

Jadavpur University

Data Communication and Computer Networks Lab Report

Name : Himangshu Bauri

Roll : 002210503009

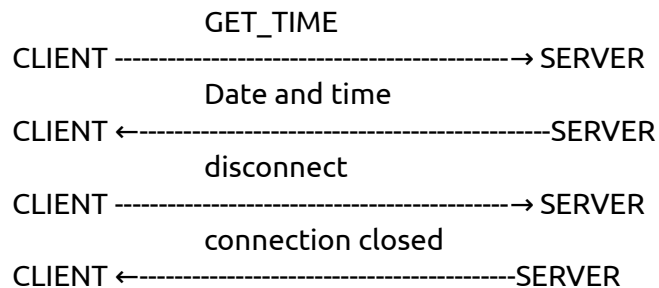
Dept. : CSE

Class : MCA 2nd Year- 1st Sem

Assignment 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 request to the server to get the current time and date. Choose your own formats for the request/reply messages.



Code:

Server side

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

#bind the port
server_socket.bind(('127.0.0.1', 12345))

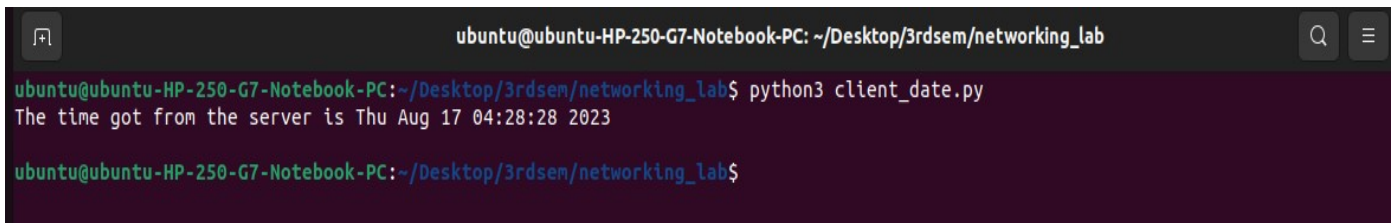
#queue upto 5 request
server_socket.listen(5)
while True:
    print("Waiting for connection")
    #establish a connection
    client_socket, addr = server_socket.accept()
    print("Got a connection from %s" %str(addr))
    currentTime = time.ctime(time.time()) + "\r\n"
    try:
        client_socket.send(currentTime.encode('ascii'))
    except:
        print("Exited by the user")
    client_socket.close()
server_socket.close();
```

Client Side

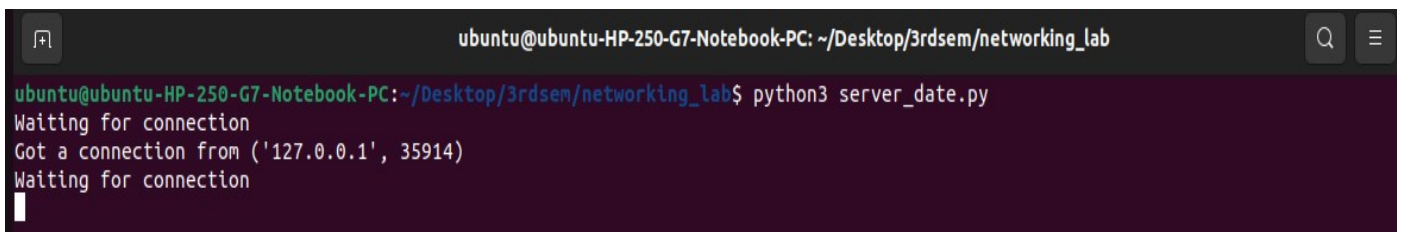
```
import socket
#create a socket object
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#connection to hostname on the port
client_socket.connect(('127.0.0.1', 12345))
data = client_socket.recv(1024) #receive the data from the server
client_socket.close()
print("The time got from the server is %s" %data.decode('ascii'))
```

Sample Run :

A terminal window with a dark background. The title bar reads 'ubuntu@ubuntu-HP-250-G7-Notebook-PC: ~/Desktop/3rdsem/networking_lab'. The prompt is 'ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem/networking_lab\$'. The user enters 'python3 client_date.py'. The output is 'The time got from the server is Thu Aug 17 04:28:28 2023'. The prompt returns.

