Design & Implement your protocol (40 points):

Socket Type:

Socket Type: SOCK_DGRAM (UDP)
UDP sends packets which are delivered independently, with no built in flow or congestion control. It allows our protocol to be able to implement all the features below.

Packet Structure:

The total size of the checksum is 16 bytes + data.
For header- 4 bytes for sequence number of the packet, 4 bytes for acknowledgment number, 2 bytes for bit flags, 2 bytes for receiver window size.

Packet Flags - SYN (synchronize), ACK (acknowledge), FIN (finish)

Checksum is 4 bytes using MD5.
How the checksum works - MD5 over header and data, and then take the first 4 bytes.

Serialize method - Convert byte object into byte sequence
Deserialize method - Check length of raw packet is at least 16 byte, extract header from the raw packet and deserialize using struct.unpack. Build the packet again with all the deserialized data and check if the checksum matches.

Transport Layer:

The main purpose of this class is to provide a **connection-oriented, reliable transport layer.** It maintains state, sequence numbers, ACK tacking and basic congestion control parameters (cwnd, ssthresh).

It provide a 3 way handshake:
Client -> Server: SYN
Server -> Client: SYN + ACK
Client -> Server: ACK

This class wraps an UnreliableChannel, which simulates Packet loss and Packet corruption.

Client-Side Handshake: connect(addr)

1) Client constructs a packet with an initial sequence number and SYN flag, which is sent through the unreliable channel. (Client -> Server SYN)
2) Client wait to receive a reply packet which must contain SYN and ACK, with a valid checksum (Client <- Server SYN + ACK)

3) Send a final ACK and complete the connection (Client -> Server: Final ACK)
4) Marks connection state as ESTABLISHED and resets the socket timeout

Server-Side Handshake: accept()
The run_receiver code uses ReliableTransport to
1) Listen for incoming packet
2) Detect incoming SYN from client
3) Respond with a SYN + ACK packet
4) Accept final ACK from the client before marking the connection as ESTABLISHED

Sender:

The GBNSender class uses Go-Back-N data transfer protocol, using the transport layer. It handles sending of data, receiving ACKs, flow control, congestion control and retransmissions.

Sending Data: send_data()
1) Splits the data into chunks of 1024 bytes
2) Sends each chunk as a packet through the channel
3) Ensure that the number of packets in delivery does not exceed minimum(congestion window, receiver window)
4) Starts timer for the first unacknowledged packet
5) Waits until all packets are acknowledged before returning

Receiving ACKs: (_receive_loop and receive_ack)
1) Continuously listens for incoming ACK packets
2) Cumulative ACKs: Updates *base* to the highest sequence number
3) Updates receiver window for flow control
4) Congestion control: Slow start (cwnd grows exponentially until it reaches ssthresh) and Congestion avoidance (cwnd grows linearly after ssthresh)

Retransmission Timer: (_start_timer, _stop_timer, _on_timeout)
1) Timer triggers retransmission of all unacknowledged packets if no ACKs are received within a timeout
2) On timeout, ssthresh is halved, cwnd resets to 1 and all unacknowledged packets are retransmitted.

Receiver:

This class implements the receiver side of GBN data transfer protocol. It handles receiving packets, enforcing in-order delivery, send cumulative ACKs, and implement flow control.

Handle Incoming Packets (handle_packet)

1) Checks if the incoming packet's sequence number matches the expected sequence number:
    a) If in order:
        i) Accepts the packet and increments expected
        ii) Sends cumulative ACK with updated expected value to sender
        iii) Returns the packet data for processing
    b) If out of order:
        i) Discard the packet
        ii) Resend the last ACK for the highest in-order packet received, so sender retransmit the missing packets

Flow Control:

1) *advertised_window* limits the sender's allowed in-flight packets
2) ACK packets includes the window size so the sender can adjust the sending rate

Run_sender:

This class acts as the client in the protocol. It uses ReliableTransport and GBNSender to connect to the server, send data and handle acknowledgements.

Key Functions:

1) Creates socket and initializes a ReliableTransport instance, with a configurable loss rate to adjust the extent of network unreliability.
2) Call ReliableTransport.connect(addr) to perform a 3 way handshake with the server.
3) If connection is established, it will create a GBNSender instance, which starts a thread that listen for incoming ACKs
4) Data Transmission (send_data) sends a large dummy payload (50000 bytes of 'A') in chunks of 1024 bytes using GBN.

