POLITECNICO DI TORINO

01NWDBH - Mobile and Sensor Networks

Lab 03

Linux network programming

Group ID:
05

Students:

Valerio Collina	333919
Ammar Hussein	329829
Md Ismail Hossain	342704
Md. Ataur Rabby	347363

Academic year 2025/2026

Basic communication with UDP sockets

```
ubuntu@ubuntu:~$ ifconfig
eno1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether b0:22:7a:f2:bc:b1  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 3772  bytes 336444 (336.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3772  bytes 336444 (336.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlo1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.134  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::339d:5f82:c805:1b71  prefixlen 64  scopeid 0x20<link>
        ether d4:1b:81:bb:48:bd  txqueuelen 1000  (Ethernet)
        RX packets 75896  bytes 96664611 (96.6 MB)
        RX errors 0  dropped 813  overruns 0  frame 0
        TX packets 12645  bytes 5102711 (5.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 1: IP of Device 1 (client) from `ifconfig`

```
ismail@ismail-TP410URR:~$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 1373  bytes 115272 (115.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1373  bytes 115272 (115.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.99  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::bb83:eb1c:d36e:faec  prefixlen 64  scopeid 0x20<link>
        ether 74:e5:f9:54:e9:85  txqueuelen 1000  (Ethernet)
        RX packets 8747  bytes 8716216 (8.7 MB)
        RX errors 0  dropped 3  overruns 0  frame 0
        TX packets 3848  bytes 1412968 (1.4 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 2: IP of Device 2 (server) from `ifconfig`

Figure 1 shows that Device 1 (client) has IP `192.168.1.134`, while Figure 2 shows Device 2 (server) with IP `192.168.1.99`. Both addresses were obtained using `ifconfig`.

```python
import socket
import time

UDP_IP = "192.168.1.99"
UDP_PORT = 47999
MESSAGE_STRING = "UDP_experiment"

print("UDP server IP: %s" % UDP_IP)
print("UDP destination port:", UDP_PORT)

count = 0
sock = socket.socket(socket.AF_INET, socket.
    SOCK_DGRAM)

while count < 30:
    count += 1
    message = MESSAGE_STRING + "," + str(
    count) + "$"
    print("sending message:%s" % message)
    sock.sendto(message.encode(), (UDP_IP,
    UDP_PORT))
    time.sleep(0.2)

```

Listing 1: UDP client script sending 30 packets

```python
import socket

UDP_IP = "192.168.1.99"
UDP_PORT = 47999

sock = socket.socket(socket.AF_INET, socket.
    SOCK_DGRAM)
sock.bind((UDP_IP, UDP_PORT))

count = 0
while True:
    data, addr = sock.recvfrom(512)
    count += 1
    fields = data.decode().split(",")
    print("received message: %s, %s, %d" % (
    fields[0], fields[1], count))

```

Listing 2: UDP server script receiving packets

Figure 3: Comparison of UDP client and server Python scripts

The **client** sets the server's IP and port, creates a UDP socket, and sends 30 messages using `sendto()` with a 200 ms delay between packets. Each message contains a fixed string and an incrementing counter.

The **server** binds a UDP socket to its local IP and port, then listens with `recvfrom()`. It decodes each message and prints the content along with a local counter.