```
ubuntu@ubuntu-HP-250-G7-Notebook-PC: ~/Desktop/3rdsem/networking_lab
ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem/networking_lab$ python3 client_date.py
The time got from the server is Thu Aug 17 04:28:28 2023
ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem/networking_lab$
```

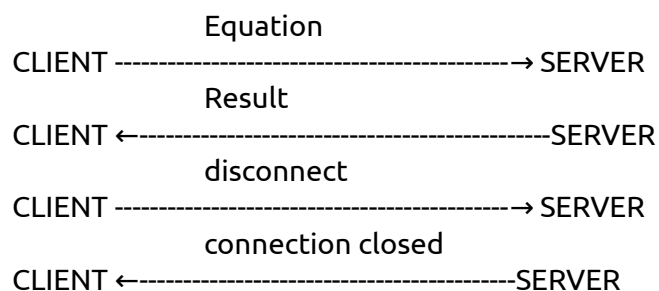
A terminal window with a dark background. The title bar reads 'ubuntu@ubuntu-HP-250-G7-Notebook-PC: ~/Desktop/3rdsem/networking_lab'. The prompt is 'ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem/networking_lab\$'. The user enters 'python3 server_date.py'. The output is 'Waiting for connection', 'Got a connection from ('127.0.0.1', 35914)', and 'Waiting for connection'. The prompt returns.

```
ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem/networking_lab$ python3 server_date.py
Waiting for connection
Got a connection from ('127.0.0.1', 35914)
Waiting for connection

```

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.



Code :

Server side

```
import socket
import ast

def evaluate_expression(expression):
    try:
        # Using the 'eval' function to evaluate the mathematical expression received from the client.
        result = eval(expression)
        return str(result)
    except Exception as e:
        return "Error: " + str(e)

def main():
    host = '127.0.0.1'
    port = 5050
    #create a socket object
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #bind the port
    server_socket.bind((host, port))
    server_socket.listen(5) # Allow up to 5 queued connections
    print("Server is listening on {}:{}".format(host, port))

    while True:
        #establish a connection
        client_socket, client_address = server_socket.accept()
        print("Connection established with {}:{}".format(client_address[0], client_address[1]))

        try:
            while True:
                data = client_socket.recv(1024).decode('utf-8')

                if data.lower() == "disconnect":
                    print("Client {}:{} disconnected.".format(client_address[0], client_address[1]))
                    break

                result = evaluate_expression(data)
                """ Converting the result of the expression evaluation to a UTF-8 encoded byte string and sending it back to the client over the network using the 'client_socket'."""
                client_socket.sendall(result.encode('utf-8'))

            except Exception as e:
                print("Error occurred while handling client request:", e)
            finally:
                # Close the client socket
                client_socket.close()
                print("Connection with {}:{} closed\n".format(client_address[0], client_address[1]))

if __name__ == "__main__":
    main()
```

Client Side

```
import socket

def main():
    host = '127.0.0.1'
    port = 5050
    #create a socket object
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

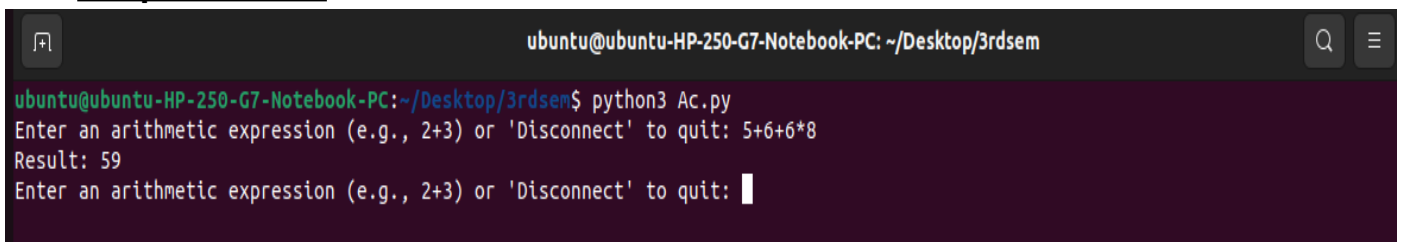
    try:
        #connection to hostname on the port
        client_socket.connect((host, port))

