pair:
    Distinct_Pair[Pair][2] = sum(Distinct_Pair[Pair][1]) / Distinct_Pair[Pair][0]

# Print a table header
print("The Table : ")
print("Source IP | Destination IP | Protocol | Source Port | Destination Port | Number of
Segments | Average Payload Length")

# Iterate through distinct 5-tuples and print their information
for Data in Distinct_Pair:
    print(Data[0],'|', Data[2],'|', Data[4],'|', Data[1],'|', Data[3],'|', Distinct_Pair[Data][0],'|',
Distinct_Pair[Data][2])
```

Sample Run :

```
devfrost@penguin:~/Workspace/JU_SUBMISSIONS/Semester 3/Computer_Network/Assignment4$ sudo python3 4.py
How many packets to capture?1000


Question 4 :
The Table :
Source IP | Destination IP | Protocol | Source Port | Destination Port | Number of Segments | Average Payload Length
100.115.92.193 | 239.255.255.250 | UDP | 1900 | 1900 | 95 | 304.0105263157895
100.115.92.199 | 100.115.92.193 | UDP | 58904 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 58904 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 44991 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 44991 | 1 | 85.0
100.115.92.193 | 224.0.0.251 | UDP | 5353 | 5353 | 640 | 50.975
100.115.92.199 | 100.115.92.193 | UDP | 53320 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 53320 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 56087 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 56087 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 52257 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 52257 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 60024 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 60024 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 56546 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 56546 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 51557 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 51557 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 49319 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 49319 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 54396 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 54396 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 35076 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 35076 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 44441 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 44441 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 55618 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 55618 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 55575 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 55575 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 45592 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 45592 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 43159 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 43159 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 38929 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 38929 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 35139 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 35139 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 57441 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 57441 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 42770 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 42770 | 1 | 85.0
```

```
100.115.92.199 | 100.115.92.193 | UDP | 42770 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 42770 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 43575 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 43575 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 51808 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 51808 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 42603 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 42603 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 35849 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 35849 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 41126 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 41126 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 49491 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 49491 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 36100 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 36100 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 56807 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 56807 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 40556 | 53 | 2 | 28.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 40556 | 2 | 79.5
100.115.92.199 | 100.115.92.193 | UDP | 41599 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 41599 | 1 | 85.0
100.115.92.199 | 20.207.73.82 | TCP | 52034 | 443 | 21 | 58.095238095238095
20.207.73.82 | 100.115.92.199 | TCP | 443 | 52034 | 17 | 244.05882352941177
100.115.92.199 | 100.115.92.193 | UDP | 47124 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 47124 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 36470 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 36470 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 50466 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 50466 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 56141 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 56141 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 42624 | 53 | 1 | 50.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 42624 | 1 | 176.0
100.115.92.199 | 40.74.98.194 | TCP | 50834 | 443 | 15 | 384.8
40.74.98.194 | 100.115.92.199 | TCP | 443 | 50834 | 14 | 492.85714285714283
100.115.92.199 | 100.115.92.193 | UDP | 54188 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 54188 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 33210 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 33210 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 39207 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 39207 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 53173 | 53 | 1 | 46.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 53173 | 1 | 85.0
100.115.92.199 | 100.115.92.193 | UDP | 58750 | 53 | 2 | 50.0
100.115.92.193 | 100.115.92.199 | UDP | 53 | 58750 | 2 | 203.5
100.115.92.199 | 20.42.65.84 | TCP | 53722 | 443 | 12 | 314.6666666666667
20.42.65.84 | 100.115.92.199 | TCP | 443 | 53722 | 9 | 558.5555555555555
devfrost@penguin:~/Workspace/JU_SUBMISSIONS/Semester 3/Computer_Network/Assignment4$
```

As the list is too large, We decided to run the program while capturing only 1000 packets.

Assignment 5

We have created a python package named 'Additional_Package' with all the additional functions with redundant processing.

Problem Statement 1: Write a python program which gets a network address from the user and generates all possible host IP addresses within that network. Then it sends a dummy ICMP echo request message to all the hosts. Display only those hosts from which you receive corresponding ICMP echo reply messages within a predefined time out period.

My solution approach and Data Structure used: The program takes an USER input of Network Address.

The program expects the user to input like : Network_Address/Network_Bits
Example: 192.168.0.1/25

Steps:

1. Once input taken, The input is processed to extract the Network Address and Network Bits.
2. Now the Subnet is generated using the Network Bits.
3. Once Subnet is acquired, It is taken on a bitwise AND operation with the Network Address provided to get the Network ID.
4. Now using the Network Bits total number of possible Host IDs is calculated.
5. By the number of total possible Hosts, Every host ID is generated and stored in a list.
6. Then the list is iterated pinging a echo request using srloop with count 1.
7. Then we are to add the corresponding host to a set if the packet received contains Result['ICMP'].type = 0. (flag type == 0 means that echo ping is successful)
8. Finally printing the set to display the Alive hosts on the network.

Source Code :

Additional_Functions.py

