Data Communication and Computer Network Laboratory

ASSIGNMENT

MCA 1st year 2nd Sem

Name	Roll-no
Dwip Shekhar Mondal	002210503037

Assignment - I

Questions →

1. 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. 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.

3. 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.

Your report should contain at least the following sections →

1. Problem Statement
2. Your design of request/reply protocol
3. Source code (with appropriate comments)
4. Sample run

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 of Request/Reply protocol →

➔ Client sends a request to the server in the format of the of a serialized JSON object with the following structure

```
request = {
    method: 'GET | POST | PUT | DELETE',
    msg: "message"
}
```

The method field represents the type of the request i.e the method of the request that is sent to the server. (The server is designed to only accept a request with method type GET)

The msg field holds the command string that the client sends the server.

➔ The server responds to a requesting client by →

- Deserialize the request sent by the client to get the JSON object sent by the client.
- If the request method is not GET then send an Error message in the format of serialized JSON object of the following structure →

```
response = {
    status: 500,
    data: 'unknown command or command not supported'
}
```

- Otherwise send the client the current date and time in the format of serialized JSON object of the following structure →

```
response = {
    status: 200,
    data: 'date-time string'
}
```

- General Response object structure →

```
response = {
    status: 200 | 500 (for non-error or error response),
    data: 'date-time string | Error message'
}
```

Code →

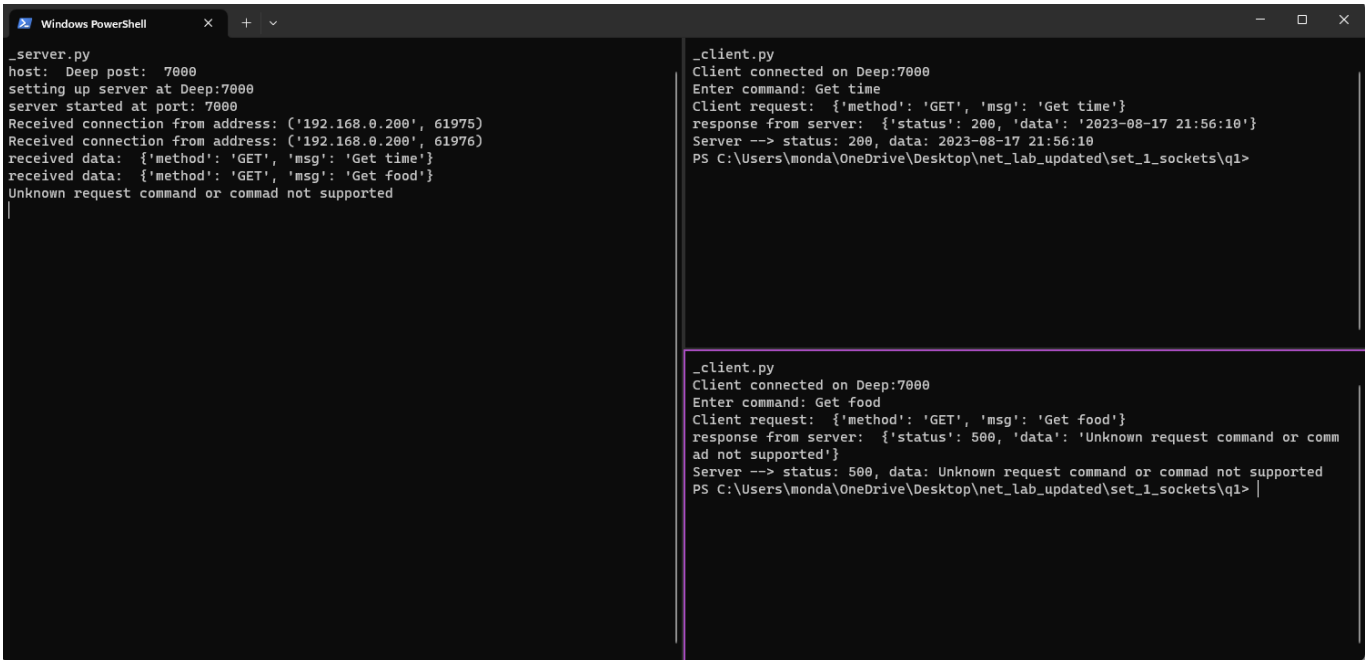
```
#server.py

from datetime import datetime
import socket
import signal
import sys
import json
import threading


class SokServer(object):
    """
    Socket server class

    reject any request with methods other than GET (for now)

    request struct --> {
        method : GET | POST | PUT | DELETE
        msg: "message"
    }

    response struct --> {
        status: NUMBER
        data: response data OR error msg
    }

    """

    def __init__(self, port=9999):
        self.host = socket.gethostname().split('.')[0]
        self.port = port
        print("host: ", self.host, "post: ", self.port)


    def start_server(self):
        """
        Start socket server
        """

        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:

            print("setting up server at {host}:{port}".format(host=self.host,
port=self.port))
            self.socket.bind((self.host, self.port))
            print("server started at port: {port}".format(port=self.port))
```

```
        except Exception as e:
            print("Error setting up server")
            self.stop_server()
            sys.exit(1)

        # start listening for incoming connection
        self.listen()

    def listen(self):
        """
        listen for incoming connection/request to the server
        """

        self.socket.listen(5)

        while True:

            clientSok, addr = self.socket.accept()
            # clientSok.settimeout(30)
            print(f"Received connection from address: {addr}")

            ## TODO: handle concurrent connection(Threads)

            threading.Thread(target=self.handle_request, args=(clientSok, addr)).start()
            # self.handle_request(clientSok, addr)

    def handle_request(self, client, address):
        """
            handle incoming request and send appropriate response
        """
        PACKET = 1024

        try:
            req = client.recv(PACKET)
            decoded_data = json.loads(req.decode())
            print("received data: ", decoded_data)


            # Reject any request other than req with GET method


            if decoded_data["method"] != "GET":
                raise Exception("Unknown request method or method not supported")


            if decoded_data['msg'] != "Get time":
                raise Exception("Unknown request command or command not supported")
```

```
            response = {
                "status" : 200,
                "data": str(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
            }

            json_response = json.dumps(response)
            client.sendall(json_response.encode())
            client.close()

        except Exception as e:
            print(e)
            err_res = {
                "status" : 500,
                "data": str(e)
            }

            client.sendall(json.dumps(err_res).encode())
            client.close()

    def stop_server(self):
        """
        Shutdown server
        """
        try:
            print("Shutting down server ... ")
            self.socket.shutdown(socket.SHUT_RDWR)
            self.socket.close()
            sys.exit(1)

        except Exception as e:
            pass


def shutdownServer(sig, unused):
    """
    Shutdown server from a SIGINT received signal
    """
    server.stop_server()
    sys.exit(1)

signal.signal(signal.SIGINT, shutdownServer)
server = SokServer(7000)
server.start_server()
```

```python
#Client.py

import socket
import json



HOST = socket.gethostname().split('.')[0]
PORT = 7000
BUFF_SIZE = 1024


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as skt:

    skt.connect((HOST, PORT))
    req = {}
    req["method"] = "GET"

    print(f"Client connected on {HOST}:{PORT}")

    msg = input("Enter command: ")

    # msg = "What's the current date ?>"

    req["msg"] = msg

    skt.sendall(json.dumps(req).encode())

    print("Client request: ", req)

    data = skt.recv(BUFF_SIZE)

    res = json.loads(data.decode())

    print("response from server: ", res)

    print(f"Server --> status: {res['status']}, data: {res['data']}")

    skt.close()
```

Sample run →



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 of Request/Reply protocol →

➔ Client sends a request to the server in the format of the of a serialized JSON object with the following structure

```
request = {
    method: 'GET | POST | PUT | DELETE',
    msg: "math expression string"
}
```

The method field represents the type of the request i.e the method of the request that is sent to the server. (The server is designed to only accept a request with method type GET)

The msg field holds the mat-expression to be evaluated by the server.

➔ The server responds to a requesting client by →

- ◆ Deserialize the request sent by the client to get the JSON object sent by the client.
- ◆ If the request method is not GET then send an Error message in the format of serialized JSON object of the following structure →

```
response = {
    status: 500,
    data: 'Error message'
}
```

- ◆ Otherwise evaluate the math expression and send the client, the result of the operation in the format of serialized JSON object of the following structure →

```
response = {
    status: 200,
    data: 'evaluated math expression'
}
```

- ◆ General Response object structure →

```
response = {
    status: 200 | 500 (for non-error or error response),
    data: 'evaluated math expression | Error message'
}
```

Code →

```
#server

from datetime import datetime
import socket
import signal
import sys
import json
import threading

class SokServer(object):
    """

    Socket server class
```

```
    reject any request with methods other than GET (for now)

    request struct --> {
        method : GET | POST | PUT | DELETE
        msg: "math expression"
    }

    response struct --> {
        status: NUMBER
        data: response (math expression) data OR error msg
    }

    """
    def __init__(self, port=9999):
        self.host = socket.gethostname().split('.')[0]
        self.port = port
        print("host: ", self.host, "post: ", self.port)

    def start_server(self):
        """
        Start socket server
        """
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:
            print("setting up server at {host}:{port}".format(host=self.host,
port=self.port))
            self.socket.bind((self.host, self.port))
            print("server started at port: {port}".format(port=self.port))

        except Exception as e:
            print("Error setting up server")
            self.stop_server()
            sys.exit(1)

        # start listening for incoming connection

        self.listen()

    def listen(self):
        """
        listen for incoming connection/request to the server
        """
        self.socket.listen(5)

        while True:

            clientSok, addr = self.socket.accept()
```

```
            clientSok.settimeout(30)
            print(f"Received connection from address: {addr}")

            ## TODO: handle concurrent connection(Threads)

            threading.Thread(target=self.handle_request, args=(clientSok, addr)).start()
            # self.handle_request(clientSok, addr)

    def handle_request(self, client, address):
        """
            handle incoming request and send appropriate response
        """

        PACKET = 1024

        try:
            req = client.recv(PACKET)
            decoded_data = json.loads(req.decode())
            print("received data: ", decoded_data)

            # Reject any request other than req with GET method

            if decoded_data["method"] != "GET":
                raise Exception("Unknown request method or method not supported")

            print("client >", decoded_data['msg'])

            res = eval(decoded_data['msg'])

            response = {
                "status" : 200,
                "data": res
            }

            json_response = json.dumps(response)
            client.sendall(json_response.encode())

        except Exception as e:
            print(e)
            err_res = {
                "status" : 500,
                "data": str(e)
            }
            client.sendall(json.dumps(err_res).encode())

    def stop_server(self):
        """
        Shutdown server
```

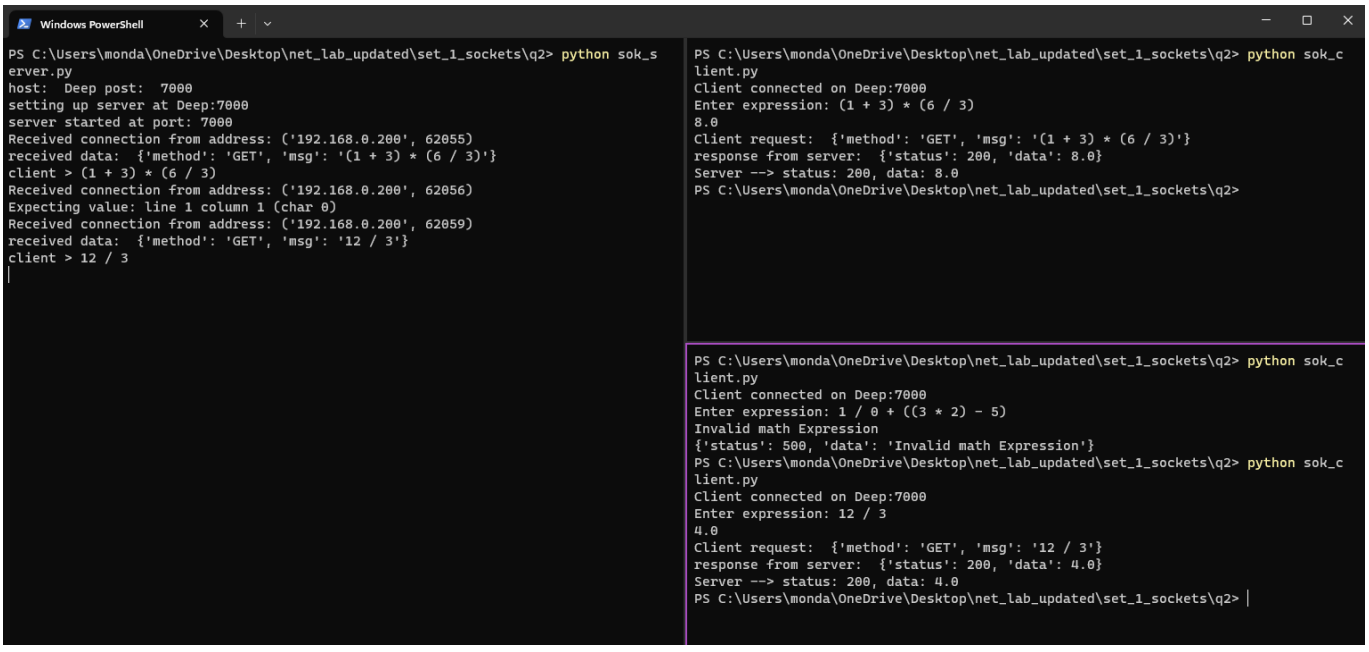
```
        """
        try:
            print("Shutting down server ... ")
            self.socket.shutdown(socket.SHUT_RDWR)
            self.socket.close()
            sys.exit(1)
        except Exception as e:
            pass


def shutdownServer(sig, unused):
    """
    Shutdown server from a SIGINT received signal
    """
    server.stop_server()
    sys.exit(1)

signal.signal(signal.SIGINT, shutdownServer)
server = SokServer(7000)
server.start_server()
```

```
#client

import socket
import json

HOST = socket.gethostname().split('.')[0]
PORT = 7000
BUFF_SIZE = 1024

def validate_first_brackets(expression):
    stack = []

    for char in expression:
        if char in ['[', '{']:
            return False
        elif char == '(':
            stack.append('(')
        elif char == ')':
            if len(stack) == 0 or stack[-1] != '(':
                return False
            stack.pop()

    return len(stack) == 0

def validate_balanced_brackets(expression):
    stack = []
```

```
    for char in expression:
        if char == '(':
            stack.append('(')
        elif char == ')':
            if not stack or stack[-1] != '(':
                return False
            stack.pop()

    return len(stack) == 0


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as skt:
    """
    request struct = {
        method: GET
        msg: math expression string
    }

    response struct --> {
        status: NUMBER
        data: response (math expression) data OR error msg
    }
    """
    try:
        skt.connect((HOST, PORT))
        req = {}
        req["method"] = "GET"

        print(f"Client connected on {HOST}:{PORT}")
        msg = input("Enter expression: ")

        if not validate_first_brackets(msg):
            raise Exception(f"Please use only first brackets in the expression")

        if not validate_balanced_brackets(msg):
            raise Exception(f"Brackets are not balanced in the expression")
        try:
            res = eval(msg)
            print(res)
        except Exception as e:
            raise Exception("Invalid math Expression")

        req["msg"] = msg
        skt.sendall(json.dumps(req).encode())

        print("Client request: ", req)
```

```
        data = skt.recv(BUFF_SIZE)
        res = json.loads(data.decode())

        print("response from server: ", res)

        print(f"Server --> status: {res['status']}, data: {res['data']}")

        skt.close()
    except Exception as e:
        print(e)
        err_res = {
            "status" : 500,
            "data": str(e)
        }
        print(err_res)
        skt.close()
```

Sample run →



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 local to the server. Choose your own formats for the request/reply messages.

Design of Request/Reply protocol →

- ➔ The UDP server has a database of the students in a csv file format with student info such that each entry contains the student name and the student address.
- ➔ The UDP client sends the server with a student's name as a message.
- ➔ In response, the server,
  - ◆ If finds the student name in the records of the database →
    - It sends the client the student info as a utf-8 encoded string with the following format: `'Name: {student name} Address: {student-address}'`
  - ◆ If does not find the name in the database records →
    - It sends the client generic error message as a utf-8 encoded string with the following format: `'student not found'`

Code →

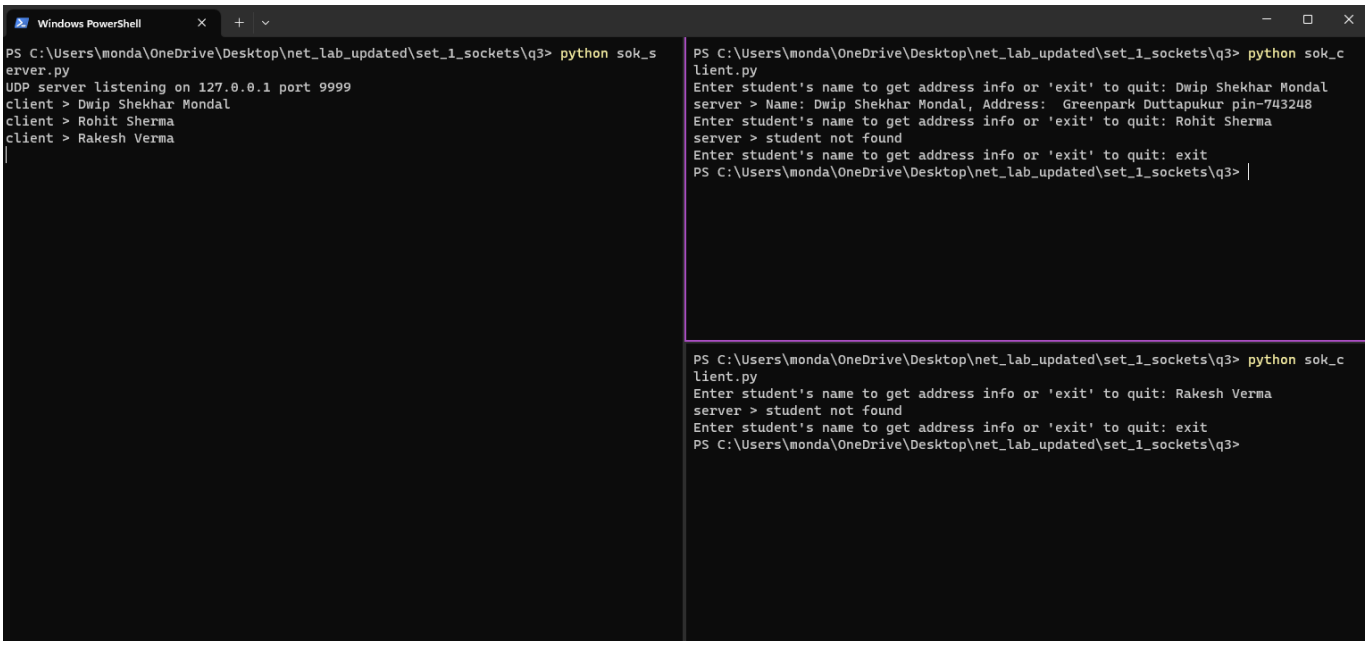
```
#server

import socket
import threading
import csv

def load_student_data(file_path):
    student_data = {}
    with open(file_path, 'r') as file:
        csvreader = csv.reader(file)
        for row in csvreader:
```

```
            name, address = row
            student_data[name] = address
    return student_data

def find_student_info(file_path, student_name):
    res = "student not found"
    with open(file_path, 'r') as file:
        csvreader = csv.reader(file)
        for row in csvreader:
            name, address = row
            if name == student_name:
                res = f"Name: {name}, Address: {address}"
                break
            # student_data[name] = address
    return res

def handle_UDP_requests(client_address, server_socket, data, file_path):
    student_name = data.decode()
    student_data = find_student_info(file_path, student_name)
    print("client >", student_name)
    reply = student_data.encode()
    server_socket.sendto(reply, client_address)


def main():
    host = '127.0.0.1'
    port = 9999
    file_path = './data.csv'

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind((host, port))
    print("UDP server listening on", host, "port", port)

    while True:
        data, address = server_socket.recvfrom(1024)
        client_handler = threading.Thread(target=handle_UDP_requests, args=(address,
server_socket, data, file_path))
        client_handler.start()

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

```
#client
import socket

def main():
    host = '127.0.0.1'
```

```
    port = 9999

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

    while True:
        student_name = input("Enter student's name to get address info or 'exit' to quit:
")
        if student_name.lower() == 'exit':
            break

        client_socket.sendto(student_name.encode(), (host, port))
        data, _ = client_socket.recvfrom(1024)
        print("server >", data.decode())

    client_socket.close()

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

Sample run →



Assignment - II

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.