        while True:
            expression = input("Enter an arithmetic expression (e.g., 2+3) or 'Disconnect' to quit: ")

            if expression.lower() == 'disconnect':
                """ Converting the result of the expression evaluation to a UTF-8 encoded
                byte string and sending it back to the client over the network using the
                'client_socket'."""
                client_socket.sendall(expression.encode('utf-8'))
                print("Disconnected from the server.")
                break
            """ Converting the result of the expression evaluation to a UTF-8 encoded
            byte string and sending it back to the client over the network using the
            'client_socket'."""
            client_socket.sendall(expression.encode('utf-8'))
            result = client_socket.recv(1024).decode('utf-8')
            print("Result:", result)
        except Exception as e:
            print("Error occurred:", e)
        finally:
            client_socket.close()

if __name__ == "__main__":
    main()
```

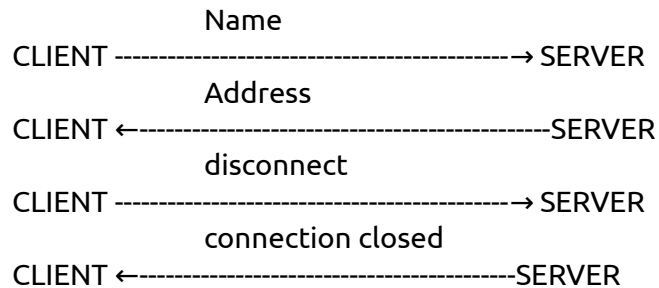
Sample Run :



```
ubuntu@ubuntu-HP-250-G7-Notebook-PC: ~/Desktop/3rdsem
ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem$ python3 Ac.py
Enter an arithmetic expression (e.g., 2+3) or 'Disconnect' to quit: 5+6*8
Result: 59
Enter an arithmetic expression (e.g., 2+3) or 'Disconnect' to quit: █
```

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. Choose your own formats for the request/reply messages.



Code:

Server side

```
import socket

def find_address(student_name):
    try:
        with open('student_addresses.txt', 'r') as file:
            for line in file:
                name, address = line.strip().split(',')
                if name.lower() == student_name.lower():
                    return address
    except Exception as e:
        print("Error:", e)
    return "Address not found"

def main():
    #create a socket object
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    host = '127.0.0.1'
    port = 12345
    #bind the port
    server_socket.bind((host, port))

    print(f"UDP server listening on {host}:{port}")

    while True:
        # Receiving data from the client using the UDP socket 'server_socket'.

        data, client_address = server_socket.recvfrom(1024)
        # Decoding the received data, which is in bytes, into a UTF-8 encoded string.

        student_name = data.decode('utf-8')

        if student_name.lower() == 'exit':
            print("Client disconnected.")
            break

        address = find_address(student_name)
```

```
""" Converting the 'address' string into a UTF-8 encoded byte strin and sending
it back to the client using the 'server_socket'. """
```

```
server_socket.sendto(address.encode('utf-8'), client_address)
```

```
server_socket.close()
```

```
if __name__ == '__main__':
    main()
```

Client Side

```
import socket
```

```
def main():
```

```
    #create a socket object
```

```
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
    host = '127.0.0.1'
```

```
    port = 12345
```

```
    print("UDP client running.")
```

```
    while True:
```

```
        student_name = input("Enter student's name (or 'exit' to quit): ")
```

```
        """ Converting the 'student_name' string into a UTF-8 encoded byte string
and sending it to the server using the 'client_socket'. """
```

```
        client_socket.sendto(student_name.encode('utf-8'), (host, port))
```

```
        if student_name.lower() == 'exit':
```

```
            print("Exiting.")
```

```
            break
```

```
        # Receiving data from the server using the UDP socket 'client_socket'.
```

```
        address, server_address = client_socket.recvfrom(1024)
```

```
        print(f"Permanent address of {student_name}: {address.decode('utf-8')}")
```

```
    client_socket.close()
```

```
if __name__ == '__main__':
    main()
```

Sample Run :

```
ubuntu@ubuntu-HP-250-G7-Notebook-PC: ~/Desktop/3rdsem/networking_lab
ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem/networking_lab$ python3 udp_server.py
UDP server listening on 127.0.0.1:12345
```