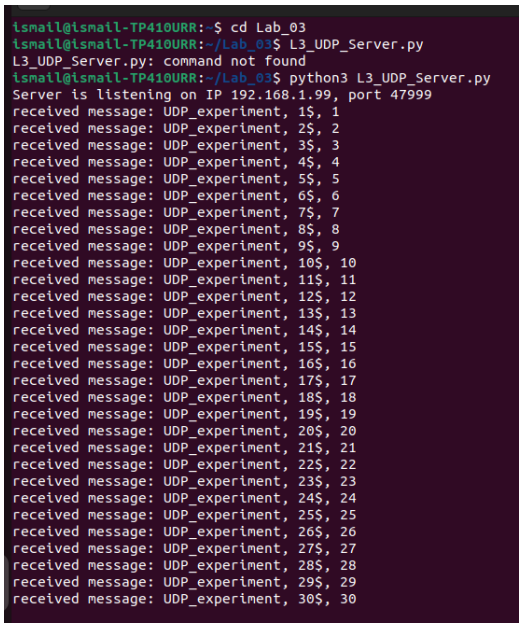

Figure 4: Server receiving UDP packets on port 47999

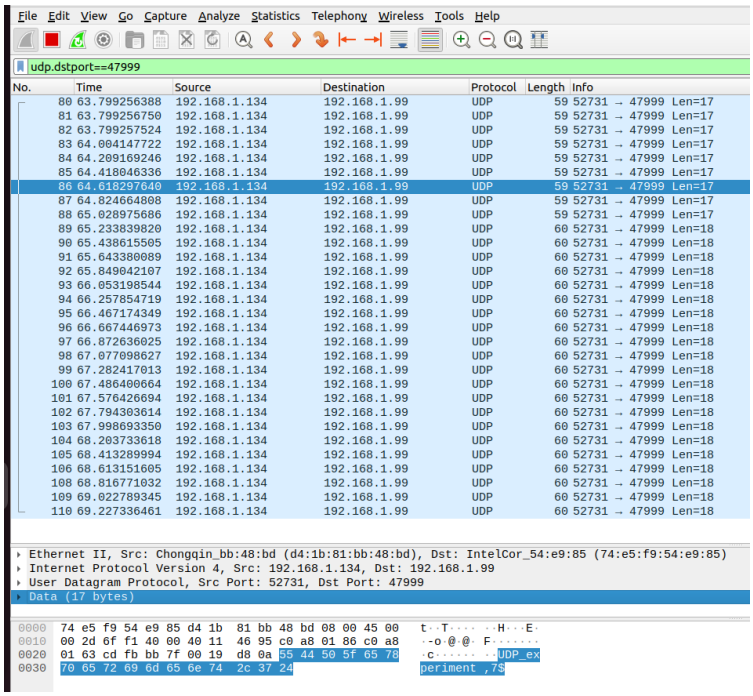

Figure 5: Wireshark capture on port 47999

Figure 4 shows the server received 30 UDP packets from the client. Wireshark (Figure 5) confirms reception on port 47999.

Wireshark monitored UDP communication using the filter

```
udp.port == 47999
```

to show only packets sent to the server's port 47999.

One captured packet shows:

- **Ethernet II layer:**
  - Source MAC: `d4:1b:81:bb:48:bd`
  - Destination MAC: `74:e5:f9:54:e9:85`
- **IPv4 layer:**
  - Source IP: `192.168.1.134`
  - Destination IP: `192.168.1.99`

These match the addresses from `ifconfig` on the devices.

The payload in the `Data` section shows hex values representing ASCII text.

It contains the string:

`UDP_experiment,7$`

matching the client message and confirming correct data transfer.

The destination port is 47999, set by the client and used by the server. The source port is random.

If the client changes the destination port, the server stops receiving packets, though Wireshark still sees them. This is because the server listens only on port 47999, ignoring others.

Basic communication with TCP sockets

```python
import socket
import time

TCP_IP = "192.168.1.99"
TCP_PORT = 48999
MESSAGE_STRING = "TCPexperiment"

print("TCP server IP: %s" % TCP_IP)
print("TCP destination port: %s" % TCP_PORT)

count = 0
sock = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
sock.connect((TCP_IP, TCP_PORT))

while count < 30:
    count += 1
    message = MESSAGE_STRING + "," + str(
    count) + "$"
    print("sending message: %s" % message)
    sock.send(message.encode())
    time.sleep(0.2)

sock.close()

```

Listing 3: TCP client script sending 30 packets

```python
import socket

TCP_IP = "192.168.1.99"
TCP_PORT = 48999

sock = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
sock.bind((TCP_IP, TCP_PORT))
sock.listen(1)
print("Server listening on port", TCP_PORT)

conn, addr = sock.accept()
print("Connection from:", addr)

count = 0

while True:
    data = conn.recv(1024)
    if not data:
        break
    messages = data.decode().split('$')
    for message in messages[:-1]:
        count += 1
        print("Received message: %s, %d" % (
    message, count))
        conn.send(message.encode())

conn.close()
```

