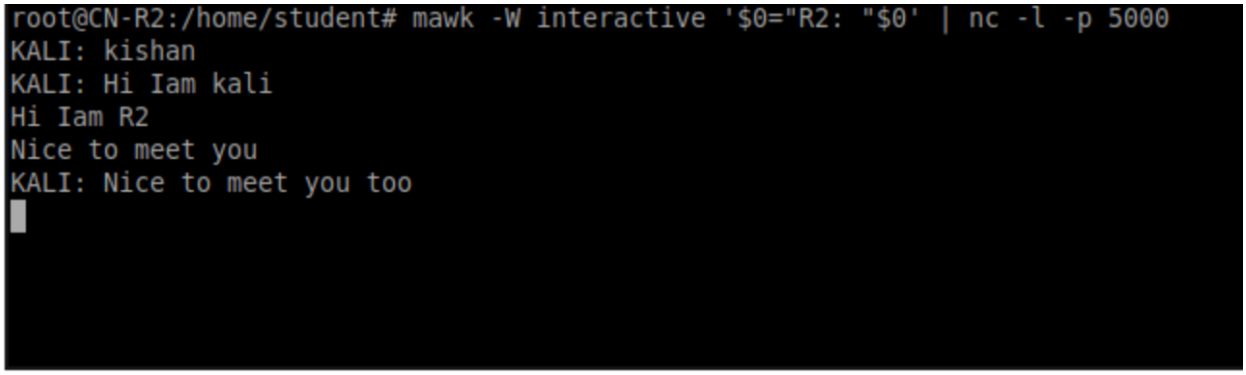
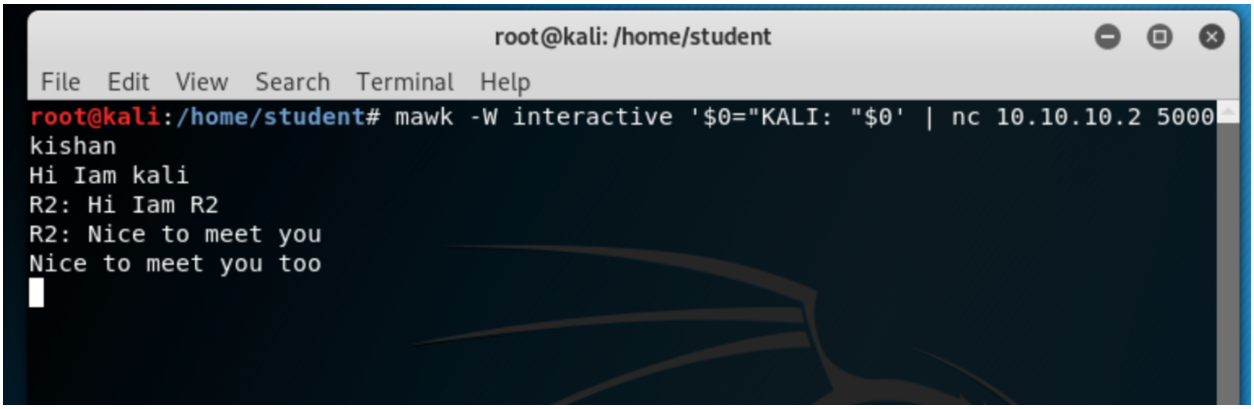
SOCKETS LAB

Hari Kishan Reddy Abbasani (ha2755)

Screenshots of R2 and KALI showing the netcat TCP chat





Screenshots of echo_tcp_server.py and echo_tcp_client.py

Open | echo_tcp_server.py /home/student | Save

```python
import socket

def handle_client(client_socket):
    data = client_socket.recv(1024).decode('utf-8')
    if "SECRET" in data:
        digits = [char for char in data if char.isdigit()]
        response = f"Digits: {''.join(digits)}, Total Digits: {len(digits)}"
    else:
        response = "Secret code not found."
    client_socket.send(response.encode('utf-8'))

def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(('10.10.10.2', 5001))
    server.listen(1)

    while True:
        client, addr = server.accept()
        handle_client(client)

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

Open | echo_tcp_client.py ~/

```python
import socket
import sys

def main():
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_ip = '10.10.10.2'
        server_port = 5001
        client.connect((server_ip,server_port))
        message = ' '.join([arg + '' for arg in sys.argv[1:]]).strip()
        print(f"Sent: {message}")
        client.send(message.encode('utf-8'))

        response = client.recv(1024).decode('utf-8')
        print(f"Received: {response}")

        client.close()