```
ubuntu@ubuntu-HP-250-G7-Notebook-PC: ~/Desktop/3rdsem/networking_lab
ubuntu@ubuntu-HP-250-G7-Notebook-PC:~/Desktop/3rdsem/networking_lab$ python3 udp_clint.py
UDP client running.
Enter student's name (or 'exit' to quit): Himangshu
Permanent address of Himangshu: purulia
Enter student's name (or 'exit' to quit):
```

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

(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==12345						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	56504 → 12345 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSva
2	0.000036406	127.0.0.1	127.0.0.1	TCP	74	12345 → 56504 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SAC
3	0.000067878	127.0.0.1	127.0.0.1	TCP	66	56504 → 12345 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=492703536 TS

Destination Port: 12345
[Stream index: 0]
[Conversation completeness: Incomplete, ESTABLISHED (7)]
[TCP Segment Len: 0]
Sequence Number: 0 (relative sequence number)
Sequence Number (raw): 2813459265
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 0
Acknowledgment number (raw): 0
1010 = Header Length: 40 bytes (10)
Flags: 0x002 (SYN)
Window: 65495
[Calculated window size: 65495]
Checksum: 0xfe30 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation
[Timestamps]

0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 0
0010 00 3c 1b f7 40 00 40 06 20 c3 7f 00 00 01 7f 0
0020 00 01 dc b8 30 39 a7 b1 fb 41 00 00 00 00 a0 0
0030 ff d7 fe 30 00 00 02 04 ff d7 04 02 08 0a 1d 5
0040 0f 30 00 00 00 00 01 03 03 07

Frame 2: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, i
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:0
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 12345, Dst Port: 33386, Seq: 0, Ack: 1, Len:
Source Port: 12345
Destination Port: 33386
[Stream index: 0]
[Conversation completeness: Incomplete, ESTABLISHED (7)]
[TCP Segment Len: 0]
Sequence Number: 0 (relative sequence number)
Sequence Number (raw): 3013306436
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 368384901
1010 = Header Length: 40 bytes (10)
Flags: 0x012 (SYN, ACK)
Window: 65483

0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 0
0010 00 3c 00 00 40 00 40 06 3c ba 7f 00 00 01 7f 0
0020 00 01 30 39 82 6a b3 9b 68 44 15 f5 1b 85 a0 1
0030 ff cb fe 30 00 00 02 04 ff d7 04 02 08 0a 1d 6