Step 1: 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 server.
(vi) Stop Wireshark capture

Step 2: 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.

Step 3: 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.

Step 4: TCP Connection Termination

Once the data transfer is over, the client initiates the connection termination by sending a TCP segment with FIN flag set, to the server. Server acknowledges it and sends its 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.

1. Capturing a Trace using Wireshark →

For this task, the client/server program of question 1 from assignment-1 is used, in which the TCP client program sends requests to the server to get the current time and date. The server returns the date and time along with status code (200 for success, 500 for error). Using wireshark the packets sent/received by the client/server will be observed.

Note: there will be two instances of clients, both will request the server for current date, one will send the appropriate command (for which the server will respond with appropriate date), the other will send an arbitrary command to server (for which the server will respond with an error msg).

- ➔ Wireshark capture with Loopback interface as input and with filter of “ip.addr==127.0.0.1 and tcp.port==7000” (Port 7000 is used by the TCP server)
- ➔ Two instances of the client connect to the server.

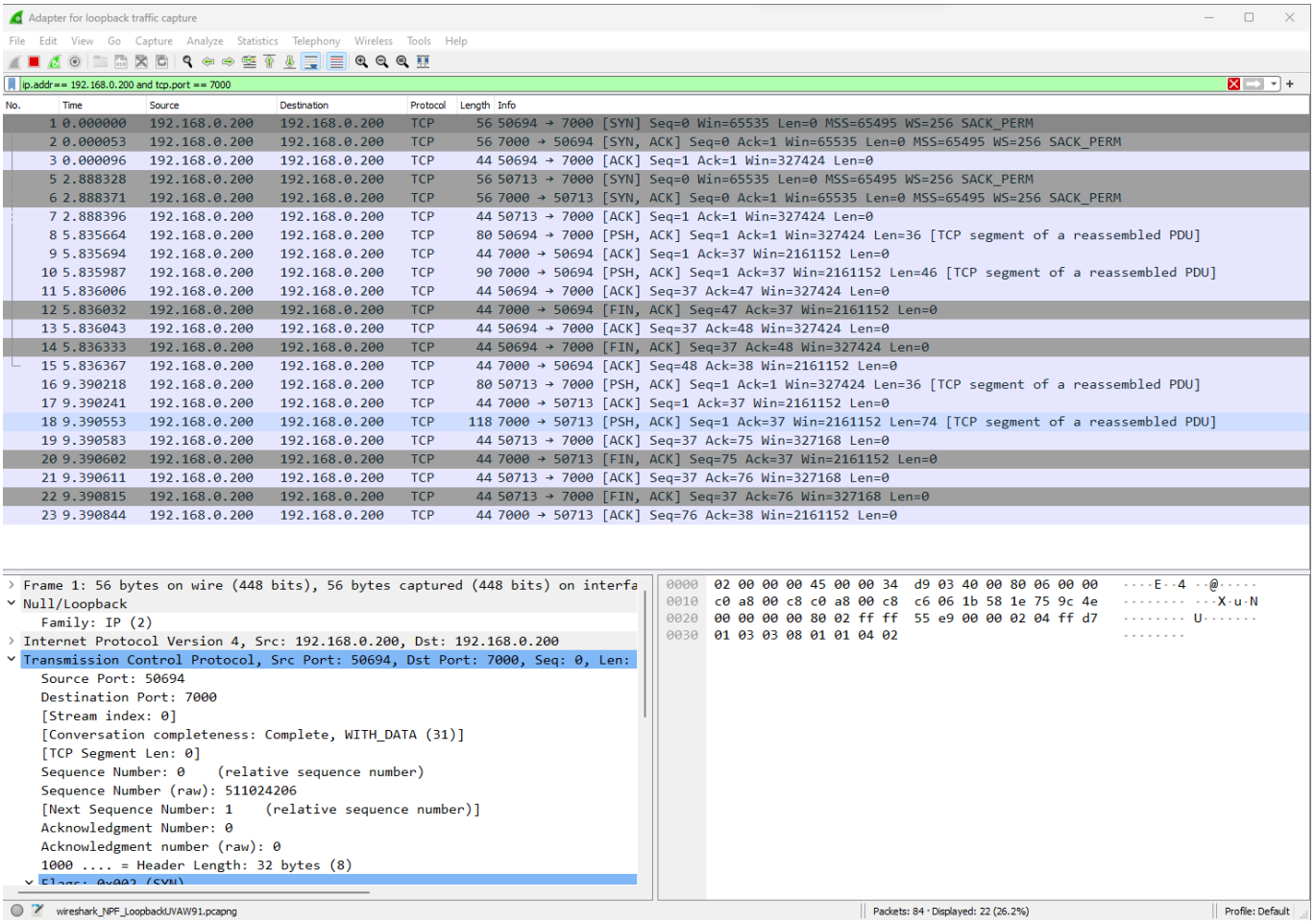


Figure 1: Capture of the packets from the client/server (TCP)

	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.0.200	192.168.0.200	TCP	56	50694 → 7000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
2	0.000053	192.168.0.200	192.168.0.200	TCP	56	7000 → 50694 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
3	0.000096	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
5	2.888328	192.168.0.200	192.168.0.200	TCP	56	50713 → 7000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
6	2.888371	192.168.0.200	192.168.0.200	TCP	56	7000 → 50713 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
7	2.888396	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
8	5.835664	192.168.0.200	192.168.0.200	TCP	80	50694 → 7000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=36 [TCP segment of a reassembled PDU]
9	5.835694	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [ACK] Seq=1 Ack=37 Win=2161152 Len=0
10	5.835987	192.168.0.200	192.168.0.200	TCP	90	7000 → 50694 [PSH, ACK] Seq=1 Ack=37 Win=2161152 Len=46 [TCP segment of a reassembled PDU]
11	5.836006	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=37 Ack=47 Win=327424 Len=0
12	5.836032	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [FIN, ACK] Seq=47 Ack=37 Win=2161152 Len=0
13	5.836043	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=37 Ack=48 Win=327424 Len=0
14	5.836333	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [FIN, ACK] Seq=37 Ack=48 Win=327424 Len=0
15	5.836367	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [ACK] Seq=48 Ack=38 Win=2161152 Len=0
16	9.390218	192.168.0.200	192.168.0.200	TCP	80	50713 → 7000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=36 [TCP segment of a reassembled PDU]
17	9.390241	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [ACK] Seq=1 Ack=37 Win=2161152 Len=0
18	9.390553	192.168.0.200	192.168.0.200	TCP	118	7000 → 50713 [PSH, ACK] Seq=1 Ack=37 Win=2161152 Len=74 [TCP segment of a reassembled PDU]
19	9.390583	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=37 Ack=75 Win=327168 Len=0
20	9.390602	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [FIN, ACK] Seq=75 Ack=37 Win=2161152 Len=0
21	9.390611	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=37 Ack=76 Win=327168 Len=0
22	9.390815	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [FIN, ACK] Seq=37 Ack=76 Win=327168 Len=0
23	9.390844	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [ACK] Seq=76 Ack=38 Win=2161152 Len=0

Figure 2: TCP handshakes

From the above capture it is evident that →

- ➔ Two clients connect to the server at port 7000.
- ➔ Client 1 connects from 50694 → 7000 (3 way hand-shake with server)
- ➔ Client 2 connects from 50713 → 7000 (3 way hand-shake with server)
- ➔ Client 1 sends the message to server: {'method': 'GET', 'msg': 'Get time'}
  - ◆ Gets response from server: {'status': 200, 'data': '2023-08-14 13:27:46'}
  - ◆ After that client 1 terminates.
  - ◆ 4 way hand-shake with server to close connection.
- ➔ Client 2 sends the message to server: {'method': 'GET', 'msg': 'Get food}
  - ◆ Gets response from server: {'status': 500, 'data': 'Unknown request command or command not supported'}
  - ◆ After that client 2 terminates.
  - ◆ 4 way hand-shake with server to close connection.

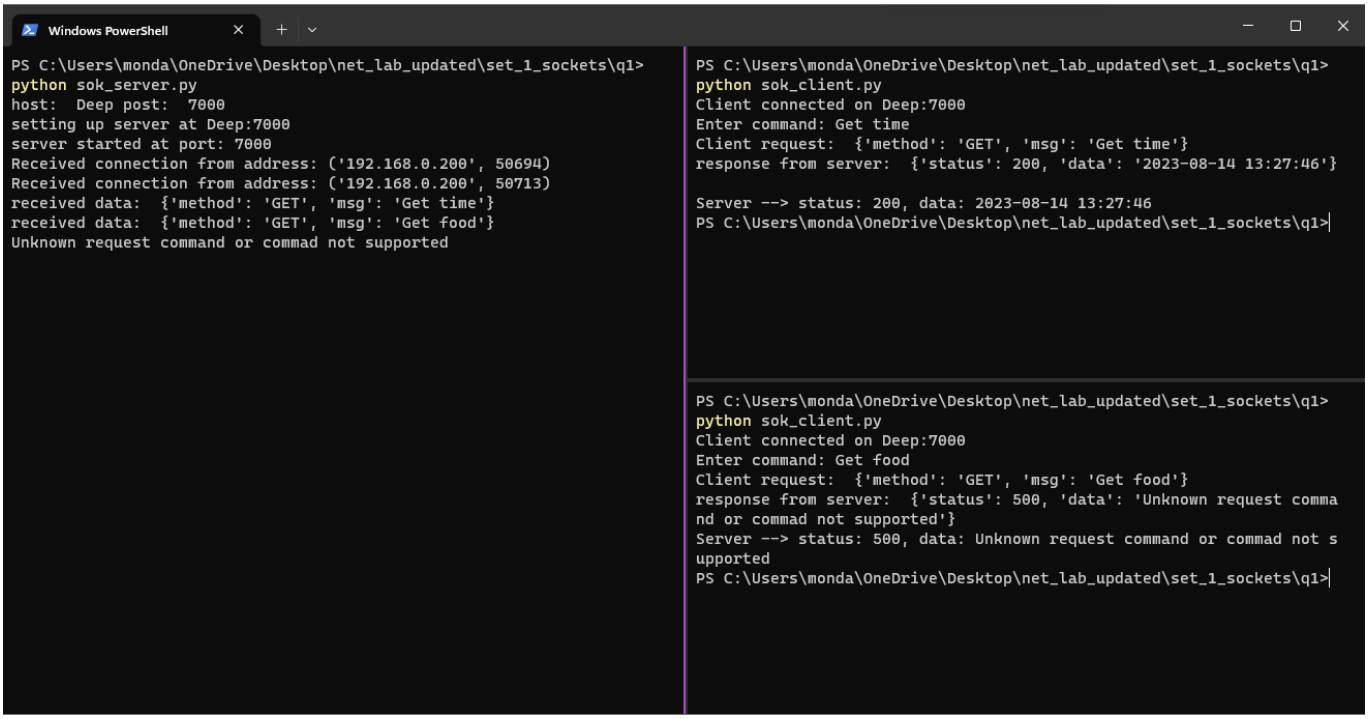
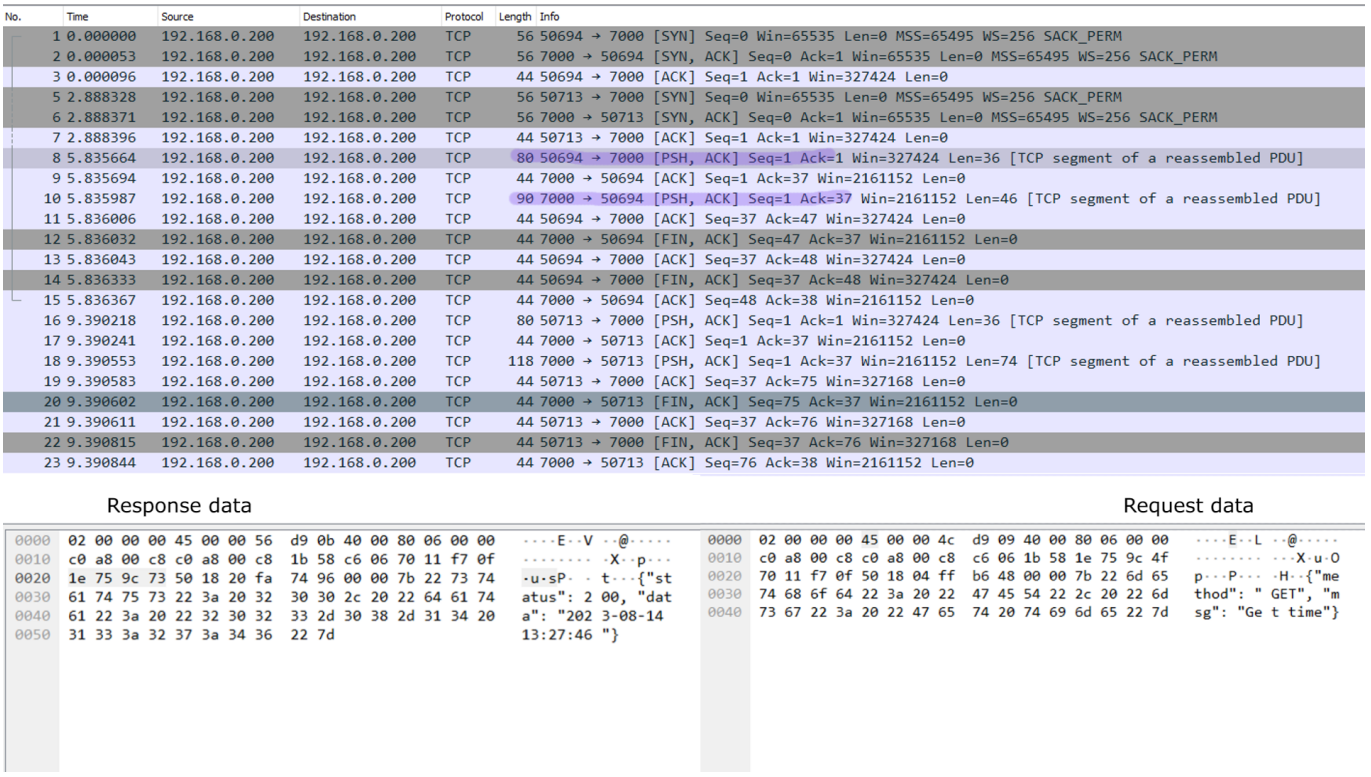


Figure 3: Requests and responses in the client/server console



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.0.200	192.168.0.200	TCP	56	50694 → 7000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
2	0.000053	192.168.0.200	192.168.0.200	TCP	56	7000 → 50694 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
3	0.000096	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
5	2.888328	192.168.0.200	192.168.0.200	TCP	56	50713 → 7000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
6	2.888371	192.168.0.200	192.168.0.200	TCP	56	7000 → 50713 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
7	2.888396	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
8	5.835664	192.168.0.200	192.168.0.200	TCP	80	50694 → 7000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=36 [TCP segment of a reassembled PDU]
9	5.835694	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [ACK] Seq=1 Ack=37 Win=2161152 Len=0
10	5.835987	192.168.0.200	192.168.0.200	TCP	90	7000 → 50694 [PSH, ACK] Seq=1 Ack=37 Win=2161152 Len=46 [TCP segment of a reassembled PDU]
11	5.836006	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=37 Ack=47 Win=327424 Len=0
12	5.836032	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [FIN, ACK] Seq=47 Ack=37 Win=2161152 Len=0
13	5.836043	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=37 Ack=48 Win=327424 Len=0
14	5.836333	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [FIN, ACK] Seq=37 Ack=48 Win=327424 Len=0
15	5.836367	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [ACK] Seq=48 Ack=38 Win=2161152 Len=0
16	9.390218	192.168.0.200	192.168.0.200	TCP	80	50713 → 7000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=36 [TCP segment of a reassembled PDU]
17	9.390241	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [ACK] Seq=1 Ack=37 Win=2161152 Len=0
18	9.390553	192.168.0.200	192.168.0.200	TCP	118	7000 → 50713 [PSH, ACK] Seq=1 Ack=37 Win=2161152 Len=74 [TCP segment of a reassembled PDU]
19	9.390583	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=37 Ack=75 Win=327168 Len=0
20	9.390602	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [FIN, ACK] Seq=75 Ack=37 Win=2161152 Len=0
21	9.390611	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=37 Ack=76 Win=327168 Len=0
22	9.390815	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [FIN, ACK] Seq=37 Ack=76 Win=327168 Len=0
23	9.390844	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [ACK] Seq=76 Ack=38 Win=2161152 Len=0

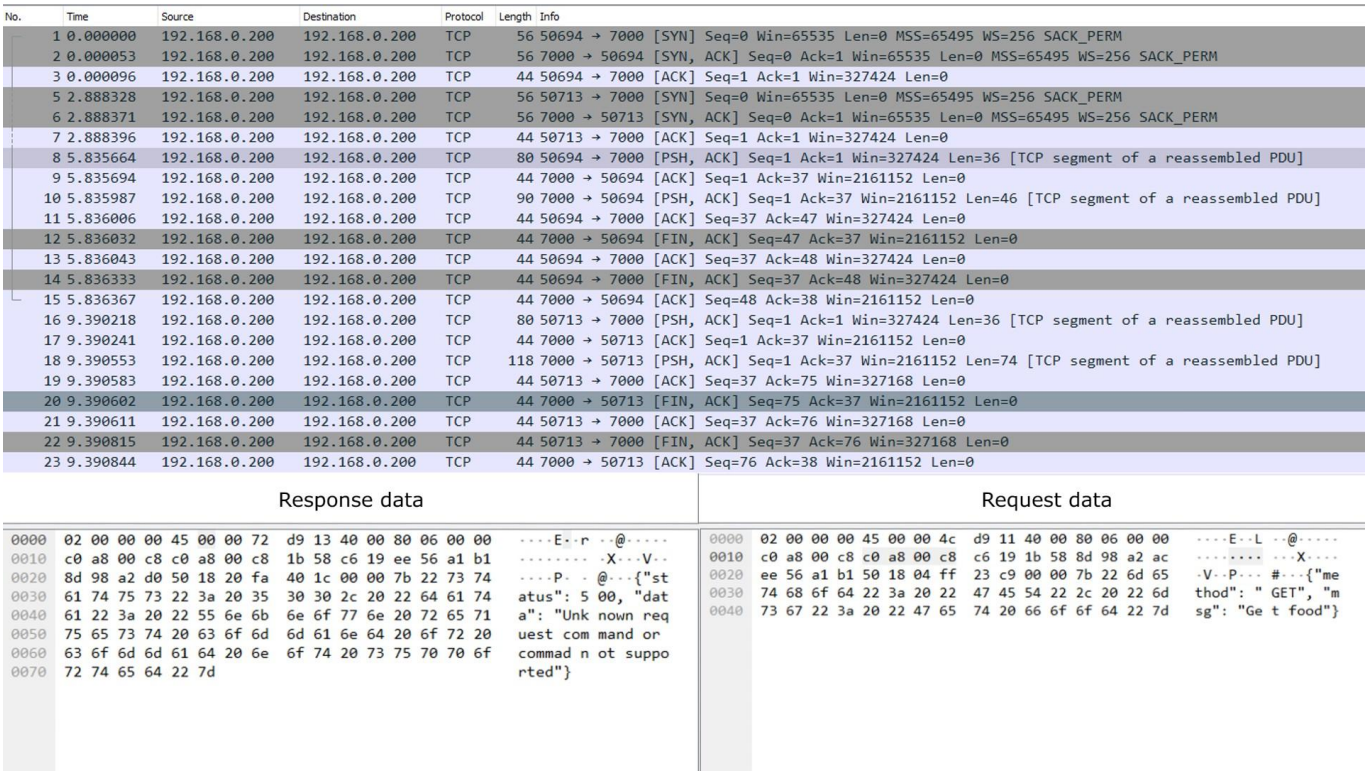
Response data

```
0000  02 00 00 00 45 00 00 56  d9 0b 40 00 80 06 00 00   ····E··V ··@·····
0010  c0 a8 00 c8 c0 a8 00 c8  1b 58 c6 06 70 11 f7 0f   ········ ·X··p···
0020  1e 75 9c 73 50 18 20 fa  74 96 00 00 7b 22 73 74   ·u·sP· · t···{"st
0030  61 74 75 73 22 3a 20 32  30 30 2c 20 22 64 61 74   atus": 2 00, "dat
0040  61 22 3a 20 22 32 30 32  33 2d 30 38 2d 31 34 20   a": "202 3-08-14
0050  31 33 3a 32 37 3a 34 36  22 7d                     13:27:46 "}
```

