ASSIGNMENT 1.1

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.

2. Design of the Request/Reply Protocol

- **Request Message (Client to Server)**: The client sends a request message ("get_time") to request the current time and date.
- **Reply Message (Server to Client):**The server program listens for incoming connections on IP address '127.0.0.1' and port 5000. When a client connects, it sends the current date and time (formatted as "%Y-%m-%d %H:%M:%S") to the client and then closes the connection.

3. Source Code:

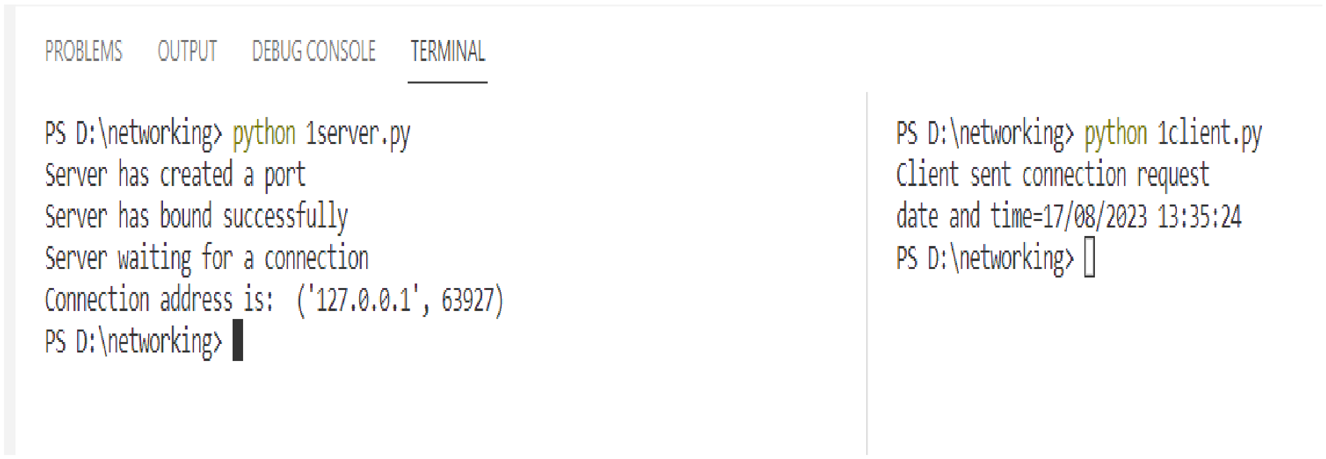
1server.py

```
from datetime import datetime
import socket
port = 5000
ip = '127.0.0.1'
BUF_SIZE = 1024
k = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Server has created a port")
k.bind((ip, port))
print("Server has bound successfully")
k.listen(1)    # listening for connections.
print("Server waiting for a connection")

con, addr = k.accept()  # When client connects, it returns a new socket 'con'
print('Connection address is: ',addr)
dt = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
msg = f"date and time={dt}".encode()   # encodes into bytes
con.send(msg) # server send msg to client
con.close()
```

1client.py

```
import socket
port = 5000
ip = '127.0.0.1' # loopback address
BUF_SIZE = 1024
k = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
k.connect((ip,port))
print("Client sent connection request")

k.send("get_time".encode()) #send msg to server
data = k.recv(BUF_SIZE)  # client receive data from the server
print(data.decode())   # decodes received data into string and prints it
k.close()
```

4. Sample Run:



```
PS D:\networking> python 1server.py
Server has created a port
Server has bound successfully
Server waiting for a connection
Connection address is:  ('127.0.0.1', 63927)
PS D:\networking>
```

```
PS D:\networking> python 1client.py
Client sent connection request
date and time=17/08/2023 13:35:24
PS D:\networking>
```

ASSIGNMENT 1.2

1. 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.

2. Design of the Request/Reply Protocol

- **Request Message (Client to Server)**:The client sends a request message to the server. In the request, an integer arithmetic expression given by the user is sent.
- **Reply Message (Server to Client):**The server responds with the evaluated result of the arithmetic expression (using “eval” function) given by the user.

3. Source Code:

2server.py

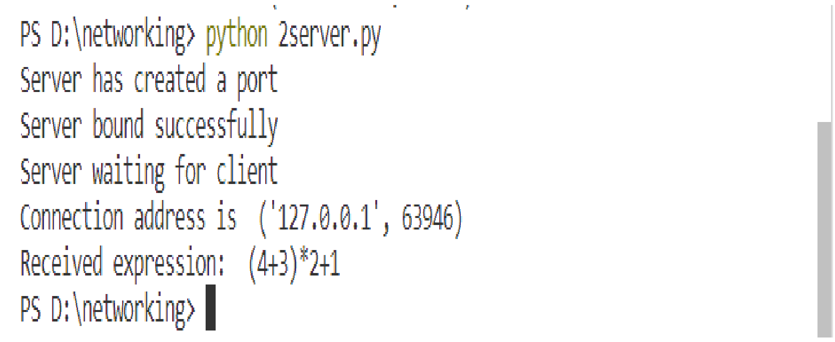
```
import socket
TCP_PORT = 5000
IP_ADDR = '127.0.0.1'
BUF_SIZE = 1024
k = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Server has created a port")
k.bind((IP_ADDR,TCP_PORT))
print("Server bound successfully")
k.listen(1)
print("Server waiting for client")

con, addr = k.accept()
print("Connection address is ", addr)
data = con.recv(BUF_SIZE)  # server received exp from client
print("Received expression: ",data.decode())  #byte to string

try:
   res = str(eval(data))  #evaluation
   res = "Result: "+res
   con.send(res.encode())  #encode the result into bytes and send to client
```

```
except Exception as x:
    print(x)
    err = "Invalid integer expression: %s"%(x)
    err=str(err)
    con.send(err.encode())
finally:
    con.close()
```