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

Screenshots of tcp_file_transfer_server.py and tcp_file_transfer_client.py

Open | tcp_file_transfer_server.py /home/student | Save

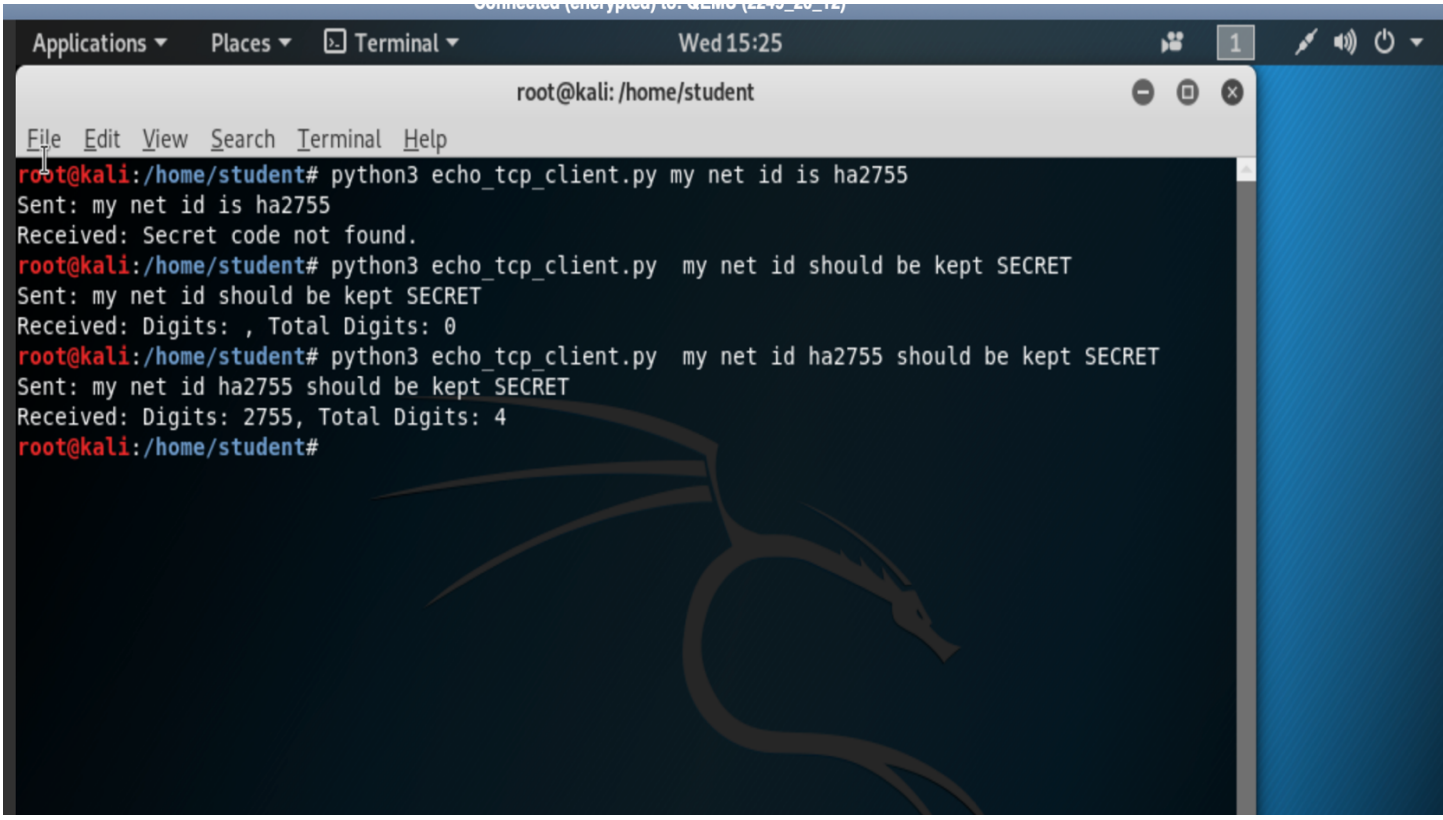
```
import socket

def handle_client(client_socket):
        file_data = b""
        while True:
                data = client_socket.recv(1024)
                if not data:
                        break
                file_data += data

        with open('received_file.txt', 'wb') as file:
                file.write(file_data)


def main():
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind(('10.10.10.2',5002))
        server.listen(1)

        while True:
                client, addr = server.accept()
                handle_client(client)

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

Open | tcp_file_transfer_client.py /home/student

```
import socket

def send_file(file_path, server_ip, server_port):
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client.connect((server_ip,server_port))

        with open(file_path,'rb') as file:
                file_data = file.read()
                client.sendall(file_data)

        client.shutdown(socket.SHUT_WR)
        client.close()

def main():
        server_ip = '10.10.10.2'
        server_port = 5002
        file_path = '/home/student/send.txt'

        send_file(file_path, server_ip, server_port)

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

Screenshots showing the behavior of Part 2.