```
import copy
from scapy.all import IP, TCP, ICMP, srloop, sr1

# Function for sending a TCP SYN packet and checking if the host is alive
def TCP_SEND(Host, Port, Alive_Hosts):
    # Create a TCP SYN packet
    Packet = IP(dst=Host) / TCP(dport=Port, flags="S")

    # Send the packet and wait for a response (SYN-ACK)
    Response = sr1(Packet, timeout=1)

    # If a response is received and it has the SYN-ACK flags set, mark the host as alive
    if Response != None and Response['TCP'].flags == "SA":
        print(Response['TCP'].flags)
        Alive_Hosts.add((Host, Port))

# Function for sending an ICMP echo request (ping) and checking if the host responds
def Echo_Ping(Host_Address):
    # Create an ICMP echo request packet
    Packet = IP(dst=Host_Address) / ICMP() / "Hello"

    # Send the packet and check for a response within the timeout
    Result = srloop(Packet, count=1, timeout=1)[0]

    # If a response is received and it has ICMP type 0 (echo reply), consider the host alive
    if len(Result) == 0:
        return False
    elif Result[0][1]['ICMP'].type == 0:
        return True
    else:
        return False

# Function to convert a decimal number to binary as an integer
def NumToBin(num):
    bin = ''
    while num >= 2:
        bin += str(num % 2)
        num = num // 2
    bin += str(num)
    return int(bin[::-1])

# Function to parse a network tuple in the format "IP/NetMask"
def Network_Tuple(incoming):
    incoming = incoming.replace(" ", "")
```

```
    incoming = incoming.split("/")
    return (incoming[0], int(incoming[1]))

# Function to split an IP address string into octets and convert them to integers
def Octates(Address):
    Oct = Address.split(".")
    for i in range(len(Oct)):
        Oct[i] = int(Oct[i])
    return Oct

# Function to calculate the wildcard mask for a given set of octets
def Wild_Card(Octates):
    Result = []
    for Octate in Octates:
        Result.append(Octate ^ 255)
    return Result

# Function to determine the IP address class based on the first octet
def Return_Class(Network_Address):
    Octates = Octates(Network_Address)
    FOctate = int(Octates[0])
    if FOctate < 128:
        return "A"
    elif FOctate < 192:
        return "B"
    elif FOctate < 224:
        return "C"
    elif FOctate < 240:
        return "D"
    elif FOctate < 256:
        return "E"
    else:
        return None

# Function to generate a subnet mask as a list of octets
def Subnet_Generator(Network_Bits):
    Host_Bits = 32 - Network_Bits
    Host_Octates = Host_Bits // 8
    build = []
    Net_Octates = Network_Bits // 8
    Network_Bits = Network_Bits % 8
    Limit = 7
    build += [255] * Net_Octates
    if Network_Bits != 0:
        S = 0
        while True:
            if Network_Bits != 0:
                S += 2**(Limit)
```

```
            Network_Bits -= 1
            Limit -= 1
        else:
            break
    build.append(S)
build += [0] * Host_Octates
return build

# Function to calculate the network ID by applying a subnet mask
def Network_ID(Network_Address, Network_Bits):
    Net_ID = []
    Net_Octates = Octates(Network_Address)
    Sub_Octates = Subnet_Generator(Network_Bits)
    for i in range(len(Net_Octates)):
        Net_ID.append(Net_Octates[i] & Sub_Octates[i])
    return Net_ID

# Function to calculate the total number of host IP addresses in a subnet
def Total_Hosts(Network_Bits):
    Host_Bits = 32 - Network_Bits
    Host = 0
    while Host_Bits != 0:
        Host += 2**(Host_Bits - 1)
        Host_Bits -= 1
    return Host - 1