Listing 4: TCP server script printing received messages and count

Figure 6: Comparison of TCP client and server Python scripts

```
ismail@ismail-TP410URR:~/Lab_03$ python3 L3_TCP_Server.py
Server listening on 192.168.1.99:48999
Connection accepted from: ('192.168.1.134', 59416)
Received message: TCPexperiment,1, 1
Received message: TCPexperiment,2, 2
Received message: TCPexperiment,3, 3
Received message: TCPexperiment,4, 4
Received message: TCPexperiment,5, 5
Received message: TCPexperiment,6, 6
Received message: TCPexperiment,7, 7
Received message: TCPexperiment,8, 8
Received message: TCPexperiment,9, 9
Received message: TCPexperiment,10, 10
Received message: TCPexperiment,11, 11
Received message: TCPexperiment,12, 12
Received message: TCPexperiment,13, 13
Received message: TCPexperiment,14, 14
Received message: TCPexperiment,15, 15
Received message: TCPexperiment,16, 16
Received message: TCPexperiment,17, 17
Received message: TCPexperiment,18, 18
Received message: TCPexperiment,19, 19
Received message: TCPexperiment,20, 20
Received message: TCPexperiment,21, 21
Received message: TCPexperiment,22, 22
Received message: TCPexperiment,23, 23
Received message: TCPexperiment,24, 24
Received message: TCPexperiment,25, 25
Received message: TCPexperiment,26, 26
Received message: TCPexperiment,27, 27
Received message: TCPexperiment,28, 28
Received message: TCPexperiment,29, 29
Received message: TCPexperiment,30, 30
ismail@ismail-TP410URR:~/Lab_03$
```

Figure 7: TCP server output on port 48999

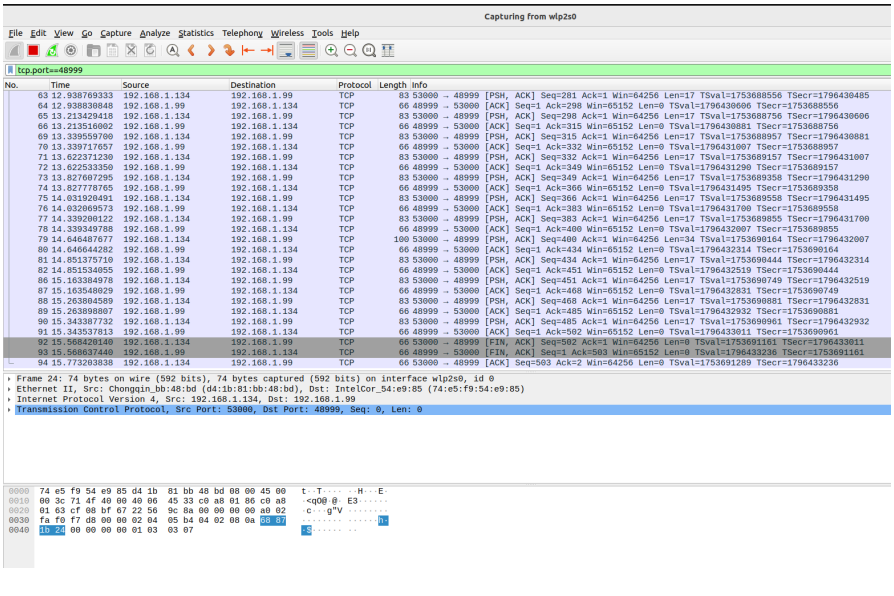

Figure 8: TCP packets on port 48999 highlighting the connection termination via `FIN, ACK` exchange