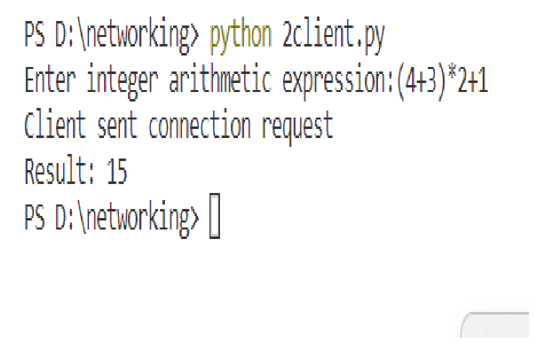
2client.py

```
import socket
TCP_PORT = 5000
IP_ADDR = '127.0.0.1' # loopback address
BUF_SIZE = 1024
exp = input("Enter integer arithmetic expression:")
k = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
k.connect((IP_ADDR,TCP_PORT))
print("Client sent connection request")
k.send(exp.encode())    # client send exp to the server

res = k.recv(BUF_SIZE)   #client received data from server
print(res.decode())    # decodes the received data into string
k.close()
```

4. Sample Run:



```
PS D:\networking> python 2server.py
Server has created a port
Server bound successfully
Server waiting for client
Connection address is  ('127.0.0.1', 63946)
Received expression:  (4+3)*2+1
PS D:\networking>
```



```
PS D:\networking> python 2client.py
Enter integer arithmetic expression:(4+3)*2+1
Client sent connection request
Result: 15
PS D:\networking>
```

ASSIGNMENT 1.3

1. 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 local to the server.

2. Design of the Request/Reply Protocol

- **Request Message (Client to Server)**: The client sends a request message to the server. In the request, a name is sent by the client. The client sends a ‘quit’ message to close the connection.
- **Reply Message (Server to Client):**The server responds with the address of the name if present in the text file and if not present sends a “Address invalid” message.

3. Source Code:

3server.py

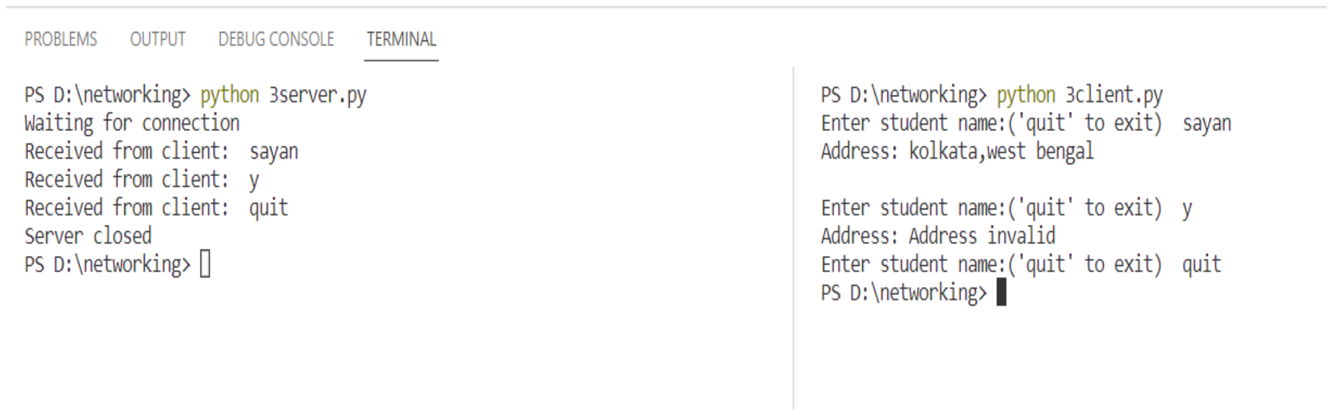
```
import socket
def getAddr(name):
   with open("student.txt") as f:
      data = f.readlines()
      for line in data:
         sname, addr = line.split(":")  #split the line
         if sname==name:
            return addr
   return "Address invalid"

IP_ADDR='127.0.0.1'
UDP_PORT=5000
BUF_SIZE=1024  #buffer size of received data
k=socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
k.bind((IP_ADDR,UDP_PORT))
print("Waiting for connection")
try:
   while True:
      con, addr = k.recvfrom(BUF_SIZE)
      name = con.decode() #bytes to string
      print("Received from client: ",name)
```

```
        if name == "quit":
            break
        address=getAddr(name) #gets the address
        k.sendto(address.encode(),addr) #send data to client
except Exception as e:
    print(e)
finally:
    print("Server closed")
    k.close()
```

<u>3client.py</u>

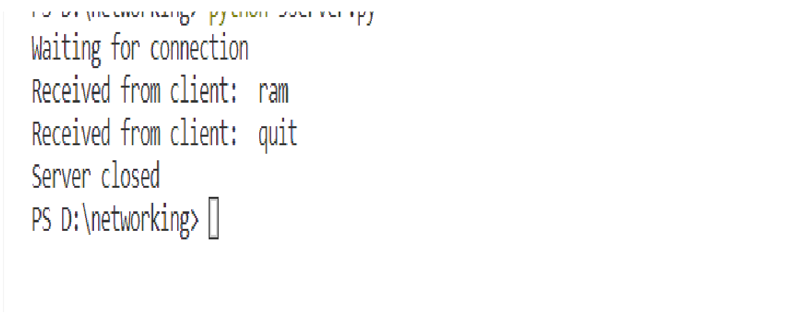
```
import socket
IP_ADDR='127.0.0.1'
UDP_PORT=5000
BUF_SIZE=1024
k=socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
while True:
    name = input("Enter student name:('quit' to exit)  ")
    if name.lower() == "quit":
        k.sendto(name.encode(),(IP_ADDR,UDP_PORT))
        break
    k.sendto(name.encode(),(IP_ADDR,UDP_PORT))
    data,_ = k.recvfrom(BUF_SIZE) #server address is ignored and data received
    print("Address:",data.decode()) #decode the result into string
k.close()
```