Request data

```
0000  02 00 00 00 45 00 00 4c  d9 09 40 00 80 06 00 00   ····E··L ··@·····
0010  c0 a8 00 c8 c0 a8 00 c8  c6 06 1b 58 1e 75 9c 4f   ········ ···X·u·O
0020  70 11 f7 0f 50 18 04 ff  b6 48 00 00 7b 22 6d 65   p···P··· ·H··{"me
0030  74 68 6f 64 22 3a 20 22  47 45 54 22 2c 20 22 6d   thod": " GET", "m
0040  73 67 22 3a 20 22 47 65  74 20 74 69 6d 65 22 7d   sg": "Ge t time"}
```

Figure 4: Request and response from Client 1 to server



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.0.200	192.168.0.200	TCP	56	50694 → 7000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
2	0.000053	192.168.0.200	192.168.0.200	TCP	56	7000 → 50694 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
3	0.000096	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
5	2.888328	192.168.0.200	192.168.0.200	TCP	56	50713 → 7000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
6	2.888371	192.168.0.200	192.168.0.200	TCP	56	7000 → 50713 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
7	2.888396	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=1 Ack=1 Win=327424 Len=0
8	5.835664	192.168.0.200	192.168.0.200	TCP	80	50694 → 7000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=36 [TCP segment of a reassembled PDU]
9	5.835694	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [ACK] Seq=1 Ack=37 Win=2161152 Len=0
10	5.835987	192.168.0.200	192.168.0.200	TCP	90	7000 → 50694 [PSH, ACK] Seq=1 Ack=37 Win=2161152 Len=46 [TCP segment of a reassembled PDU]
11	5.836006	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=37 Ack=47 Win=327424 Len=0
12	5.836032	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [FIN, ACK] Seq=47 Ack=37 Win=2161152 Len=0
13	5.836043	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [ACK] Seq=37 Ack=48 Win=327424 Len=0
14	5.836333	192.168.0.200	192.168.0.200	TCP	44	50694 → 7000 [FIN, ACK] Seq=37 Ack=48 Win=327424 Len=0
15	5.836367	192.168.0.200	192.168.0.200	TCP	44	7000 → 50694 [ACK] Seq=48 Ack=38 Win=2161152 Len=0
16	9.390218	192.168.0.200	192.168.0.200	TCP	80	50713 → 7000 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=36 [TCP segment of a reassembled PDU]
17	9.390241	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [ACK] Seq=1 Ack=37 Win=2161152 Len=0
18	9.390553	192.168.0.200	192.168.0.200	TCP	118	7000 → 50713 [PSH, ACK] Seq=1 Ack=37 Win=2161152 Len=74 [TCP segment of a reassembled PDU]
19	9.390583	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=37 Ack=75 Win=327168 Len=0
20	9.390602	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [FIN, ACK] Seq=75 Ack=37 Win=2161152 Len=0
21	9.390611	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [ACK] Seq=37 Ack=76 Win=327168 Len=0
22	9.390815	192.168.0.200	192.168.0.200	TCP	44	50713 → 7000 [FIN, ACK] Seq=37 Ack=76 Win=327168 Len=0
23	9.390844	192.168.0.200	192.168.0.200	TCP	44	7000 → 50713 [ACK] Seq=76 Ack=38 Win=2161152 Len=0

Response data

```
0000  02 00 00 00 45 00 00 72  d9 13 40 00 80 06 00 00   ····E··r ··@·····
0010  c0 a8 00 c8 c0 a8 00 c8  1b 58 c6 19 ee 56 a1 b1   ········ ·X···V··
0020  8d 98 a2 d0 50 18 20 fa  40 1c 00 00 7b 22 73 74   ····P··· @···{"st
0030  61 74 75 73 22 3a 20 35  30 30 2c 20 22 64 61 74   atus": 5 00, "dat
0040  61 22 3a 20 22 55 6e 6b  6e 6f 77 6e 20 72 65 71   a": "Unk nown req
0050  75 65 73 74 20 63 6f 6d  6d 61 6e 64 20 6f 72 20   uest com mand or
0060  63 6f 6d 6d 61 64 20 6e  6f 74 20 73 75 70 70 6f   commad n ot suppo
0070  72 74 65 64 22 7d                                 rted"}
```

Request data

```
0000  02 00 00 00 45 00 00 4c  d9 11 40 00 80 06 00 00   ····E··L ··@·····
0010  c0 a8 00 c8 c0 a8 00 c8  c6 19 1b 58 8d 98 a2 ac   ········ ···X····
0020  ee 56 a1 b1 50 18 04 ff  23 c9 00 00 7b 22 6d 65   ·V··P··· #···{"me
0030  74 68 6f 64 22 3a 20 22  47 45 54 22 2c 20 22 6d   thod": " GET", "m
0040  73 67 22 3a 20 22 47 65  74 20 66 6f 6f 64 22 7d   sg": "Ge t food"}
```

Figure 5: Request and response from Client 2 to server

2. TCP Connection Establishment →

2.1 Client 1 to server →

- ➔ The following are records of 3 way handshake of the connection establishment of client 1 to server.
- ➔ Client 1 connects from 50694 → 7000
- ➔ Client 1 sends a SYN packet (50694 → 7000) with sequence no: 511024206 and acknowledge no: 0
- ➔ Server sends a SYN, ACK packet (7000 → 50694) with sequence no: 188022550 and acknowledge no: 511024207 (511024206 + 1)
- ➔ Client 1 sends a ACK packet (50694 → 7000) with sequence no: 511024207 and acknowledge no: 188022551 (188022550 + 1)
- ➔ Hence the 3-way hand-shake is completed

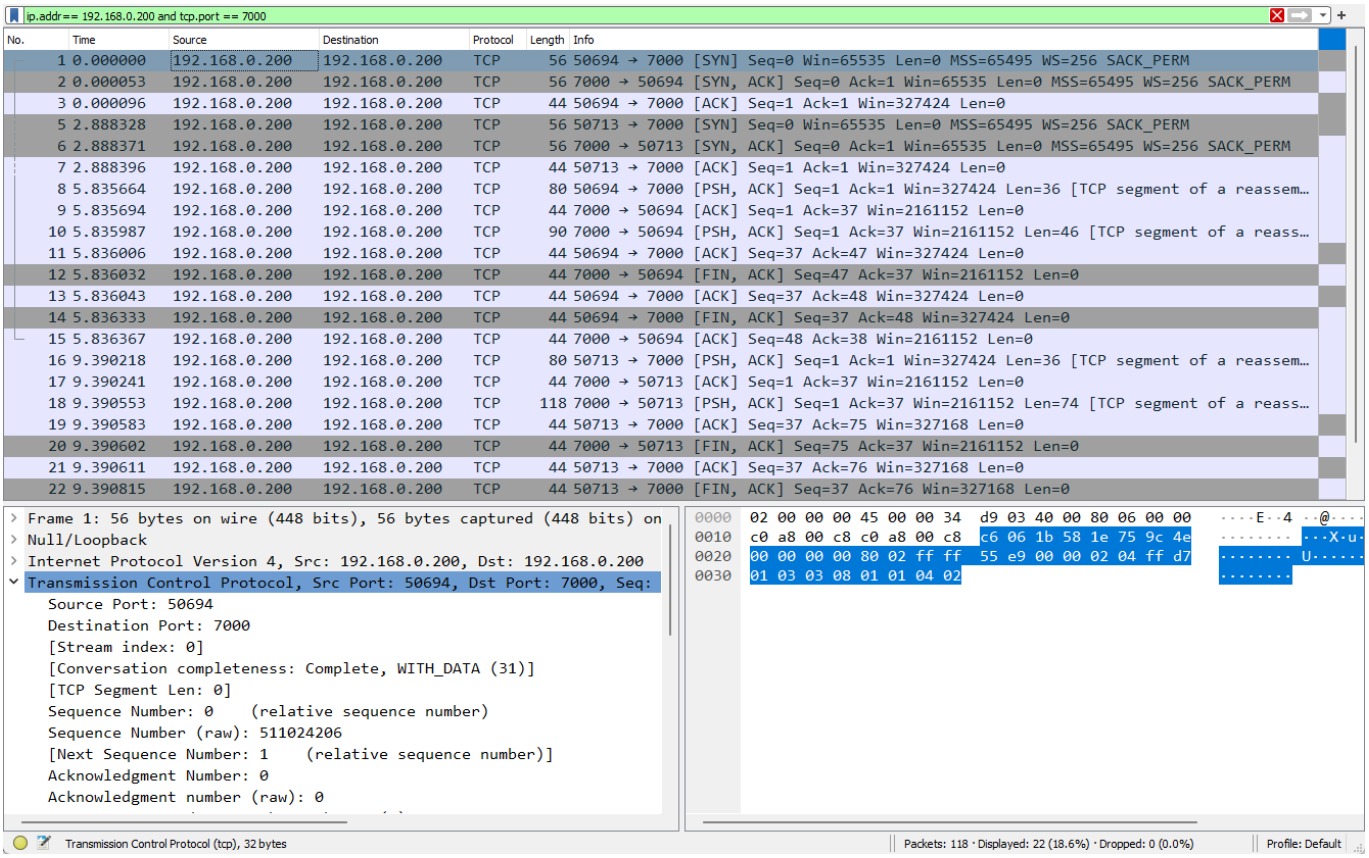


Figure 6.1: 3-way handshake step 1 (client 1/server)

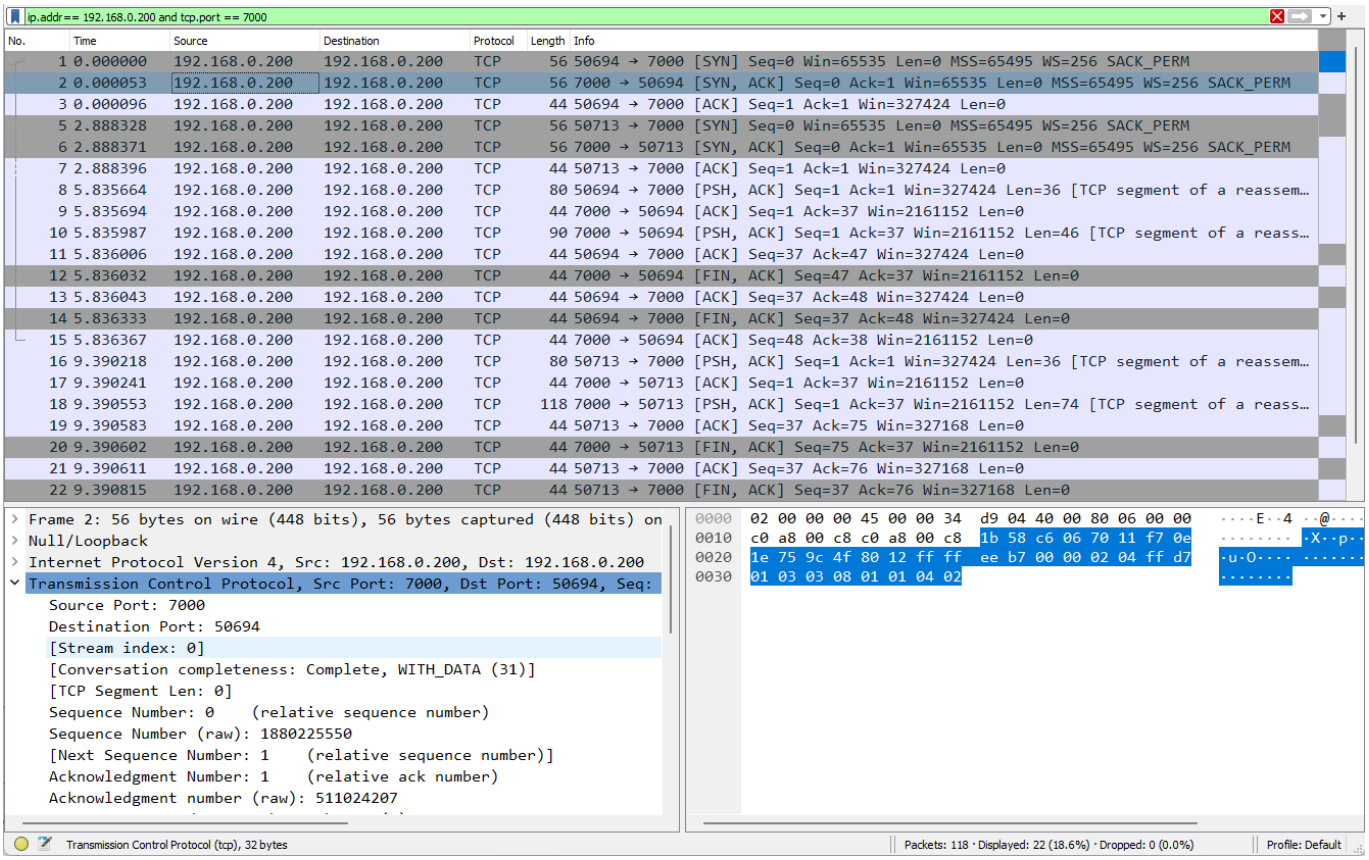


Figure 6.2: 3-way handshake step 2 (client 1/server)

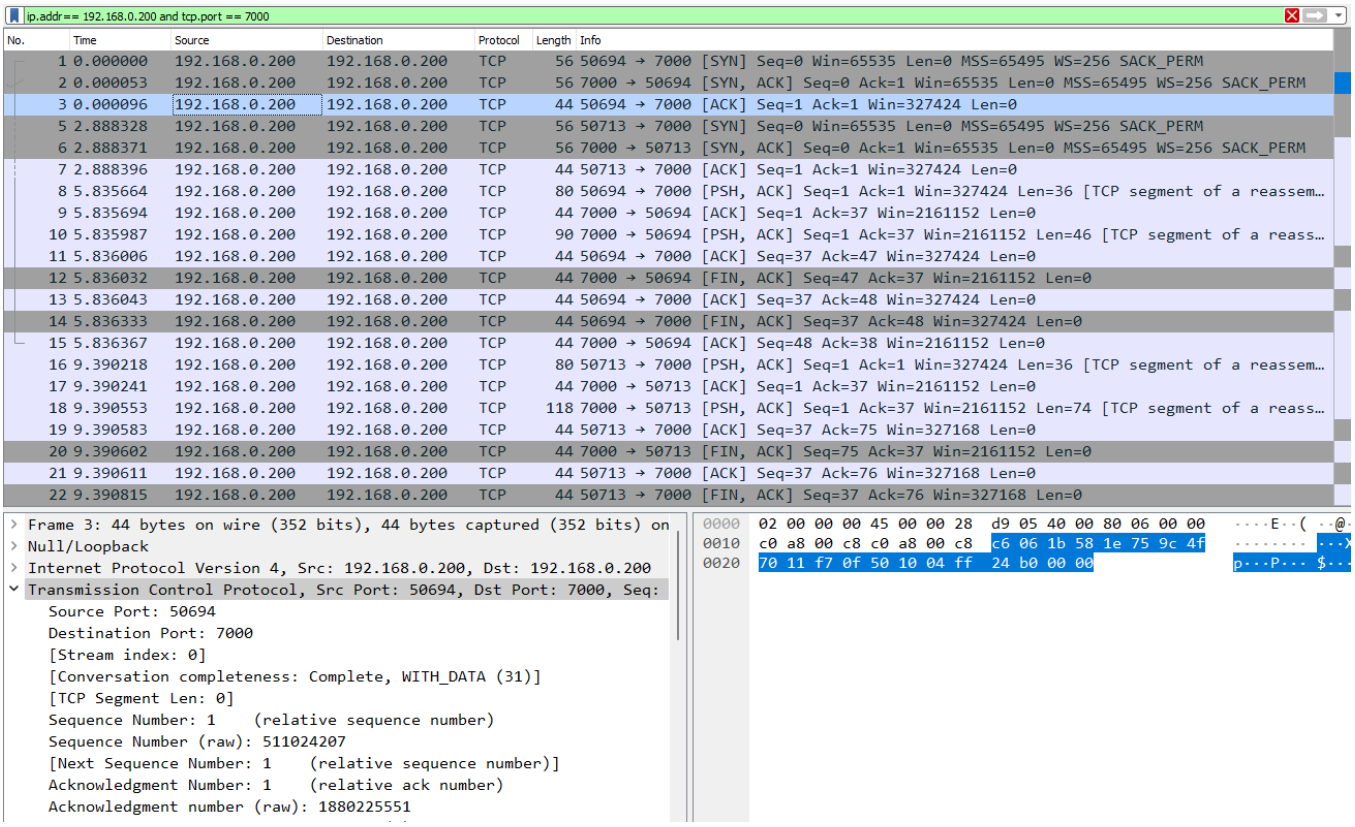


Figure 6.3: 3-way handshake step 3 (client 1/server)

2.2 Client 2 to server →

- ➔ The following are records of 3 way handshake of the connection establishment of client 2 to server.
- ➔ Client 2 connects from 50713 → 7000
- ➔ Client 1 sends a SYN packet (50713 → 7000) with sequence no: 2375590571 and acknowledge no: 0
- ➔ Server sends a SYN, ACK packet (7000 → 50713) with sequence no: 3998654896 and acknowledge no: 2375590572 (2375590571 + 1)
- ➔ Client 1 sends a ACK packet (50713 → 7000) with sequence no: 2375590572 and acknowledge no: 3998654897 (3998654896 + 1)
- ➔ Hence the 3-way hand-shake is completed

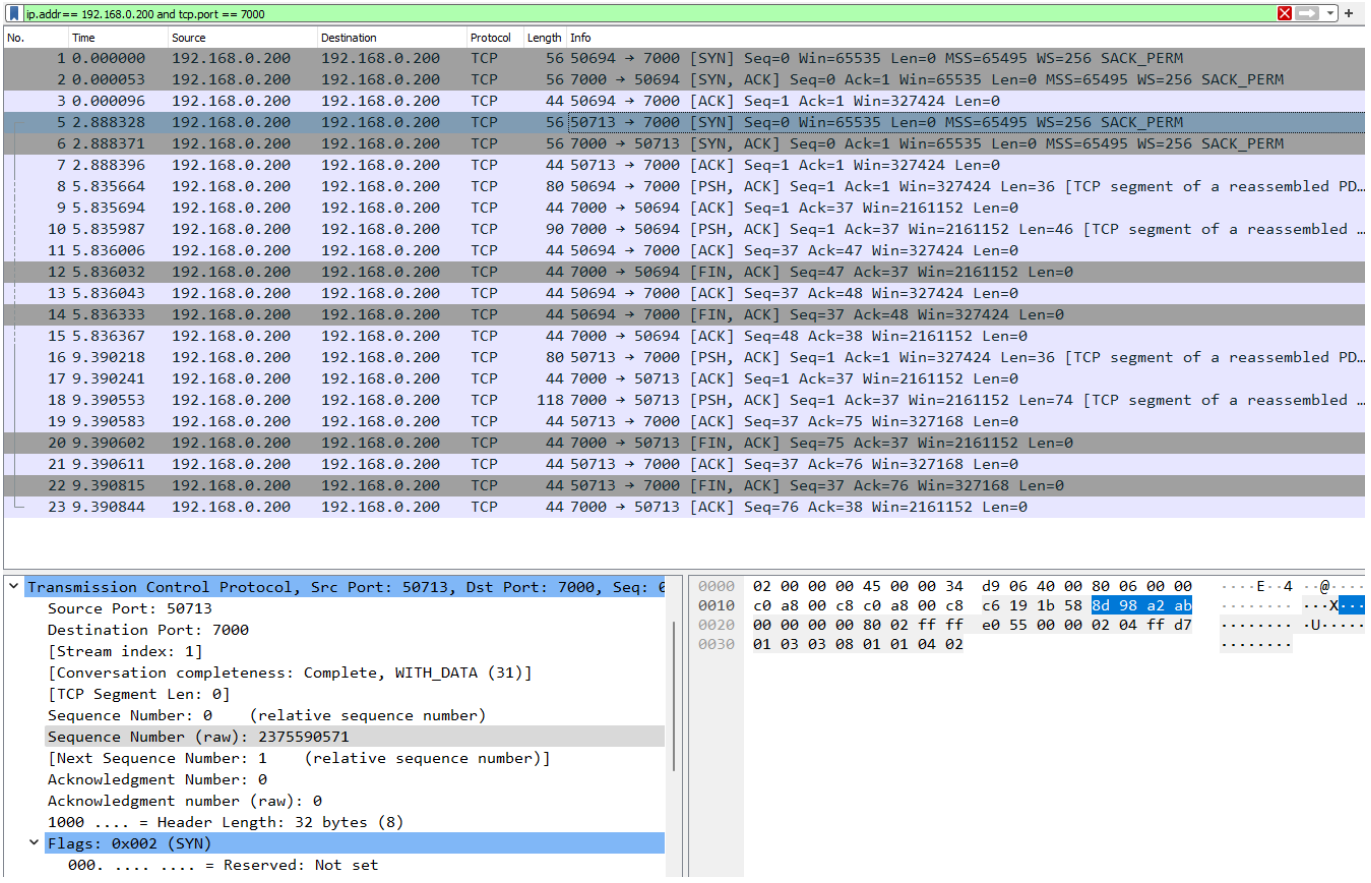


Figure 7.1: 3-way handshake step 1 (client 2/server)