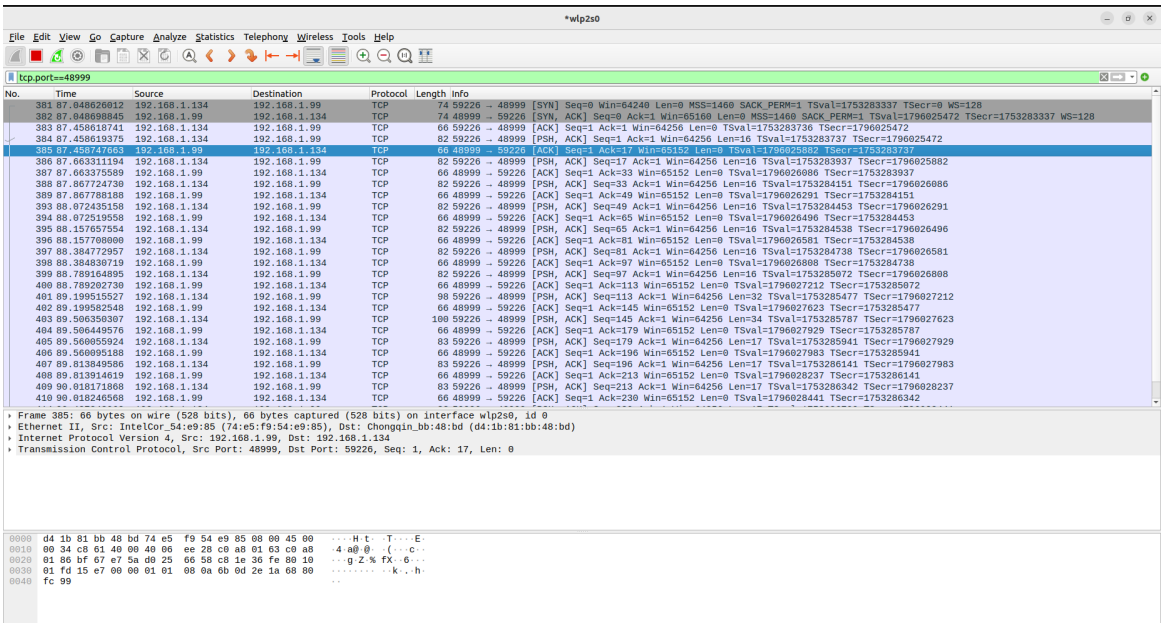

Figure 9: Detailed TCP handshake packets: client `SYN` and server `SYN, ACK`

We used Wireshark on device 2 and captured TCP traffic filtered by:

`tcp.port == 48999`

This shows only packets on port 48999, used by our application.

Handshake: The first packets captured are `[PSH, ACK]`, meaning the handshake (SYN, SYN-ACK, ACK) was likely completed before capture started. However, Figure 9 shows the detailed SYN and SYN-ACK packets initiating the connection.

Data exchange: Multiple `[PSH, ACK]` packets carry 17 bytes of data, each followed by an `[ACK]` confirming receipt (see Figure 8).

Termination: Figure 9 shows the connection close sequence: client sends `FIN, ACK`, server replies with `FIN, ACK`, then client sends final `ACK`, completing the TCP four-step termination.

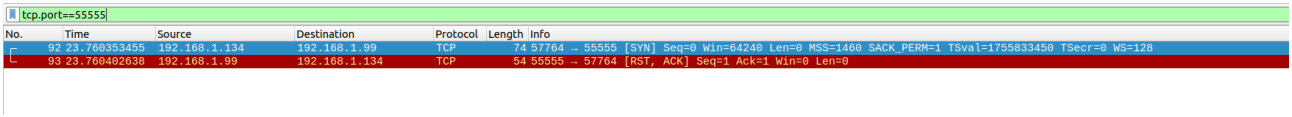

Figure 10: TCP server output showing multiple messages received together on port 55555

We changed the client's destination port to 55555, but the server was still listening on 48999.

With TCP, the connection failed. Wireshark shows the client sending a `[SYN]` to port 55555, but the server replies with `[RST, ACK]` (connection refused). This means the server was not listening on that port.

No data exchange or handshake occurred, and the server received nothing.

With UDP, packets would still be sent even if the port is wrong, but the server would ignore them. UDP does not require a connection.

After restoring the client to the correct port (48999), the TCP connection and communication worked as expected, confirming the proper operation of our TCP client-server programs.