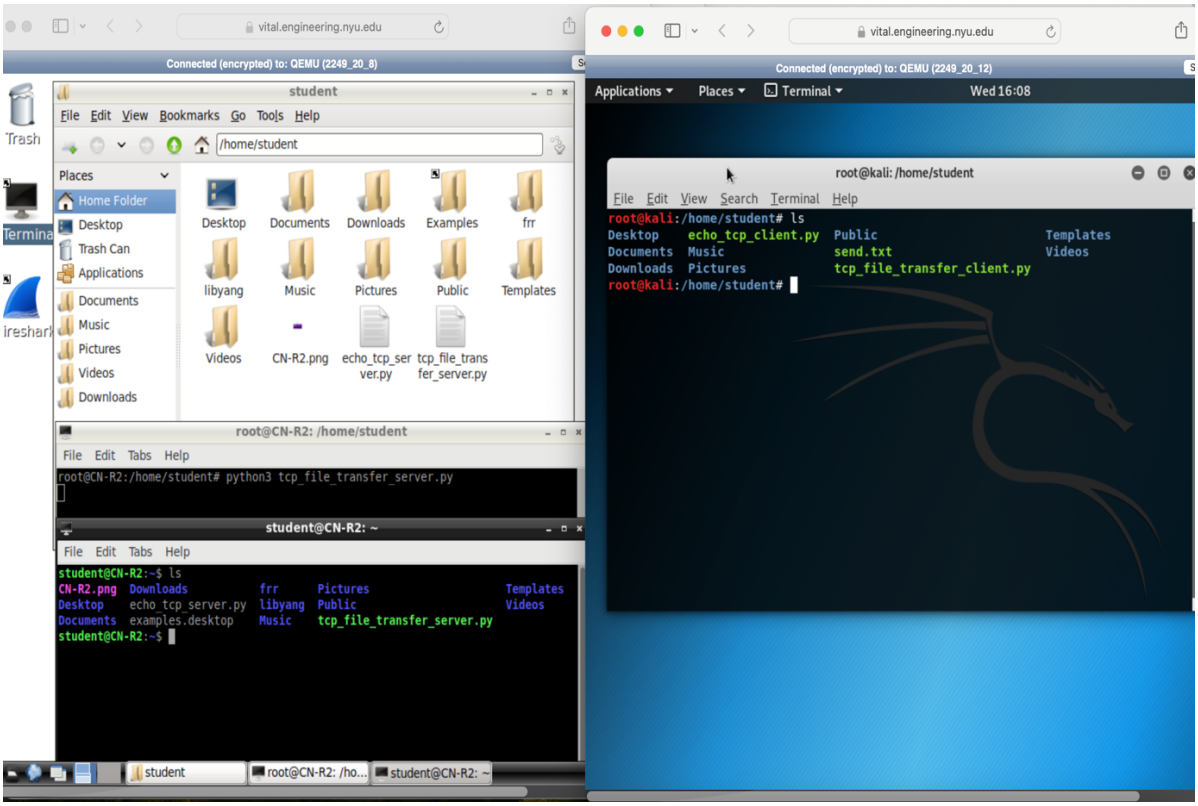
Make sure to include cases with and without the secret code.



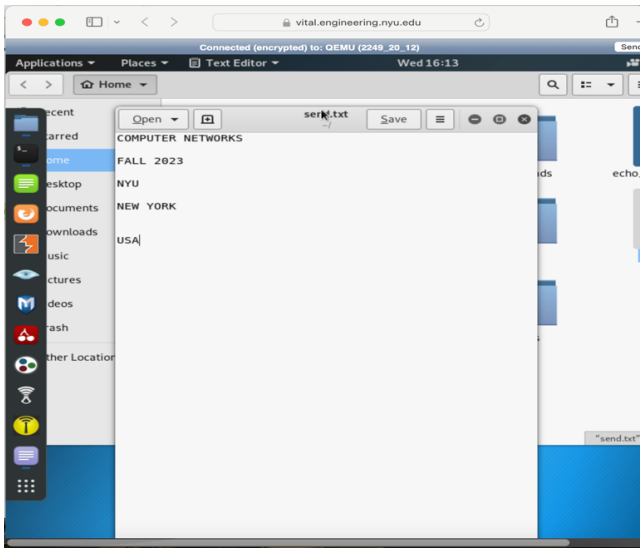
Screenshots showing the file transfer in Part 3:

show the original file on KALI, the KALI terminal after transferring, and the transferred file on R2.

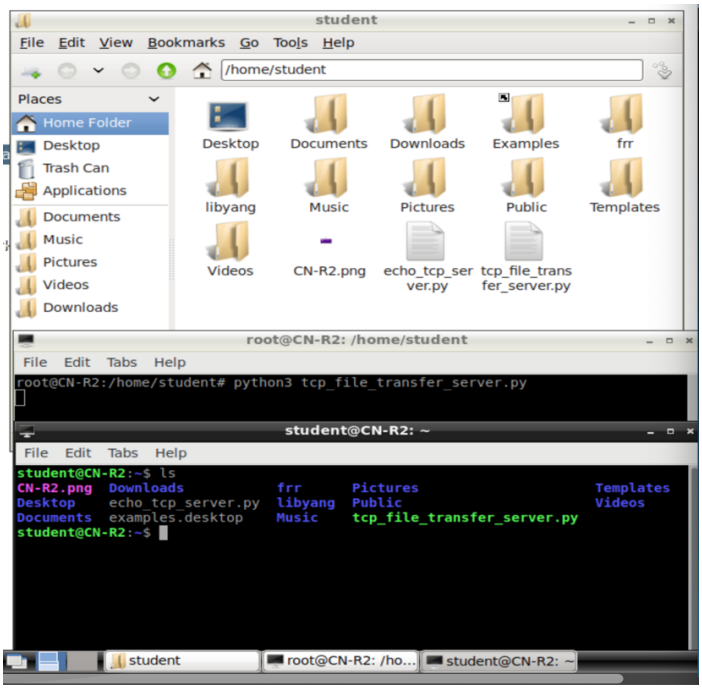
Before executing/transferring the file and you can see the file “send.txt” in kali:



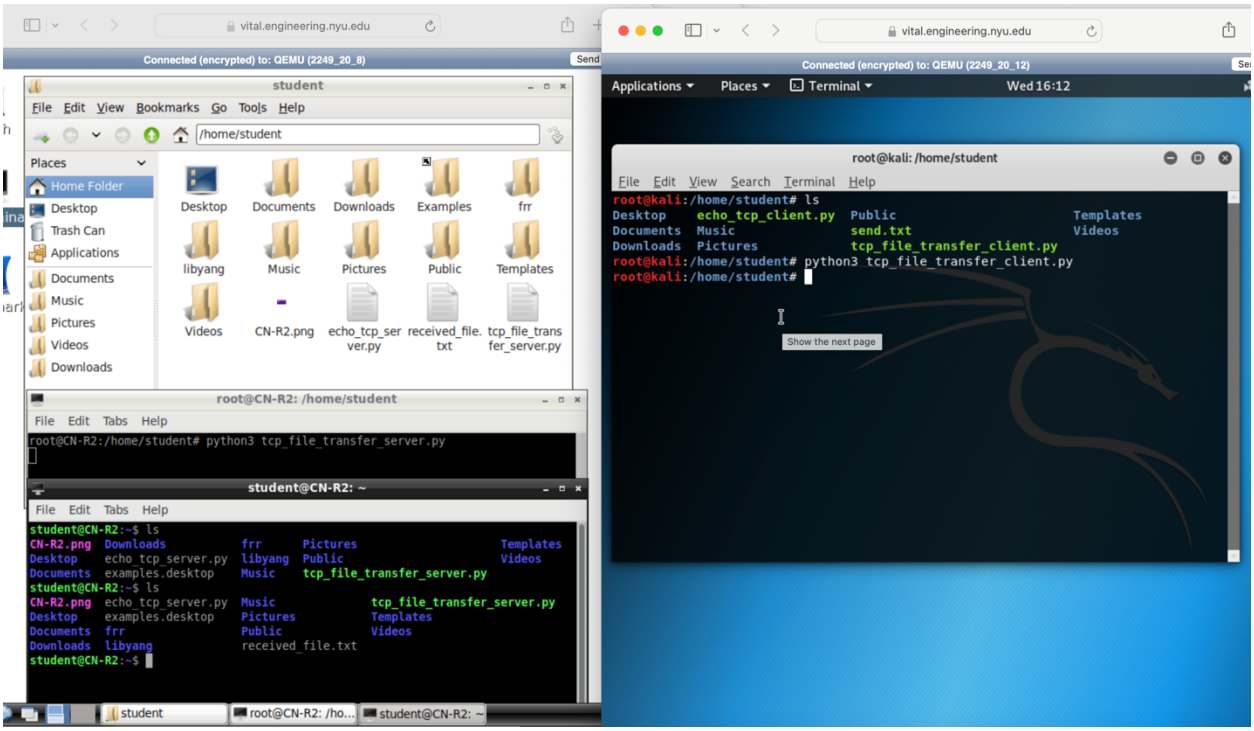
Send.txt file in Kali linux machine:



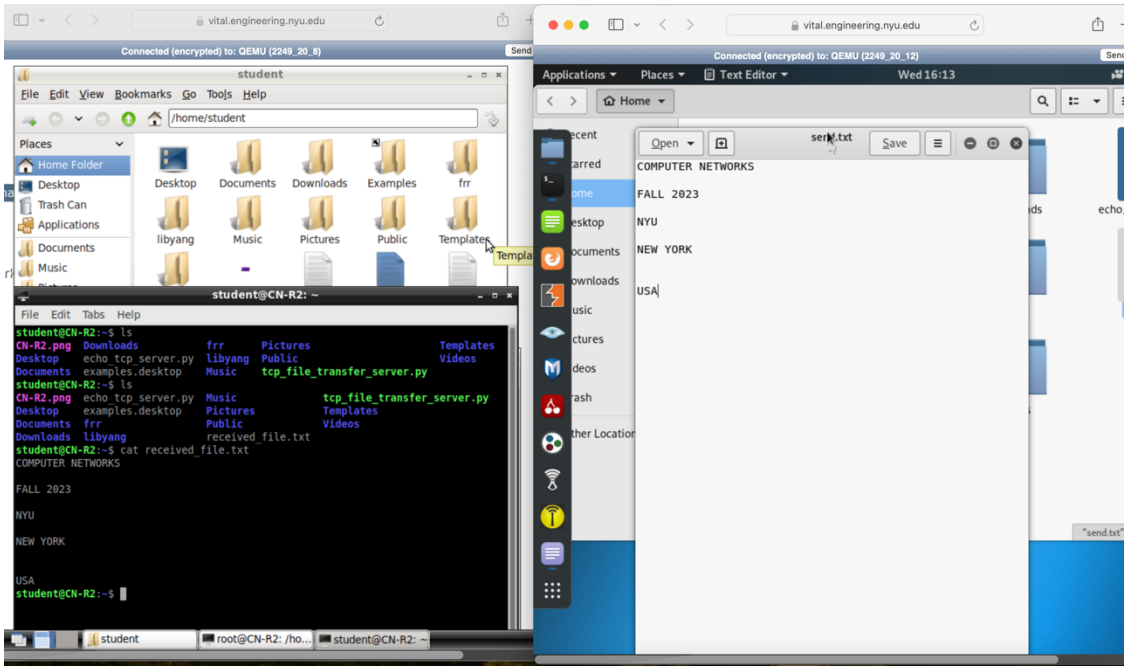
On R2:(executing the tcp_file_transfer_server.py)