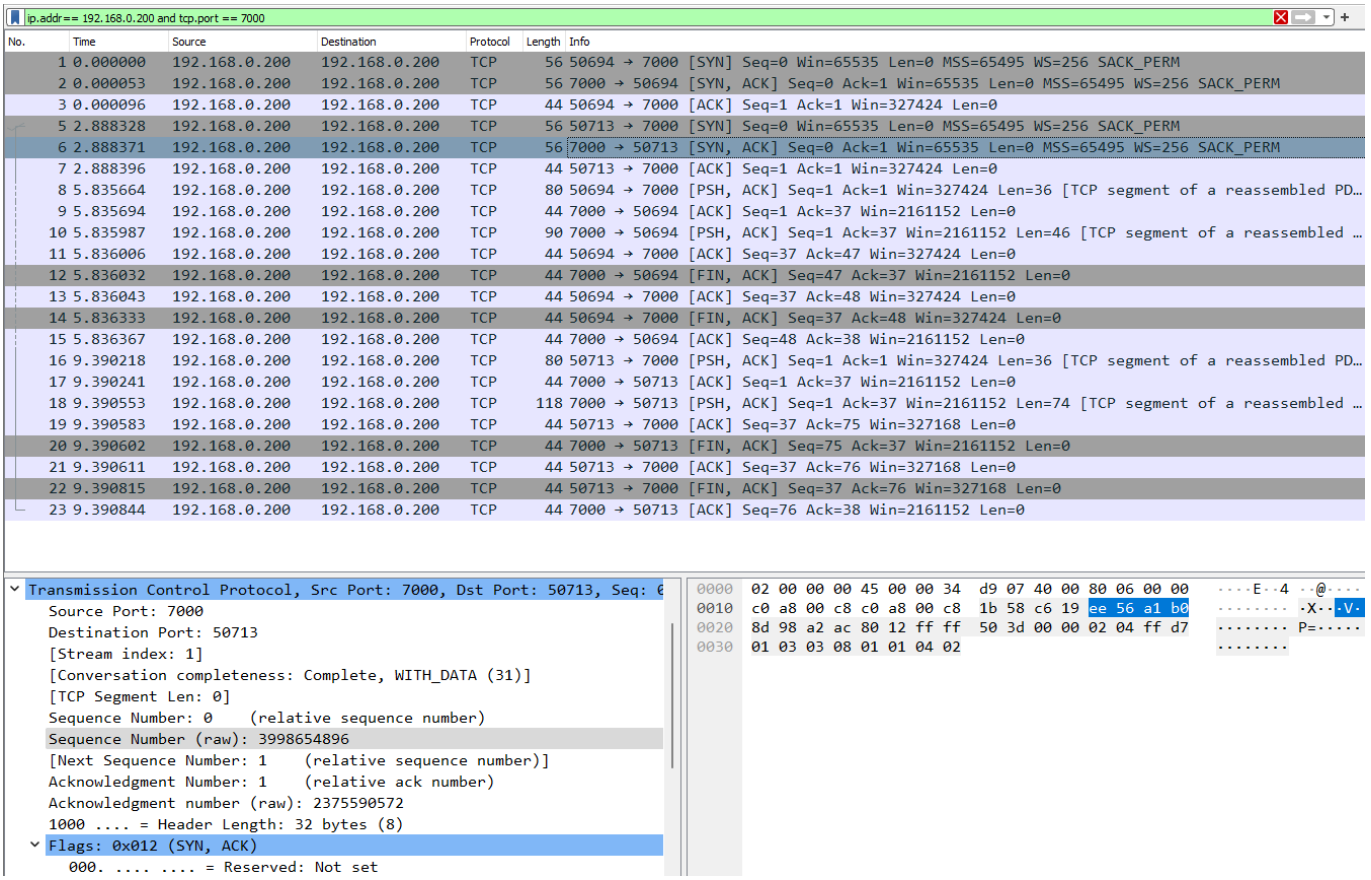


Figure 7.2: 3-way handshake step 2 (client 2/server)

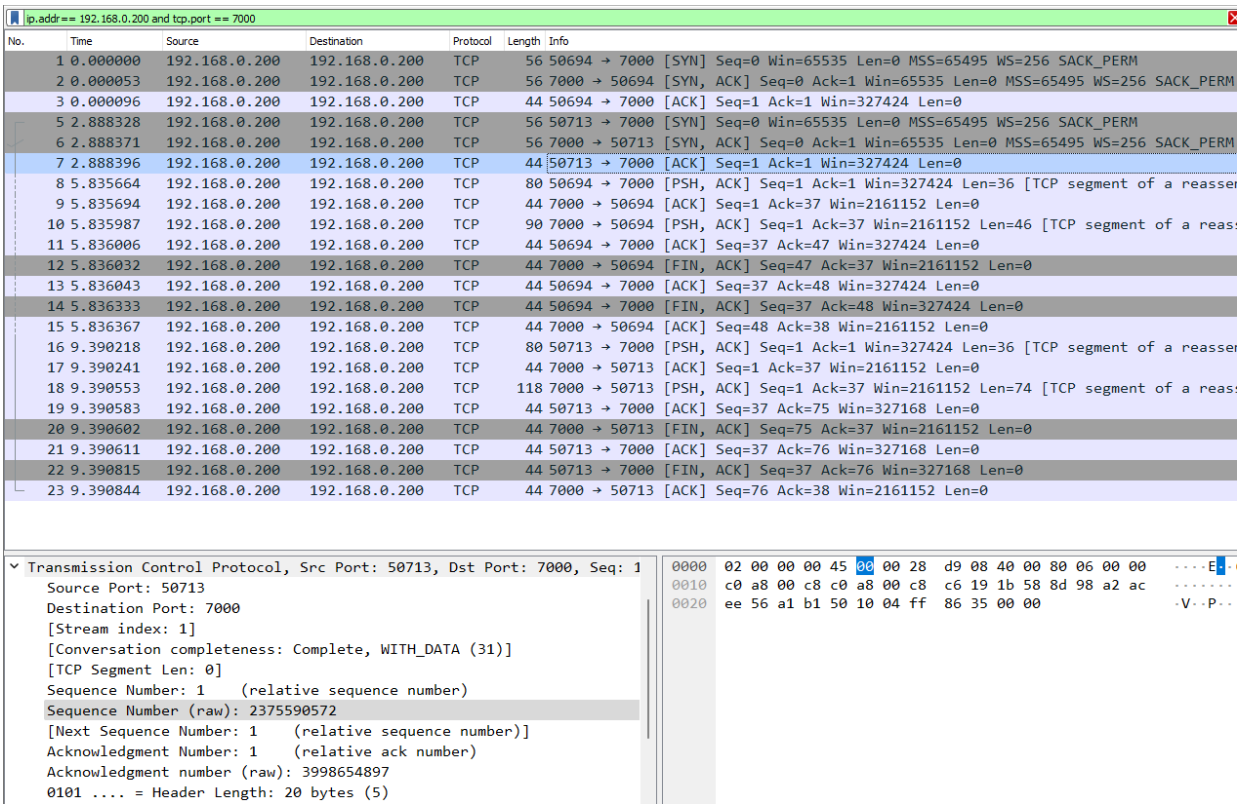


Figure 7.3: 3-way handshake step 3 (client 2/server)

Timing Sequence diagram for Connection establishment from client 1 to server →

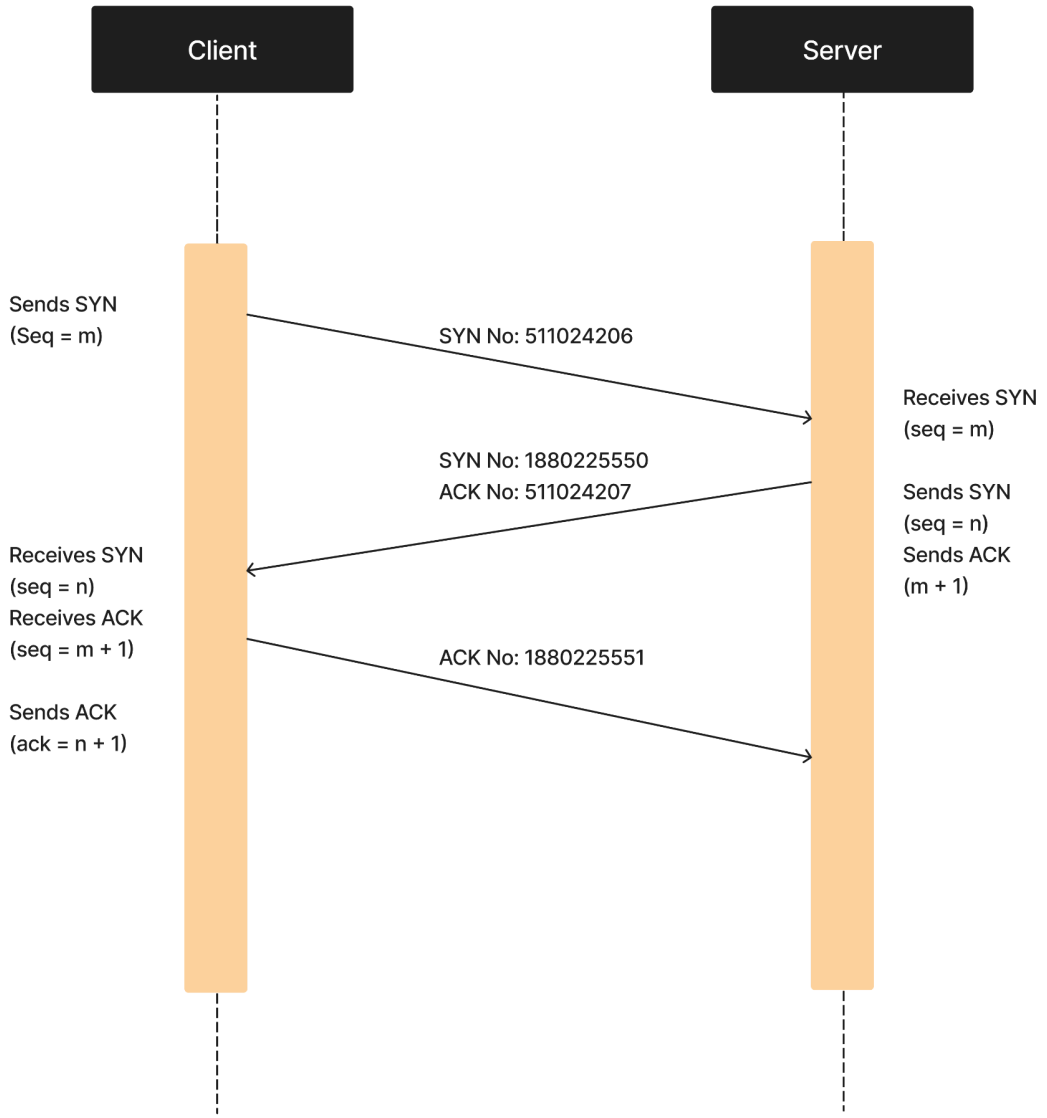


Figure 8: 3-way handshake connection establishment phase (client 1/server)

Timing Sequence diagram for Connection establishment from client 2 to server →

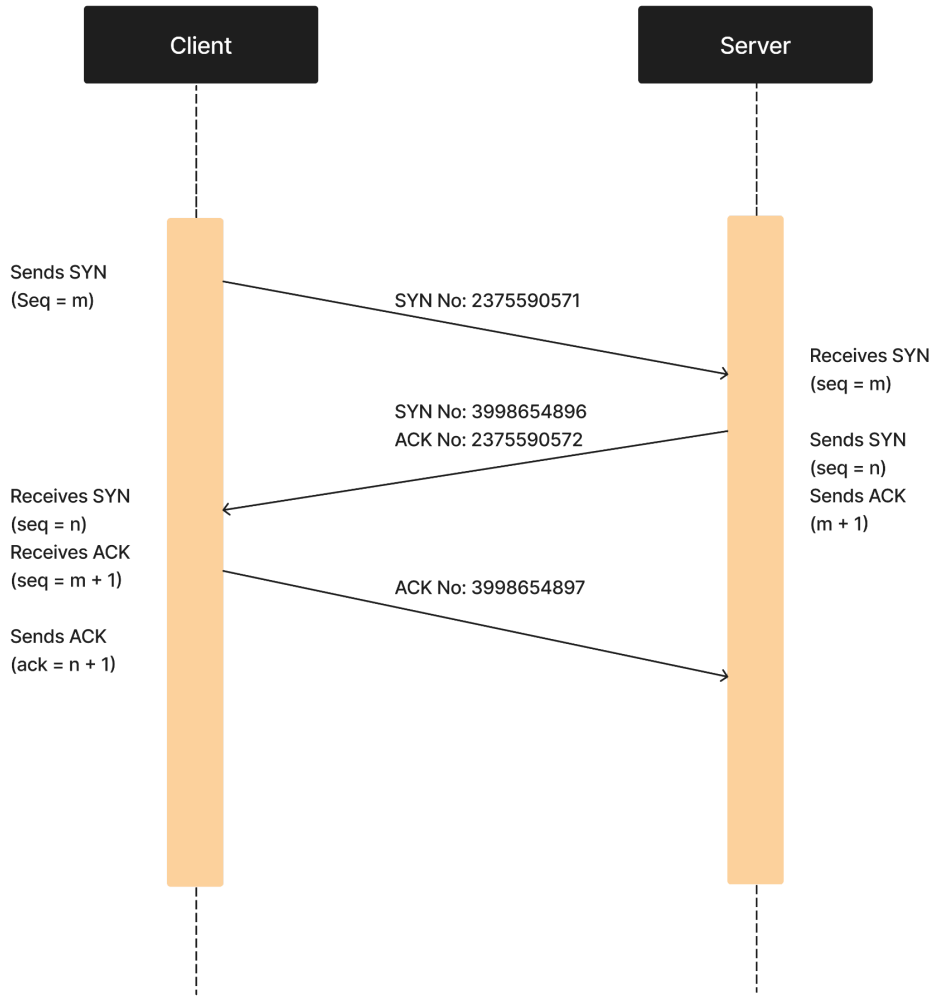


Figure 9: 3-way handshake connection establishment phase (client 2/server)

3. TCP Data Transfer →

3.1 Client 1 - server →

- ➔ The following are records of TCP data transfer phase from client 1 to server.
- ➔ Client 1 sends a PSH, ACK packet (50694 → 7000) with sequence no: 511024207 and acknowledge no: 1880225551
- ➔ Server sends a ACK packet (7000 → 50694) with sequence no: 1880225551 and acknowledge no: 511024243
- ➔ Server sends a PSH, ACK packet (7000 → 50694) with sequence no: 1880225551 and acknowledge no: 511024243
- ➔ Client 1 sends a ACK packet (50694 → 7000) with sequence no: 511024243 and acknowledge no: 1880225597

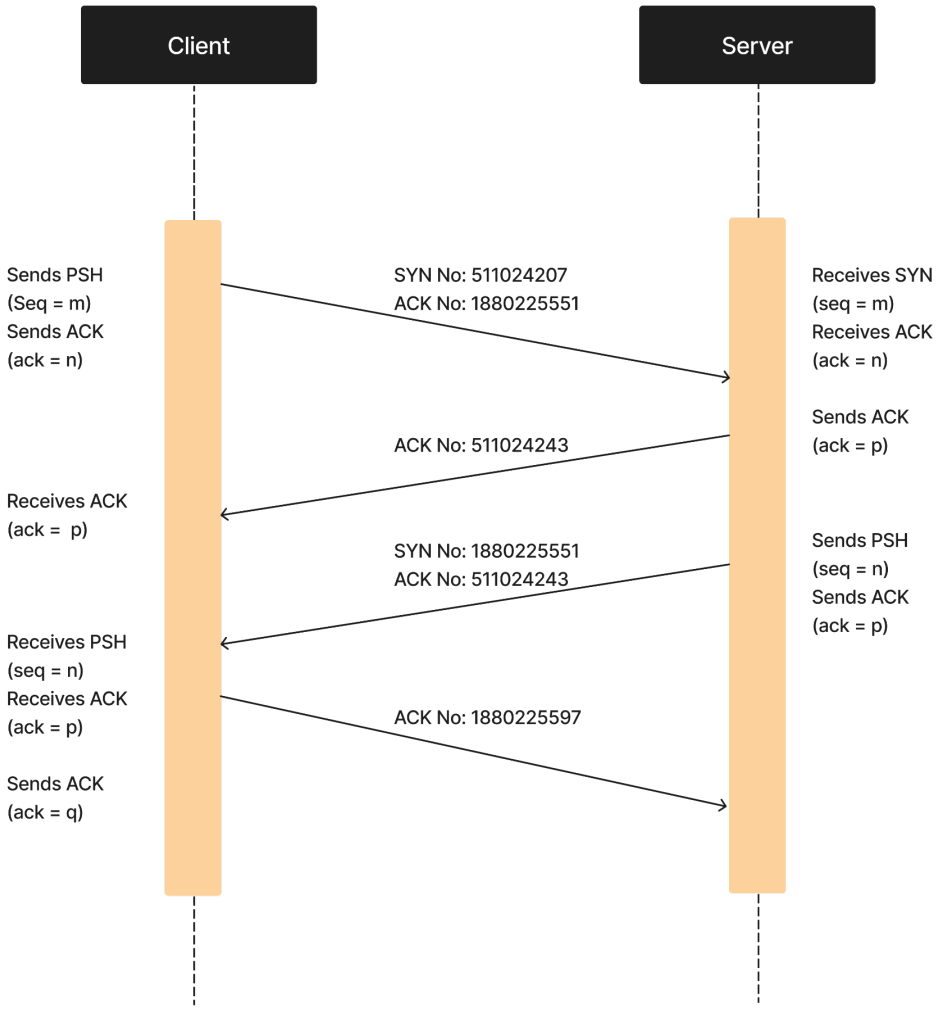


Figure 10: Sequence diagram TCP data-transfer phase (client 1/server)