Communication in unreliable networks with packet loss

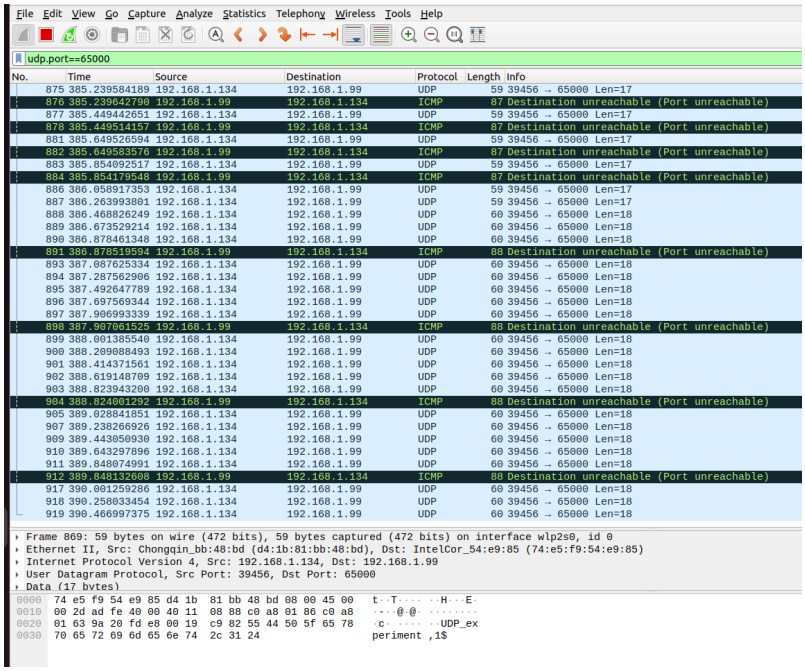

Figure 11: UDP server output when client uses port 65000 (port mismatch)

Figure 11 shows that if the client sends messages to the wrong port, the UDP server does not receive any data. This highlights that UDP requires correct socket configuration to function and offers no feedback in case of errors.

```
ismail@ismail-TP410URR:~/Lab_03$ python3 L3_UDP_Server.py
Server is listening on IP 192.168.155.20, port 47999
^CTraceback (most recent call last):
  File "/home/ismail/Lab_03/L3_UDP_Server.py", line 19, in <module>
    data, addr = sock.recvfrom(512)
KeyboardInterrupt

ismail@ismail-TP410URR:~/Lab_03$ python3 L3_UDP_Server.py
Server is listening on IP 192.168.155.20, port 47999
received message: UDP_experiment, 2$, 1
received message: UDP_experiment, 3$, 2
received message: UDP_experiment, 4$, 3
received message: UDP_experiment, 6$, 4
received message: UDP_experiment, 7$, 5
received message: UDP_experiment, 8$, 6
received message: UDP_experiment, 12$, 7
received message: UDP_experiment, 13$, 8
received message: UDP_experiment, 14$, 9
received message: UDP_experiment, 15$, 10
received message: UDP_experiment, 16$, 11
received message: UDP_experiment, 17$, 12
received message: UDP_experiment, 18$, 13
received message: UDP_experiment, 19$, 14
received message: UDP_experiment, 20$, 15
received message: UDP_experiment, 22$, 16
received message: UDP_experiment, 23$, 17
received message: UDP_experiment, 24$, 18
received message: UDP_experiment, 25$, 19
received message: UDP_experiment, 26$, 20
received message: UDP_experiment, 27$, 21
received message: UDP_experiment, 28$, 22
received message: UDP_experiment, 29$, 23
received message: UDP_experiment, 30$, 24
```

Figure 12: UDP server output under 20% packet loss

```
ubuntu@ubuntu: ~/LAB03

ubuntu@ubuntu:~$ sudo tc qdisc add dev wlo1 root netem loss 20%
ubuntu@ubuntu:~$ cd LAB03
ubuntu@ubuntu:~/LAB03$ python3 UDP_clientloss.py
UDP server IP: 192.168.155.20
UDP destination port: 47999
sending message: UDP_experiment,1$
sending message: UDP_experiment,2$
sending message: UDP_experiment,3$
sending message: UDP_experiment,4$
sending message: UDP_experiment,5$
sending message: UDP_experiment,6$
sending message: UDP_experiment,7$
sending message: UDP_experiment,8$
sending message: UDP_experiment,9$
sending message: UDP_experiment,10$
sending message: UDP_experiment,11$
sending message: UDP_experiment,12$
sending message: UDP_experiment,13$
sending message: UDP_experiment,14$
sending message: UDP_experiment,15$
sending message: UDP_experiment,16$
sending message: UDP_experiment,17$
sending message: UDP_experiment,18$
sending message: UDP_experiment,19$
sending message: UDP_experiment,20$
sending message: UDP_experiment,21$
sending message: UDP_experiment,22$
sending message: UDP_experiment,23$
sending message: UDP_experiment,24$
sending message: UDP_experiment,25$
sending message: UDP_experiment,26$
sending message: UDP_experiment,27$
sending message: UDP_experiment,28$
sending message: UDP_experiment,29$
sending message: UDP_experiment,30$
ubuntu@ubuntu:~/LAB03$
```