Executing tcp_file_transfer_client.py on Kali and you can see that the file “received.txt” has been received on R2 from Kali



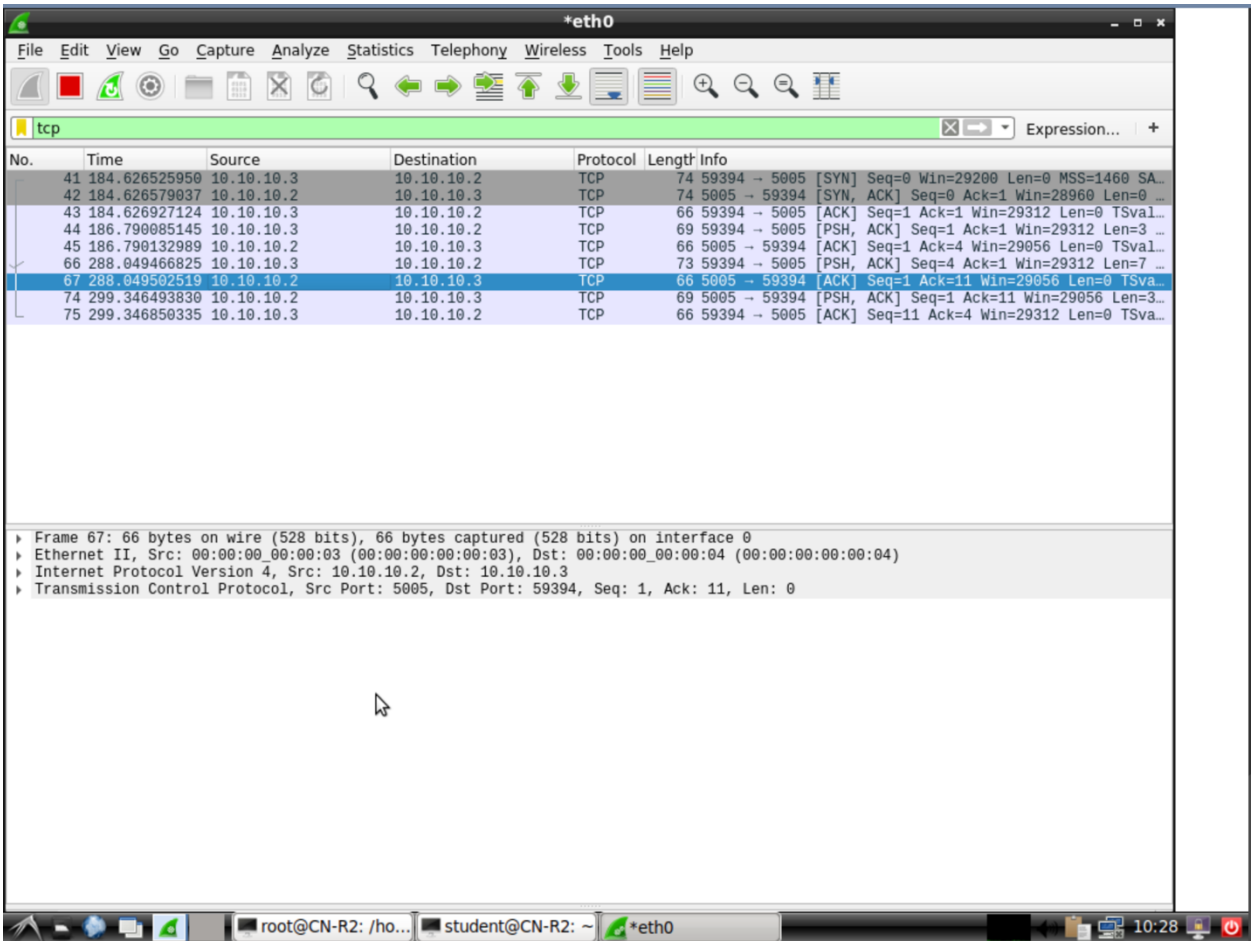
Showing files content on both the machines:



Part 4: Questions

a) In netcat, you specified the port on which the server should listen but did not specify the port the server should use to send a message to the client. Which client port does your netcat server send to? Use Wireshark to answer the question and include a screenshot.

To determine the port the netcat server sends messages to the client, we can use Wireshark to capture network traffic during the netcat chat. We can see that it's using **port number 59394 for TCP** and I have specified port number 5005.So, it's using both the ports for TCP connection.



b) Briefly explain your code from Part 2 and Part 3. In your explanation, focus not on the syntax but on the TCP communication establishment and flow.

