NETWORKING ASSIGNMENT – 1,2,3

Jadavpur university

Name: Arka Das
MCA 2nd Year 3rd Sem
Roll number: 002210503046
2022-2024

Assignment- 1

Question: 1

Problem Statement:

Write a TCP Day-Time server program that returns the current time and date. Also write a TCP client program that sends requests to the server to get the current time and date. Choose your own formats for the request/reply messages.

Design for request and reply:

Here we have a single server and a single client who communicates among themselves. After the server is successfully created and has been bound it waits for the client to send requests.

The client after connecting with the server can send a message to the server. But if the client sends the message "GET_TIME" only then the server will reply with the current date and time.

The server formats the current date and time as Date: dd/mm/yy, Time: hh:mm:ss and sends this as reply to the client. The server keeps running and multiple clients can connect with the server at any given time.

Source Code:

The code for both the server and the client has been written using Python language and the socket library of python is used.

Code for Server:

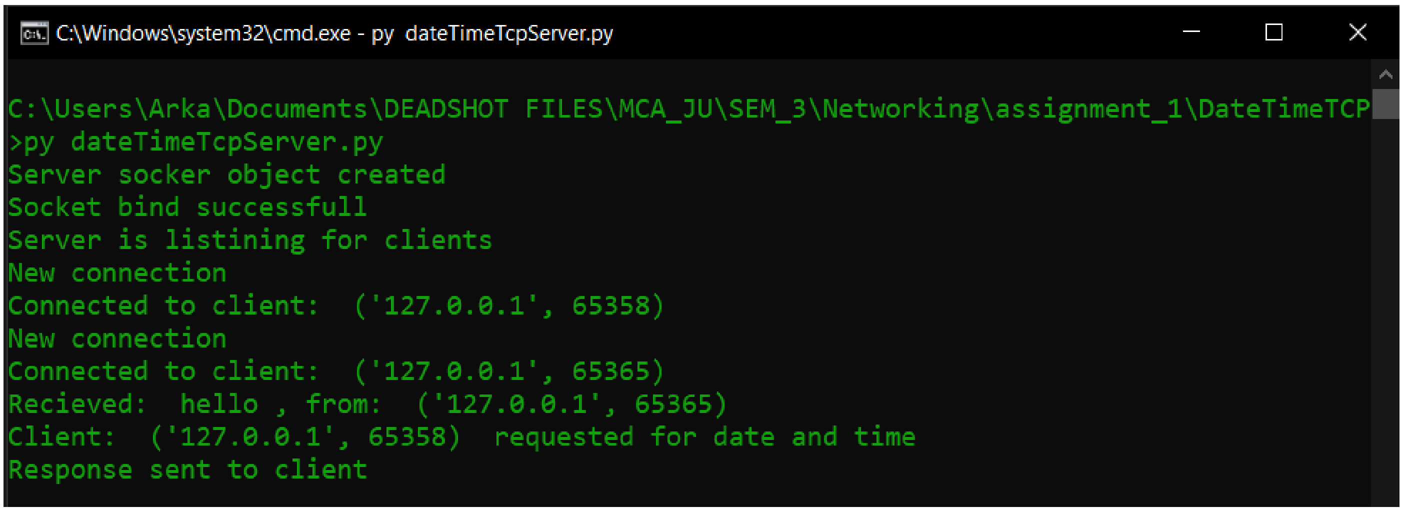
```
import socket
import threading
from datetime import datetime

ipAddr = "127.0.0.1"    #address for localhost
port = 5555

#thread for new client
def onNewClient(con, addr):
    data = con.recv(1024)
    if(data.decode() == "GET_TIME"):
        print("Client: ", addr , " requested for date and time")
        now = datetime.now()
        curr = now.strftime("Date: %d/%m/%Y, Time: %H:%M:%S")
        con.send(curr.encode()) #send this string to client
        print("Response sent to client")
    else:
        print("Recieved: ", data.decode(), ", from: ", addr)
        msg = "Different request given"
        con.send(msg.encode())
    con.close()

serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Server socker object created")
serverSocket.bind((ipAddr, port))
print("Socket bind successfull")
print("Server is listining for clients")

while True:
    serverSocket.listen(5)  #at most 5 client connection
    con, clientAddr = serverSocket.accept()
    print("New connection")
    print("Connected to client: ", clientAddr)
    t = threading.Thread(target = onNewClient, args = (con, clientAddr, ))
    t.start()

serverSocket.close()
```

Code for Client:

```
import socket

ipAddr = "127.0.0.1"
port = 5555

clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Client socker object created")
clientSocket.connect((ipAddr, port))    #connect with server with particular port
print("Connected with server")

#ask for date-time to server
message = input("Enter message: ")  #GET_TIME for time
clientSocket.send(message.encode())

#recieve date time from server
if(message == "GET_TIME"):
    print("Request for current date and time sent to server")
    currTime = clientSocket.recv(1024)
    print("Current date and time recieved from server")
    print(currTime.decode())
else:
    msg = clientSocket.recv(1024)
    print(msg.decode())

clientSocket.close()    #close connections
```

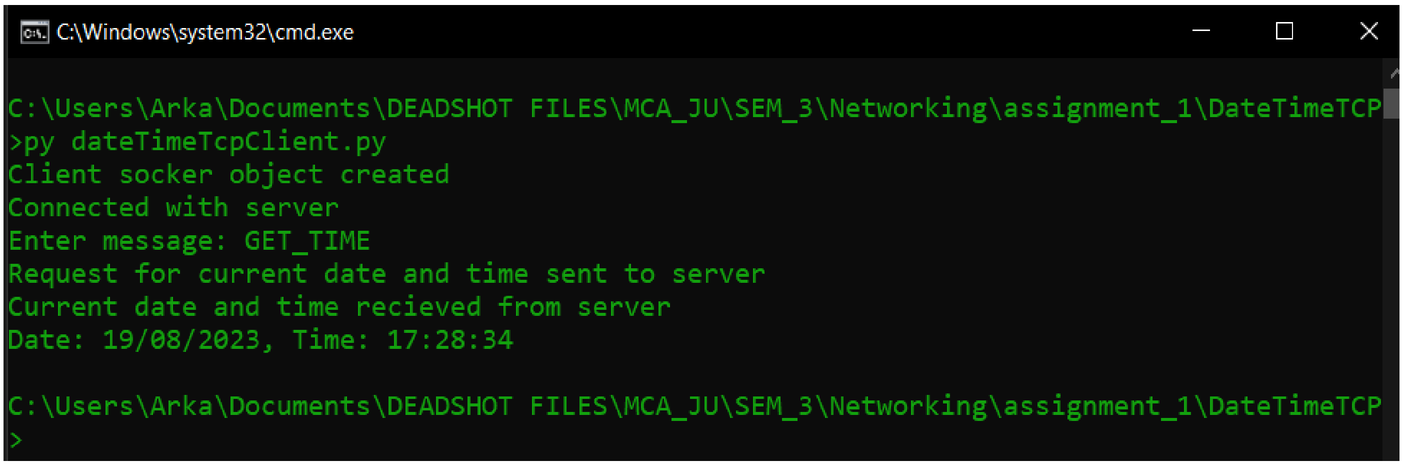
Output:

Server



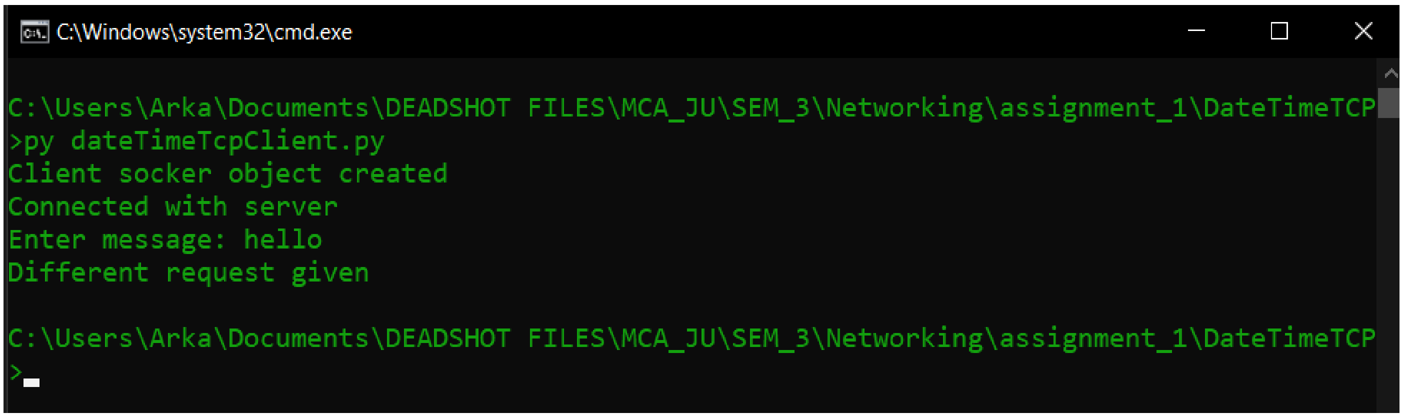
```
C:\Windows\system32\cmd.exe - py  dateTimeTcpServer.py

C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\DateTimeTCP
>py dateTimeTcpServer.py
Server socker object created
Socket bind successfull
Server is listining for clients
New connection
Connected to client:  ('127.0.0.1', 65358)
New connection
Connected to client:  ('127.0.0.1', 65365)
Recieved:  hello , from:  ('127.0.0.1', 65365)
Client:  ('127.0.0.1', 65358)  requested for date and time
Response sent to client
```

First client



```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\DateTimeTCP
>py dateTimeTcpClient.py
Client socker object created
Connected with server
Enter message: GET_TIME
Request for current date and time sent to server
Current date and time recieved from server
Date: 19/08/2023, Time: 17:28:34

C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\DateTimeTCP
>
```

Second client



```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\DateTimeTCP
>py dateTimeTcpClient.py
Client socker object created
Connected with server
Enter message: hello
Different request given

C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\DateTimeTCP
>
```

Question: 2

Problem Statement:

Write a TCP Math server program that accepts any valid integer arithmetic expression, evaluates it and returns the value of the expression. Also write a TCP client program that accepts an integer arithmetic expression from the user and sends it to the server to get the result of evaluation. Choose your own formats for the request/reply messages.

Design for request and reply:

Here we have a single server and a single client who communicates among themselves. After the server is successfully created and has been bound it waits for the client to send requests.

The client after connecting with the server can send a message to the server. Here the client can send any arithmetic expression either valid or invalid.

The server will receive the expression and will try to evaluate that expression. Now if the given expression is a valid one then the server will send the result back to the client.