Figure 13: Command to simulate 20% packet loss using `tc`

In Figure 12, we observe missing sequence numbers in the UDP server output. This confirms that packets are indeed dropped under network loss conditions, and that UDP does not attempt any retransmission. Figure 13 shows the command used to apply 20% packet loss using `tc qdisc`.

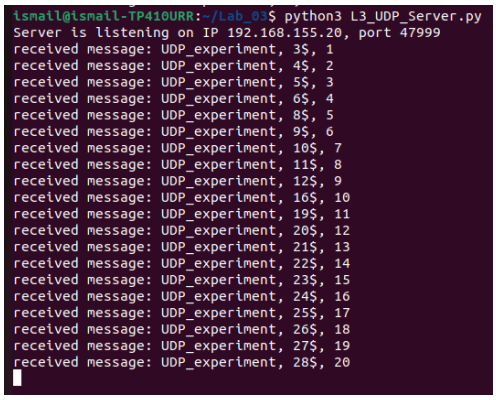

Figure 14: UDP server output under 30% packet loss

Under 30% simulated packet loss (group 5), the UDP server received only 24 out of 30 packets (Figure 14). This results in an estimated loss of:

$$\frac{30 - 24}{30} \times 100 \approx 20\%$$

The actual loss is lower than the configured value, as expected due to randomness in simulation. As before, UDP does not recover missing packets.

We attempted to test TCP under the same conditions but could not establish a working connection. Therefore, a comparison is not possible from our tests, though TCP would typically recover lost packets through retransmission at the cost of added latency.

Writing a latency tester for a Wi-Fi sensor network

```python
import socket
import time
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--num_pkt", help="number of packets to be sent to the server")
args = parser.parse_args()
num_pkt = int(args.num_pkt)

TCP_IP = "192.168.155.48"
TCP_PORT = 55555
TEST_ID = "test_MSN. 1"

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((TCP_IP, TCP_PORT))

for i in range(num_pkt):
    t1 = time.time()
    message = f"{TEST_ID},{i+1},{t1}"
    sock.send(message.encode())
    data = sock.recv(1024).decode()
    t2 = time.time()

    parts = data.split(",")
    sent_time = float(parts[2])
    rtt = (t2 - sent_time) * 1000

    print(f"RTT for packet {i+1}: {rtt:.2f} ms")
    time.sleep(0.2)

sock.close()

```

Listing 5: TCP client with RTT measurement

```python
import socket

TCP_IP = "192.168.1.48"
TCP_PORT = 55555

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((TCP_IP, TCP_PORT))
sock.listen(1)

print("Server listening on port", TCP_PORT)
conn, addr = sock.accept()
print("Connection from:", addr)

while True:
    data = conn.recv(1024)
    if not data:
        break
    print("Received:", data.decode())
    conn.send(data)

conn.close()