# Function to calculate the broadcast address by applying a wildcard mask to the network ID
def Broadcast_ID(Network_Address, Network_Bits):
    Broad_ID = []
    Net_ID = Network_ID(Network_Address, Network_Bits)
    WildCard = Wild_Card(Subnet_Generator(Network_Bits))
    for i in range(len(Net_ID)):
        Broad_ID.append(Net_ID[i] | WildCard[i])
    return Broad_ID

# Function to generate a list of host IP addresses within a subnet
def Host_IP_Generator(Network_Address, Network_Bits):
    Result = []
    Net_ID = Network_ID(Network_Address, Network_Bits)
    Total_Host = Total_Hosts(Network_Bits)
    j = 3
    for i in range(1, Total_Host + 1):
        if Net_ID[j] < 255:
            Net_ID[j] += 1
        else:
            Net_ID[j] = 0
            j -= 1
            if Net_ID[j] < 255:
```

```
                Net_ID[j] += 1
            else:
                Net_ID[j] = 0
                j -= 1
                if Net_ID[j] < 255:
                    Net_ID[j] += 1
                else:
                    Net_ID[j] = 0
                    j -= 1
                    Net_ID[j] += 1
                    j += 1
                j += 1
            j += 1
        Result.append(Strfy(copy.copy(Net_ID)))
    return Result

# Function to convert a list of octets representing an IP address back to a string format
def Strfy(IPV4):
    temp = ""
    for Oct in IPV4:
        temp += "." + str(Oct)
    return temp[1:]

# Function to convert a list of IP addresses represented as lists of octets to string format
def Data_Stringyfy(Result):
    for i in range(len(Result)):
        Result[i] = Strfy(Result[i])
    return Result
```

1.py

```
# Import necessary Scapy modules
from scapy.all import IP, ICMP, srp1

# Import additional functions from the "Additional_Functions" module
import Additional_Functions as AF

# Ask the user to input a network address in the format "IP/NetMask"
USER_INPUT = input("Enter the Network Address: ")

# Parse the user input to extract the network address and network bits
Network_Address, Network_Bits = AF.Network_Tuple(USER_INPUT)

# Calculate the network ID, subnet mask, broadcast ID, total hosts, and generated host addresses
Network_ID = AF.Strfy(AF.Network_ID(Network_Address, Network_Bits))
Subnet_Mask = AF.Strfy(AF.Subnet_Generator(Network_Bits))
Broadcast_ID = AF.Strfy(AF.Broadcast_ID(Network_Address, Network_Bits))
```

```python
Total_Hosts = AF.Total_Hosts(Network_Bits)
Generated_Host_Addresses = AF.Host_IP_Generator(Network_Address, Network_Bits)

# Print the network information
print(Network_Address, Network_Bits)
print(Network_ID, Subnet_Mask, Broadcast_ID, Total_Hosts, Generated_Host_Addresses)

# Create a set to store alive hosts
Alive_Hosts = set()

# Display processing message
print("Processing...")

# Send echo request (ping) to all generated host addresses and check for responses
print("Sending echo request to all hosts...")
for Host in Generated_Host_Addresses:
    print("Ping to", Host, " : ")

    # Check if the host responds to the echo request and add it to the set of alive hosts
    if AF.Echo_Ping(Host):
        Alive_Hosts.add(Host)