Part 2 - TCP Echo Server (`echo_tcp_server.py`):

- `socket.socket(socket.AF_INET, socket.SOCK_STREAM)`: This line creates a TCP socket (`SOCK_STREAM`), specifying the IPv4 address family (`AF_INET`).
- `server.bind(('10.10.10.2', 5001))`: The server binds to the specified IP address (`10.10.10.2`) and port number (`5001`). This ensures that the server is reachable at a specific network location.
- `server.listen(1)`: The server listens for incoming connections, allowing up to 1 connection in the queue. This line prepares the server to accept client connections.
- `client, addr = server.accept()`: The server accepts an incoming connection from a client. This line is blocking until a connection is established.
- `handle_client(client)`: The `handle_client` function is called to process the client's request. This function is responsible for receiving data from the client, modifying it, and sending the response back.

TCP Echo Client (`echo_tcp_client.py`):

- `client.connect((server_ip, server_port))`: The client connects to the server using the R2 IP address (`10.10.10.2`) and port number (`5001`).
- `client.send(message.encode('utf-8'))`: The client sends the message to the server after encoding it to bytes.
- `client.recv(1024).decode('utf-8')`: The client receives the response from the server , decodes it from bytes to a string.
- `print(response)`: The client prints the received response from the server.
- `client.close()`: The client closes the socket after completing the communication.

TCP Communication Flow:

1. Server Setup: The server creates a TCP socket, binds to a specific address and port, and listens for incoming connections.
2. Client Setup: The client creates a TCP socket and connects to the server using the server's IP address and port number.
3. Connection Establishment: The server accepts the incoming connection, and a TCP connection is established.
4. Data Exchange: The client sends a message to the server. The server receives the message, modifies it ("R2: "), and sends the modified response back to the client.
5. Connection Closure: The server and client close their respective sockets.

Part 3 - TCP File Transfer Server (`tcp_file_transfer_server.py`):

- The server follows a similar structure to the TCP Echo Server from Part 2 but with modifications for file transfer.
- handle_client(client_socket): The function responsible for receiving file data from the client and writing it to a new file (`received_file.txt`).
- file_data += data: The server appends the received data to the `file_data` byte string until the entire file is received.
- with open('received_file.txt', 'wb') as file: The server opens a new file in binary write mode and writes the received file data to the file.

TCP File Transfer Client (`tcp_file_transfer_client.py`):

- The client sends a file to the server using the `send_file` function.
- with open(file_path, 'rb') as file: The client opens the file to be transferred in binary read mode and reads its content into `file_data`.
- client.sendall(file_data): The client sends the entire file data to the server using `sendall`. This ensures that all data is sent, even if it requires multiple calls to `send`.
- The client then closes its socket after sending the file.

TCP File Transfer Flow:

1. Server Setup: The server creates a TCP socket, binds to a specific address and port, and listens for incoming connections.

2. Client Setup: The client creates a TCP socket and connects to the server using the server's IP address and port number.

3. Connection Establishment: The server accepts the incoming connection, and a TCP connection is established.

4. File Transfer: The client reads the content of the file to be transferred (`send.txt`).
The client sends the entire file data to the server.

5. File Reception: The server receives the file data, appends it to a byte string (`file_data`), and writes it to a new file (`received_file.txt`).

6. Connection Closure: The server and client close their respective sockets.

c) What does the socket system call return?

The `socket` system call in Python returns a new socket object. This socket object represents an endpoint for communication and is used to send and receive data over the network. The returned socket object has methods and attributes for configuring and interacting with the socket.

d) What does the bind system call return? Who calls bind (client/server)?

The **bind** system call in Python is used to associate a socket with a specific network address (IP address and port number) on the local machine. It returns an error code or raises an exception if there's an issue.

Server Calls bind: The server calls **bind** to bind the socket to a specific address before listening for incoming connections. This ensures that the server is reachable at a specific IP address and port.

e) Suppose you wanted to send an urgent message from a remote client to a server as fast as possible. Would you use UDP or TCP? Why? (Hint: compare RTTs.)