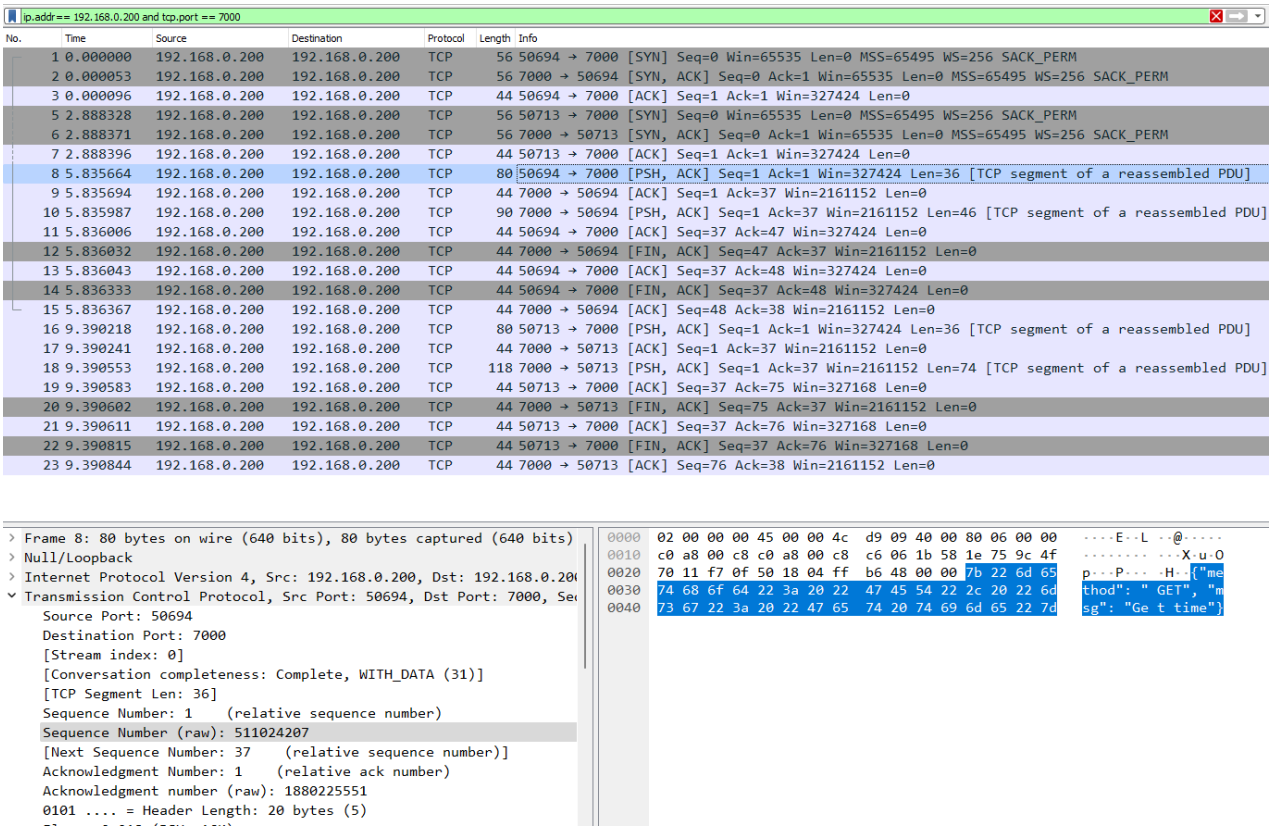


Figure 11.1: Sequence diagram TCP data-transfer phase (client 1/server) step 1

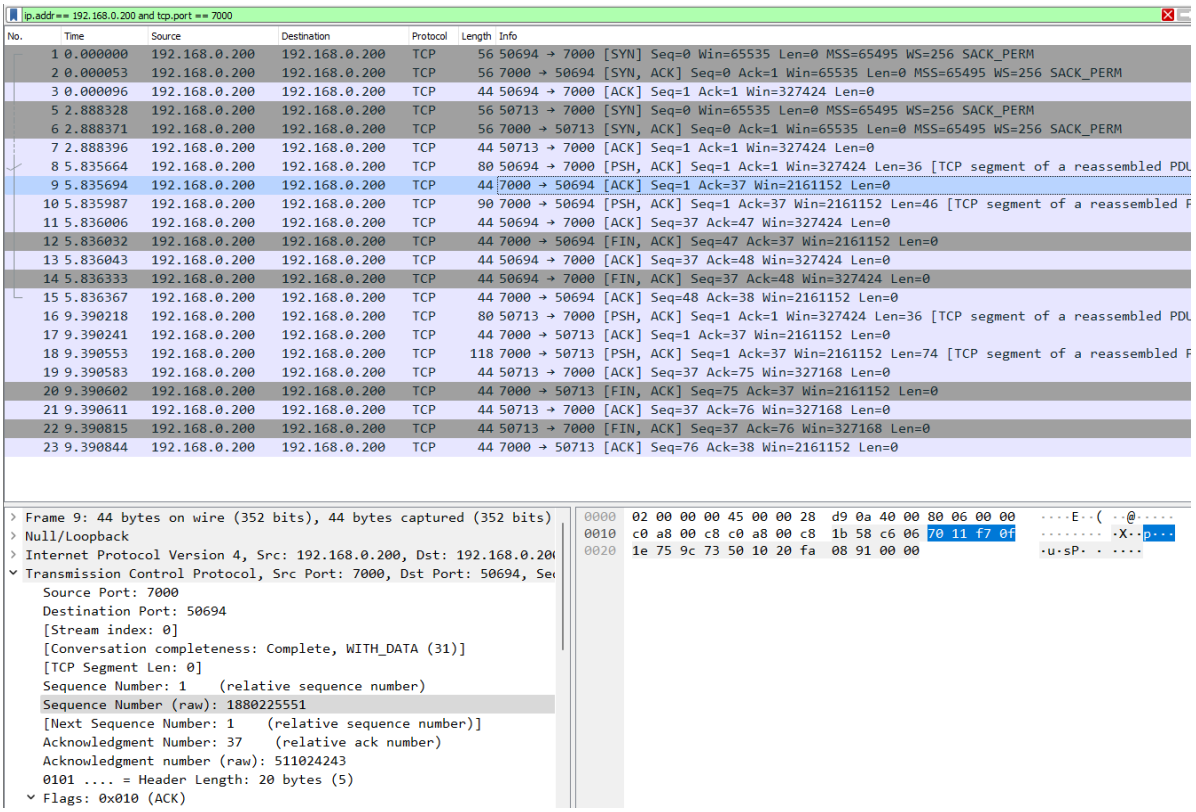


Figure 11.2: Sequence diagram TCP data-transfer phase (client 1/server) step 2

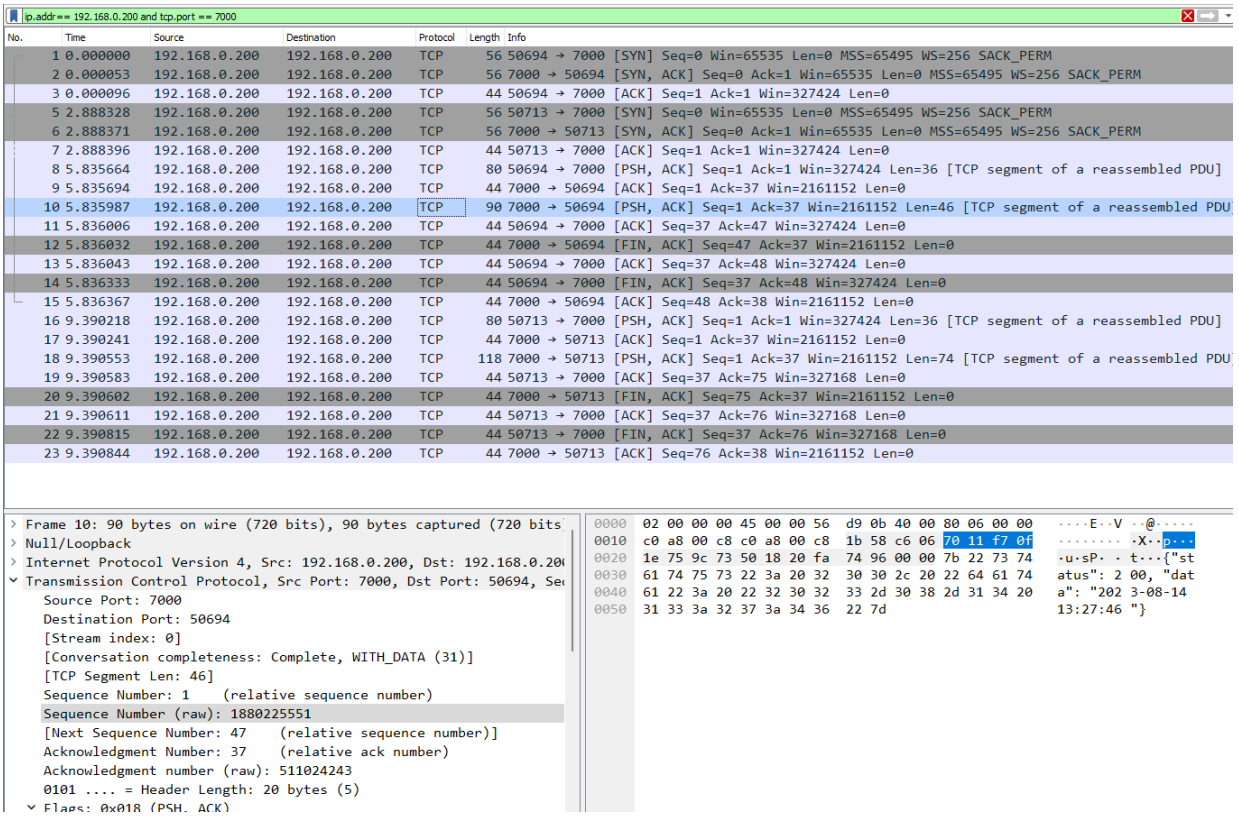


Figure 11.3: Sequence diagram TCP data-transfer phase (client 1/server) step 3

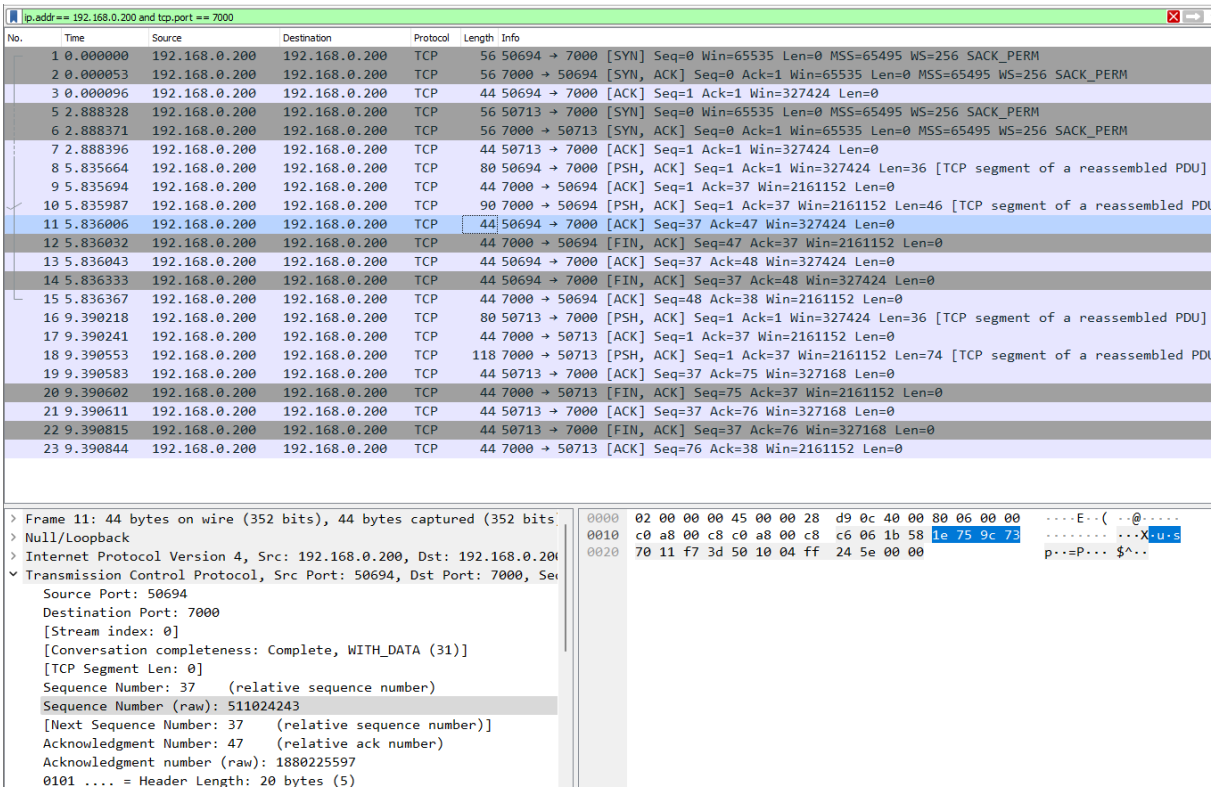


Figure 11.4: Sequence diagram TCP data-transfer phase (client 1/server) step 4

3.2 Client 2 - server →

- ➔ The following are records of TCP data transfer phase from client 1 to server.
- ➔ Client 1 sends a PSH, ACK packet (50713 → 7000) with sequence no: 2375590572 and acknowledge no: 3998654897
- ➔ Server sends a ACK packet (7000 → 50713) with sequence no: 3998654897 and acknowledge no: 2375590608
- ➔ Server sends a PSH, ACK packet (7000 → 50694) with sequence no: 3998654897 and acknowledge no: 2375590608
- ➔ Client 1 sends a ACK packet (50694 → 7000) with sequence no: 2375590608 and acknowledge no: 3998654971

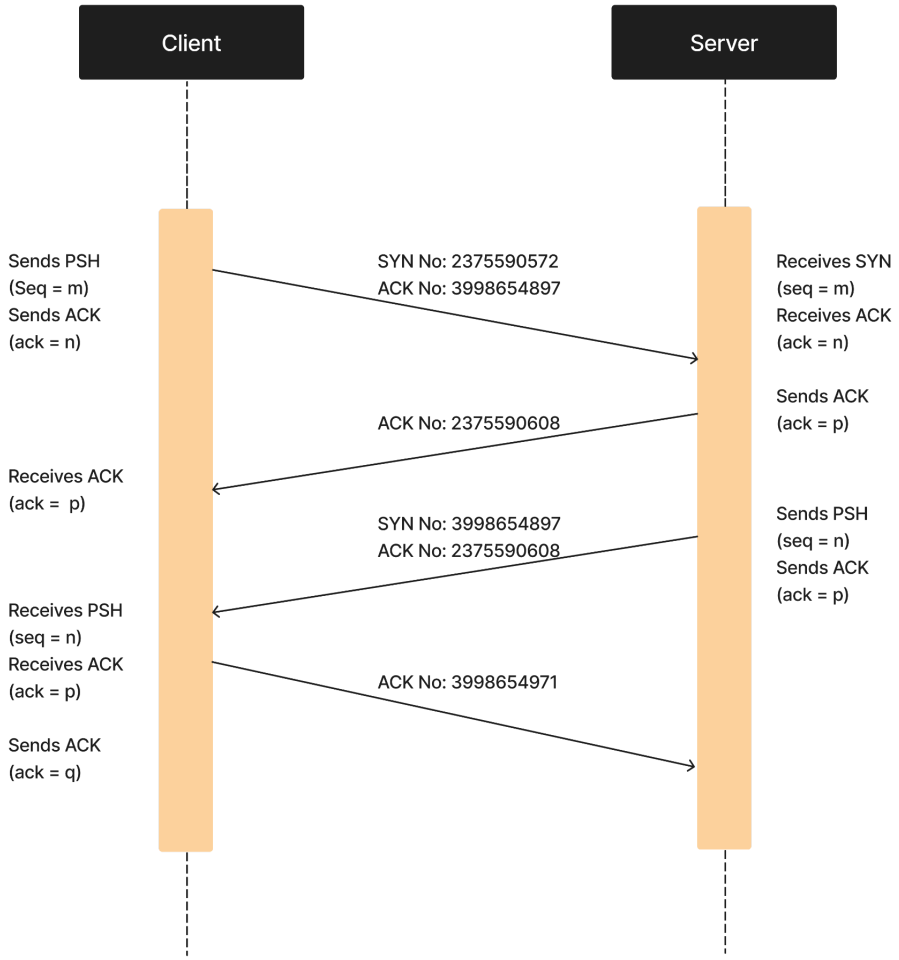


Figure 12: Sequence diagram TCP data-transfer phase (client 2/server)