0040 57 f3 1d 64 57 f3 01 03 03 07

Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:0
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 33386, Dst Port: 12345, Seq: 1, Ack: 1, Len:
Source Port: 33386
Destination Port: 12345
[Stream index: 0]
[Conversation completeness: Incomplete, ESTABLISHED (7)]
[TCP Segment Len: 0]
Sequence Number: 1 (relative sequence number)
Sequence Number (raw): 368384901
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 3013306437
1000 = Header Length: 32 bytes (8)
Flags: 0x010 (ACK)
Window: 512
[Calculated window size: 65536]
[Window size scaling factor: 128]
Checksum: 0xfe28 [unverified]

0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 0
0010 00 34 4a cf 40 00 40 06 f1 f2 7f 00 00 01 7f 0
0020 00 01 82 6a 30 39 15 f5 1b 85 b3 9b 68 45 80 1
0030 02 00 fe 28 00 00 01 01 08 0a 1d 64 57 f3 1d 6
0040 57 f3

TCP Data Transfer:

Capturing from Loopback: lo

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr==127.0.0.1 and tcp.port==12345

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	33386 → 12345 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=
2	0.000029388	127.0.0.1	127.0.0.1	TCP	74	12345 → 33386 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK
3	0.000058962	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=493115379 TS
4	273.219107...	127.0.0.1	127.0.0.1	TCP	78	33386 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=49338859
5	273.219140...	127.0.0.1	127.0.0.1	TCP	66	12345 → 33386 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=493388598 TS
6	273.219366...	127.0.0.1	127.0.0.1	TCP	100	12345 → 33386 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=34 TSval=49338859
7	273.219390...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=13 Ack=35 Win=65536 Len=0 TSval=493388599

▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ Transmission Control Protocol, Src Port: 33386, Dst Port: 12345, Seq: 1, Ack: 1, Len: 0

Source Port: 33386

Destination Port: 12345

[Stream index: 0]

[Conversation completeness: Incomplete, DATA (15)]

[TCP Segment Len: 12]

Sequence Number: 1 (relative sequence number)

Sequence Number (raw): 368384901

[Next Sequence Number: 13 (relative sequence number)]

Acknowledgment Number: 1 (relative ack number)

Acknowledgment number (raw): 3013306437

1000 = Header Length: 32 bytes (8)