If the expression is invalid then the server will catch any exception during evaluation and will send appropriate messages to the client.

The program is written in such way that server will keep running and multiple clients can connect with the server. Clients can connect and disconnect and all client operations are in isolation from other clients

Source Code:

The code for both the server and the client has been written using Python language and the socket library of python is used.

Code for Server:

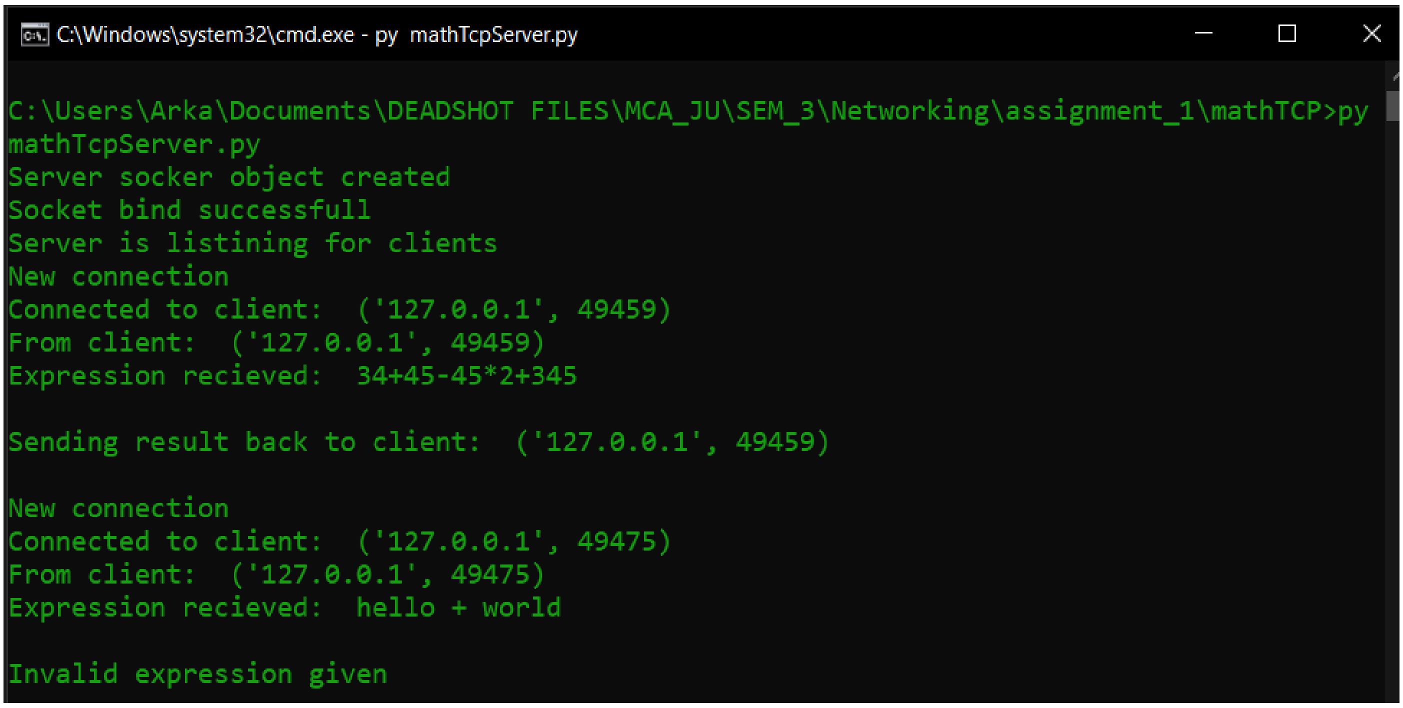
```
import socket
import threading

ipAddr = "127.0.0.1"    #address of localhost
port = 5555

def onNewThread(con, addr):
    expression = con.recv(1024).decode()    #recieve expression from client
    print("From client: ", addr)
    print("Expression recieved: ", expression, "\n")
    try:
        result = eval(expression)   #evaluate the expression
    except: #if any exception occurs
        msg = "Invalid expression given"
        print(msg, "\n")
        con.send(msg.encode())
    else:   #if no exception occurs
        print("Sending result back to client: ", addr, "\n")
        con.send(str(result).encode())
    con.close()

serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Server socker object created")
serverSocket.bind((ipAddr, port))   #bind socket with localhost and port
print("Socket bind successfull")
print("Server is listining for clients")

while True:
    serverSocket.listen(5)
    con, clientAddr = serverSocket.accept() #accept incoming connection
    print("New connection")
    print("Connected to client: ", clientAddr)
    t = threading.Thread(target=(onNewThread), args=(con, clientAddr, ))
    t.start()

serverSocket.close()
```

Code for Client:

```
import socket

ipAddr = "127.0.0.1"
port = 5555

clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM);
print("Client socker object created")
clientSocket.connect((ipAddr, port))    #connect with server
print("Connected with server")

#ask for expression
print("Enter and integer expression to evaluate")
message = input()
clientSocket.send(message.encode()) #send the expression to server
print("Request for expression evaluation sent to server")

result = clientSocket.recv(1024)    #recieve result of expression
print("Result recieved from server")
print("Result of expression is: ", result.decode())

clientSocket.close()    #close all connection
```

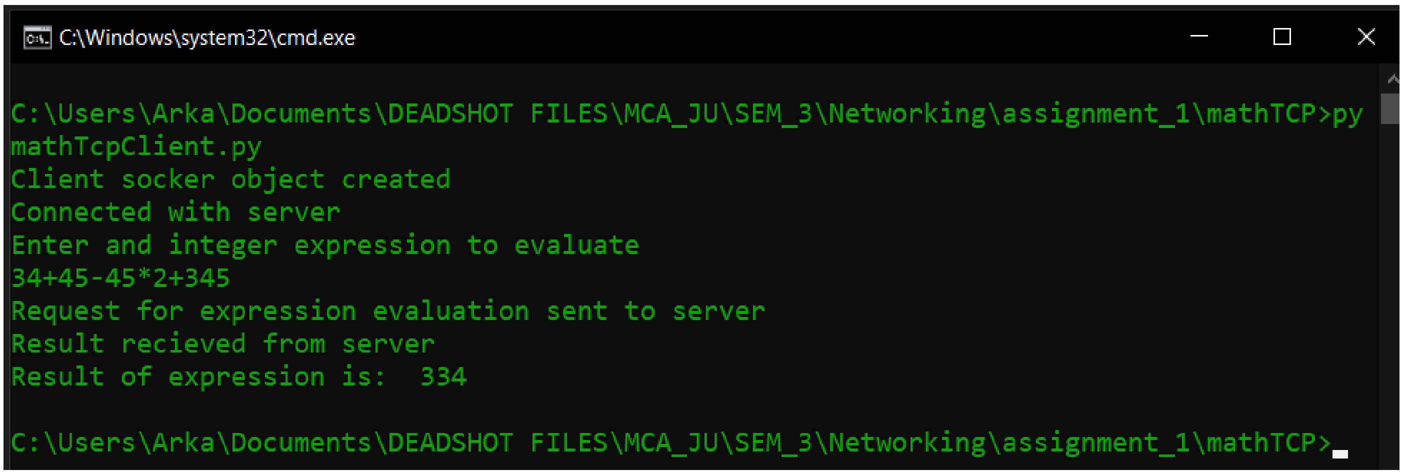
Output:

Server


C:\Windows\system32\cmd.exe - py mathTcpServer.py

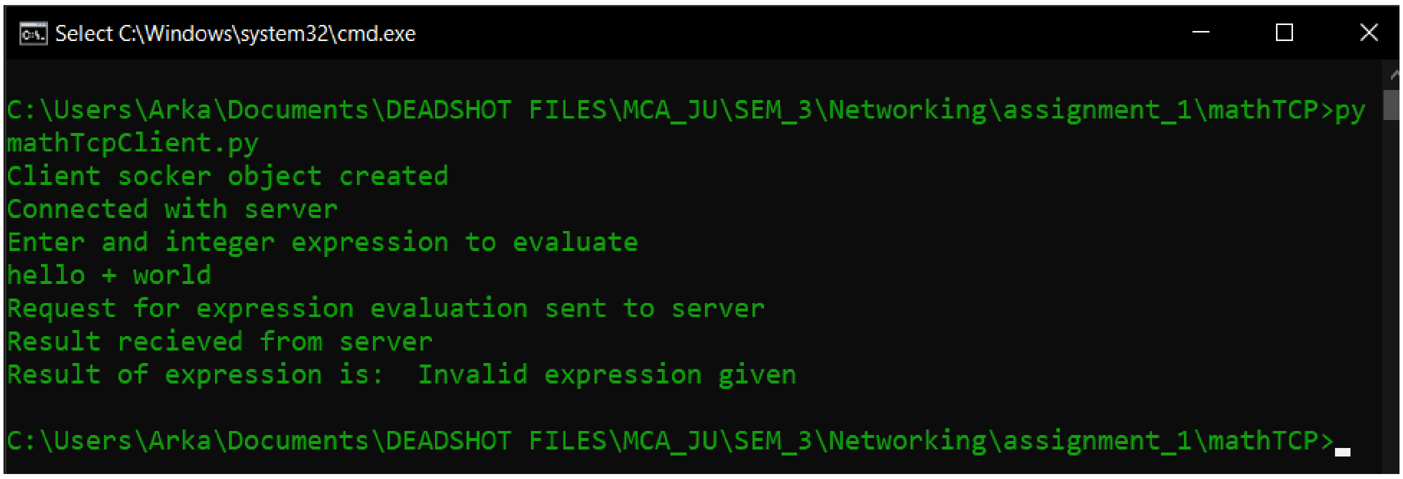
```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\mathTCP>py
mathTcpServer.py
Server socker object created
Socket bind successfull
Server is listining for clients
New connection
Connected to client:  ('127.0.0.1', 49459)
From client:  ('127.0.0.1', 49459)
Expression recieved:  34+45-45*2+345

Sending result back to client:  ('127.0.0.1', 49459)

New connection
Connected to client:  ('127.0.0.1', 49475)
From client:  ('127.0.0.1', 49475)
Expression recieved:  hello + world

Invalid expression given
```

Frist client



```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\mathTCP>py
mathTcpClient.py
Client socker object created
Connected with server
Enter and integer expression to evaluate
34+45-45*2+345
Request for expression evaluation sent to server
Result recieved from server
Result of expression is:  334

C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\mathTCP>
```

Second client



```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\mathTCP>py
mathTcpClient.py
Client socker object created
Connected with server
Enter and integer expression to evaluate
hello + world
Request for expression evaluation sent to server
Result recieved from server
Result of expression is:  Invalid expression given

C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\mathTCP>
```

Question: 3

Problem Statement:

Implement a UDP server program that returns the permanent address of a student upon receiving a request from a client. Assume that a text file that stores the names of students and their permanent addresses is available locally to the server. Choose your own formats for the request/reply messages.