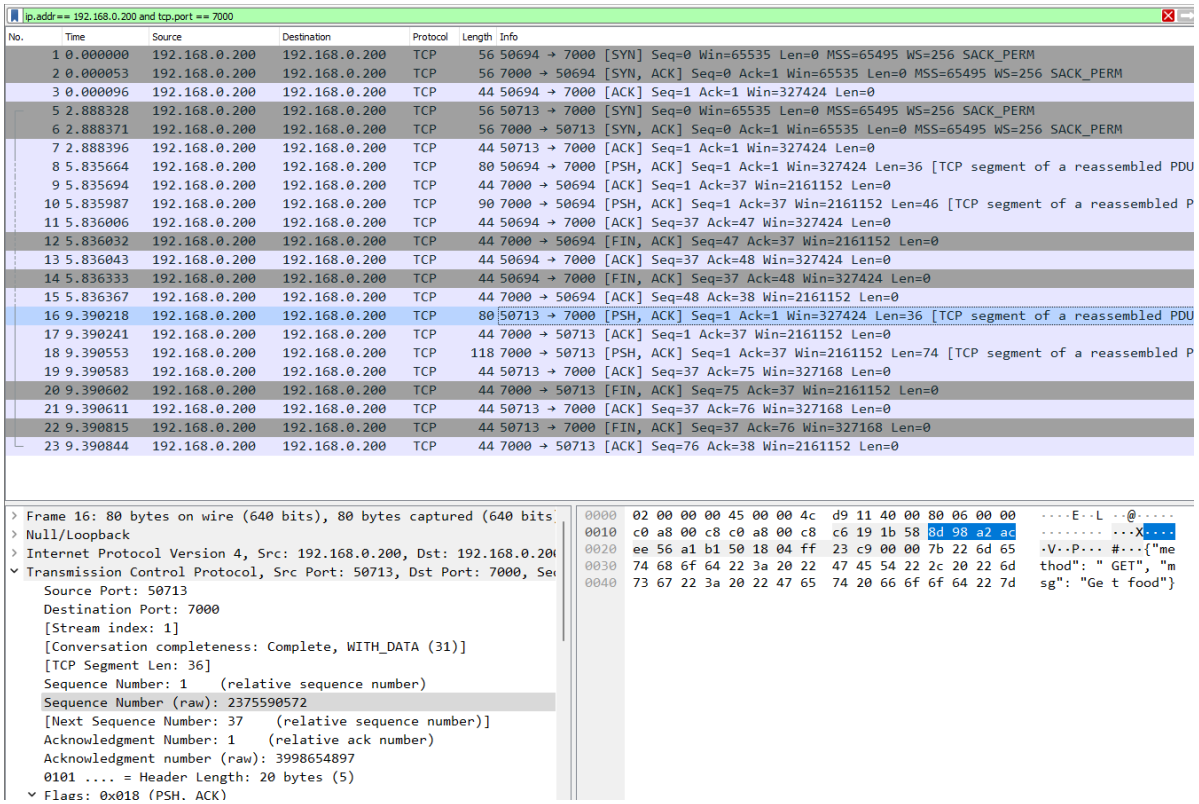


Figure 13.1: Sequence diagram TCP data-transfer phase (client 2/server) step 1

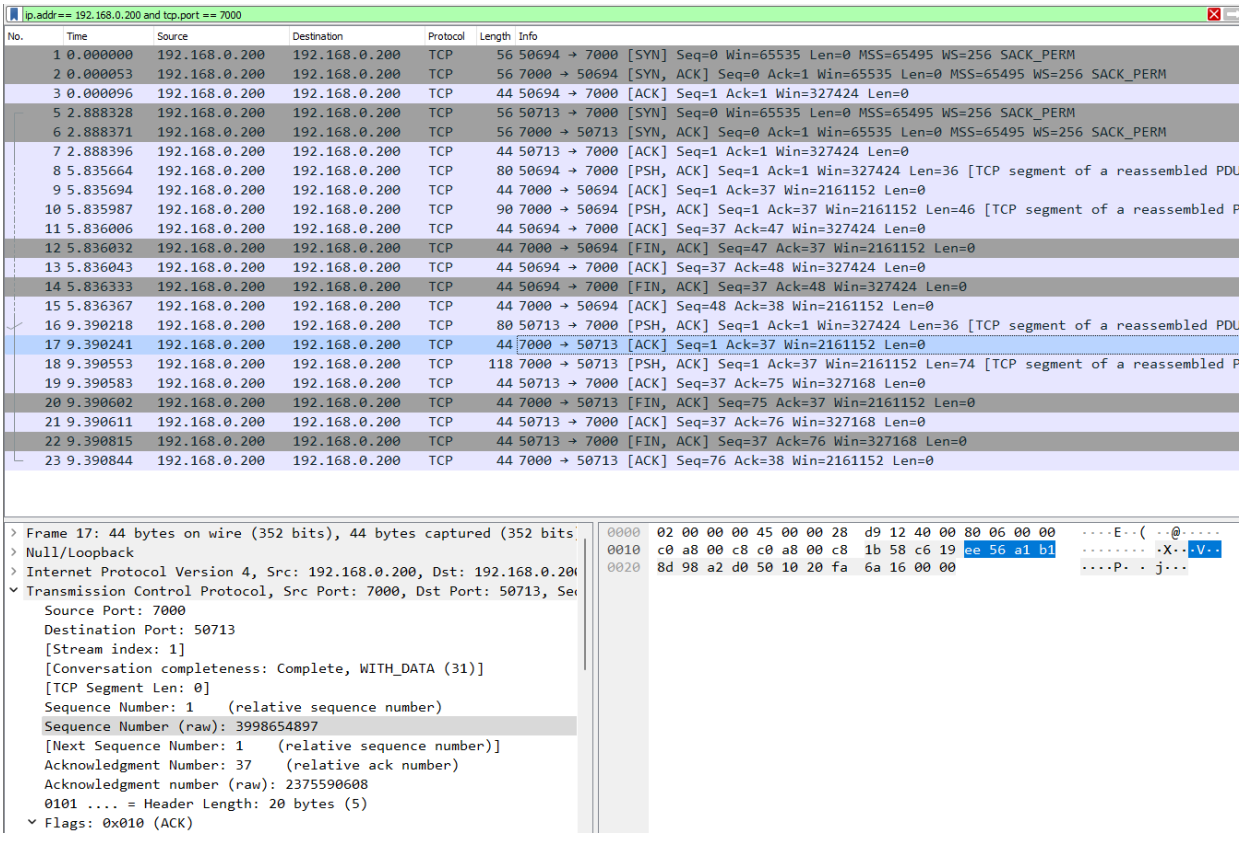


Figure 13.2: Sequence diagram TCP data-transfer phase (client 2/server) step 2

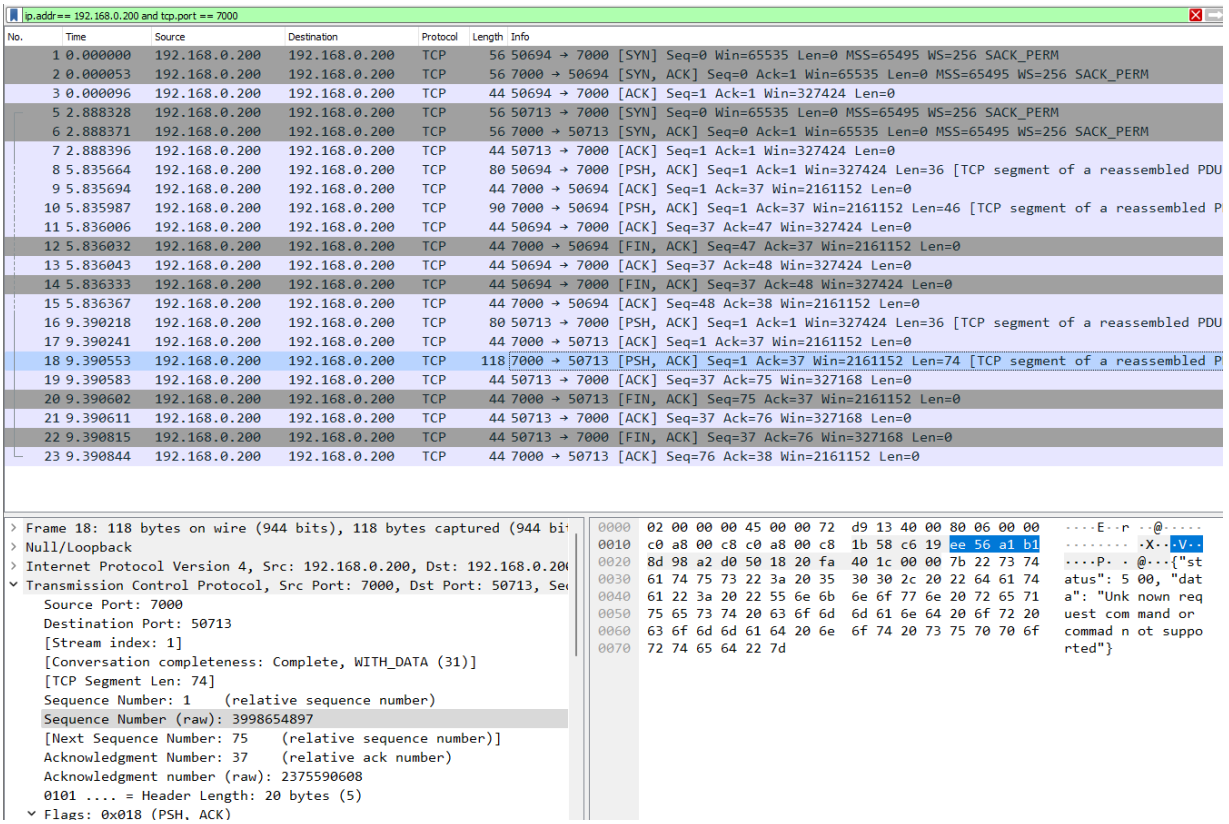


Figure 13.3: Sequence diagram TCP data-transfer phase (client 2/server) step 3

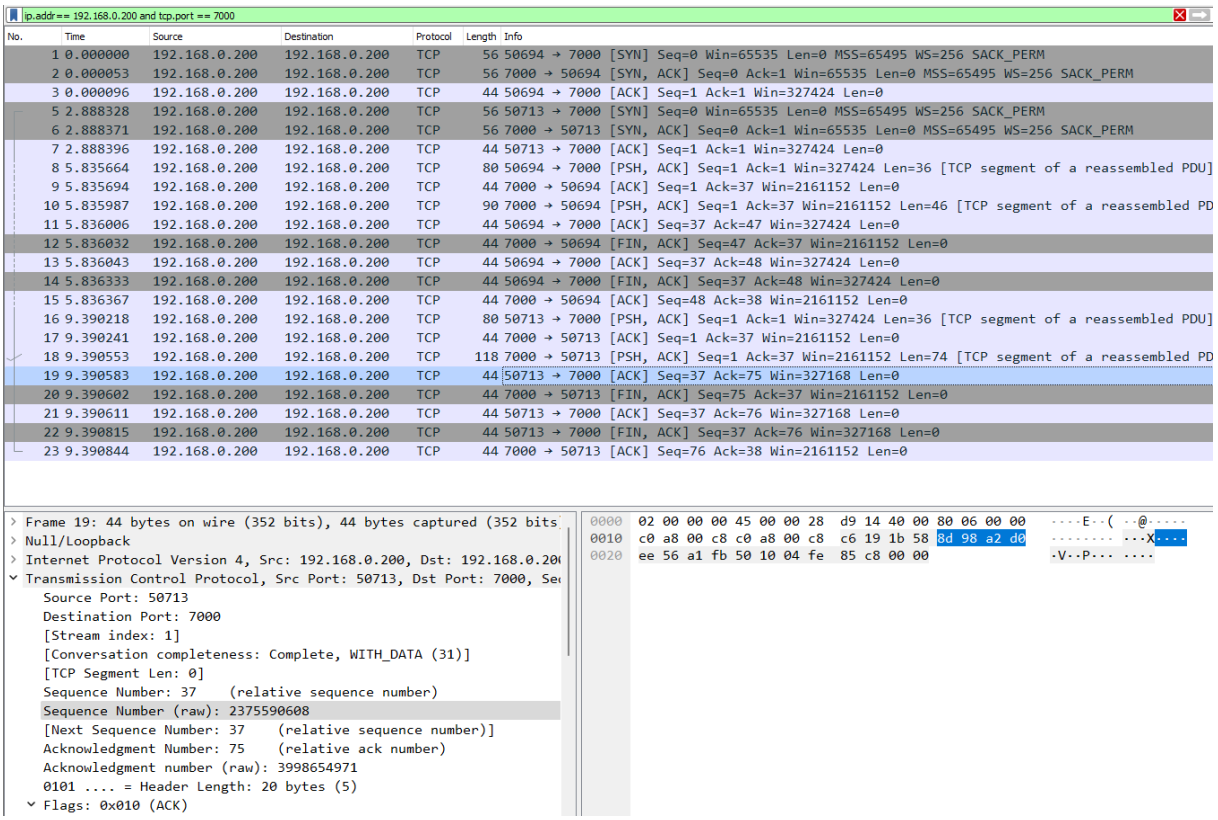


Figure 13.4: Sequence diagram TCP data-transfer phase (client 2/server) step 4

4. TCP Connection Termination

4.1 Client 1 - server

- ➔ Server sends FIN, ACK packet to client 1 (7000 → 50694) with sequence no: 1880225597 and acknowledge no: 511024243
- ➔ Client 1 sends back an ACK packet in response (50694 → 7000) with sequence no: 511024243, acknowledge no: 1880225598
- ➔ Client 1 send a FIN, ACK packet to server (50694 → 7000) which signifies that client wants to close the connection with sequence no: 511024243, acknowledge no: 1880225598
- ➔ Server sends ACK packet to client 1 (7000 → 50694) in response with sequence no: 1880225598, acknowledge no: 511024244