<u>4. Sample Run:</u>

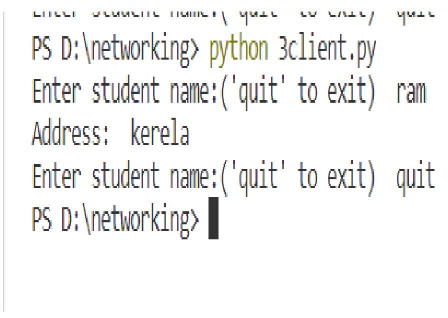


```
PS D:\networking> python 3server.py
Waiting for connection
Received from client:  sayan
Received from client:  y
Received from client:  quit
Server closed
PS D:\networking>
```

```
PS D:\networking> python 3client.py
Enter student name:('quit' to exit)  sayan
Address: kolkata,west bengal

Enter student name:('quit' to exit)  y
Address: Address invalid
Enter student name:('quit' to exit)  quit
PS D:\networking>
```



```
PS D:\networking> python 3server.py
Waiting for connection
Received from client:  ram
Received from client:  quit
Server closed
PS D:\networking>
```



```
Enter student name:('quit' to exit)  quit
PS D:\networking> python 3client.py
Enter student name:('quit' to exit)  ram
Address:  kerela
Enter student name:('quit' to exit)  quit
PS D:\networking>
```

<u>student.txt</u>

sayan:kolkata,west bengal
atul:kolkata,west bengal
ram: kerela
raj: bt road, kolkata

Assignment 2

1.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
(i) Launch Wireshark
(ii) From Capture→Options select Loopback interface
(iii) 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.
(iv) Run the TCP server program on a terminal.
(v) Run two instances of the TCP client program on two separate terminals and send some dummy data to the
sever.
(vi) 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.

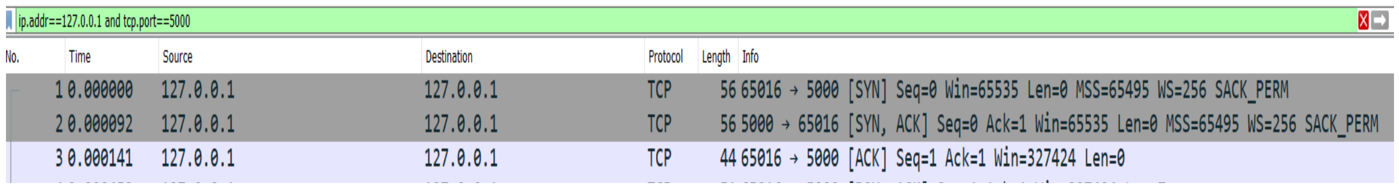
Step3: TCP Data Transfer
For all data segments sent by the client, record the value of the sequence number and acknowledge number
fields. Also, record the same for the corresponding acknowledgements sent by the server. Draw a time sequence
diagram of the data transfer in your trace. Do it for all the client connections.

Step4: TCP Connection Termination

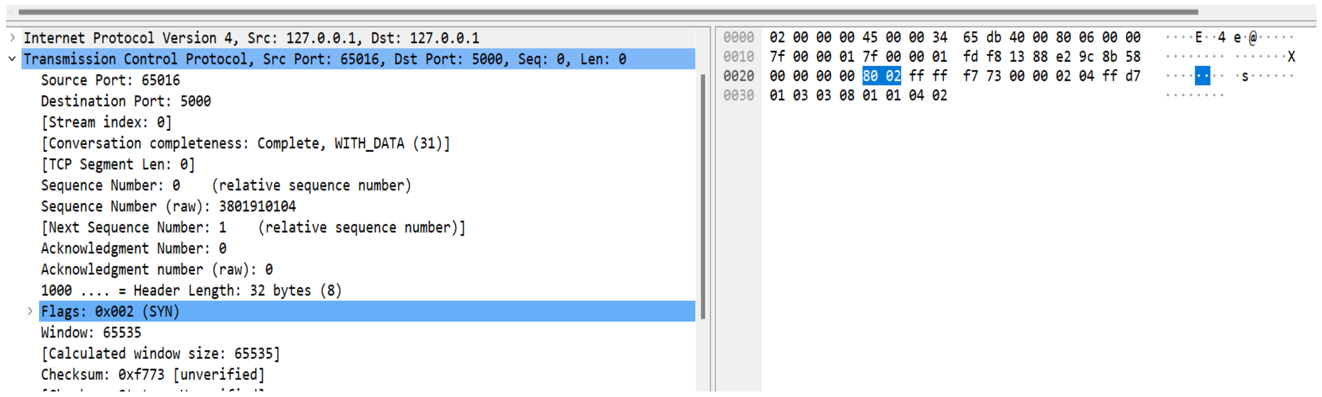
Once the data transfer is over, the client initiates the connection termination by sending TCP segment with FIN
flag set, to the server. Server acknowledges it and sends it's own intention to terminate the connection by sending
a TCP segment with FIN and ACK flags set. The client finally sends an ACK 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.