Design for request and reply:

Here we have a single server and a single client who communicates among themselves. After the server is successfully created and has been bound it waits for the client to send requests. Here the protocol will be used is UDP.

The client will now send a message using the server's address and port number. There is no connection object here like TCP. Both server and client will communicate using each other's address and port.

Server will have a CSV file containing the data about students and their addresses. Before creating the socket, it will load the data of the CSV file and will create a dictionary with the student name being the key and corresponding addresses being the value.

If a student name (key) doesn't exist in the dictionary then it will send an appropriate message to the client.

Source Code:

The code for both the server and the client has been written using Python language and the socket library of python is used.

Code for Server:

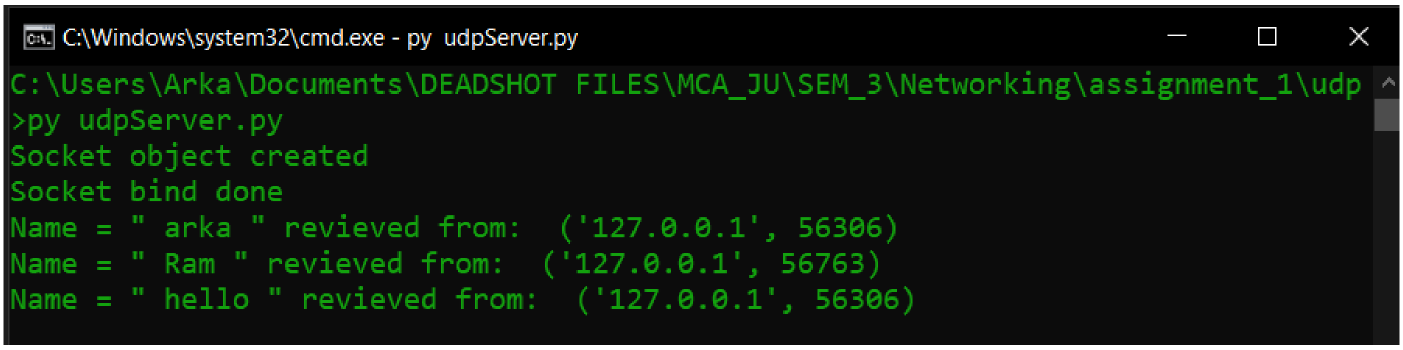
```
import socketimport csv

ipAddr = "127.0.0.1"
port = 5555

#creating map of addresses
data = {}
with open('data.csv', mode = 'r') as file:
      csvFile = csv.reader(file)   #load data from a CSV file
      for lines in csvFile:
            data[lines[0]] = lines[1]    #populate the dictonary


udpServerSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #udp socket
object
#if connection in port already exists then reuse that connection
udpServerSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
print("Socket object created")
udpServerSocket.bind((ipAddr, port))
print("Socket bind done")

#recieve student name from client
while True:
      conn = udpServerSocket.recvfrom(1024)    #recieve data from client
      #conn[0] contains the message send by client
      #conn[1] contains the address of client who has sent the data
      studentName = conn[0].decode()
      print("Name = \"", studentName, "\" revieved from: ", conn[1])
      add = data.get(studentName)  #get data from dictionary
      if add == None:  #if address not found
            msg = "Student record not found"
            udpServerSocket.sendto(msg.encode(), conn[1])
      else:
            udpServerSocket.sendto(add.encode(), conn[1])

udpServerSocket.close()      #close connection
```

Code for Client:

```
import socket
ipAddr = "127.0.0.1"
port = 5555

udpClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
print("Socket object created")

while True:
      print("Enter a student name: ")
      msg = input()
      if(msg == "exit"):
            break
      udpClientSocket.sendto(msg.encode(), (ipAddr, port)) #send message to server
      data = udpClientSocket.recvfrom(1024)   #recieve response
      #data[0] contains the message from server
      #data[1] contains the address of server
      add = data[0].decode()
      print("Address for student: ", end="")
      print(add, "\n")

udpClientSocket.close()
```

Output:

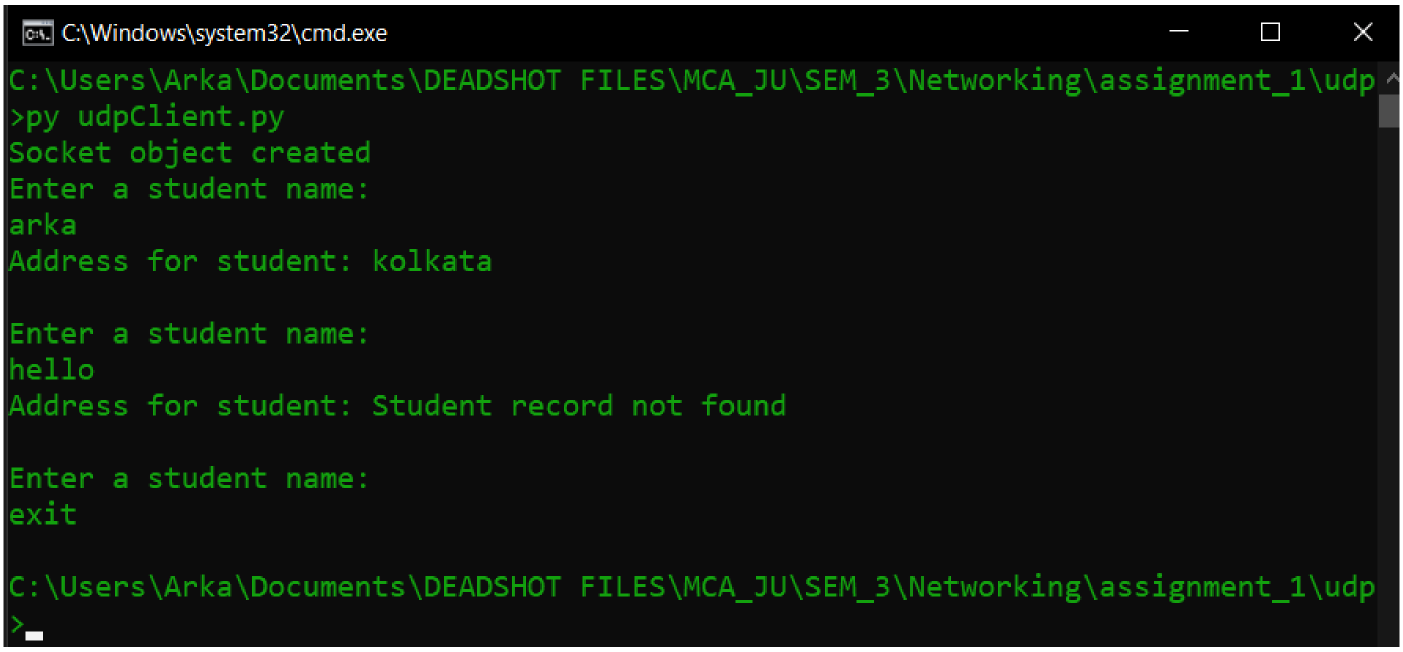
Server



C:\Windows\system32\cmd.exe - py udpServer.py

```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\udp
>py udpServer.py
Socket object created
Socket bind done
Name = " arka " revieved from:  ('127.0.0.1', 56306)
Name = " Ram " revieved from:  ('127.0.0.1', 56763)
Name = " hello " revieved from:  ('127.0.0.1', 56306)
```

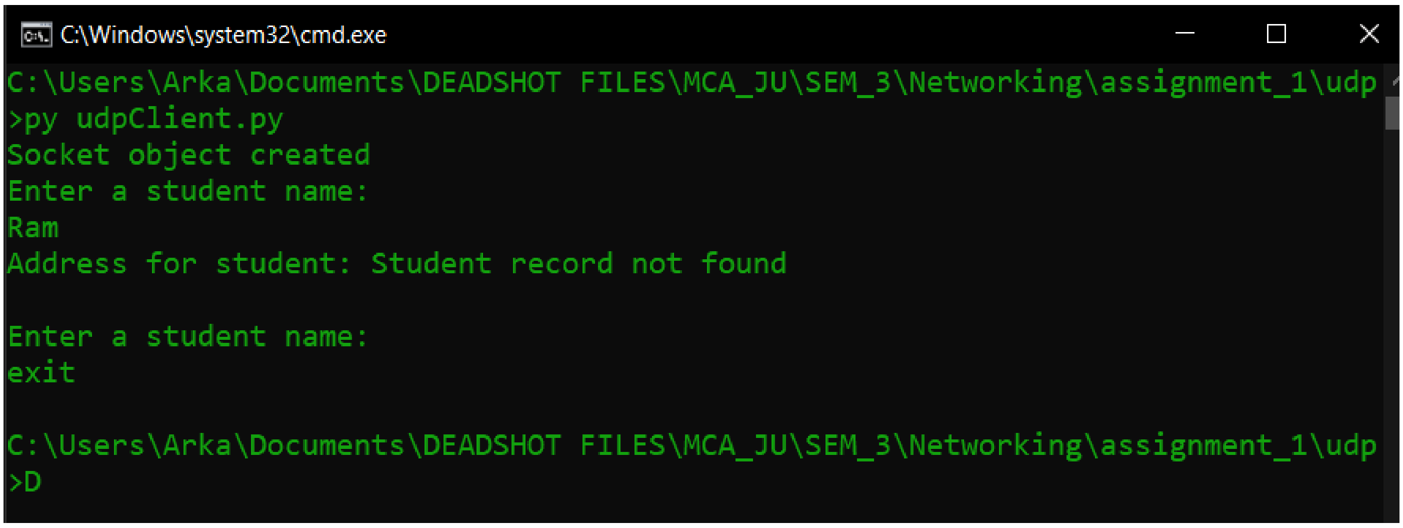
First client



C:\Windows\system32\cmd.exe

```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\udp
>py udpClient.py
Socket object created
Enter a student name:
arka
Address for student: kolkata

Enter a student name:
hello
Address for student: Student record not found

Enter a student name:
exit

C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\udp
>
```

Second Client



C:\Windows\system32\cmd.exe

```
C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\udp
>py udpClient.py
Socket object created
Enter a student name:
Ram
Address for student: Student record not found

Enter a student name:
exit

C:\Users\Arka\Documents\DEADSHOT FILES\MCA_JU\SEM_3\Networking\assignment_1\udp
>D
```

Assignment- 2

Problem Statement:

The objective of this laboratory exercise is to look at the details of the Transmission Control Protocol (TCP). TCP is a transport layer protocol. It is used by many application protocols like HTTP, FTP, SSH etc., where guaranteed and reliable delivery of messages is required. To do this exercise you need to install the Wireshark tool. This tool would be used to capture and examine a packet trace. Wireshark can be downloaded from www.wireshark.org.