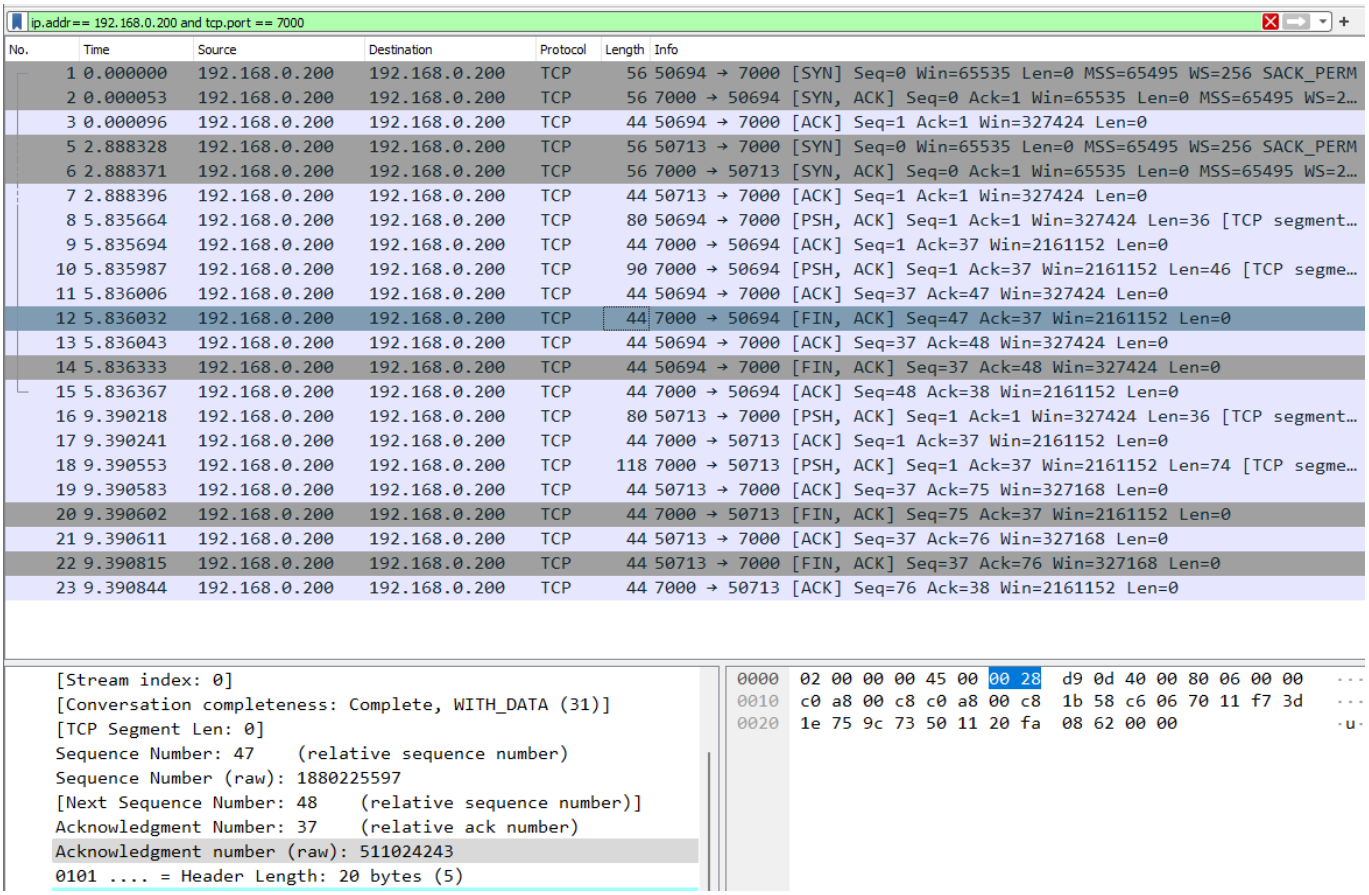


Figure 14.1: TCP termination phase (client 1/server) step 1

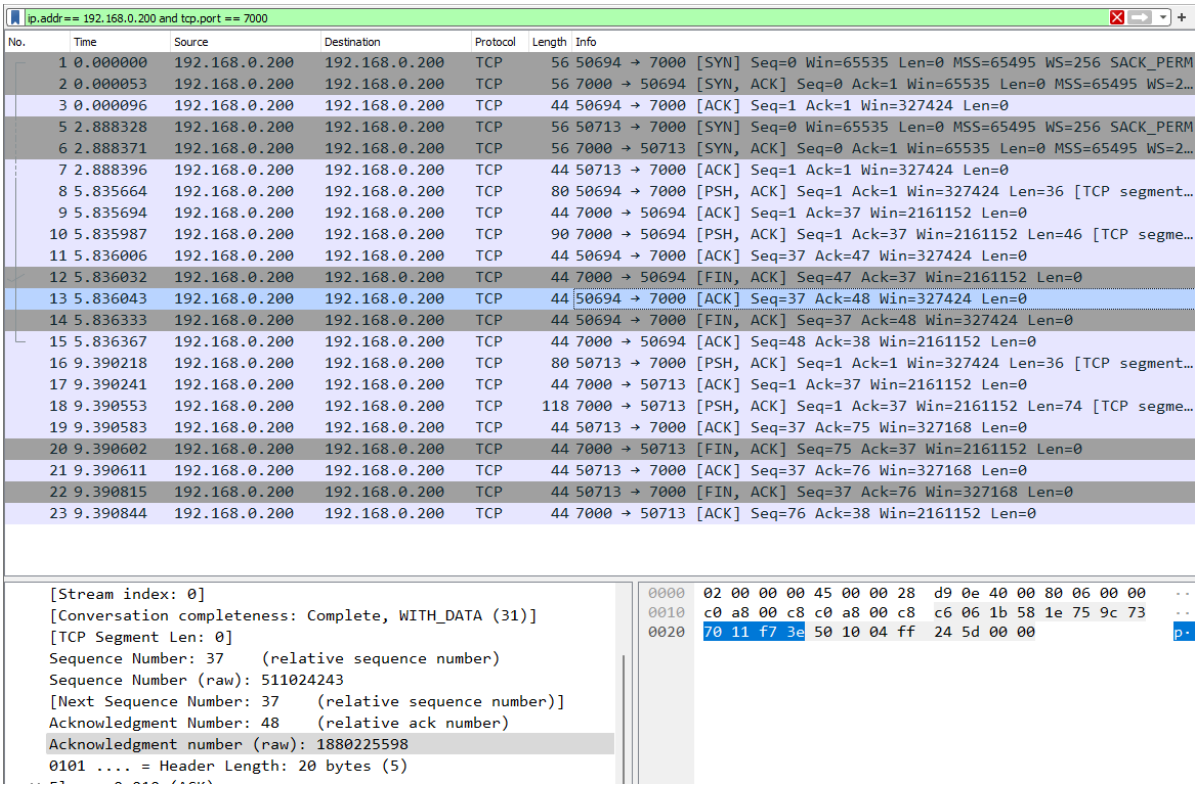


Figure 14.2: TCP termination phase (client 1/server) step 2

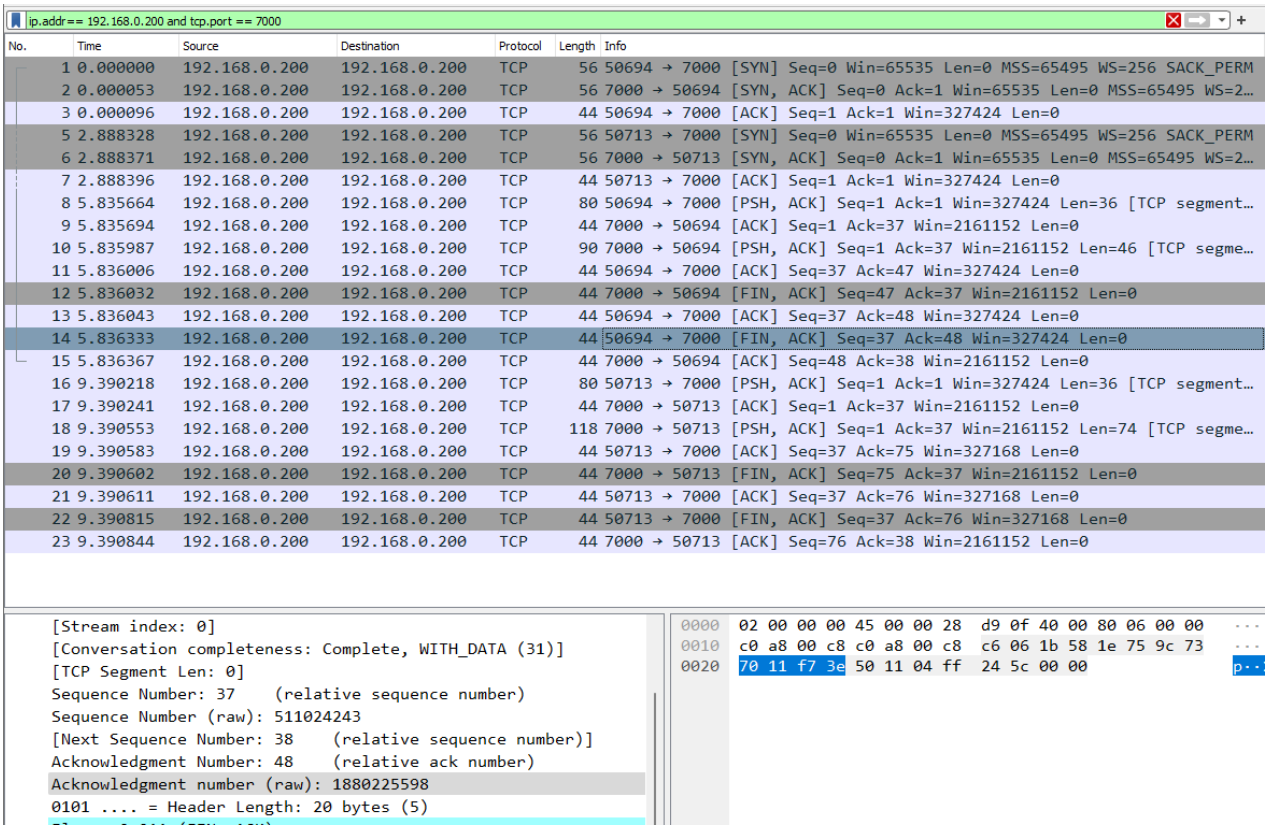


Figure 14.3: TCP termination phase (client 1/server) step 3

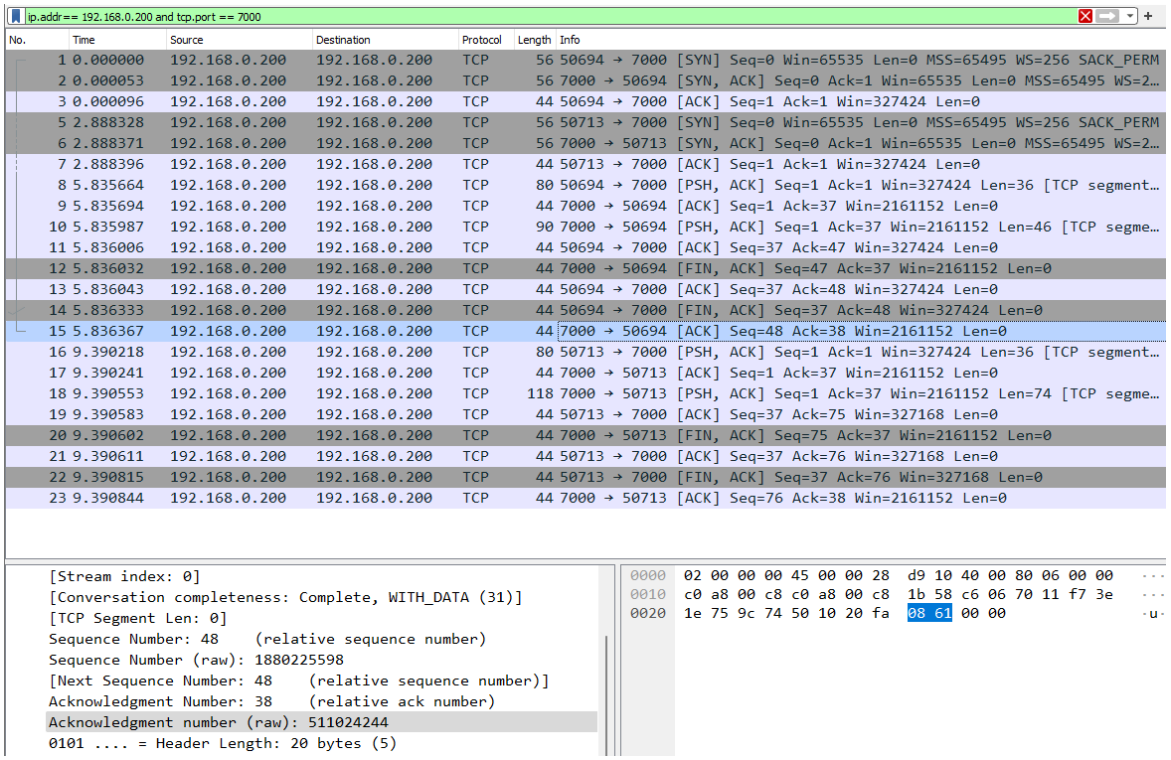


Figure 14.4: TCP termination phase (client 1/server) step 4

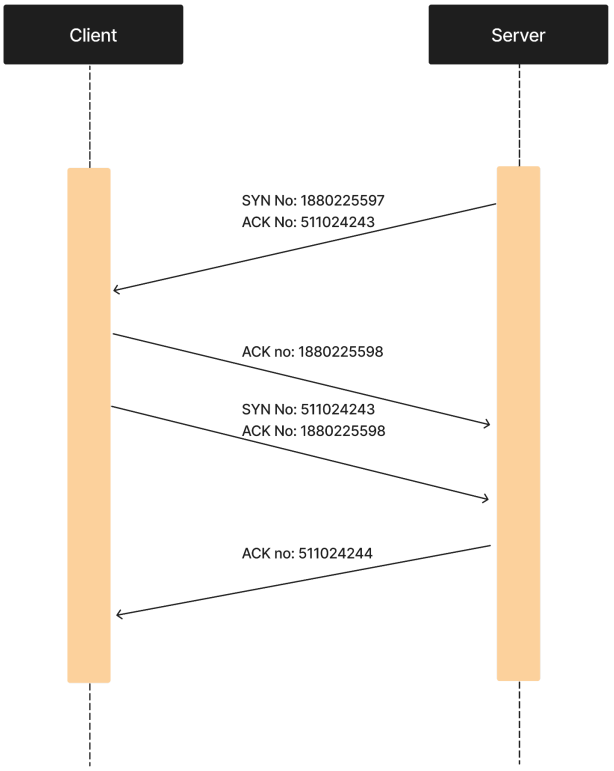


Figure 15: Sequence diagram TCP termination phase (client 1/server)

4.2 Client 2 - server

- ➔ Server sends FIN, ACK packet to client 2 (7000 → 50713) with sequence no: 3998654971 and acknowledge no: 2375590608
- ➔ Client 1 sends back an ACK packet in response (50713 → 7000) with sequence no: 2375590608, acknowledge no: 3998654972
- ➔ Client 1 send a FIN, ACK packet to server (50713 → 7000) which signifies that client wants to close the connection with sequence no: 2375590608, acknowledge no: 3998654972
- ➔ Server sends ACK packet to client 2 (7000 → 50713) in response with sequence no: 399865972, acknowledge no: 2375590609

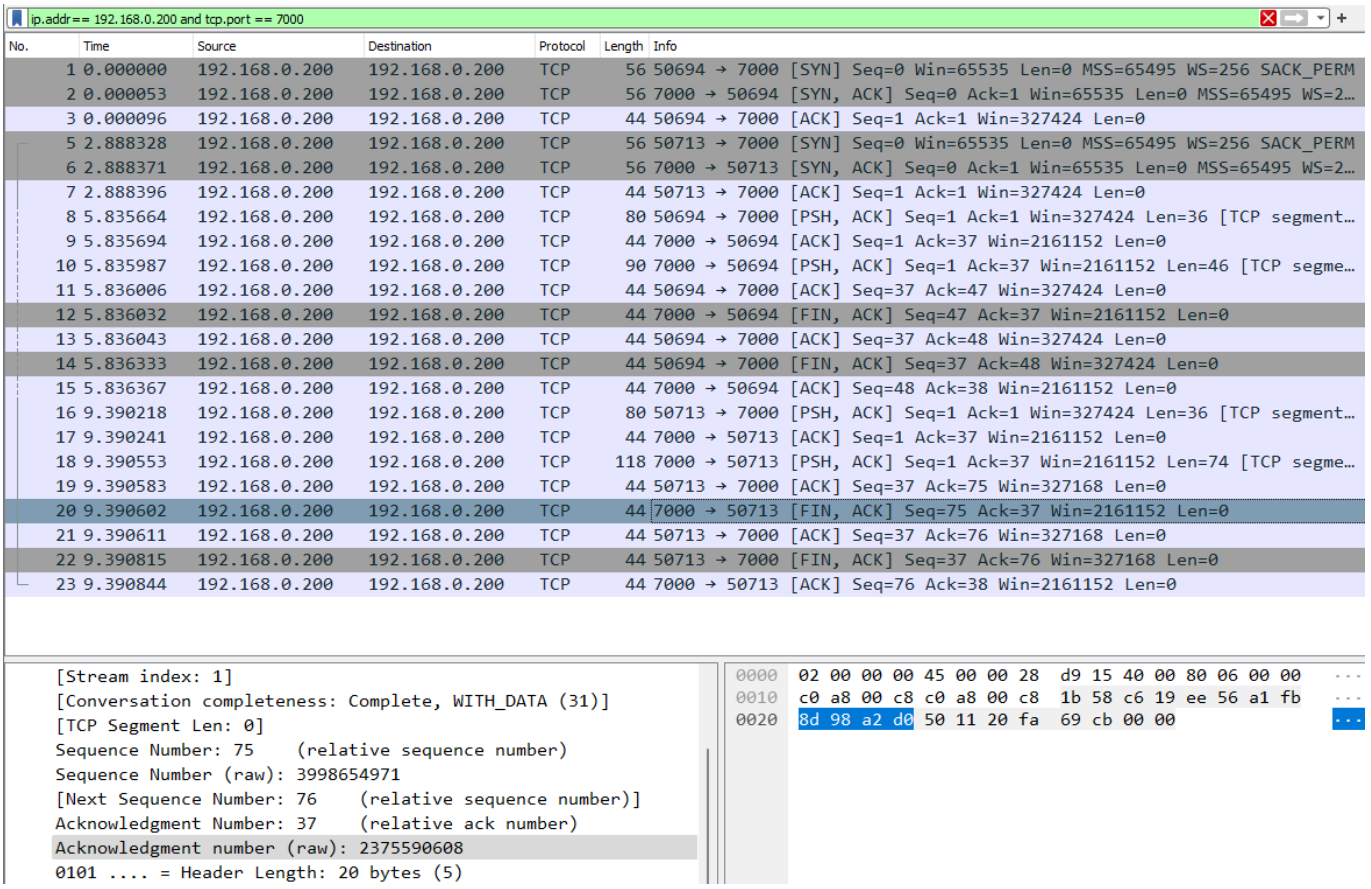


Figure 16.1: TCP termination phase (client 2/server) step 1

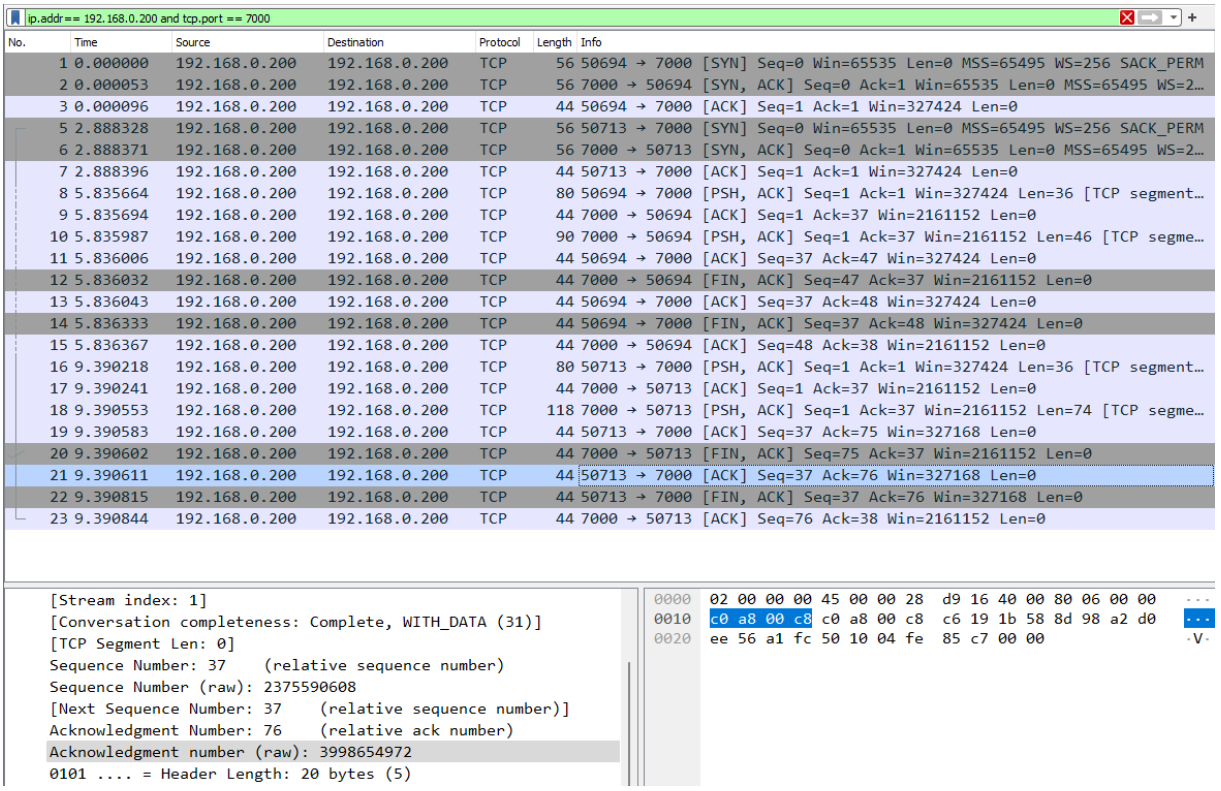


Figure 16.2: TCP termination phase (client 2/server) step 2

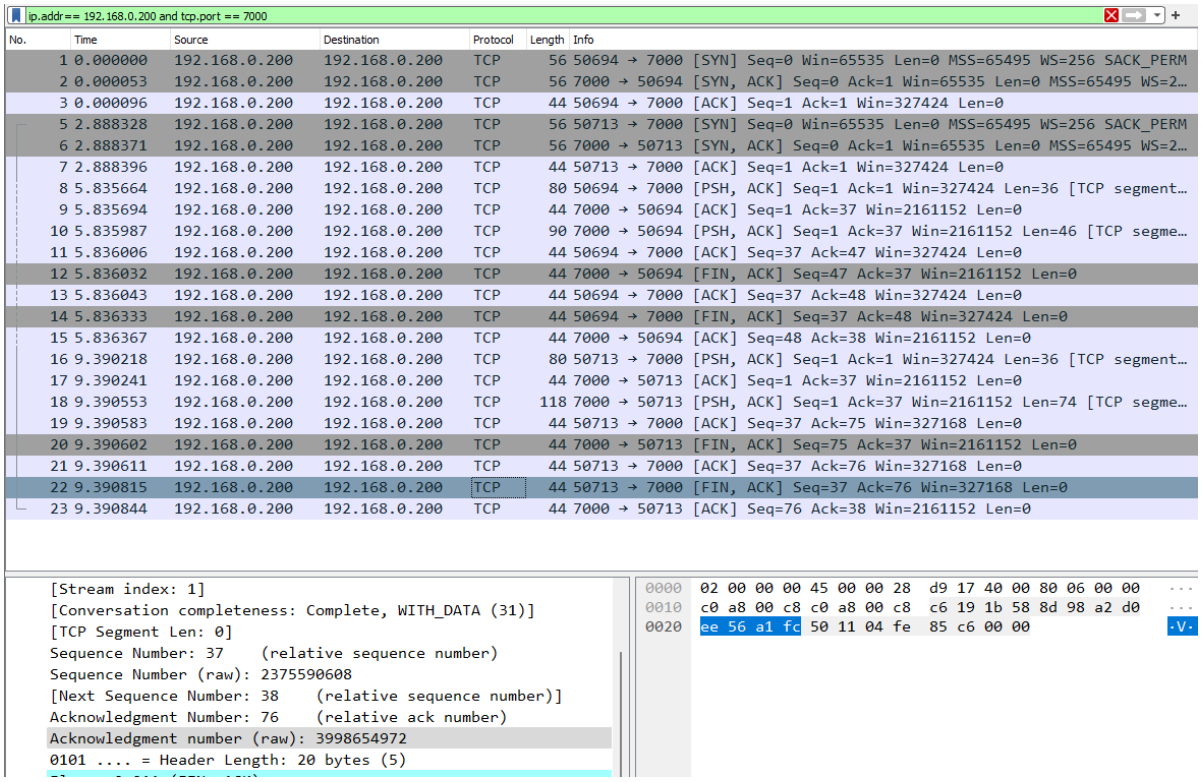


Figure 16.3: TCP termination phase (client 2/server) step 3

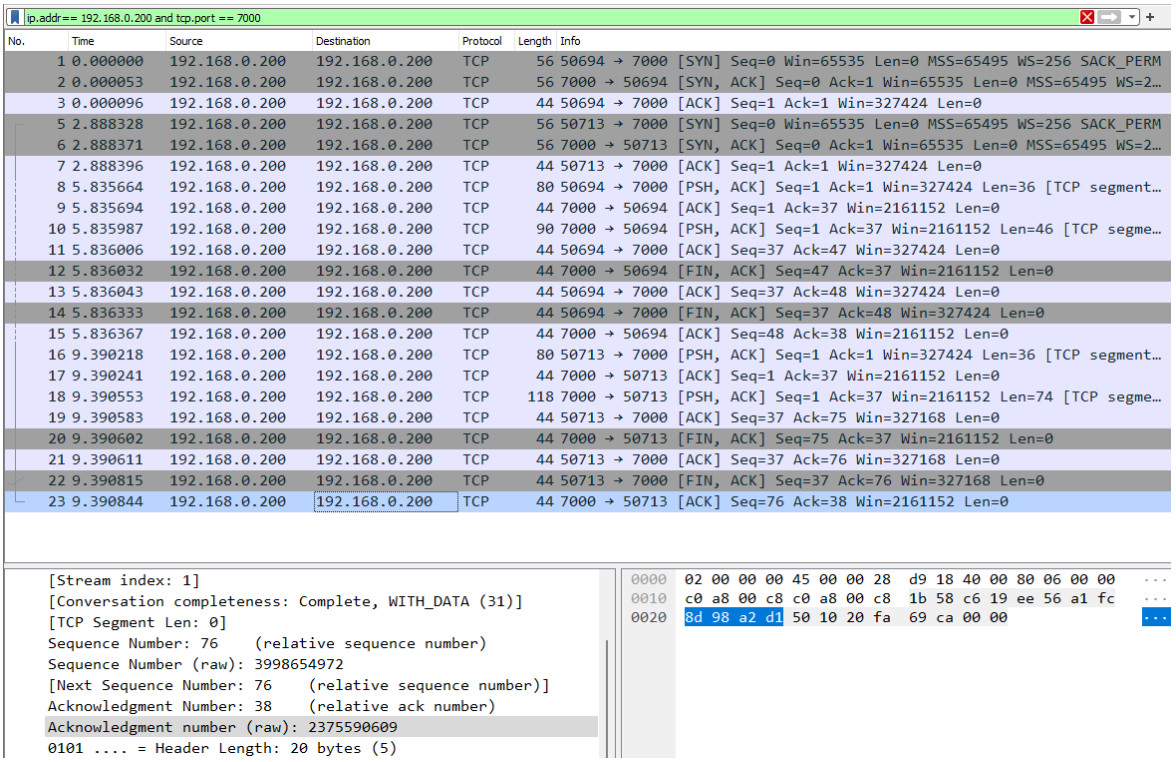


Figure 16.4: TCP termination phase (client 2/server) step 4

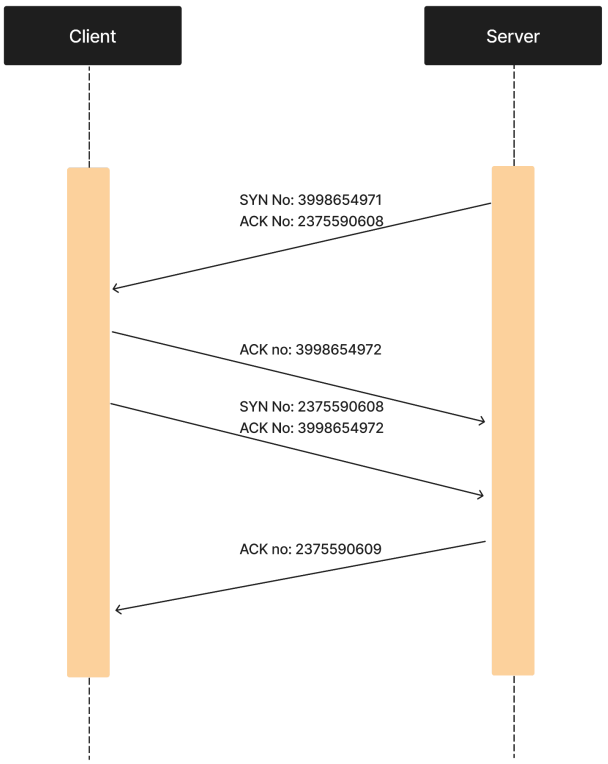


Figure 17: Sequence diagram TCP termination phase (client 2/server)

Assignment - III

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.

Step 1: 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 server.
(vi) Stop Wireshark capture

Step 2: 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.

Step 1: Capture a Trace →

For this task, the client/server program of question 3 from assignment-1 is used, in which the UDP client program sends requests to the server with a student name to get the requested student data (name, address). The server returns the student info if the student exists in the local student database (a csv file stored in the server) otherwise it sends that student does not exist message. Using wireshark the packets sent/received by the client/server will be observed.

Note: there will be two instances of clients, both will request the server for student info, one will request for a student name that exists in the database the other client will request for a student that does not exist in the database.