Run_receiver:

This class acts as the server in the protocol. It uses ReliableTransport and Receiver to accept client connections, receive data and send cumulative ACKs.

Key Steps:

1) Creates a UDP socket to a fixed local host and initialize a ReliableTransport instance and a Receiver instance
2) Calls *ReliableTransport.accept()* to perform the 3 way handshake. The procedure is as mentioned in the transport layer
3) Uses *Receiver.handle_packet()* to accept in order packets, send cumulative ACKs reflecting the highest sequence number received in order, and also implement flow control via the advertised window

Unreliable Channel (Simulation of Loss):

This class simulates an unreliable network channel. It is used to test the reliability mechanisms of the transport layer, as well as the sender and receiver. It introduces packet loss and data corruption as the means of testing.

Sending Packets (send):

1) There is a probability of *loss_rate* that a packet is dropped
2) There is a probability of *corrupt_rate* that a packet is corrupted by flipping a random byte

Data Corruption (_corrupt_data):

1) Function to convert the packet to bytearray and flip a random byte with XOR, which is then returned

Analyze Traffic (20 points):

Test 1: Stress Test (Reliability and Congestion Control)

This test is to prove that the protocol is able to work reliably under sub-optimal conditions (2% loss), being able to establish a connection using a 3-way handshake, correctly implementing sequence numbers for retransmission, and dynamically adjust the congestion window (slow start and congestion avoidance.

Parameters used:

- Sender Loss (run_sender.py) = 2%
- Receiver Loss (run_receiver.py) = 0%
- Corruption (channel.py) = 0%
- Window (receiver.py) = 64

Analysis of results

Successful 3-Way Handshake - As shown in the figure below, the protocol is able to establish a connection. The initial packet was dropped, and the client's timer triggered a timeout. The protocol automatically retransmitted the SYN packet, and successfully established the connection.

```
Connecting...
Sending SYN (Attempt 1)...
[LOSS] Packet dropped
Handshake timed out, retrying...
Sending SYN (Attempt 2)...
Connection Established!
```

Congestion Control (Slow Start) - As shown in the figure below, the slow start phase starts immediately after the connection is established. The congestion window (CWND) increases exponentially by 1 MSS for every ACK.

```
Sender listening for ACKs...
ACK 1 received. CWND: 2.00, RWND: 64
ACK 2 received. CWND: 3.00, RWND: 64
ACK 3 received. CWND: 4.00, RWND: 64
ACK 4 received. CWND: 5.00, RWND: 64
ACK 5 received. CWND: 6.00, RWND: 64
ACK 6 received. CWND: 7.00, RWND: 64
ACK 7 received. CWND: 8.00, RWND: 64
[LOSS] Packet dropped
ACK 8 received. CWND: 9.00, RWND: 64
ACK 9 received. CWND: 10.00, RWND: 64
ACK 10 received. CWND: 11.00, RWND: 64
ACK 11 received. CWND: 12.00, RWND: 64
ACK 12 received. CWND: 13.00, RWND: 64
ACK 13 received. CWND: 14.00, RWND: 64
ACK 14 received. CWND: 15.00, RWND: 64
```