Step1: Capture a Trace

1. Launch Wireshark
2. From Capture→Options select Loopback interface
3. Start a capture with a filter of "ip.addr==127.0.0.1 and tcp.port==xxxx", where xxxx is the port number used by the TCP server.
4. Run the TCP server program on a terminal.
5. Run two instances of the TCP client program on two separate terminals and send some dummy data to the sever.
6. Stop Wireshark capture

Step2: TCP Connection Establishment

To observe the three-way handshake in action, look for a TCP segment with SYN flag set. A "SYN" segment is the start of the three-way handshake and is sent by the TCP client to the TCP server. The server then replies with a TCP segment with SYN and ACK flag set. And finally the client sends an "ACK" to the server. For all the above three segments record the values of the sequence number and acknowledgment fields. Draw a time sequence diagram of the three-way handshake for TCP connection establishment in your trace. Do it for all the client connections.

Step3: TCP Data Transfer

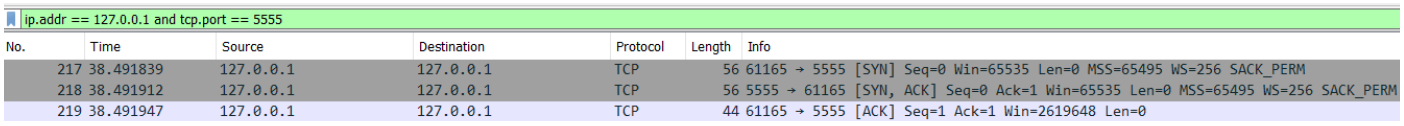
For all data segments sent by the client, record the value of the sequence number and acknowledge number fields. Also, record the same for the corresponding acknowledgements sent by the server. Draw a time sequence diagram of the data transfer in your trace. Do it for all the client connections.

Step4: TCP Connection Termination

Once the data transfer is over, the client initiates the connection termination by sending TCP segment with FIN flag set, to the server. Server acknowledges it and sends it's own intention to terminate the connection by sending a TCP segment with FIN and ACK flags set. The client finally sends an ACK segment to the server. For all the above three segments record the values of the sequence number and acknowledgment fields. Draw a time sequence diagram of the three-way handshake for TCP connection termination in your trace. Do it for all the client connections.

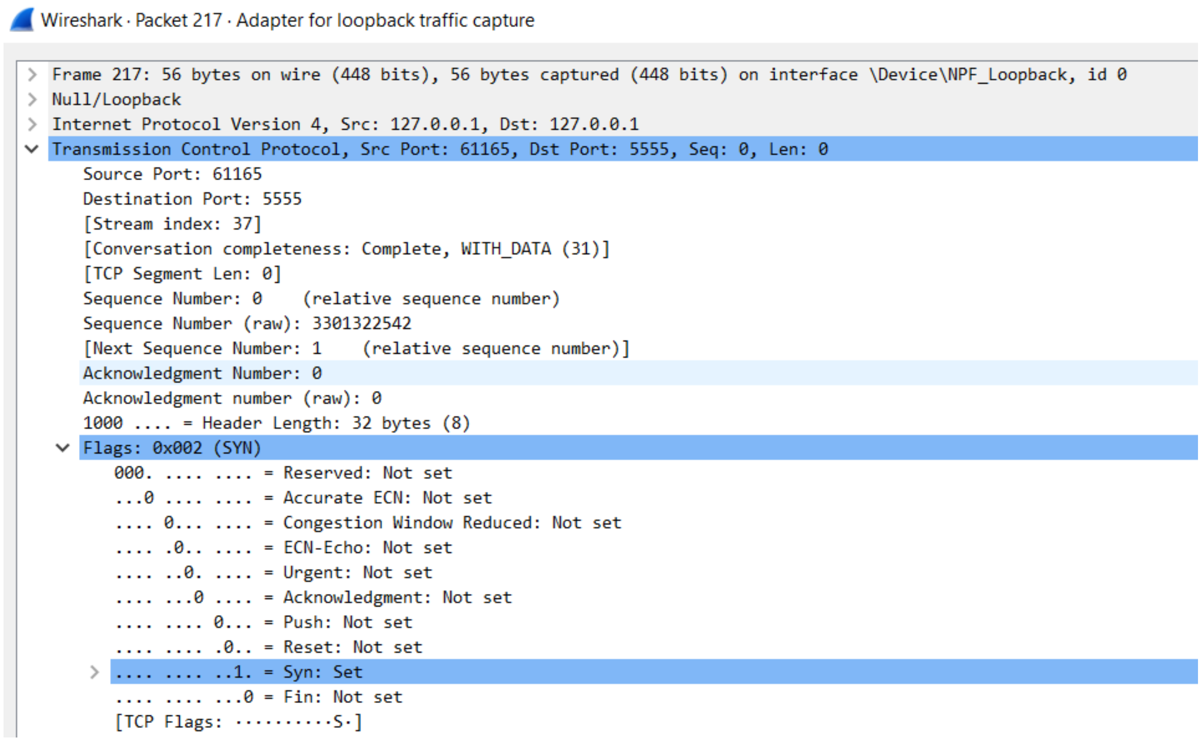
TCP connection establishment:

A snapshot of the 3-way handshake for TCP connection establishment.



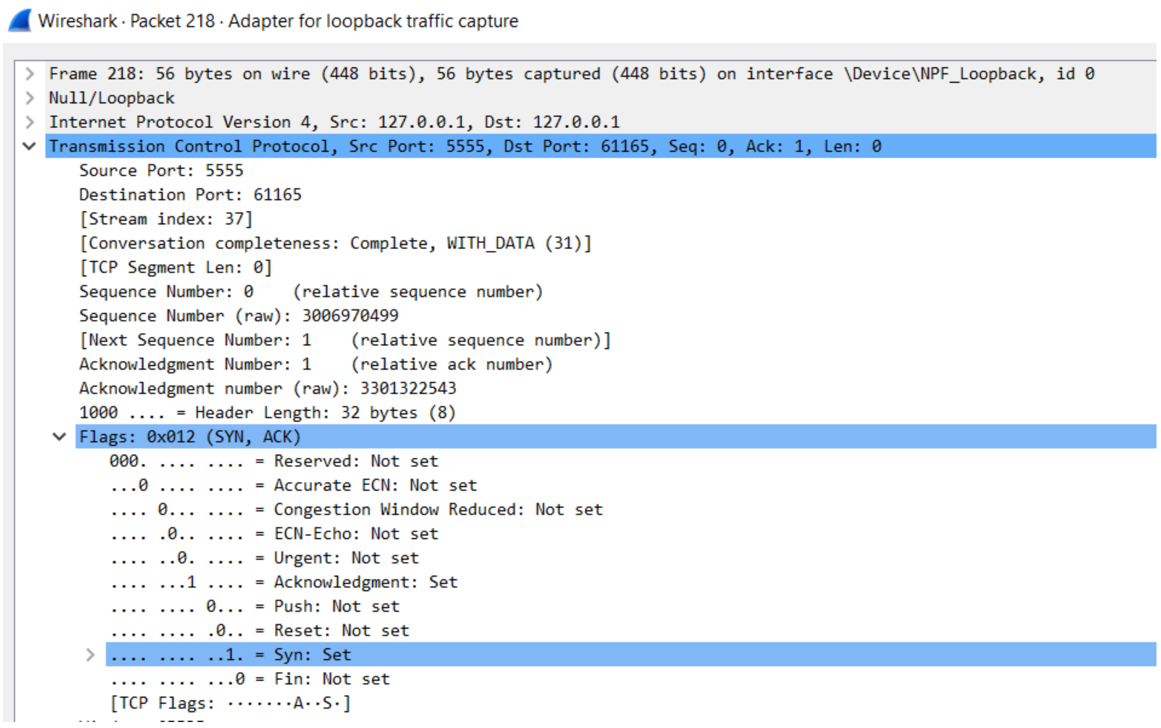
```
ip.addr == 127.0.0.1 and tcp.port == 5555
No.     Time         Source       Destination   Protocol  Length  Info
  217 38.491839    127.0.0.1    127.0.0.1     TCP          56 61165 → 5555 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  218 38.491912    127.0.0.1    127.0.0.1     TCP          56 5555 → 61165 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  219 38.491947    127.0.0.1    127.0.0.1     TCP          44 61165 → 5555 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
```

Step-1: Client sends SYN (Synchronize) message to the server. The message includes SYN flag set to 1 and it contains a unique sequence number which is any 32-bit number and acknowledge number which is set to 0.