- ➔ Wireshark capture with Loopback interface as input and with filter of "ip.addr==127.0.0.1 and udp.port==9999" (Port 9999 is used by the UDP server)
- ➔ Two instances of the client connect to the server.

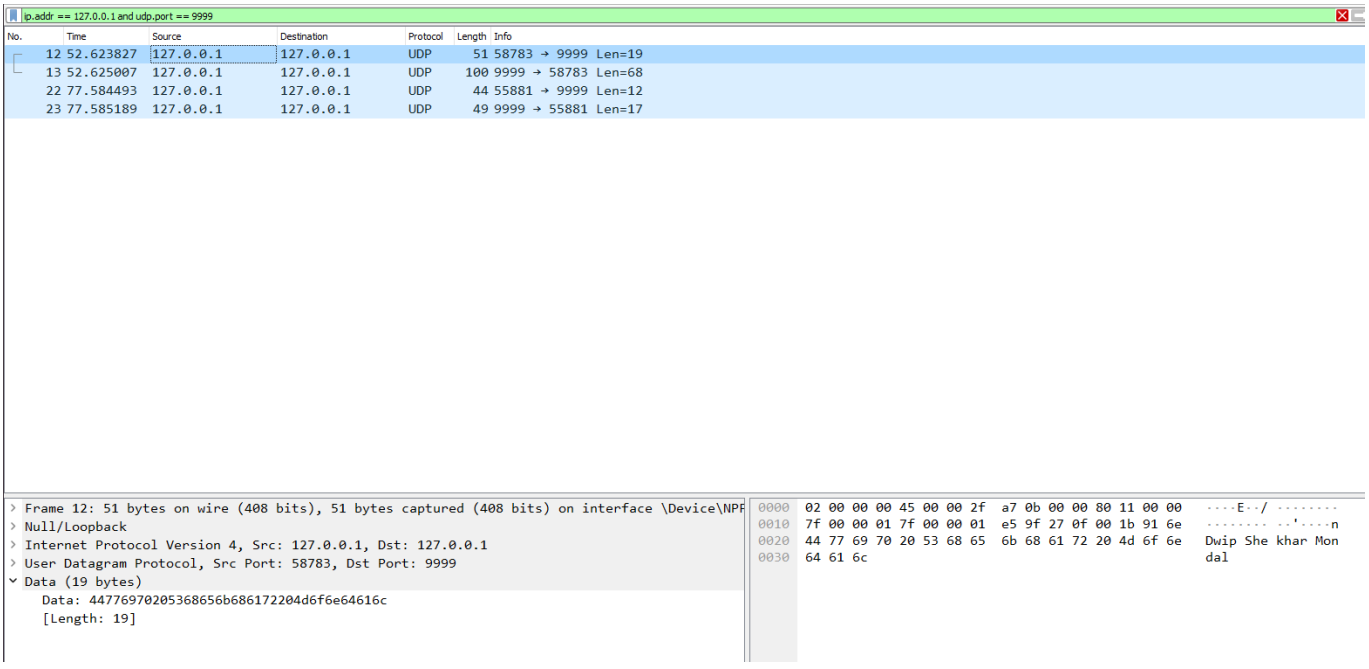


Figure 17: Capture of the packets from the client/server (UDP)

From the above capture it it's evident that →

- ➔ Two clients connects to the UDP server
- ➔ Server listens at port 9999
- ➔ 2 instances of client connects to the server
  - ◆ Client 1 connects from 58789 → 9999
  - ◆ Client 2 connects from 55881 → 9999
- ➔ Client 1 sends a request to get a student info “Dwip Shekhar Mondal” that exists in the student database stored at the server, and gets response “Name: Dwip Shekhar Mondal, Address: Greenpark Duttapukur pin-743248”
- ➔ Client 2 sends a request to get a student info “Rohit Sharma” that does not exist in the student database stored in the server, and gets a generic error message “Student does not exist”

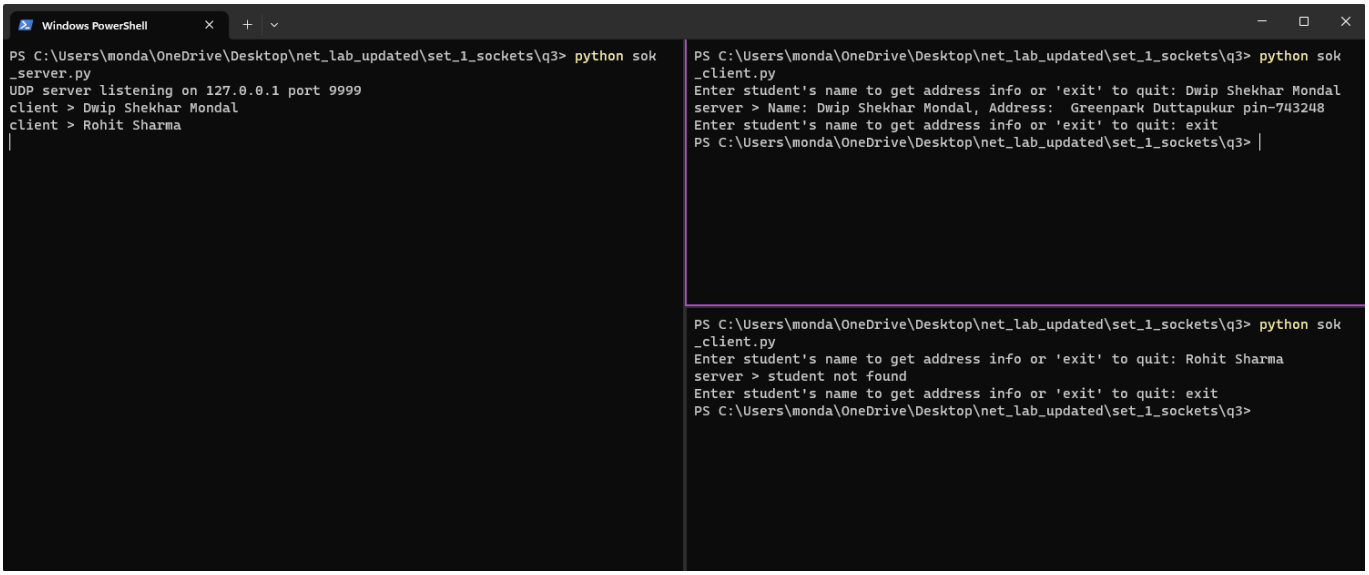


Figure 18: Request/response from client/server console (UDP)

Step 2: Inspect the Trace

Client 1 - server →

➔ Client 1 connects from port 58783 → 9999

- ◆ It sends the request with source port: 58783 and destination port: 9999
- ◆ It sends data with length 19 bytes i.e the UDP payload size is 19 bytes. Data = "Dwip Sekhar Mondal"
- ◆ Server Responds with appropriate message.
  - Source port → 9999
  - Destination port → 58783
  - Payload size → 68 Bytes
  - Data = "Name: Dwip Shekhar Mondal, Address: Greenpark Duttapukur pin-743248"

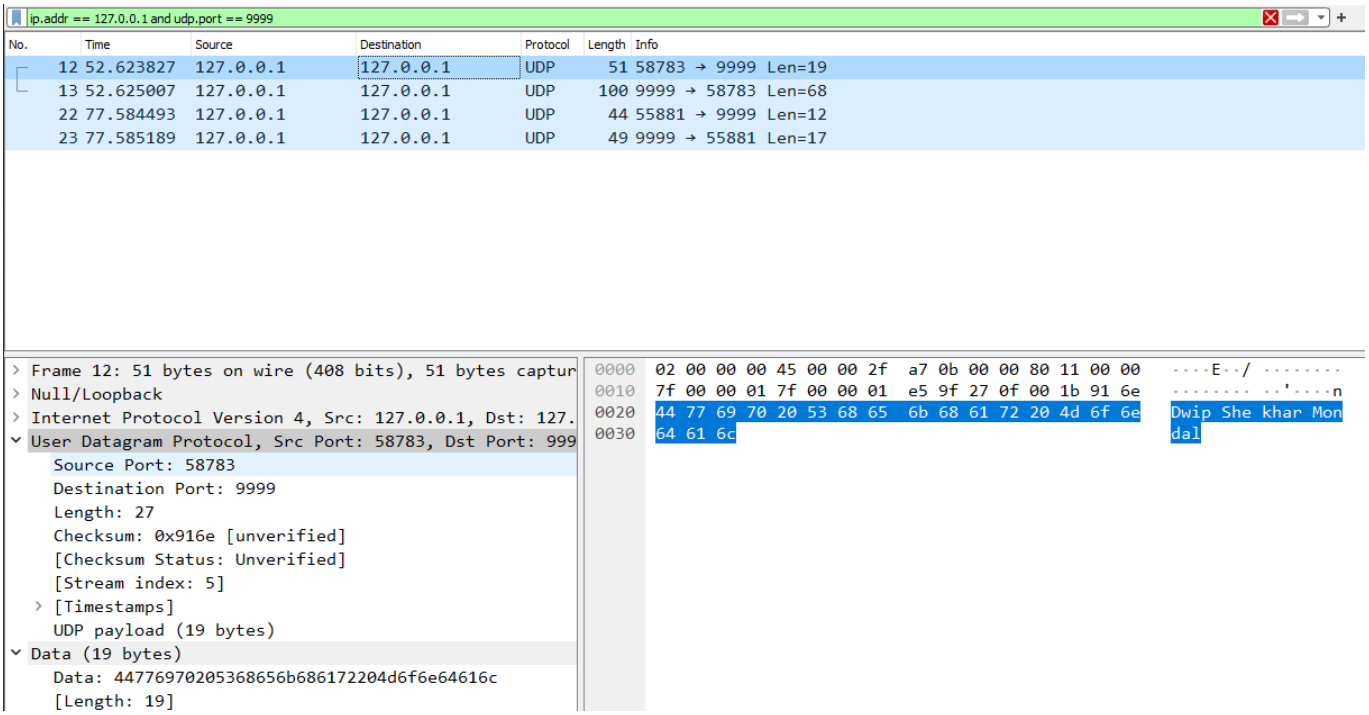


Figure 19: Request from client 1 to server (UDP)

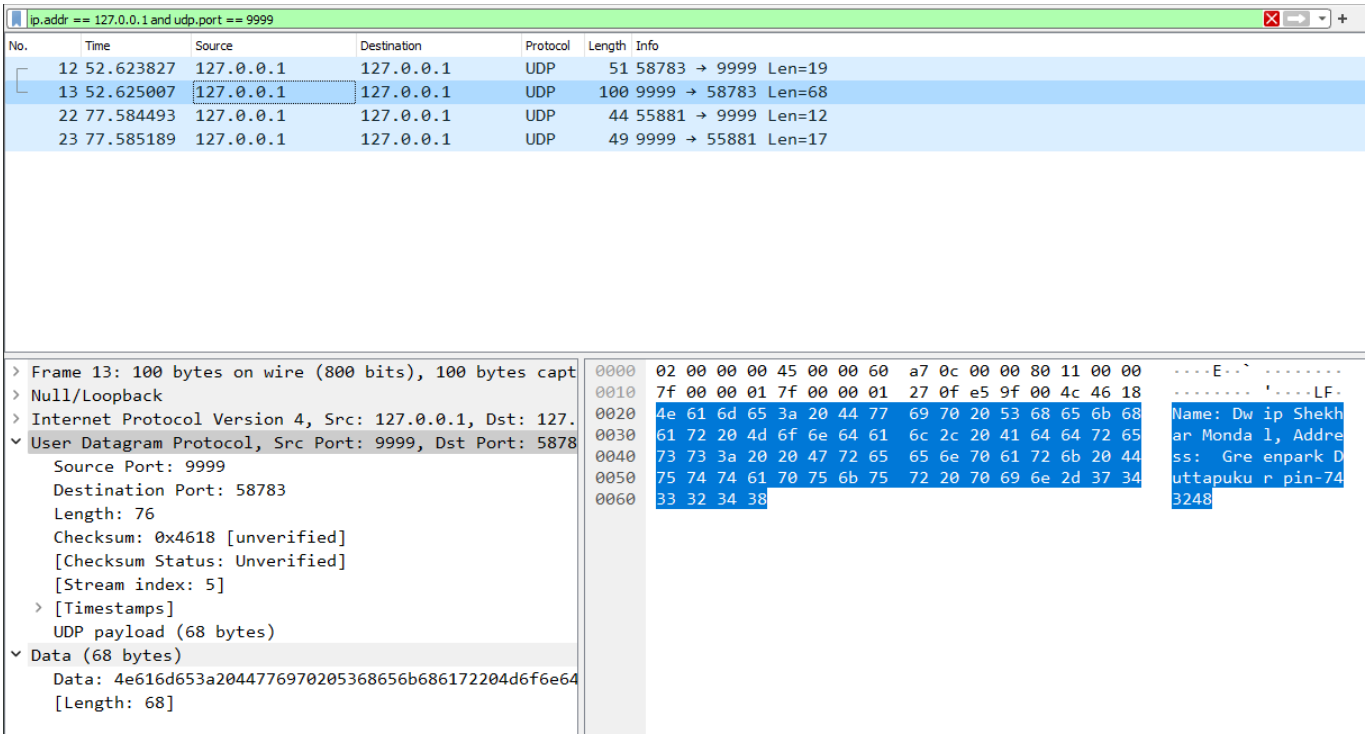


Figure 20: Response from server to client 1 (UDP)

Client 2 - server →

➔ Client 2 connects from port 55881 → 9999

- ◆ It sends the request with source port: 55881 and destination port: 9999
- ◆ It sends data with length 12 bytes i.e the UDP payload size is 12 bytes.
  Data = “Rohit Sharma”
- ◆ Server Responds with appropriate message.
  - Source port → 9999
  - Destination port → 55881
  - Payload size → 17 Bytes
  - Data = “student not found”

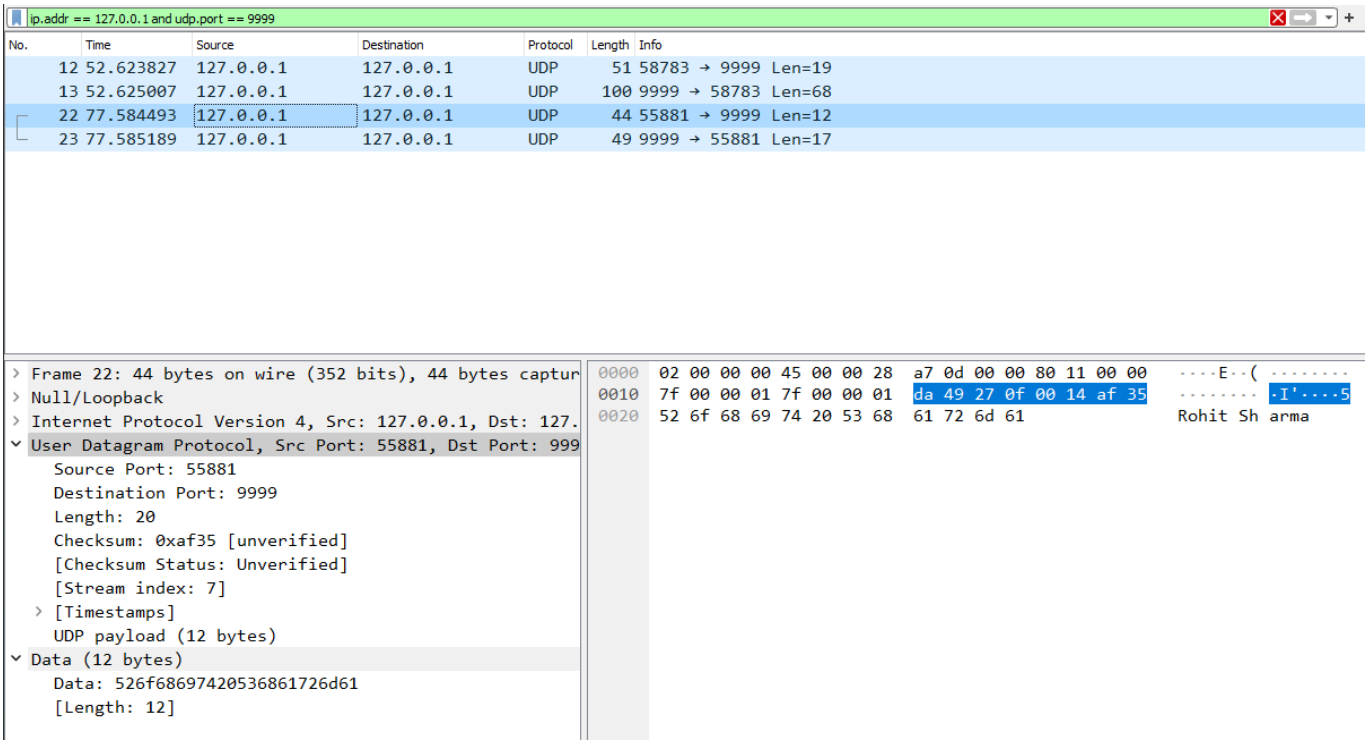


Figure 21: Request from client 2 to server (UDP)

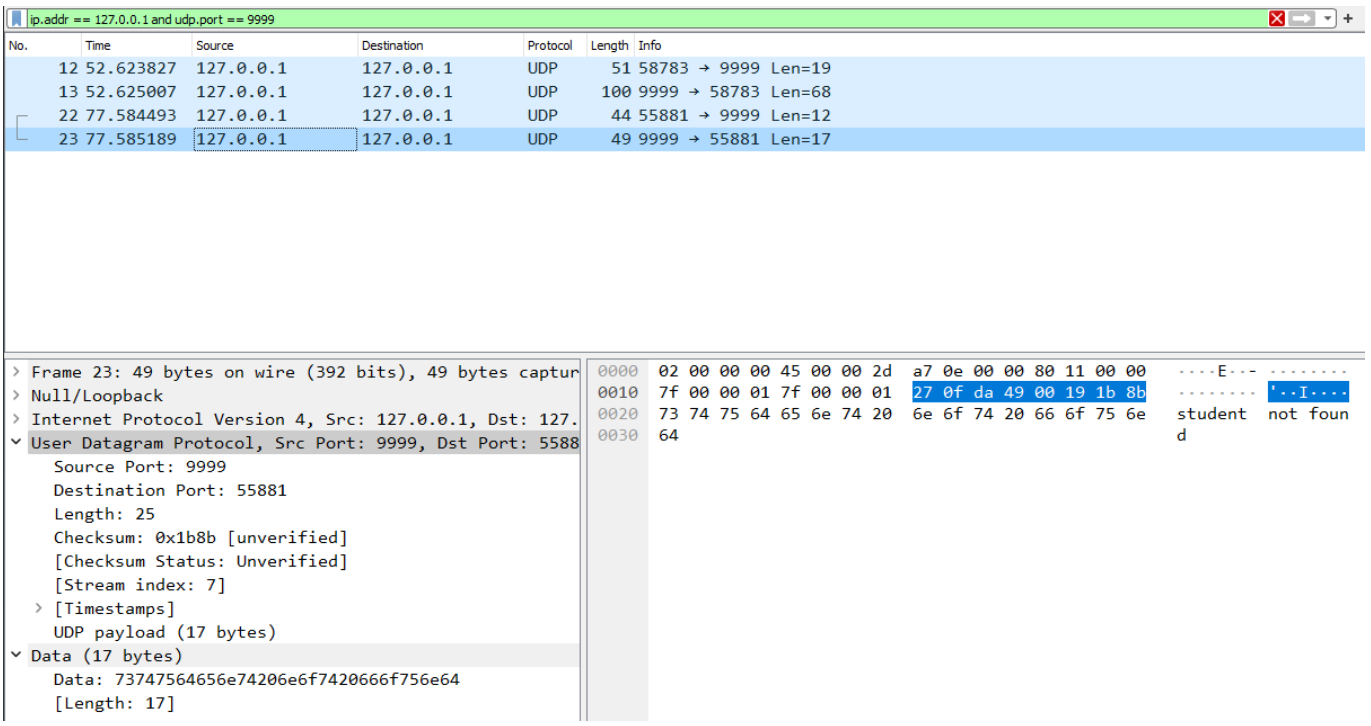


Figure 22: Response from server to client 2 (UDP)