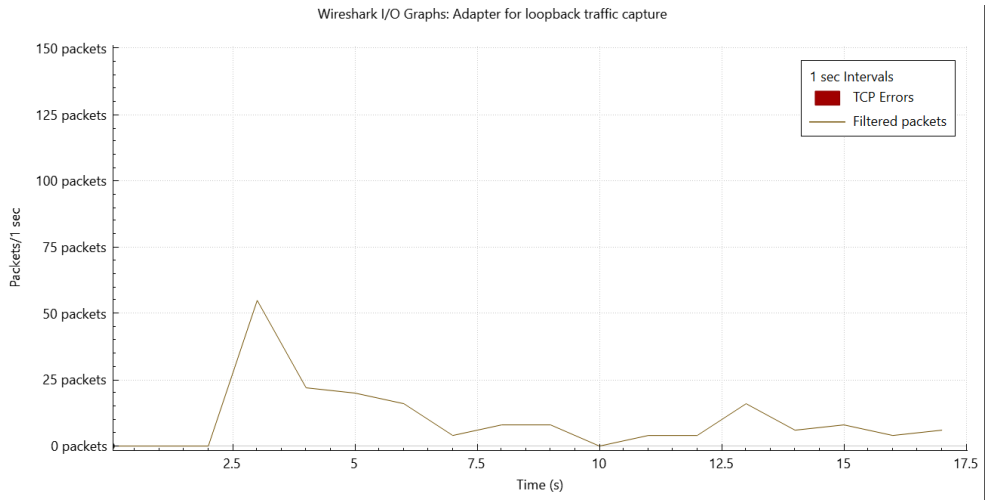
Timeout and retransmission - The figure below shows when packet 14 was lost, the sender stopped receiving ACKs for this sequence number, causing the retransmission timer to timeout. The sender managed to identify the unacknowledged data and retransmitted the lost packet.

```
ACK 14 received. CWND: 15.00, RWND: 64
[LOSS] Packet dropped
[LOSS] Packet dropped
[TIMEOUT] Loss detected.
Congestion Control: Dropped CWND to 1, SSTHRESH to 7
[LOSS] Packet dropped
[LOSS] Packet dropped
[LOSS] Packet dropped
[LOSS] Packet dropped
ACK 15 received. CWND: 2.00, RWND: 64
```

Congestion Avoidance - The figure below shows the transition to congestion avoidance. After a loss, the *ssthresh* was set to 2. When after the window reaches this threshold, the growth switches from exponential to linear growth.

```
Congestion Control: Dropped CWND to 1, SSTHRESH to 2
[LOSS] Packet dropped
[LOSS] Packet dropped
ACK 18 received. CWND: 2.00, RWND: 64
ACK 19 received. CWND: 2.50, RWND: 64
ACK 20 received. CWND: 2.90, RWND: 64
ACK 21 received. CWND: 3.24, RWND: 64
ACK 22 received. CWND: 3.55, RWND: 64
ACK 23 received. CWND: 3.83, RWND: 64
ACK 24 received. CWND: 4.10, RWND: 64
[TIMEOUT] Loss detected.
```

Wireshark I/O Graph - The Wireshark I/O graph confirms the output from the logs (Filtered UDP Port == 9001). The spike on the left corresponds to the slow start phase where the throughput peaked. The drop corresponds to the packet loss event. The valley confirms the 1 second timeout wait. The recovery is shown by the smaller climbs which represent the start of the transmission loops.



Test 2: Flow Control

This test is to prove that the flow control correctly prevents the receiver from overflowing the receivers buffer. The sender must be able to limit the packets in flight according to the receiver's advertised window (RWND) regardless of the network's capacity (CWND).

Parameters used:

- Sender Loss (run_sender.py) = 0%
- Receiver Loss (run_receiver.py) = 0%
- Corruption (channel.py) = 0%
- Window (receiver.py) = 2

Analysis of results

Logs - The figure below shows that the flow control is working as expected. The advertised window (RWND) consistently advertises a window of 2 packets. Since there are no packet losses in this case, the congestion window (CWND) increases normally (exponentially) up to the set threshold (32). Despite the CWND allowing much more packets, the sender is forced to limit its transaction to **min(CWND, RWND) = 2** packets in flight. This proves that the sender follows the receivers buffer limits.