Flags: 0x018 (PSH, ACK)

Window: 512

[Calculated window size: 65536]

[Window size scaling factor: 128]

Checksum: 0xfe34 [unverified]

This shows the raw value of the sequence number (tcp.seq_raw), 4 bytes

Packets: 7 · Displayed: 7 (100.0%)

Profile: Default

Capturing from Loopback: lo

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr==127.0.0.1 and tcp.port==12345

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	33386 → 12345 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=
2	0.000029388	127.0.0.1	127.0.0.1	TCP	74	12345 → 33386 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK
3	0.000058962	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=493115379 TS
4	273.219107...	127.0.0.1	127.0.0.1	TCP	78	33386 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=49338859
5	273.219140...	127.0.0.1	127.0.0.1	TCP	66	12345 → 33386 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=493388598 TS
6	273.219366...	127.0.0.1	127.0.0.1	TCP	100	12345 → 33386 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=34 TSval=49338859
7	273.219390...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=13 Ack=35 Win=65536 Len=0 TSval=493388599

▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ Transmission Control Protocol, Src Port: 12345, Dst Port: 33386, Seq: 1, Ack: 13, Len: 0

Source Port: 12345

Destination Port: 33386

[Stream index: 0]

[Conversation completeness: Incomplete, DATA (15)]

[TCP Segment Len: 34]

Sequence Number: 1 (relative sequence number)

Sequence Number (raw): 3013306437

[Next Sequence Number: 35 (relative sequence number)]

Acknowledgment Number: 13 (relative ack number)

Acknowledgment number (raw): 368384913

1000 = Header Length: 32 bytes (8)

Flags: 0x018 (PSH, ACK)

Window: 512

[Calculated window size: 65536]

[Window size scaling factor: 128]

Checksum: 0xfe4a [unverified]

This shows the raw value of the sequence number (tcp.seq_raw), 4 bytes

Packets: 7 · Displayed: 7 (100.0%)

Profile: Default

TCP Connection Termination:

Capturing from Loopback: lo

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr==127.0.0.1 and tcp.port==12345

No.	Time	Source	Destination	Protocol	Length	Info
2	0.000029388	127.0.0.1	127.0.0.1	TCP	74	12345 → 33386 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SA...
3	0.000058962	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=493115379 T...