Capturing in Wireshark

TCP Connection Establishment:

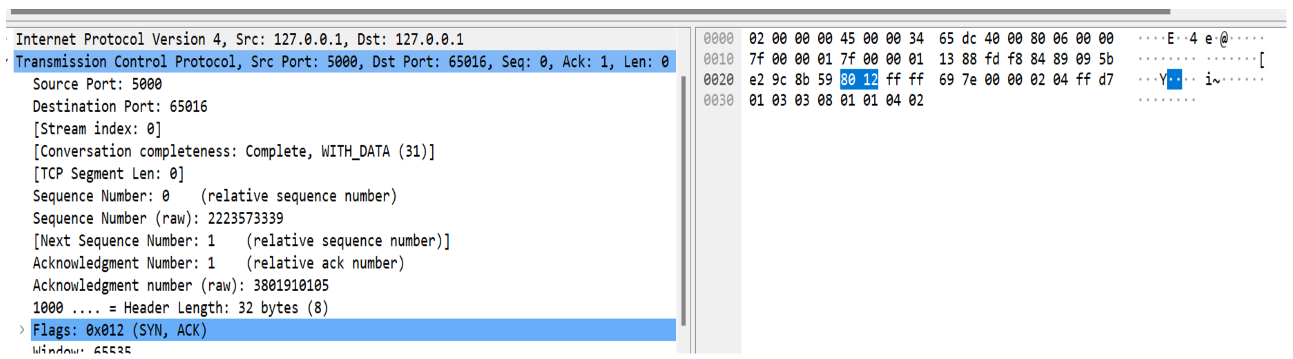


```
ip.addr==127.0.0.1 and tcp.port==5000

No.  Time      Source     Destination  Protocol  Length  Info
  1  0.000000  127.0.0.1  127.0.0.1    TCP       56      65016 → 5000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  2  0.000092  127.0.0.1  127.0.0.1    TCP       56      5000 → 65016 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  3  0.000141  127.0.0.1  127.0.0.1    TCP       44      65016 → 5000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
```



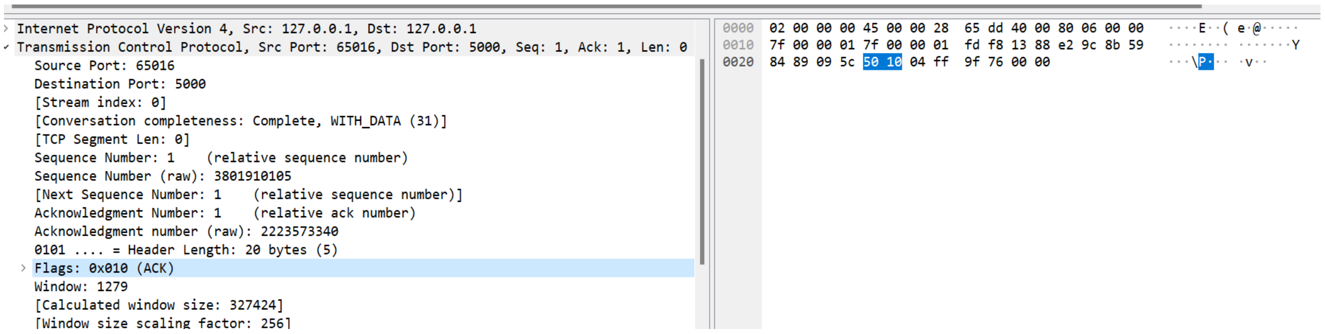
```
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 65016, Dst Port: 5000, Seq: 0, Len: 0
    Source Port: 65016
    Destination Port: 5000
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 3801910104
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x002 (SYN)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0xf773 [unverified]
```

```
0000  02 00 00 00 45 00 00 34  65 db 40 00 80 06 00 00   ····E··4 e·@·····
0010  7f 00 00 01 7f 00 00 01  fd f8 13 88 e2 9c 8b 58   ········ ·······X
0020  00 00 00 00 80 02 ff ff  f7 73 00 00 02 04 ff d7   ········ ·s······
0030  01 03 03 08 01 01 04 02                            ········
```



```
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 5000, Dst Port: 65016, Seq: 0, Ack: 1, Len: 0
    Source Port: 5000
    Destination Port: 65016
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 2223573339
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3801910105
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x012 (SYN, ACK)
```

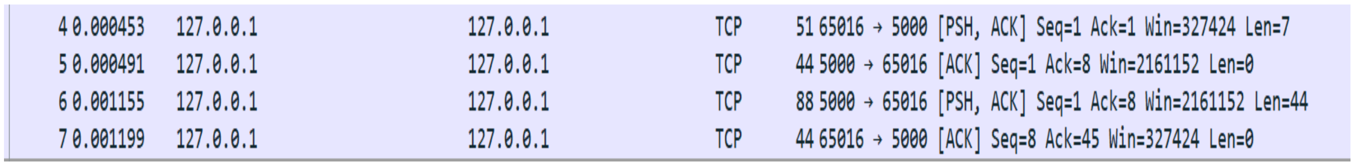
```
0000  02 00 00 00 45 00 00 34  65 dc 40 00 80 06 00 00   ····E··4 e·@·····
0010  7f 00 00 01 7f 00 00 01  13 88 fd f8 84 89 09 5b   ········ ·······[
0020  e2 9c 8b 59 80 12 ff ff  69 7e 00 00 02 04 ff d7   ···Y···· i~······
0030  01 03 03 08 01 01 04 02                            ········
```