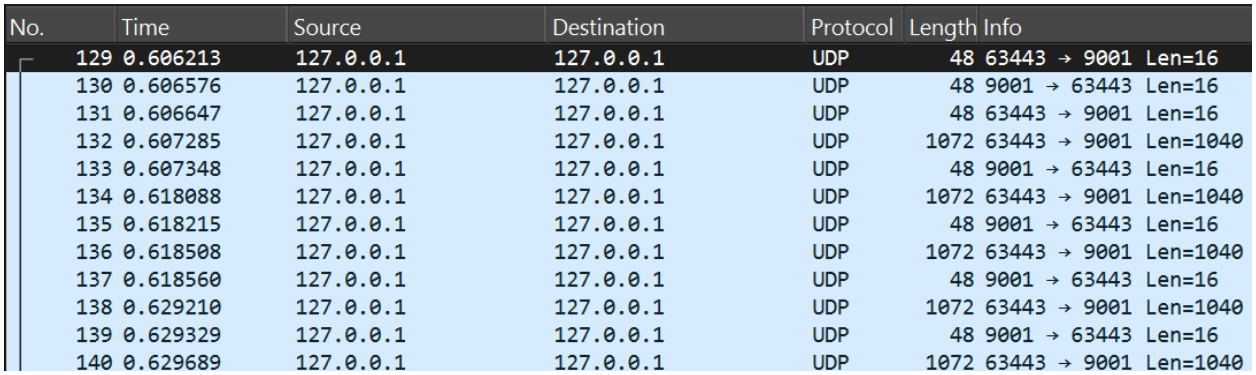
```
Sending...
Sender listening for ACKs...
ACK 1 received. CWND: 2.00, RWND: 2
ACK 2 received. CWND: 3.00, RWND: 2
ACK 3 received. CWND: 4.00, RWND: 2
ACK 4 received. CWND: 5.00, RWND: 2
ACK 5 received. CWND: 6.00, RWND: 2
```

```
ACK 29 received. CWND: 30.00, RWND: 2
ACK 30 received. CWND: 31.00, RWND: 2
ACK 31 received. CWND: 32.00, RWND: 2
ACK 32 received. CWND: 32.03, RWND: 2
ACK 33 received. CWND: 32.06, RWND: 2
ACK 34 received. CWND: 32.09, RWND: 2
```

Wireshark Results - The figure below shows the wireshark trace from running the programs (Filtered UDP Port == 9001). Packet 132 marks the start of the data transmission. The sender initially transmits only one packet since the CWND initializes at 1 during slow start, making the effective window **min(CWND = 1, RWND = 2) = 1**.

As the connection stabilizes and CWND grows, the transmission becomes constrained only by the receiver window (RWND = 2). The trace confirms this limit as we can see the pattern going Data → ACK → Data. This shows that the sender fills up the limit of 2,

and waits for an ACK before sending the next packet. This proves that it never exceeds the receivers buffer limit.



No.	Time	Source	Destination	Protocol	Length	Info
129	0.606213	127.0.0.1	127.0.0.1	UDP	48	63443 → 9001 Len=16
130	0.606576	127.0.0.1	127.0.0.1	UDP	48	9001 → 63443 Len=16
131	0.606647	127.0.0.1	127.0.0.1	UDP	48	63443 → 9001 Len=16
132	0.607285	127.0.0.1	127.0.0.1	UDP	1072	63443 → 9001 Len=1040
133	0.607348	127.0.0.1	127.0.0.1	UDP	48	9001 → 63443 Len=16
134	0.618088	127.0.0.1	127.0.0.1	UDP	1072	63443 → 9001 Len=1040
135	0.618215	127.0.0.1	127.0.0.1	UDP	48	9001 → 63443 Len=16
136	0.618508	127.0.0.1	127.0.0.1	UDP	1072	63443 → 9001 Len=1040
137	0.618560	127.0.0.1	127.0.0.1	UDP	48	9001 → 63443 Len=16
138	0.629210	127.0.0.1	127.0.0.1	UDP	1072	63443 → 9001 Len=1040
139	0.629329	127.0.0.1	127.0.0.1	UDP	48	9001 → 63443 Len=16
140	0.629689	127.0.0.1	127.0.0.1	UDP	1072	63443 → 9001 Len=1040

Test 3: Handshake Test

This test is to prove that the 3-way handshake can handle losses without crashing.

Parameters used:

- Sender Loss (run_sender.py) = 10%
- Receiver Loss (run_receiver.py) = 10%
- Corruption (channel.py) = 0%
- Window (receiver.py) = 64

Analysis of results:
The screenshot of the logs confirms that the connection was established successfully on the 4th attempt. Packet 197 (t = 4.64s) represents attempt 3 of the handshake. This packet was transmitted, but no acknowledgement was received. The trace shows a silence of 1.01 seconds between packet 197 and 199 (t = 5.65s). This confirms that the sender correctly waited 1 second before retransmitting. Right after the timeout, the Server responds with a SYN-ACK (packet 199). This response indicates that the client successfully transmitted attempt 4 (packet 198), which was received and acknowledged by the server, completing the connection.