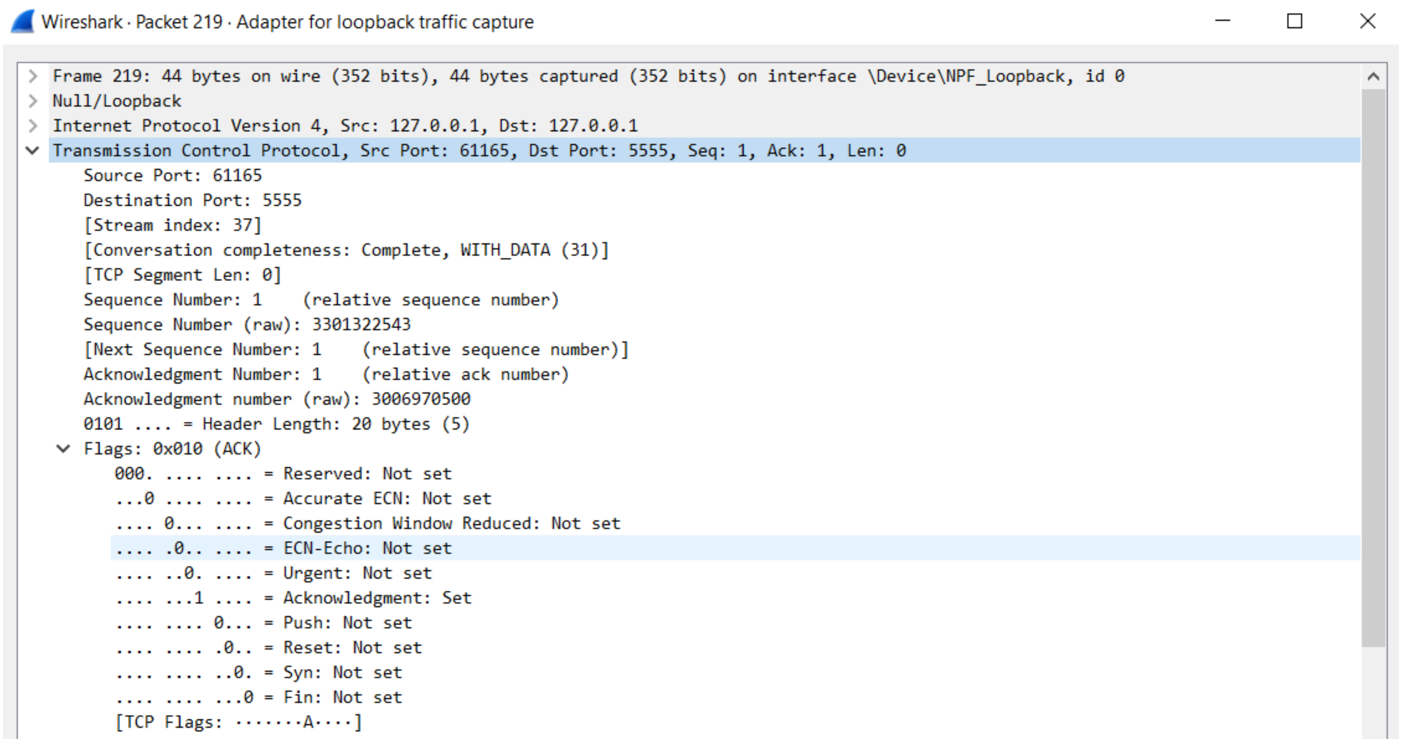
```
Wireshark · Packet 217 · Adapter for loopback traffic capture

> Frame 217: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 61165, Dst Port: 5555, Seq: 0, Len: 0
    Source Port: 61165
    Destination Port: 5555
    [Stream index: 37]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 3301322542
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1000 .... = Header Length: 32 bytes (8)
  v Flags: 0x002 (SYN)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Accurate ECN: Not set
      .... 0... .... = Congestion Window Reduced: Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...0 .... = Acknowledgment: Not set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    > .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ··········S·]
```

Step-2: Now the server responds with SYN and ACK (Acknowledge) message to the client. Here the SYN and ACK flags are set to 1. The ACK number is one higher than the SYN sequence number received by the client. But the sequence number will be different.



```
Wireshark · Packet 218 · Adapter for loopback traffic capture

> Frame 218: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 5555, Dst Port: 61165, Seq: 0, Ack: 1, Len: 0
    Source Port: 5555
    Destination Port: 61165
    [Stream index: 37]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 3006970499
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3301322543
    1000 .... = Header Length: 32 bytes (8)
  v Flags: 0x012 (SYN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Accurate ECN: Not set
      .... 0... .... = Congestion Window Reduced: Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    > .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·······A··S·]
```

Step-3: After client has received the SYN – ACK message from server it sends and Acknowledge message to the server. The ACK flag is set to 1 and the ACK number will be one higher than the previously received sequence number from server.

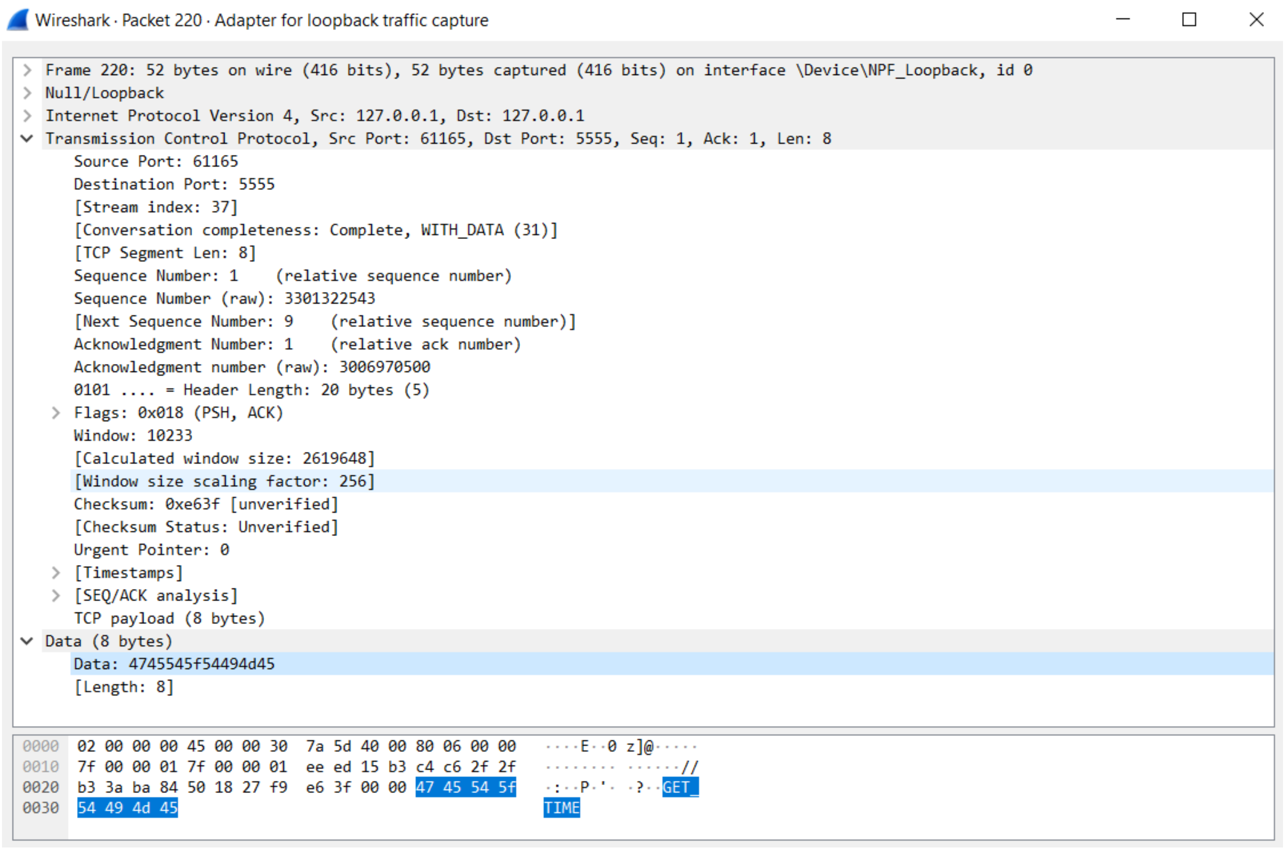


Wireshark · Packet 219 · Adapter for loopback traffic capture

```
> Frame 219: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 61165, Dst Port: 5555, Seq: 1, Ack: 1, Len: 0
    Source Port: 61165
    Destination Port: 5555
    [Stream index: 37]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3301322543
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3006970500
    0101 .... = Header Length: 20 bytes (5)
  v Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Accurate ECN: Not set
      .... 0... .... = Congestion Window Reduced: Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·······A····]
```

TCP Data Transfer:

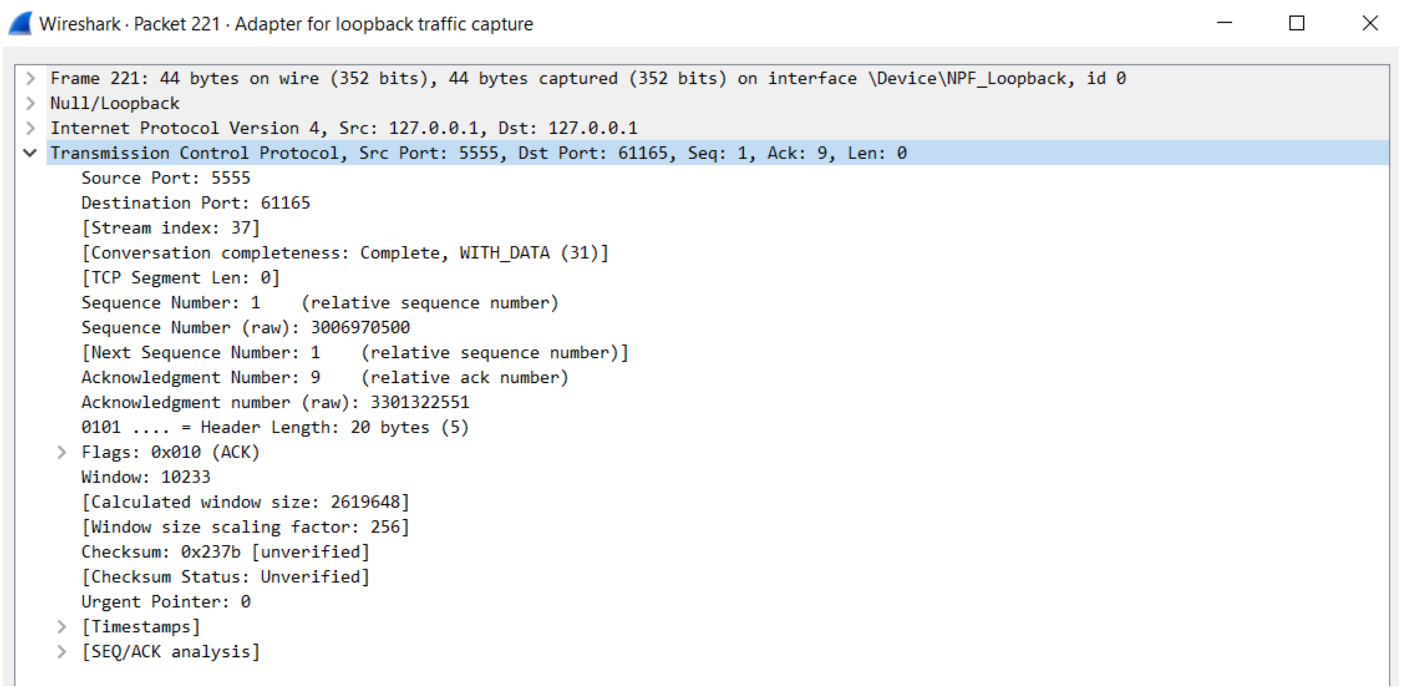
After successful connection data can be transferred in between the server and the client. Whoever wants to send data will send a message with flags PSH and ACK set to 1 along with the actual data.