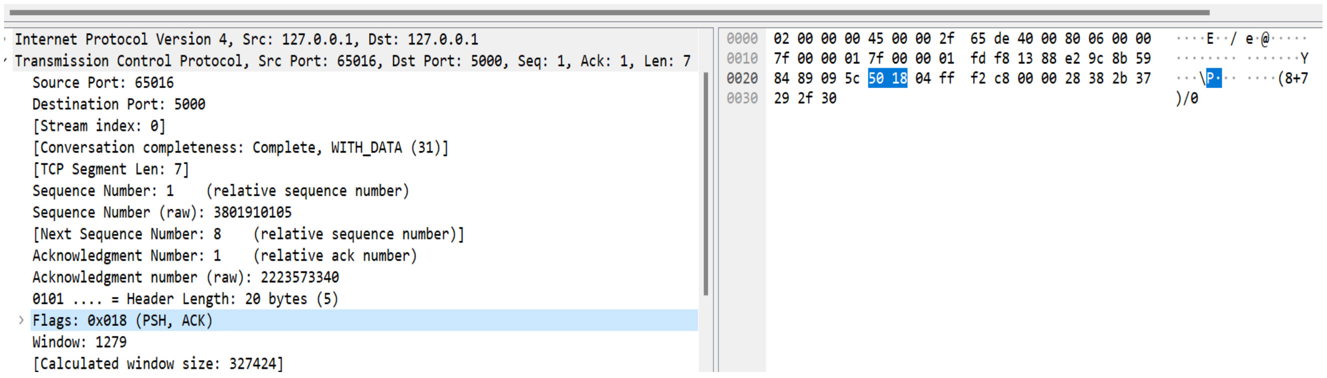
```
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
⌄ Transmission Control Protocol, Src Port: 65016, Dst Port: 5000, Seq: 1, Ack: 1, Len: 0
    Source Port: 65016
    Destination Port: 5000
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3801910105
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 2223573340
    0101 .... = Header Length: 20 bytes (5)
  › Flags: 0x010 (ACK)
    Window: 1279
    [Calculated window size: 327424]
    [Window size scaling factor: 256]
```

```
0000  02 00 00 00 45 00 00 28  65 dd 40 00 80 06 00 00   ····E··( e·@·····
0010  7f 00 00 01 7f 00 00 01  fd f8 13 88 e2 9c 8b 59   ········ ·······Y
0020  84 89 09 5c 50 10 04 ff  9f 76 00 00               ···\P··· ·v··
```

TCP Data Transfer:

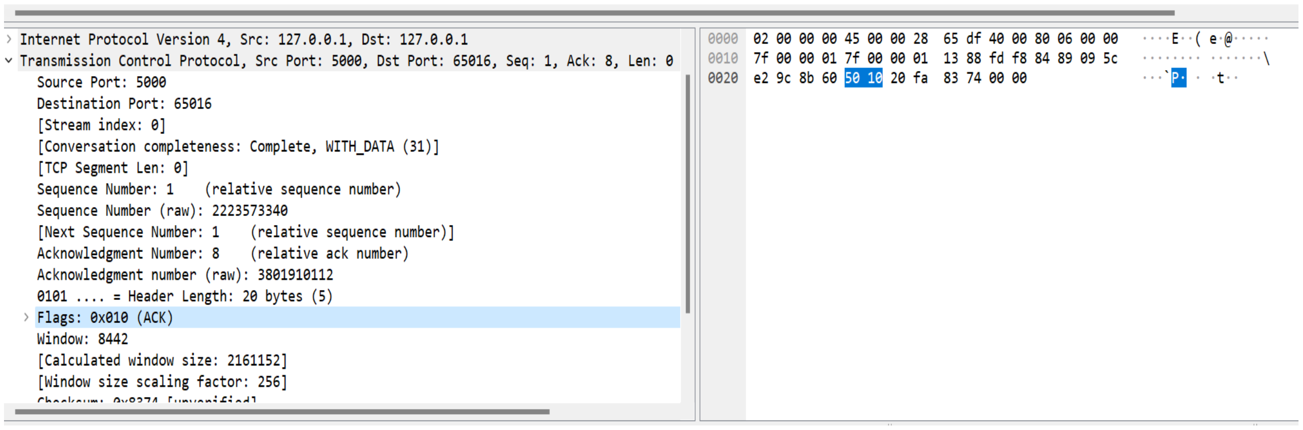


```
4 0.000453   127.0.0.1    127.0.0.1    TCP   51 65016 → 5000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=7
5 0.000491   127.0.0.1    127.0.0.1    TCP   44 5000 → 65016 [ACK] Seq=1 Ack=8 Win=2161152 Len=0
6 0.001155   127.0.0.1    127.0.0.1    TCP   88 5000 → 65016 [PSH, ACK] Seq=1 Ack=8 Win=2161152 Len=44
7 0.001199   127.0.0.1    127.0.0.1    TCP   44 65016 → 5000 [ACK] Seq=8 Ack=45 Win=327424 Len=0
```



```
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
⌄ Transmission Control Protocol, Src Port: 65016, Dst Port: 5000, Seq: 1, Ack: 1, Len: 7
    Source Port: 65016
    Destination Port: 5000
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 7]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3801910105
    [Next Sequence Number: 8    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 2223573340
    0101 .... = Header Length: 20 bytes (5)
  › Flags: 0x018 (PSH, ACK)
    Window: 1279
    [Calculated window size: 327424]
```

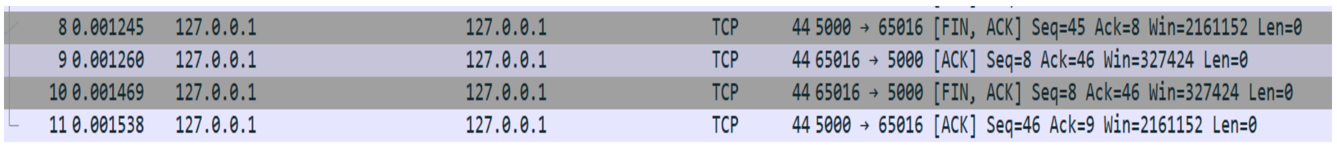
```
0000  02 00 00 00 45 00 00 2f  65 de 40 00 80 06 00 00   ····E··/ e·@·····
0010  7f 00 00 01 7f 00 00 01  fd f8 13 88 e2 9c 8b 59   ········ ·······Y
0020  84 89 09 5c 50 18 04 ff  f2 c8 00 00 28 38 2b 37   ···\P··· ····(8+7
0030  29 2f 30                                           )/0
```