# Print the list of alive hosts
print("These Hosts are alive: ", Alive_Hosts)
```

Sample Run :

```
devfrost@penguin:~/Workspace/JU_SUBMISSIONS/Semester 3/Computer_Network/Assignment5$ sudo python3 1.py
Enter the Network Address : 192.168.0.1/25
192.168.0.1 25
192.168.0.0 255.255.255.128 192.168.0.127 126 ['192.168.0.1', '192.168.0.2', '192.168.0.3', '192.168.0.4', '192.168.0.5', '192.168.0.6'
, '192.168.0.7', '192.168.0.8', '192.168.0.9', '192.168.0.10', '192.168.0.11', '192.168.0.12', '192.168.0.13', '192.168.0.14', '192.168
.0.15', '192.168.0.16', '192.168.0.17', '192.168.0.18', '192.168.0.19', '192.168.0.20', '192.168.0.21', '192.168.0.22', '192.168.0.23',
 '192.168.0.24', '192.168.0.25', '192.168.0.26', '192.168.0.27', '192.168.0.28', '192.168.0.29', '192.168.0.30', '192.168.0.31', '192.1
68.0.32', '192.168.0.33', '192.168.0.34', '192.168.0.35', '192.168.0.36', '192.168.0.37', '192.168.0.38', '192.168.0.39', '192.168.0.40
', '192.168.0.41', '192.168.0.42', '192.168.0.43', '192.168.0.44', '192.168.0.45', '192.168.0.46', '192.168.0.47', '192.168.0.48', '192
.168.0.49', '192.168.0.50', '192.168.0.51', '192.168.0.52', '192.168.0.53', '192.168.0.54', '192.168.0.55', '192.168.0.56', '192.168.0.
57', '192.168.0.58', '192.168.0.59', '192.168.0.60', '192.168.0.61', '192.168.0.62', '192.168.0.63', '192.168.0.64', '192.168.0.65', '1
92.168.0.66', '192.168.0.67', '192.168.0.68', '192.168.0.69', '192.168.0.70', '192.168.0.71', '192.168.0.72', '192.168.0.73', '192.168.
0.74', '192.168.0.75', '192.168.0.76', '192.168.0.77', '192.168.0.78', '192.168.0.79', '192.168.0.80', '192.168.0.81', '192.168.0.82',
'192.168.0.83', '192.168.0.84', '192.168.0.85', '192.168.0.86', '192.168.0.87', '192.168.0.88', '192.168.0.89', '192.168.0.90', '192.16
8.0.91', '192.168.0.92', '192.168.0.93', '192.168.0.94', '192.168.0.95', '192.168.0.96', '192.168.0.97', '192.168.0.98', '192.168.0.99'
, '192.168.0.100', '192.168.0.101', '192.168.0.102', '192.168.0.103', '192.168.0.104', '192.168.0.105', '192.168.0.106', '192.168.0.107
', '192.168.0.108', '192.168.0.109', '192.168.0.110', '192.168.0.111', '192.168.0.112', '192.168.0.113', '192.168.0.114', '192.168.0.11
5', '192.168.0.116', '192.168.0.117', '192.168.0.118', '192.168.0.119', '192.168.0.120', '192.168.0.121', '192.168.0.122', '192.168.0.1
23', '192.168.0.124', '192.168.0.125', '192.168.0.126']
Processing...
Sending echo request to all hosts...
Ping to 192.168.0.1  :
RECV 1: IP / ICMP 192.168.0.1 > 100.115.92.199 echo-reply 0 / Raw

Sent 1 packets, received 1 packets. 100.0% hits.
Ping to 192.168.0.2  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.2 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.3  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.3 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.4  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.4 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.5  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.5 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.6  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.6 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.7  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.7 echo-request 0 / Raw
```

…………………………………………………………………………………………

```
Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.116  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.116 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.117  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.117 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.118  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.118 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.119  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.119 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.120  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.120 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.121  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.121 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.122  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.122 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.123  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.123 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.124  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.124 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.125  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.125 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
Ping to 192.168.0.126  :
fail 1: IP / ICMP 100.115.92.199 > 192.168.0.126 echo-request 0 / Raw

Sent 1 packets, received 0 packets. 0.0% hits.
These Hosts are alive :  {'192.168.0.103', '192.168.0.100', '192.168.0.102', '192.168.0.101', '192.168.0.1'}
```

This is my home network and I found these Hosts alive. 192.168.0.1 is my router, 192.168.0.103 is my Laptop, 192.168.0.100 is my Google Home, 192.168.0.102 is my Printer, and 192.168.0.101 is my NAS. However, My phone is connected to the router and is on 192.168.0.104 and doesn't seem to respond against echo replies. The issue is unknown. Only my phone is not detected in this sample run.

Problem Statement 2: Write a python program which gets a host IP address from the user and sends TCP SYN segments to all
the ports within the range 0 to 1023. Display only those port numbers from which you receive corresponding TCP SYN+ACK segments within a predefined time out period.

My solution approach and Data Structure used: The program takes an USER input of Network Address.

The program expects the user to input like : 192.168.0.1

Steps:
We are to design a function that sends a TCP packet with SYN flag set expecting a SYN-ACK packet in return. Which helps to determine if the port is open.

1. Crafting the packet :

IP(dst = HOST_IP_ADDRESS) / TCP(dport = DESTINATION_PORT, flags = 'S')
Here 'S' means the flag 'SYN'.

2. Using the sr1 function of scapy to send the packet with 1 sec timeout.
3. We are expecting a package in return with flags = 'SA'. flag 'SA' means SYN-ACK. Which will help us to determine if the port is open or not. If we get other flags like RA, F and so on. We will not acknowledge the package.
4. Now our main script is designed to split the large number of ports into sections and using thread we are to make the process faster. Since every packet sending requires 1 sec of timeout(It may or may not take a full 1 sec), sending packets one by one must be time consuming.
5. Now if a valid alive port is found, it is added to a Set() data structure for displaying at last.

Source Code :

Additional_Functions.py