While the transmitted SYN packet (attempt 4) was not visible in the filtered view (Filtered UDP Port == 9001), the appearance of the SYN-ACK (packet 199) immediately after the timeout confirms that attempt 4 was successfully transmitted by the client.

```
Connecting...
Sending SYN (Attempt 1)...
[LOSS] Packet dropped
Handshake timed out, retrying...
Sending SYN (Attempt 2)...
[LOSS] Packet dropped
Handshake timed out, retrying...
Sending SYN (Attempt 3)...
Handshake timed out, retrying...
Sending SYN (Attempt 4)...
Connection Established!
Sending...
```

No.	Time	Source	Destination	Protocol	Length	Info
197	4.644509	127.0.0.1	127.0.0.1	UDP	48	61965 → 9001 Len=16
199	5.655360	127.0.0.1	127.0.0.1	UDP	48	9001 → 61965 Len=16
200	5.657697	127.0.0.1	127.0.0.1	UDP	48	61965 → 9001 Len=16
201	5.657924	127.0.0.1	127.0.0.1	UDP	48	61965 → 9001 Len=16
202	5.660640	127.0.0.1	127.0.0.1	UDP	1072	61965 → 9001 Len=1040

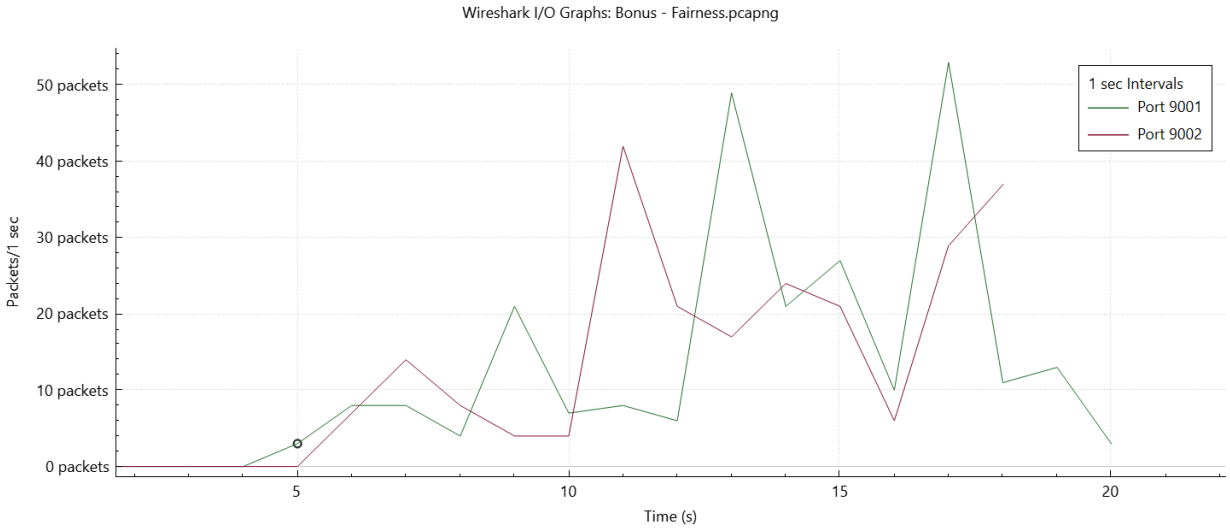
BONUS SECTION - Prove Fairness (5 Points)

To demonstrate fairness, when multiple instances of the protocol compete for resources, they should share bandwidth equally, not allowing one flow to starve the other.

To test this, we duplicated the sender and receiver programs to create two distinct flows:

- Flow 1: Port 9001
- Flow 2: Port 9002
- Parameters:
  - Sender Loss = 10%
  - Receiver Loss = 10%
  - Window = 64
- Data: The transmission size was increased from 50,000 bytes to ensure that the transfer lasted long enough.

The graph below shows that when both sets of the programs run simultaneously (between t = 7s and t = 17s), they show similar peaks, and take turns using up resources. When one set increases its window enough to cause a loss event (congestion), it reduces its rate, allowing the other set to increase its rate. This proves that the protocol prevents either set to monopolize the link, or starve. Hence, the protocol distributes the bandwidth equally, and is thus, fair.


Wireshark I/O Graphs: Bonus - Fairness.pcapng
50 packets
40 packets
30 packets
20 packets
10 packets
0 packets
Packets/1 sec
5
10
15
20
Time (s)
1 sec Intervals
Port 9001
Port 9002