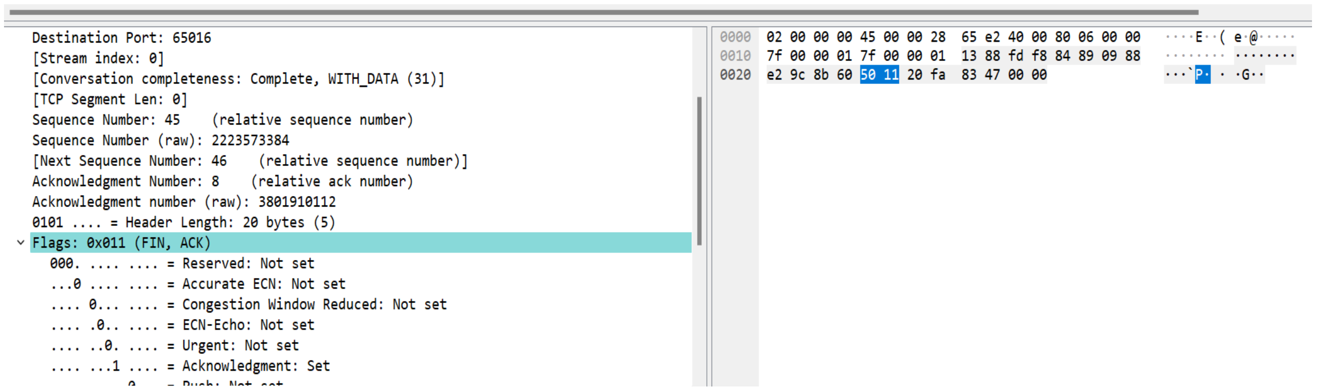
```
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
⌄ Transmission Control Protocol, Src Port: 5000, Dst Port: 65016, Seq: 1, Ack: 8, Len: 0
    Source Port: 5000
    Destination Port: 65016
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 2223573340
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 8    (relative ack number)
    Acknowledgment number (raw): 3801910112
    0101 .... = Header Length: 20 bytes (5)
  › Flags: 0x010 (ACK)
    Window: 8442
    [Calculated window size: 2161152]
    [Window size scaling factor: 256]
    Checksum: 0x8374 [unverified]
```

```
0000  02 00 00 00 45 00 00 28  65 df 40 00 80 06 00 00   ····E··( e·@·····
0010  7f 00 00 01 7f 00 00 01  13 88 fd f8 84 89 09 5c   ········ ·······\
0020  e2 9c 8b 60 50 10 20 fa  83 74 00 00               ···`P· · ·t··
```

TCP Connection Termination:



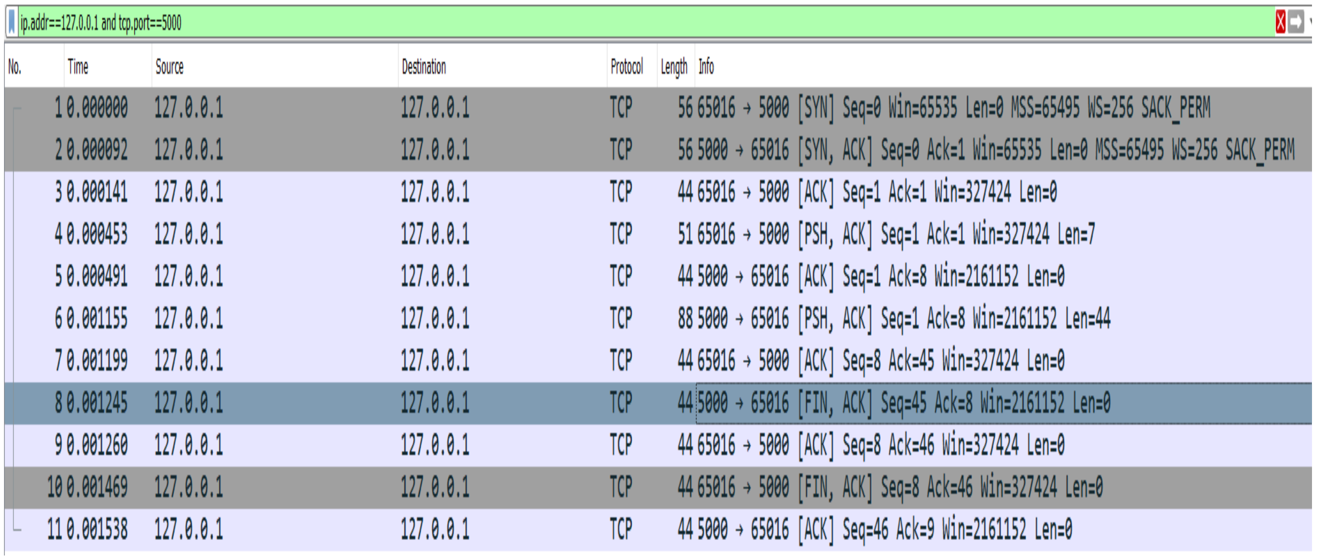
```
 8 0.001245   127.0.0.1    127.0.0.1    TCP   44 5000 → 65016 [FIN, ACK] Seq=45 Ack=8 Win=2161152 Len=0
 9 0.001260   127.0.0.1    127.0.0.1    TCP   44 65016 → 5000 [ACK] Seq=8 Ack=46 Win=327424 Len=0
10 0.001469   127.0.0.1    127.0.0.1    TCP   44 65016 → 5000 [FIN, ACK] Seq=8 Ack=46 Win=327424 Len=0
11 0.001538   127.0.0.1    127.0.0.1    TCP   44 5000 → 65016 [ACK] Seq=46 Ack=9 Win=2161152 Len=0
```



```
  Destination Port: 65016
  [Stream index: 0]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 0]
  Sequence Number: 45    (relative sequence number)
  Sequence Number (raw): 2223573384
  [Next Sequence Number: 46    (relative sequence number)]
  Acknowledgment Number: 8    (relative ack number)
  Acknowledgment number (raw): 3801910112
  0101 .... = Header Length: 20 bytes (5)
⌄ 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
```

```
0000  02 00 00 00 45 00 00 28  65 e2 40 00 80 06 00 00   ····E··( e·@·····
0010  7f 00 00 01 7f 00 00 01  13 88 fd f8 84 89 09 88   ········ ········
0020  e2 9c 8b 60 50 11 20 fa  83 47 00 00               ···`P· · ·G··
```