```
import copy
from scapy.all import IP, TCP, ICMP, srloop, sr1

# Function for sending a TCP SYN packet and checking if the host is alive
def TCP_SEND(Host, Port, Alive_Hosts):
    # Create a TCP SYN packet
    Packet = IP(dst=Host) / TCP(dport=Port, flags="S")

    # Send the packet and wait for a response (SYN-ACK)
    Response = sr1(Packet, timeout=1)

    # If a response is received and it has the SYN-ACK flags set, mark the host as alive
    if Response != None and Response['TCP'].flags == "SA":
        print(Response['TCP'].flags)
        Alive_Hosts.add((Host, Port))

# Function for sending an ICMP echo request (ping) and checking if the host responds
def Echo_Ping(Host_Address):
    # Create an ICMP echo request packet
    Packet = IP(dst=Host_Address) / ICMP() / "Hello"

    # Send the packet and check for a response within the timeout
    Result = srloop(Packet, count=1, timeout=1)[0]

    # If a response is received and it has ICMP type 0 (echo reply), consider the host alive
    if len(Result) == 0:
        return False
    elif Result[0][1]['ICMP'].type == 0:
        return True
    else:
        return False

# Function to convert a decimal number to binary as an integer
def NumToBin(num):
    bin = ''
    while num >= 2:
        bin += str(num % 2)
        num = num // 2
    bin += str(num)
    return int(bin[::-1])

# Function to parse a network tuple in the format "IP/NetMask"
def Network_Tuple(incoming):
    incoming = incoming.replace(" ", "")
```

```
    incoming = incoming.split("/")
    return (incoming[0], int(incoming[1]))

# Function to split an IP address string into octets and convert them to integers
def Octates(Address):
    Oct = Address.split(".")
    for i in range(len(Oct)):
        Oct[i] = int(Oct[i])
    return Oct

# Function to calculate the wildcard mask for a given set of octets
def Wild_Card(Octates):
    Result = []
    for Octate in Octates:
        Result.append(Octate ^ 255)
    return Result

# Function to determine the IP address class based on the first octet
def Return_Class(Network_Address):
    Octates = Octates(Network_Address)
    FOctate = int(Octates[0])
    if FOctate < 128:
        return "A"
    elif FOctate < 192:
        return "B"
    elif FOctate < 224:
        return "C"
    elif FOctate < 240:
        return "D"
    elif FOctate < 256:
        return "E"
    else:
        return None

# Function to generate a subnet mask as a list of octets
def Subnet_Generator(Network_Bits):
    Host_Bits = 32 - Network_Bits
    Host_Octates = Host_Bits // 8
    build = []
    Net_Octates = Network_Bits // 8
    Network_Bits = Network_Bits % 8
    Limit = 7
    build += [255] * Net_Octates
    if Network_Bits != 0:
        S = 0
        while True:
            if Network_Bits != 0:
                S += 2**(Limit)
```

```
            Network_Bits -= 1
            Limit -= 1
        else:
            break
    build.append(S)
build += [0] * Host_Octates
return build

# Function to calculate the network ID by applying a subnet mask
def Network_ID(Network_Address, Network_Bits):
    Net_ID = []
    Net_Octates = Octates(Network_Address)
    Sub_Octates = Subnet_Generator(Network_Bits)
    for i in range(len(Net_Octates)):
        Net_ID.append(Net_Octates[i] & Sub_Octates[i])
    return Net_ID

# Function to calculate the total number of host IP addresses in a subnet
def Total_Hosts(Network_Bits):
    Host_Bits = 32 - Network_Bits
    Host = 0
    while Host_Bits != 0:
        Host += 2**(Host_Bits - 1)
        Host_Bits -= 1
    return Host - 1