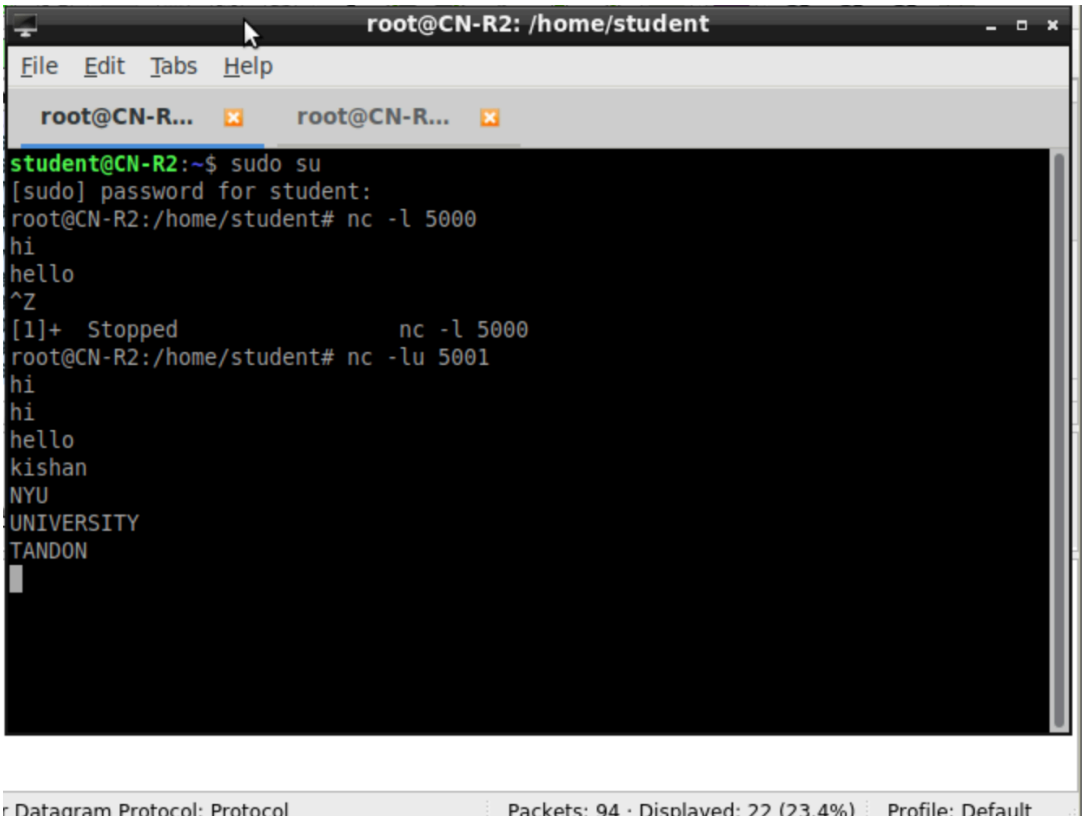
I would use **UDP** to send an urgent message from a remote client to a server as fast as possible.

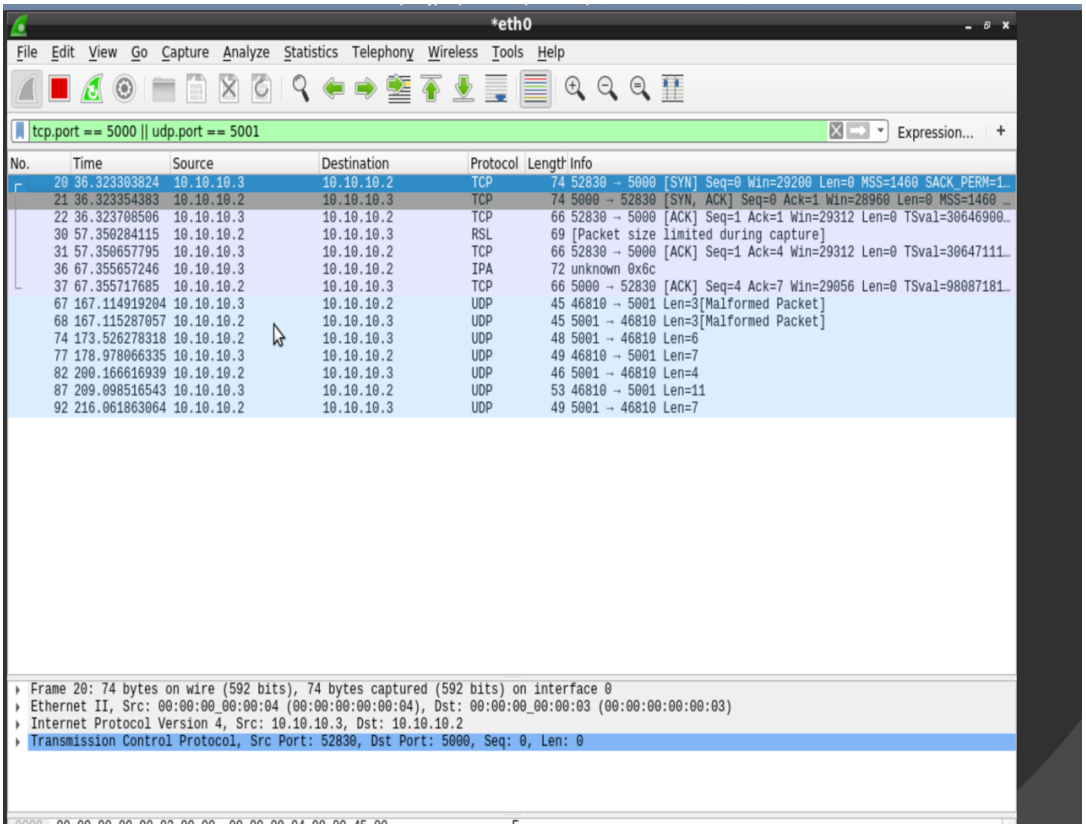
Reason:
UDP is connectionless and has lower overhead compared to TCP.TCP involves a three-way handshake, introducing additional Round-Trip Times (RTTs), which could impact latency.

TCP RTT Overhead:
TCP involves a handshake (SYN, SYN-ACK, ACK) before data exchange. This handshake introduces additional RTTs. In contrast, UDP does not have a handshake, making it quicker to send data.

Here's an example where I used **nc -u** for UDP and **nc** for TCP on R2.