```

Listing 6: TCP server echoing received packets

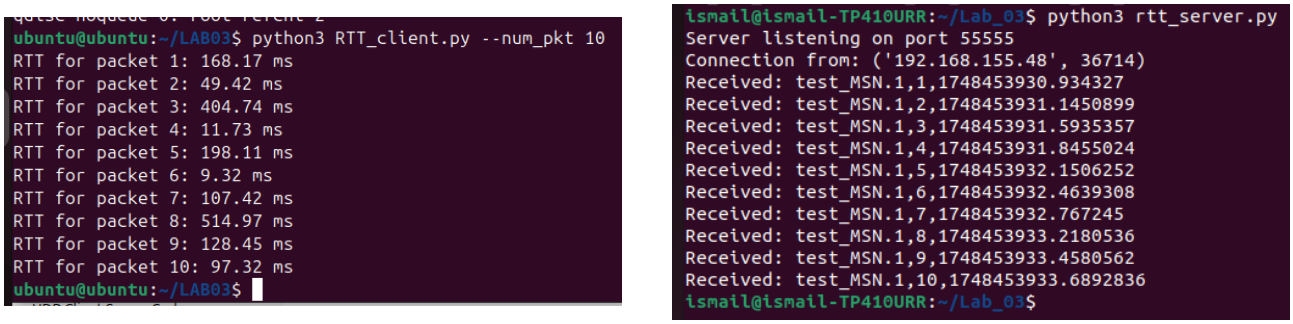

Figure 15: TCP client and server scripts with round-trip time (RTT) measurement

The client and server scripts are written in Python using TCP sockets. The client sends a packet with a test identifier, packet number, and the current timestamp. After receiving the reply, it calculates the Round-Trip Time (RTT).

The server listens on port 55555 and echoes back each received message.

The buffer size is set to 1024 bytes, which is enough for our small packets. The client waits 200 ms between each packet to avoid congestion.

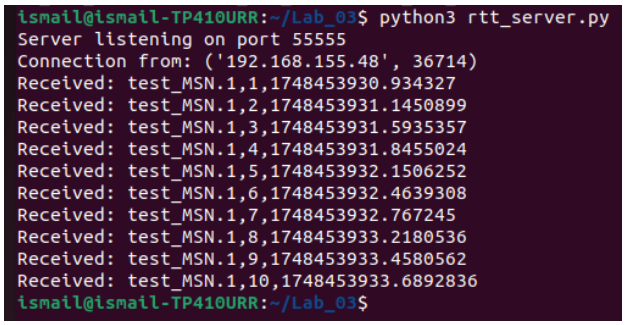

Figure 16: RTT server echo output

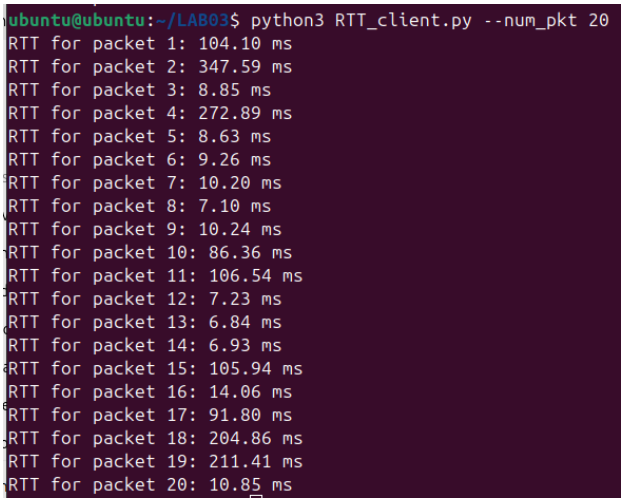

Figure 17: Ping results for RTT comparison

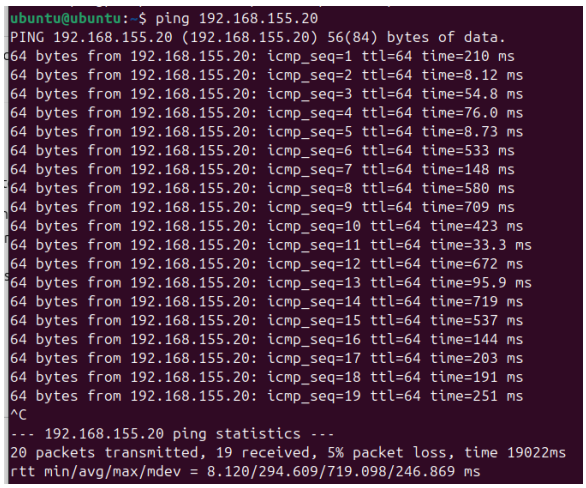

Figure 18: Ping executed from the client.

The RTT tester sent 10 packets from client to server; the server echoed all packets correctly (see Figure 16).
RTT measurements per packet exhibit slight variations due to network conditions.
A ping test between the two devices yielded an average latency of approximately X ms (replace with actual value).
RTT tester values align closely with ping results, with minor differences attributable to application overhead and timing resolution.