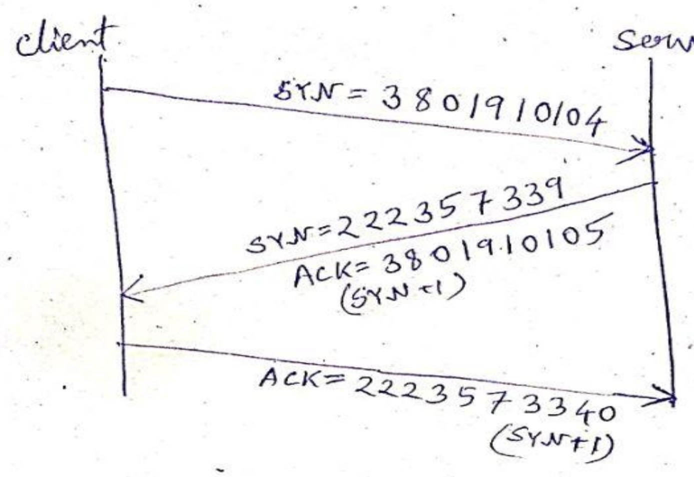
Final Capture:



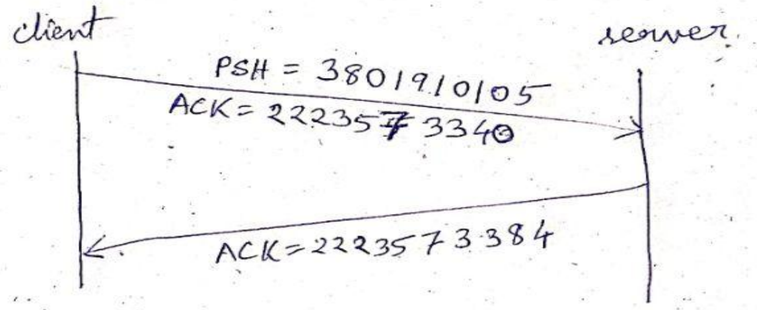
ip.addr==127.0.0.1 and tcp.port==5000

```
No.  Time       Source      Destination  Protocol  Length  Info
 1 0.000000   127.0.0.1    127.0.0.1    TCP   56 65016 → 5000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
 2 0.000092   127.0.0.1    127.0.0.1    TCP   56 5000 → 65016 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
 3 0.000141   127.0.0.1    127.0.0.1    TCP   44 65016 → 5000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
 4 0.000453   127.0.0.1    127.0.0.1    TCP   51 65016 → 5000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=7
 5 0.000491   127.0.0.1    127.0.0.1    TCP   44 5000 → 65016 [ACK] Seq=1 Ack=8 Win=2161152 Len=0
 6 0.001155   127.0.0.1    127.0.0.1    TCP   88 5000 → 65016 [PSH, ACK] Seq=1 Ack=8 Win=2161152 Len=44
 7 0.001199   127.0.0.1    127.0.0.1    TCP   44 65016 → 5000 [ACK] Seq=8 Ack=45 Win=327424 Len=0
 8 0.001245   127.0.0.1    127.0.0.1    TCP   44 5000 → 65016 [FIN, ACK] Seq=45 Ack=8 Win=2161152 Len=0
 9 0.001260   127.0.0.1    127.0.0.1    TCP   44 65016 → 5000 [ACK] Seq=8 Ack=46 Win=327424 Len=0
10 0.001469   127.0.0.1    127.0.0.1    TCP   44 65016 → 5000 [FIN, ACK] Seq=8 Ack=46 Win=327424 Len=0
11 0.001538   127.0.0.1    127.0.0.1    TCP   44 5000 → 65016 [ACK] Seq=46 Ack=9 Win=2161152 Len=0
```

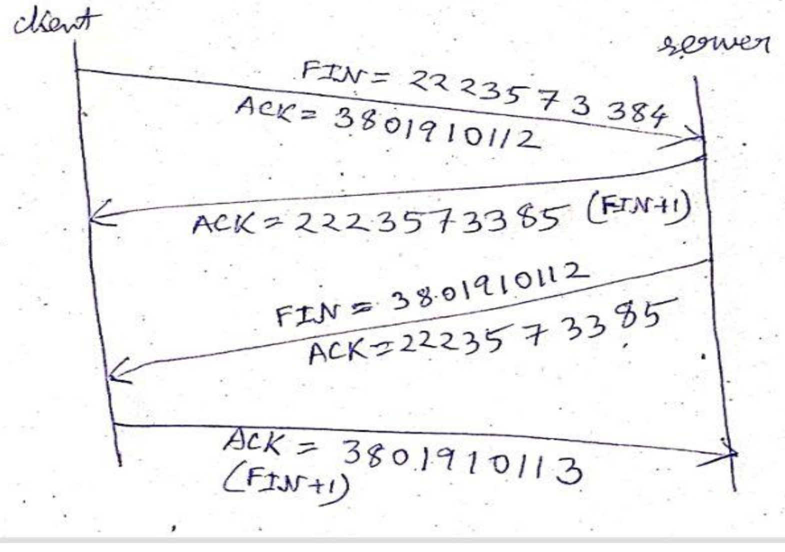
Connection establishment



Data transfer



Connection termination



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.