Wireshark · Packet 220 · Adapter for loopback traffic capture

```
> Frame 220: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 61165, Dst Port: 5555, Seq: 1, Ack: 1, Len: 8
    Source Port: 61165
    Destination Port: 5555
    [Stream index: 37]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 8]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3301322543
    [Next Sequence Number: 9    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3006970500
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x018 (PSH, ACK)
    Window: 10233
    [Calculated window size: 2619648]
    [Window size scaling factor: 256]
    Checksum: 0xe63f [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
    TCP payload (8 bytes)
v Data (8 bytes)
    Data: 4745545f54494d45
    [Length: 8]

0000  02 00 00 00 45 00 00 30  7a 5d 40 00 80 06 00 00   ····E··0 z]@·····
0010  7f 00 00 01 7f 00 00 01  ee ed 15 b3 c4 c6 2f 2f   ········ ······//
0020  b3 3a ba 84 50 18 27 f9  e6 3f 00 00 47 45 54 5f   ·:··P·'· ·?··GET_
0030  54 49 4d 45                                        TIME
```

When the other computer receives the data, it will send another ACK message to the sender. In this case the ACK number will be sequence number of previous message + number of bytes sent.

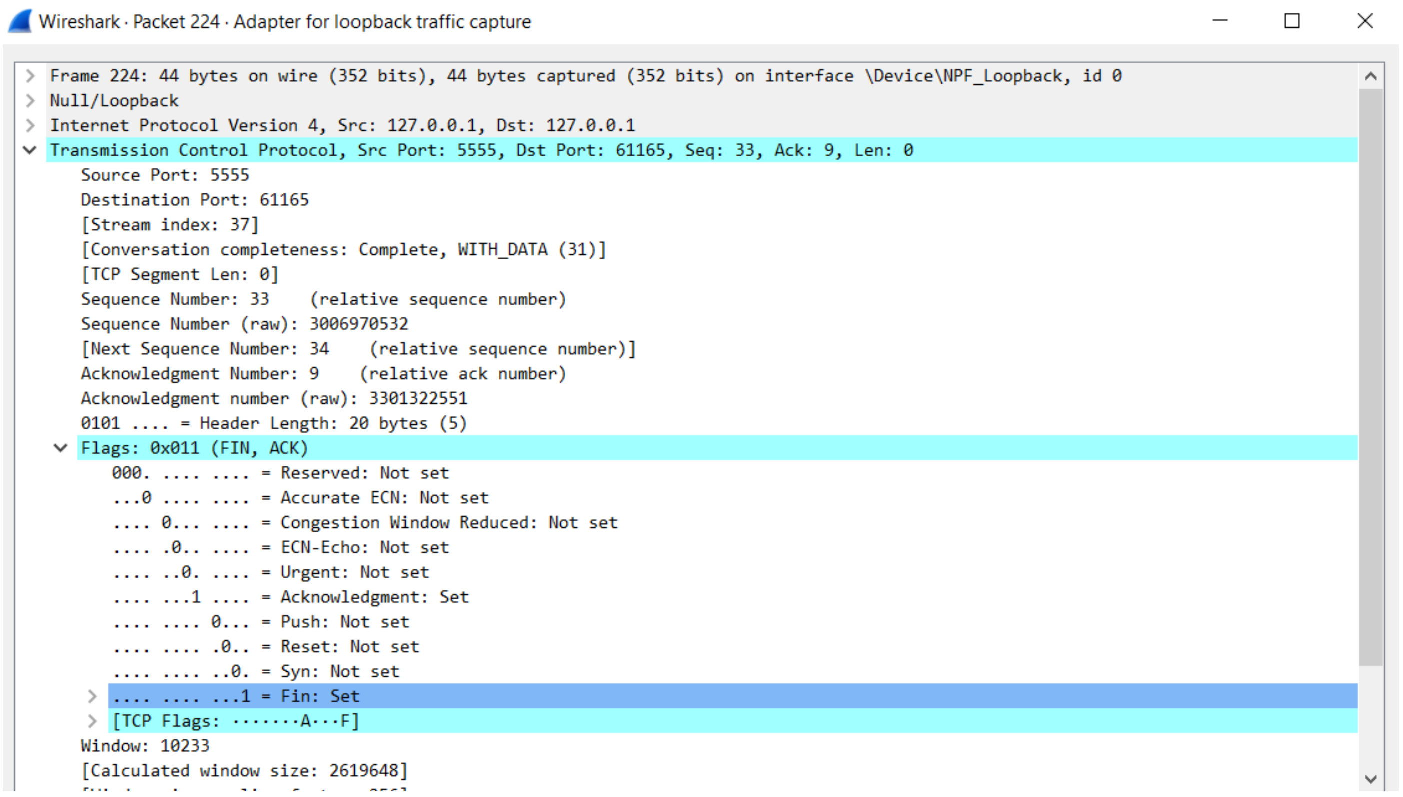


```
Wireshark · Packet 221 · Adapter for loopback traffic capture

> Frame 221: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 5555, Dst Port: 61165, Seq: 1, Ack: 9, Len: 0
    Source Port: 5555
    Destination Port: 61165
    [Stream index: 37]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3006970500
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 9    (relative ack number)
    Acknowledgment number (raw): 3301322551
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x010 (ACK)
    Window: 10233
    [Calculated window size: 2619648]
    [Window size scaling factor: 256]
    Checksum: 0x237b [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
```

TCP Connection Termination:

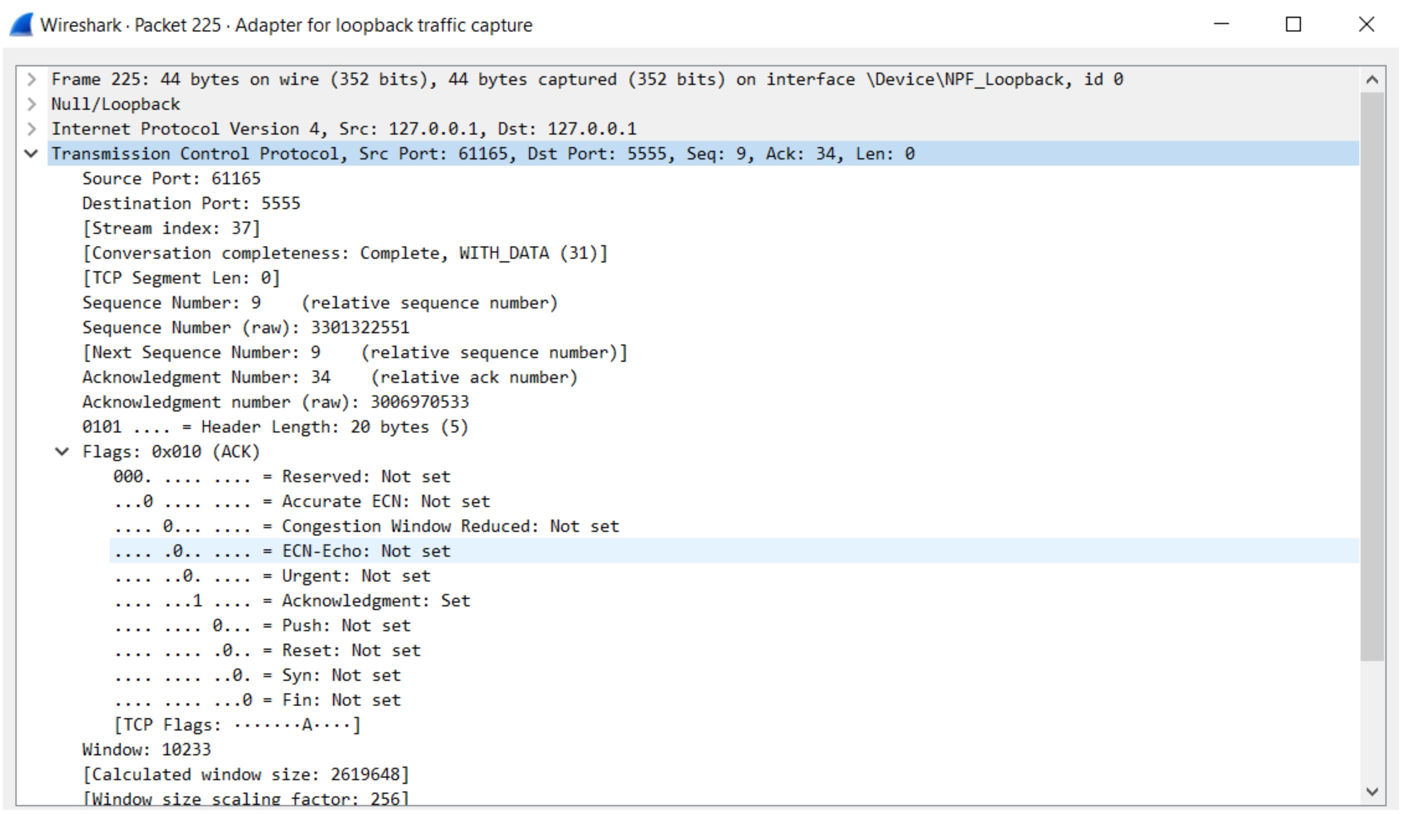
For connection termination one computer say the server will send a message with FIN and ACK flag set to 1. The client upon receiving the message will send an ACK message to the previous sender as acknowledge for closing the connection.

Similarly, the client will now send a message with FIN and ACK set to 1 to server and server will send the acknowledge as FIN, ACK back to the client.



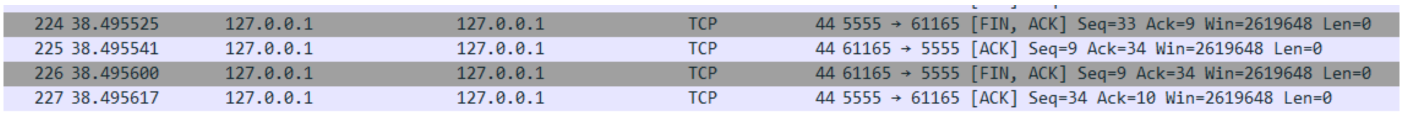
```
Wireshark · Packet 224 · Adapter for loopback traffic capture

> Frame 224: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 5555, Dst Port: 61165, Seq: 33, Ack: 9, Len: 0
    Source Port: 5555
    Destination Port: 61165
    [Stream index: 37]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 33    (relative sequence number)
    Sequence Number (raw): 3006970532
    [Next Sequence Number: 34    (relative sequence number)]
    Acknowledgment Number: 9    (relative ack number)
    Acknowledgment number (raw): 3301322551
    0101 .... = Header Length: 20 bytes (5)
  v Flags: 0x011 (FIN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Accurate ECN: Not set
      .... 0... .... = Congestion Window Reduced: Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
    > .... .... ...1 = Fin: Set
    > [TCP Flags: ·······A···F]
    Window: 10233
    [Calculated window size: 2619648]
```

For the ACK message the ACK number will be one higher than the sequence number of the previous (FIN, ACK) message.