4	273.219107...	127.0.0.1	127.0.0.1	TCP	78	33386 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=49338...
5	273.219140...	127.0.0.1	127.0.0.1	TCP	66	12345 → 33386 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=493388598...
6	273.219366...	127.0.0.1	127.0.0.1	TCP	100	12345 → 33386 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=34 TSval=4933...
7	273.219390...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=13 Ack=35 Win=65536 Len=0 TSval=493388599...
8	580.307193...	127.0.0.1	127.0.0.1	TCP	70	33386 → 12345 [PSH, ACK] Seq=13 Ack=35 Win=65536 Len=4 TSval=4936...
9	580.307273...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [FIN, ACK] Seq=17 Ack=35 Win=65536 Len=0 TSval=4936...
10	580.308518...	127.0.0.1	127.0.0.1	TCP	66	12345 → 33386 [FIN, ACK] Seq=35 Ack=18 Win=65536 Len=0 TSval=4936...
11	580.308579...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=18 Ack=36 Win=65536 Len=0 TSval=493695688...

▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ Transmission Control Protocol, Src Port: 33386, Dst Port: 12345, Seq: 17, Ack: 35, Len: 0

Source Port: 33386

Destination Port: 12345

[Stream index: 0]

[Conversation completeness: Complete, WITH_DATA (31)]

[TCP Segment Len: 0]

Sequence Number: 17 (relative sequence number)

Sequence Number (raw): 368384917

[Next Sequence Number: 18 (relative sequence number)]

Acknowledgment Number: 35 (relative ack number)

Acknowledgment number (raw): 3013306471

1000 = Header Length: 32 bytes (8)

▶ Flags: 0x011 (FIN, ACK)

Window: 512

[Calculated window size: 65536]

[Window size scaling factor: 128]

Checksum: 0xfe28 [unverified]

This shows the raw value of the sequence number (tcp.seq_raw), 4 bytes

Packets: 11 · Displayed: 11 (100.0%)

Profile: Default

FINAL CAPTURE:

Capturing from Loopback: lo

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

ip.addr==127.0.0.1 and tcp.port==12345

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	33386 → 12345 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=...
2	0.000029388	127.0.0.1	127.0.0.1	TCP	74	12345 → 33386 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK...
3	0.000058962	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=493115379 TS...
4	273.219107...	127.0.0.1	127.0.0.1	TCP	78	33386 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=493388...