# Function to calculate the broadcast address by applying a wildcard mask to the network ID
def Broadcast_ID(Network_Address, Network_Bits):
    Broad_ID = []
    Net_ID = Network_ID(Network_Address, Network_Bits)
    WildCard = Wild_Card(Subnet_Generator(Network_Bits))
    for i in range(len(Net_ID)):
        Broad_ID.append(Net_ID[i] | WildCard[i])
    return Broad_ID

# Function to generate a list of host IP addresses within a subnet
def Host_IP_Generator(Network_Address, Network_Bits):
    Result = []
    Net_ID = Network_ID(Network_Address, Network_Bits)
    Total_Host = Total_Hosts(Network_Bits)
    j = 3
    for i in range(1, Total_Host + 1):
        if Net_ID[j] < 255:
            Net_ID[j] += 1
        else:
            Net_ID[j] = 0
            j -= 1
            if Net_ID[j] < 255:
```

```
            Net_ID[j] += 1
        else:
            Net_ID[j] = 0
            j -= 1
            if Net_ID[j] < 255:
                Net_ID[j] += 1
            else:
                Net_ID[j] = 0
                j -= 1
                Net_ID[j] += 1
                j += 1
            j += 1
        j += 1
    Result.append(Strfy(copy.copy(Net_ID)))
  return Result

# Function to convert a list of octets representing an IP address back to a string format
def Strfy(IPV4):
  temp = ""
  for Oct in IPV4:
    temp += "." + str(Oct)
  return temp[1:]

# Function to convert a list of IP addresses represented as lists of octets to string format
def Data_Stringyfy(Result):
  for i in range(len(Result)):
    Result[i] = Strfy(Result[i])
  return Result
```

2.py

```
# Import necessary modules
from scapy.all import IP, TCP
from time import sleep
import threading
import Additional_Functions as AF

# Create an empty set to store alive ports
Alive_Ports = set()

# Get user input for the network address to scan
USER_INPUT = input("Enter the Network Address : ")

# Initialize variables for port range
i, j = 0, 100

# Define a function to perform port scanning within a range
def Process(i, j):
```

```
    for i in range(i, j):
        # Call a function from Additional_Functions to send TCP packets and check if a port is alive
        AF.TCP_SEND(USER_INPUT, i, Alive_Ports)

# Perform port scanning in multiple threads
while i < 1024 and j < 1024:
    # Start a new thread for port scanning within the range [i, j)
    threading.Thread(target=Process, args=(i, j)).start()

    # Update the range for the next thread
    i += 100
    if j + 100 > 1024:
        j = 1024
    else:
        j += 100

# Sleep for 15 seconds to allow threads to finish scanning
sleep(15)

# Print the total alive ports after scanning is complete
print("Total Alive Ports : ")
print(Alive_Ports)
```

Sample Run :

```
devfrost@penguin:~/Workspace/JU_SUBMISSIONS/Semester 3/Computer_Network/Assignment5$ sudo python3 2.py
Enter the Network Address : 192.168.0.1




Received 2 packets, got 1 answers, remaining 0 packets
*
Received 1 packets, got 1 answers, remaining 0 packets
*Begin emission:
Begin emission:
Begin emission:

Received 3 packets, got 1 answers, remaining 0 packets
Finished sending 1 packets.
Finished sending 1 packets.
Finished sending 1 packets.
...**
Received 3 packets, got 1 answers, remaining 0 packets
*
Received 2 packets, got 1 answers, remaining 0 packets

Received 1 packets, got 1 answers, remaining 0 packets
Begin emission:
Begin emission:
Begin emission:
Finished sending 1 packets.
Finished sending 1 packets.
Finished sending 1 packets.
*..*.*
Received 2 packets, got 1 answers, remaining 0 packets

Received 3 packets, got 1 answers, remaining 0 packets

Received 1 packets, got 1 answers, remaining 0 packets
Begin emission:
Finished sending 1 packets.
Begin emission:
Finished sending 1 packets.
.Begin emission:
*Finished sending 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
*
Received 2 packets, got 1 answers, remaining 0 packets
.*
Received 2 packets, got 1 answers, remaining 0 packets
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Total Alive Ports :
{('192.168.0.1', 22), ('192.168.0.1', 80), ('192.168.0.1', 53)}
devfrost@penguin:~/Workspace/JU_SUBMISSIONS/Semester 3/Computer_Network/Assignment5$
```