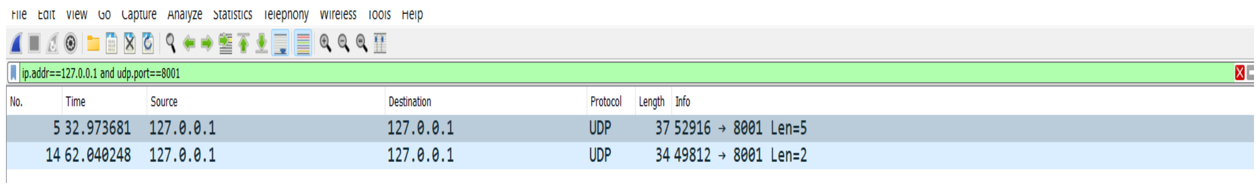
Step1: Capture a Trace

(i) Launch Wireshark
(ii) From Capture→Options select Loopback interface
(iii) 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.
(iv) Run the UDP server program on a terminal.
(v) Run multiple instances of the UDP client program on separate terminals and send requests to the sever.
(vi) Stop Wireshark capture

Step2: Inspect the Trace

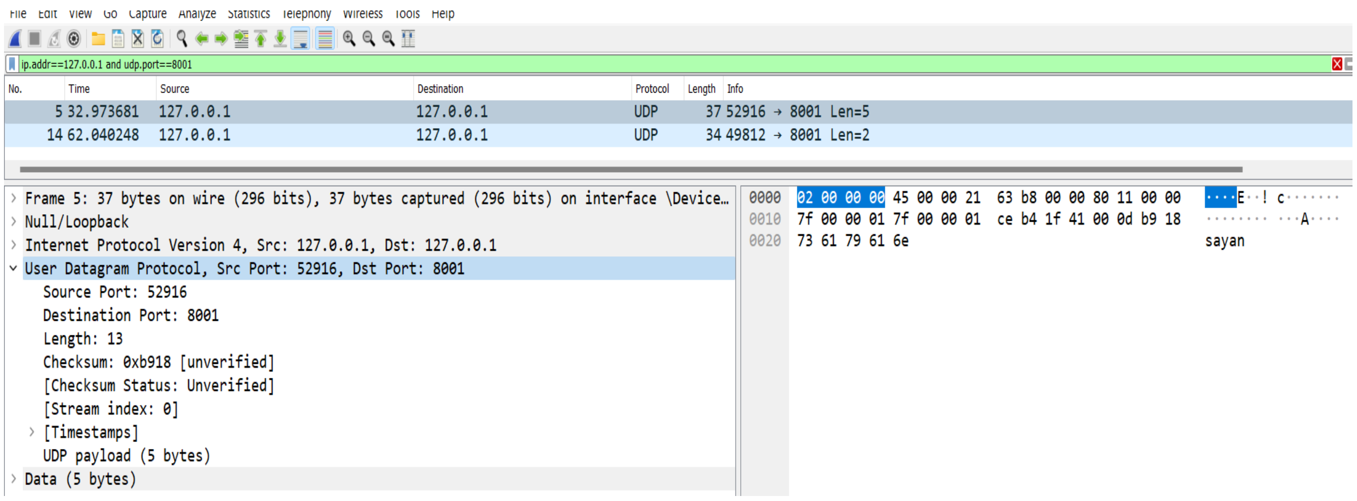
Select different packets in the trace and browse the expanded UDP header and record the following fields:
• Source Port: the port from which the udp segment is sent.
• Destination Port: the port to which the udp segment is sent.
• Length: the length of the UDP segment.

Capturing in Wireshark


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr==127.0.0.1 and udp.port==8001

No.	Time	Source	Destination	Protocol	Length	Info
5	32.973681	127.0.0.1	127.0.0.1	UDP	37	52916 → 8001 Len=5
14	62.040248	127.0.0.1	127.0.0.1	UDP	34	49812 → 8001 Len=2


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
ip.addr==127.0.0.1 and udp.port==8001

No.   Time       Source      Destination  Protocol  Length  Info
    5 32.973681  127.0.0.1   127.0.0.1    UDP       37      52916 → 8001 Len=5
   14 62.040248  127.0.0.1   127.0.0.1    UDP       34      49812 → 8001 Len=2

> Frame 5: 37 bytes on wire (296 bits), 37 bytes captured (296 bits) on interface \Device…
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v User Datagram Protocol, Src Port: 52916, Dst Port: 8001
    Source Port: 52916
    Destination Port: 8001
    Length: 13
    Checksum: 0xb918 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 0]
  > [Timestamps]
    UDP payload (5 bytes)
> Data (5 bytes)

0000  02 00 00 00 45 00 00 21  63 b8 00 00 80 11 00 00   ····E··! c·······
0010  7f 00 00 01 7f 00 00 01  ce b4 1f 41 00 0d b9 18   ········ ···A····
0020  73 61 79 61 6e                                     sayan
```

Source port : 52916

Destination port : 8001

Length : 13