5	273.219140...	127.0.0.1	127.0.0.1	TCP	66	12345 → 33386 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=493388598...
6	273.219366...	127.0.0.1	127.0.0.1	TCP	100	12345 → 33386 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=34 TSval=49338...
7	273.219390...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=13 Ack=35 Win=65536 Len=0 TSval=493388599...
8	580.307193...	127.0.0.1	127.0.0.1	TCP	70	33386 → 12345 [PSH, ACK] Seq=13 Ack=35 Win=65536 Len=4 TSval=49369...
9	580.307273...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [FIN, ACK] Seq=17 Ack=35 Win=65536 Len=0 TSval=49369...
10	580.308518...	127.0.0.1	127.0.0.1	TCP	66	12345 → 33386 [FIN, ACK] Seq=35 Ack=18 Win=65536 Len=0 TSval=49369...
11	580.308579...	127.0.0.1	127.0.0.1	TCP	66	33386 → 12345 [ACK] Seq=18 Ack=36 Win=65536 Len=0 TSval=493695688...

▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)

▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

▶ Transmission Control Protocol, Src Port: 33386, Dst Port: 12345, Seq: 17, Ack: 35, Len: 0

Source Port: 33386

Destination Port: 12345

[Stream index: 0]

[Conversation completeness: Complete, WITH_DATA (31)]

[TCP Segment Len: 0]

Sequence Number: 17 (relative sequence number)

Sequence Number (raw): 368384917

[Next Sequence Number: 18 (relative sequence number)]

This shows the raw value of the sequence number (tcp.seq_raw), 4 bytes

Packets: 13 · Displayed: 11 (84.6%)

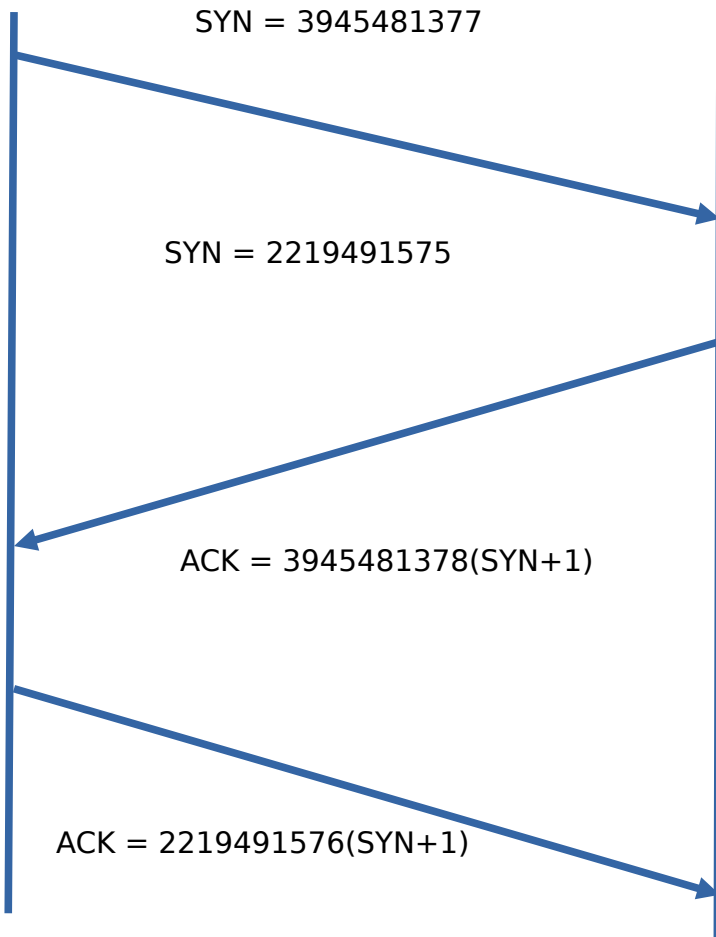
Profile: Default

Time Sequence Diagram:

Connection Establishment:

Client

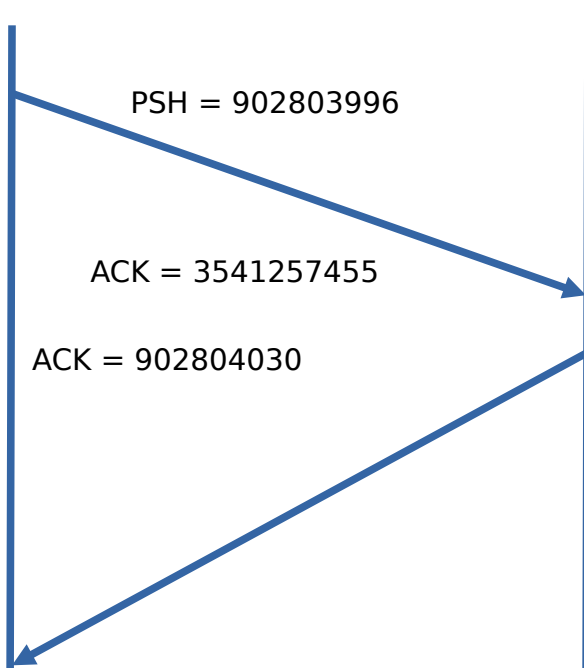
Server



Data Transfer:

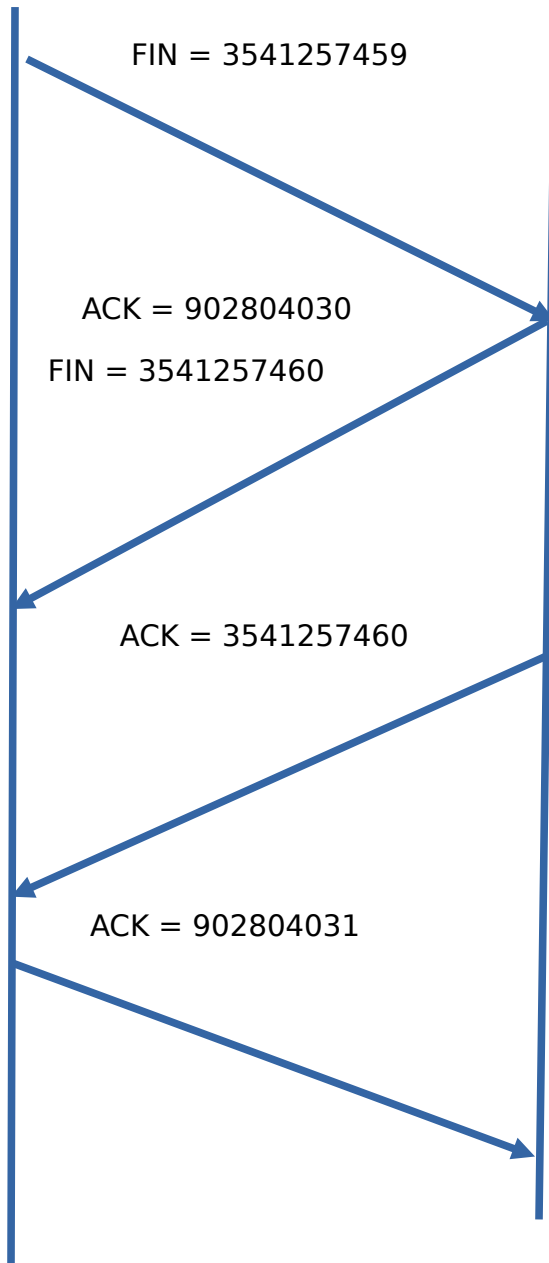
Client

Server



Connection Termination:

ACK = 902804030



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

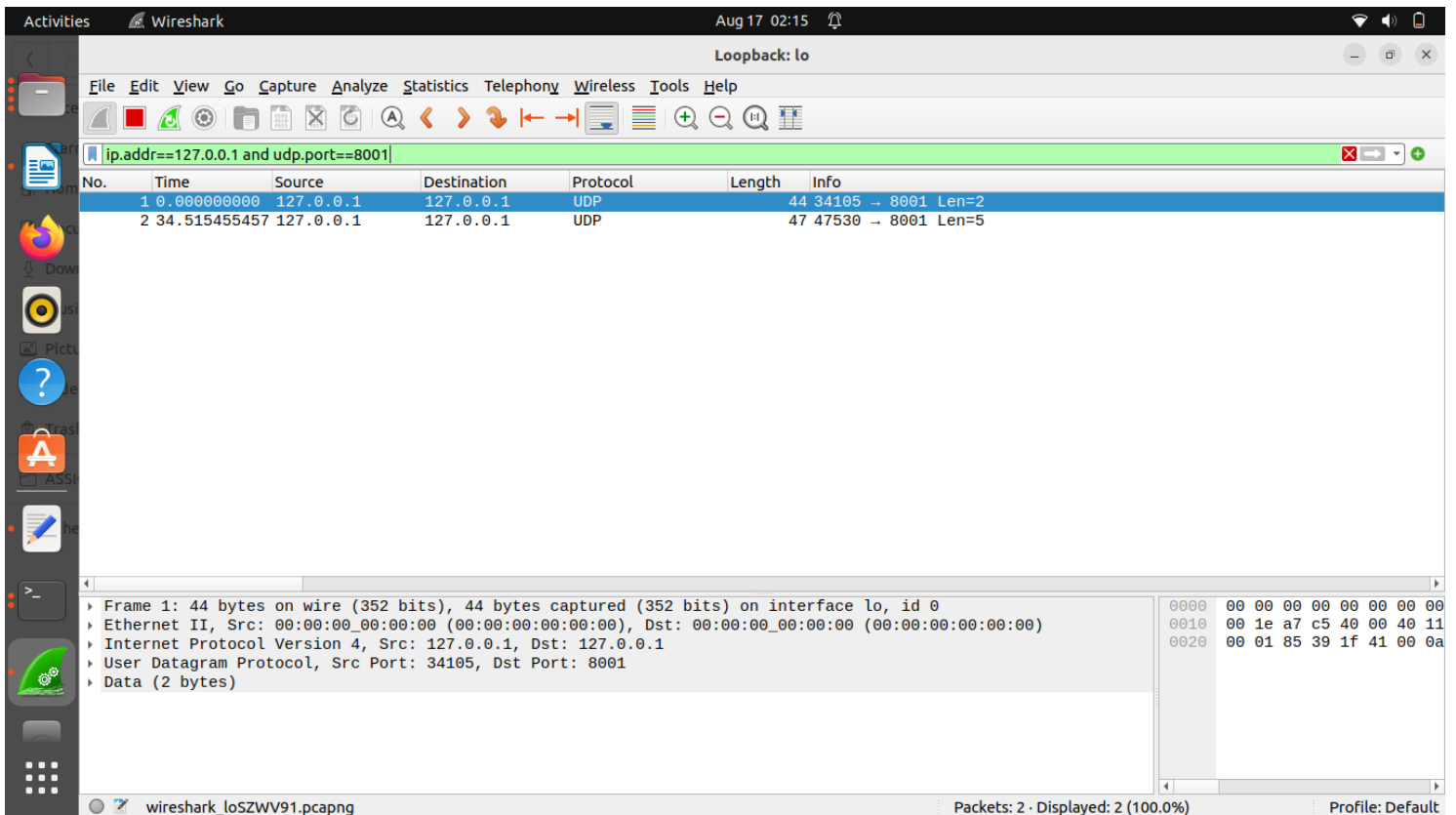
- (i) Launch Wireshark
- (ii) From Capture→Options select Loopback interface
- (iii) Start a capture with a filter of “ip.addr==127.0.0.1 and udp.port==xxxx”, where xxxx is the port number used by the UDP server.
- (iv) Run the UDP server program on a terminal.
- (v) Run multiple instances of the UDP 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



User Datagram Protocol, Src Port: 34258, Dst Port: 8001

Source Port: 34258

Destination Port: 8001

Length: 10