Wireshark · Packet 225 · Adapter for loopback traffic capture

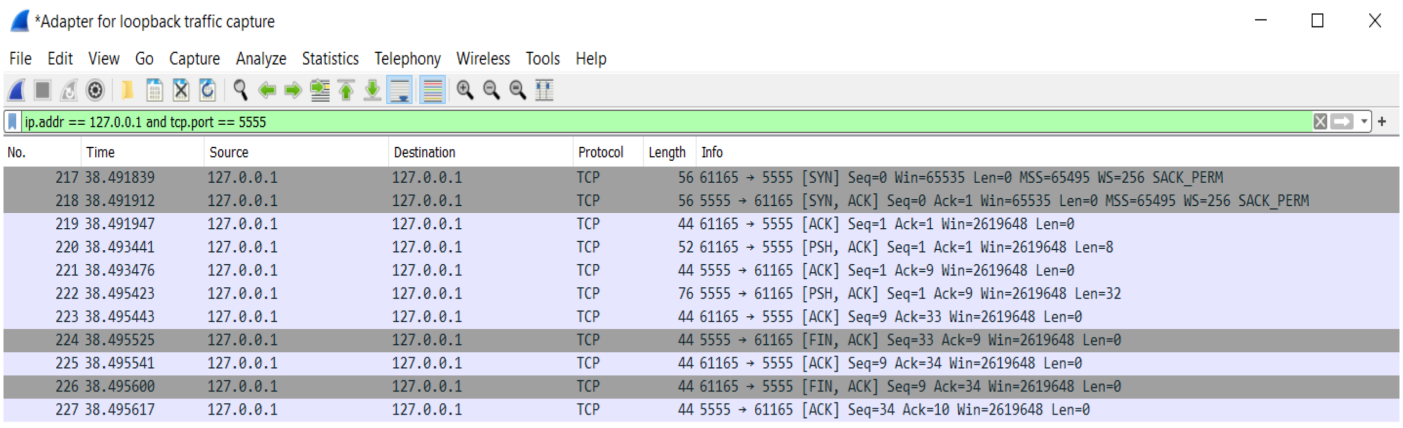
```
> Frame 225: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 61165, Dst Port: 5555, Seq: 9, Ack: 34, Len: 0
    Source Port: 61165
    Destination Port: 5555
    [Stream index: 37]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 9    (relative sequence number)
    Sequence Number (raw): 3301322551
    [Next Sequence Number: 9    (relative sequence number)]
    Acknowledgment Number: 34    (relative ack number)
    Acknowledgment number (raw): 3006970533
    0101 .... = Header Length: 20 bytes (5)
  v Flags: 0x010 (ACK)
        000. .... .... = Reserved: Not set
        ...0 .... .... = Accurate ECN: Not set
        .... 0... .... = Congestion Window Reduced: Not set
        .... .0.. .... = ECN-Echo: Not set
        .... ..0. .... = Urgent: Not set
        .... ...1 .... = Acknowledgment: Set
        .... .... 0... = Push: Not set
        .... .... .0.. = Reset: Not set
        .... .... ..0. = Syn: Not set
        .... .... ...0 = Fin: Not set
        [TCP Flags: ·······A····]
    Window: 10233
    [Calculated window size: 2619648]
    [Window size scaling factor: 256]
```

This is called a 4-way handshake, which is required for closing a TCP connection. This is actually a set of 2-way handshake.



```
224 38.495525   127.0.0.1   127.0.0.1   TCP   44 5555 → 61165 [FIN, ACK] Seq=33 Ack=9 Win=2619648 Len=0
225 38.495541   127.0.0.1   127.0.0.1   TCP   44 61165 → 5555 [ACK] Seq=9 Ack=34 Win=2619648 Len=0
226 38.495600   127.0.0.1   127.0.0.1   TCP   44 61165 → 5555 [FIN, ACK] Seq=9 Ack=34 Win=2619648 Len=0
227 38.495617   127.0.0.1   127.0.0.1   TCP   44 5555 → 61165 [ACK] Seq=34 Ack=10 Win=2619648 Len=0
```

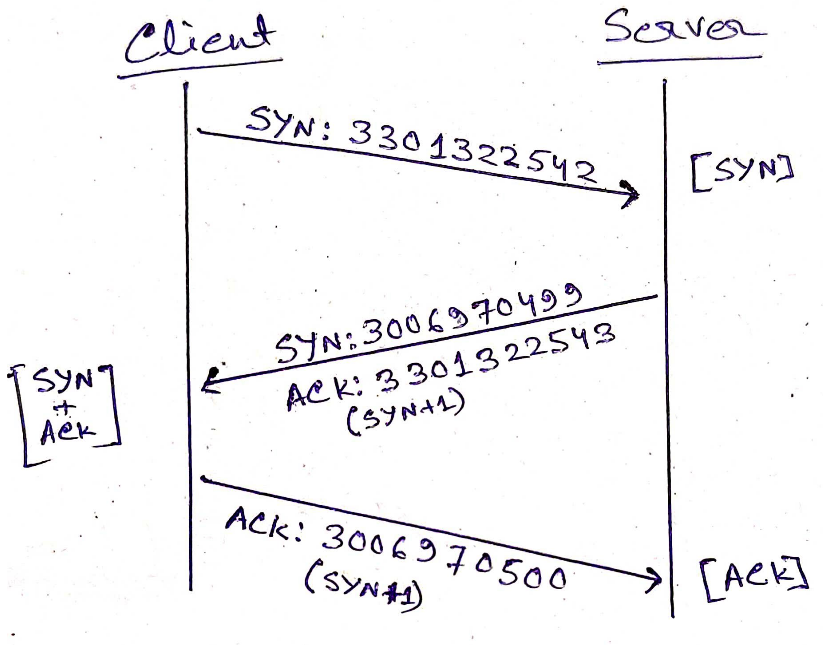
Snapshot of everything altogether:



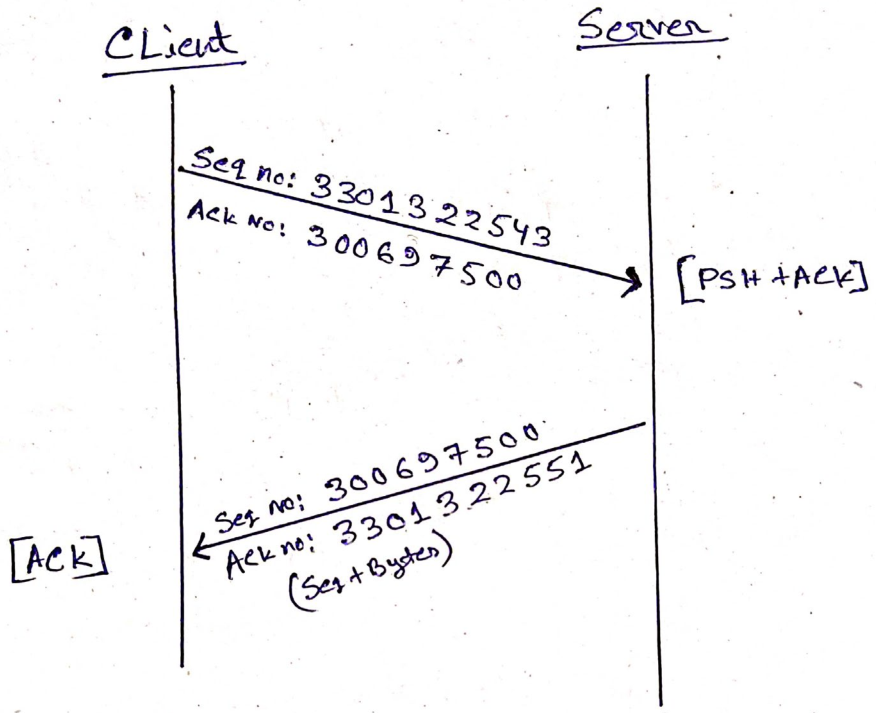
```
ip.addr == 127.0.0.1 and tcp.port == 5555

No.   Time        Source      Destination  Protocol  Length  Info
  217 38.491839   127.0.0.1   127.0.0.1    TCP       56 61165 → 5555 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  218 38.491912   127.0.0.1   127.0.0.1    TCP       56 5555 → 61165 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  219 38.491947   127.0.0.1   127.0.0.1    TCP       44 61165 → 5555 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
  220 38.493441   127.0.0.1   127.0.0.1    TCP       52 61165 → 5555 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=8
  221 38.493476   127.0.0.1   127.0.0.1    TCP       44 5555 → 61165 [ACK] Seq=1 Ack=9 Win=2619648 Len=0
  222 38.495423   127.0.0.1   127.0.0.1    TCP       76 5555 → 61165 [PSH, ACK] Seq=1 Ack=9 Win=2619648 Len=32
  223 38.495443   127.0.0.1   127.0.0.1    TCP       44 61165 → 5555 [ACK] Seq=9 Ack=33 Win=2619648 Len=0
  224 38.495525   127.0.0.1   127.0.0.1    TCP       44 5555 → 61165 [FIN, ACK] Seq=33 Ack=9 Win=2619648 Len=0
  225 38.495541   127.0.0.1   127.0.0.1    TCP       44 61165 → 5555 [ACK] Seq=9 Ack=34 Win=2619648 Len=0
  226 38.495600   127.0.0.1   127.0.0.1    TCP       44 61165 → 5555 [FIN, ACK] Seq=9 Ack=34 Win=2619648 Len=0
  227 38.495617   127.0.0.1   127.0.0.1    TCP       44 5555 → 61165 [ACK] Seq=34 Ack=10 Win=2619648 Len=0
```

Timing sequence diagram:

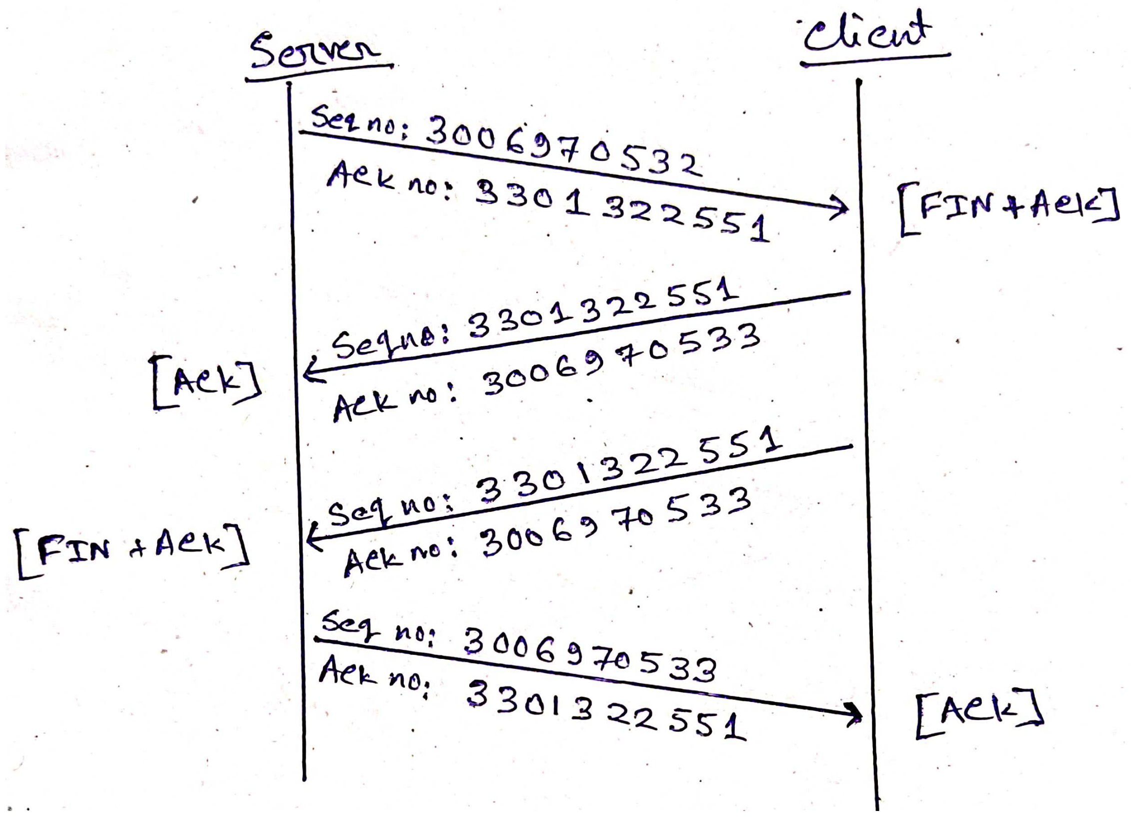
TCP connection:



Data transfer:



Closing connection:



Assignment- 3

Problem Statement:

The objective of this laboratory exercise is to look at the details of the User Datagram Protocol (UDP). UDP is a transport layer protocol. It is used by many application protocols like DNS, DHCP, SNMP etc., where reliability is not a concern. To do this exercise you need to install the Wireshark tool, which is widely used to capture and examine a packet trace. Wireshark can be downloaded from www.wireshark.org.

Step1: Capture a Trace

1. Launch Wireshark
2. From Capture→Options select Loopback interface
3. Start a capture with a filter of "ip.addr==127.0.0.1 and udp.port==xxxx", where xxxx is the port number used by the UDP server.
4. Run the UDP server program on a terminal.
5. Run multiple instances of the UDP client program on separate terminals and send requests to the sever.
6. Stop Wireshark capture

Step2: Inspect the Trace

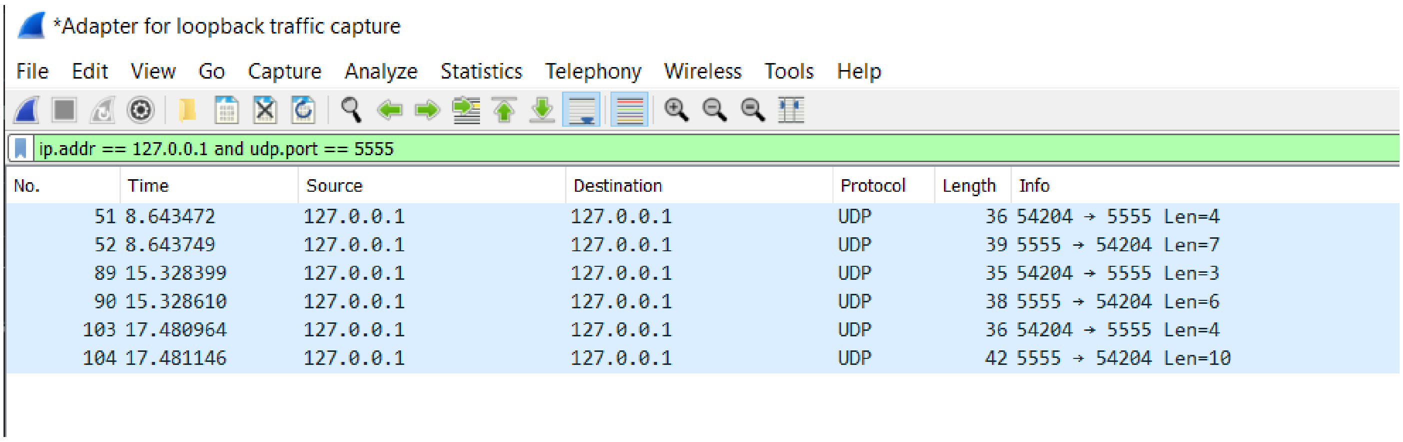
Select different packets in the trace and browse the expanded UDP header and record the following fields:

- Source Port: the port from which the udp segment is sent.
- Destination Port: the port to which the udp segment is sent.
- Length: the length of the UDP segment.

Capturing UDP packets in WireShark:

UDP is an unreliable and connectionless protocol. There is no need for establishing any connection before data transfer. UDP packets contains a UPD header which contains the port number for both source and destination then there is a length field which is the size of header and payload in Bytes and then some more checksum information.

UDP is faster than TCP as there is no need for creating a connection. But it is unreliable.


*Adapter for loopback traffic capture
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
ip.addr == 127.0.0.1 and udp.port == 5555
```

No.	Time	Source	Destination	Protocol	Length	Info
51	8.643472	127.0.0.1	127.0.0.1	UDP	36	54204 → 5555 Len=4
52	8.643749	127.0.0.1	127.0.0.1	UDP	39	5555 → 54204 Len=7
89	15.328399	127.0.0.1	127.0.0.1	UDP	35	54204 → 5555 Len=3
90	15.328610	127.0.0.1	127.0.0.1	UDP	38	5555 → 54204 Len=6
103	17.480964	127.0.0.1	127.0.0.1	UDP	36	54204 → 5555 Len=4
104	17.481146	127.0.0.1	127.0.0.1	UDP	42	5555 → 54204 Len=10

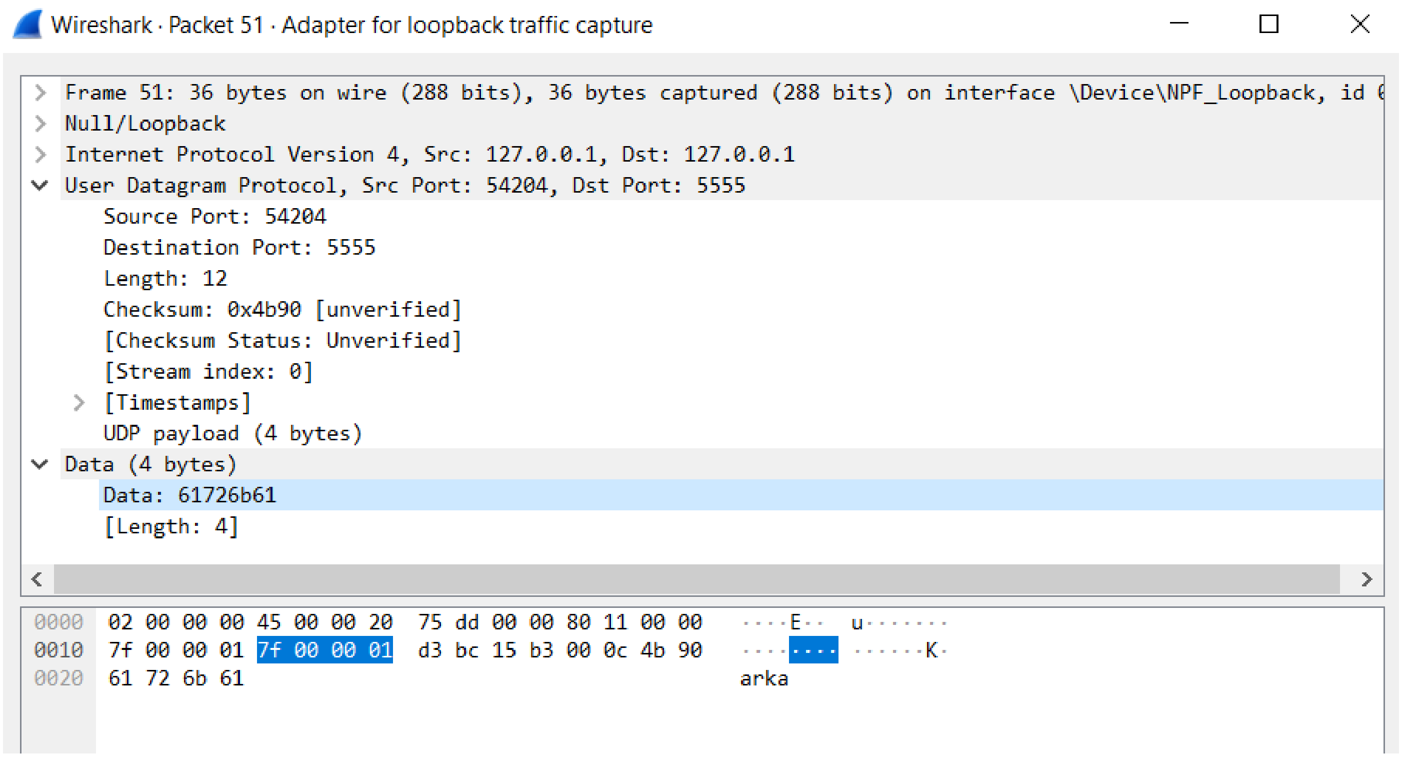
Considering the first frame sent –

The source port is : 54204   (Client in this case)

The destination port is : 5555 (Server in this case)

The length is : 27

The size of the payload is 4 Bytes.


Wireshark · Packet 51 · Adapter for loopback traffic capture

```
> Frame 51: 36 bytes on wire (288 bits), 36 bytes captured (288 bits) on interface \Device\NPF_Loopback, id
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v User Datagram Protocol, Src Port: 54204, Dst Port: 5555
    Source Port: 54204
    Destination Port: 5555
    Length: 12
    Checksum: 0x4b90 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
  > [Timestamps]
    UDP payload (4 bytes)
v Data (4 bytes)
    Data: 61726b61
    [Length: 4]

0000  02 00 00 00 45 00 00 20  75 dd 00 00 80 11 00 00   ····E·· u·······
0010  7f 00 00 01 7f 00 00 01  d3 bc 15 b3 00 0c 4b 90   ········ ······K·
0020  61 72 6b 61                                        arka
```