We can see that TCP connection involves Threeway handshake whereas UDP doesn't involve any handshake and hence avoids Round Trip Time. We can see that in the below wire shark screenshots.


root@CN-R2: /home/student
File Edit Tabs Help
root@CN-R...
root@CN-R...
student@CN-R2:~$ sudo su
[sudo] password for student:
root@CN-R2:/home/student# nc -l 5000
hi
hello
^Z
[1]+ Stopped nc -l 5000
root@CN-R2:/home/student# nc -lu 5001
hi
hi
hello
kishan
NYU
UNIVERSITY
TANDON
Datagram Protocol: Protocol
Packets: 94 · Displayed: 22 (23.4%)
Profile: Default


*eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.port == 5000 || udp.port == 5001
Expression...
No. Time Source Destination Protocol Length Info
20 36.323303824 10.10.10.3 10.10.10.2 TCP 74 52830 → 5000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1…
21 36.323354383 10.10.10.2 10.10.10.3 TCP 74 5000 → 52830 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 …
22 36.323708506 10.10.10.3 10.10.10.2 TCP 66 52830 → 5000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=30646900…
30 57.350284115 10.10.10.2 10.10.10.3 RSL 69 [Packet size limited during capture]
31 57.350657795 10.10.10.3 10.10.10.2 TCP 66 52830 → 5000 [ACK] Seq=1 Ack=4 Win=29312 Len=0 TSval=30647111…
36 67.355657246 10.10.10.3 10.10.10.2 IPA 72 unknown 0x6c
37 67.355717685 10.10.10.2 10.10.10.3 TCP 66 5000 → 52830 [ACK] Seq=4 Ack=7 Win=29056 Len=0 TSval=98087181…
67 167.114919204 10.10.10.3 10.10.10.2 UDP 45 46810 → 5001 Len=3[Malformed Packet]
68 167.115287057 10.10.10.2 10.10.10.3 UDP 45 5001 → 46810 Len=3[Malformed Packet]
74 173.526278318 10.10.10.2 10.10.10.3 UDP 48 5001 → 46810 Len=6
77 178.978066335 10.10.10.3 10.10.10.2 UDP 49 46810 → 5001 Len=7
82 200.166616939 10.10.10.2 10.10.10.3 UDP 46 5001 → 46810 Len=4
87 209.098516543 10.10.10.3 10.10.10.2 UDP 53 46810 → 5001 Len=11
92 216.061863064 10.10.10.2 10.10.10.3 UDP 49 5001 → 46810 Len=7
Frame 20: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
Internet Protocol Version 4, Src: 10.10.10.3, Dst: 10.10.10.2
Transmission Control Protocol, Src Port: 52830, Dst Port: 5000, Seq: 0, Len: 0

f) What is Nagle's algorithm? What problem does it aim to solve and how?

We use Nagle's algorithm in TCP to improve network efficiency. It solves small packet problem which occurs when applications send small amounts of data frequently, resulting in inefficient use of network resources.

Small Packet Problem:
In network communication, sending a large number of small packets can lead to inefficient use of network resources and increased overhead. Nagle's algorithm addresses the "Small Packet Problem" by attempting to group small pieces of data into larger packets.

How it solves:

Delaying Transmission:
Nagle's algorithm introduces a small delay, often measured in milliseconds, before sending small packets. During this delay, the sender collects additional small pieces of data to form a more substantial payload.

Combining Data:
While waiting for the delay period to expire, Nagle's algorithm accumulates small pieces of data into a buffer. If the delay period elapses or the size of the data reaches a predefined threshold (Maximum Segment Size, MSS), the accumulated data is sent as a single, larger packet.

Reducing Overhead:
By sending larger packets, Nagle's algorithm helps reduce the overhead associated with individual packet headers and acknowledgments. This is particularly beneficial in scenarios where frequent small updates or commands are sent, such as interactive applications.

g) Explain one potential scenario in which delayed ACK could be problematic.

Delayed ACK (Acknowledgment) can be problematic in scenarios requiring low-latency communication.

Scenario: Video call/Conference

During video conference, delayed ACK can undermine the seamless flow of conversations. Participants in video conferencing expect natural interactions and synchronized audio/video streams. Delayed acknowledgments can disrupt this synchronization, leading to lip-sync issues and degraded overall quality. Additionally, features like collaborative document sharing or interactive whiteboards may experience reduced responsiveness, affecting participants' ability to engage effectively. The sense of presence and connection is compromised as feedback from gestures or reactions is delayed, diminishing the immediacy and engagement of the virtual meeting. In both scenarios, minimizing acknowledgment delays is crucial to providing users with the responsive and immersive experiences they expect.

Another Scenario: Interactive Gaming

In online gaming, low latency is crucial for a responsive and real-time gaming experience. Each player's action, such as moving, shooting, or interacting with the environment, requires quick acknowledgment from the server to provide timely feedback to the player.

In a scenario where delayed ACK is employed, the server might intentionally delay sending acknowledgments to wait for additional data to combine into a single ACK packet. This delay could introduce unnecessary latency in acknowledging critical actions from players, leading to a perceived lag in the game. Gamers may experience delays in their actions being reflected in the game environment, negatively impacting the overall gaming experience.

In such real-time and interactive applications, where immediate acknowledgment is essential, delayed ACK can be problematic and may result in a less responsive and less enjoyable user experience.

So, we should disable or avoid using Nagle’s algorithm in these cases.