

Pipelined Reliable Transfer Protocol

**By:
Kandisa Agarwal and Puja Shah**

TABLE OF CONTENTS

Introduction.....	2
Design and Implementation.....	2
Implementation Specifics.....	8
Interesting Modifications to Explore:.....	8
Simulating Errors.....	9
Case for packet loss.....	9
Case for corrupted packet.....	10
Traffic Analysis.....	10
Three Way Handshake.....	10
Flow Control:.....	12
Connection Termination:.....	13
Congestion Control Visual Representation.....	14

Introduction

Reliable data transmission is crucial for applications that require guaranteed delivery, such as web browsing, file transfers, and real-time communications. Protocols like Transmission Control Protocol (TCP) and QUIC serve as the foundation of the Internet's reliable data transfer, providing essential services including flow control, congestion control, and error recovery. In this project, we aim to design and implement a connection-oriented, reliable transport protocol that replicates key features of TCP, specifically focusing on flow control and congestion control. To ensure the protocol is robust in real-world conditions, we will also simulate packet loss and errors during testing.

Design and Implementation

The initial phase of the project involved defining mechanisms to effectively manage flow control and congestion control over a socket type that doesn't natively offer these features. While TCP sockets provide reliability, we chose to build our protocol on top of UDP due to its simplicity and flexibility. UDP's minimalistic design allowed us the freedom to tailor our protocol according to our specific requirements, which included implementing custom mechanisms for reliability, flow control, and congestion control.

Before proceeding with the implementation, we identified the need for a custom header capable of storing all the necessary information for our protocol. The most straightforward way to structure this was to create a class for the header. Our custom header, named `ReliableTransportLayerProtocolHeader`, encapsulates all relevant fields required for our protocol's operation, such as sequence numbers, acknowledgment numbers, window size, and checksum, among others.

Source Port		Destination Port			
Sequence Number					
ACK Number					
MSS	Checksum	Sending Window	SIN	ACK	FIN
App Data					

We found this information to be sufficient to implement our mechanisms for flow control and congestion control moving forward.

The first step toward ensuring reliability was establishing a secure connection. After careful consideration, we opted for the traditional 3-way handshake process for connection establishment, as it is widely recognized for its reliability and security in TCP connections. **Figure 1.0** illustrates the steps involved in the handshake process.

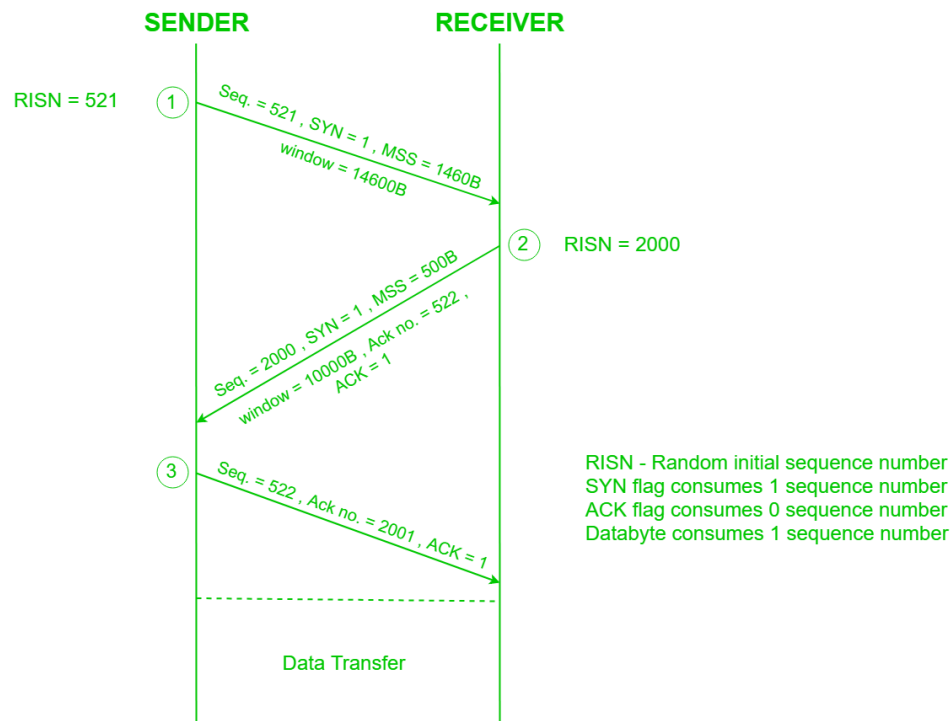


Figure 1.0 3-way Handshake

Once we were satisfied with the connection establishment, we turned our attention to implementing a pipelining protocol to handle flow control. We chose the Go-Back-N protocol, which is simple yet

effective. In this protocol, the sender operates with an advertised window size that limits the number of packets in flight at any given time. On the receiving end, the receiver processes data sequentially, discarding any out-of-order or corrupted packets. A simple illustration of this process is shown below in **Figure 2.0:**

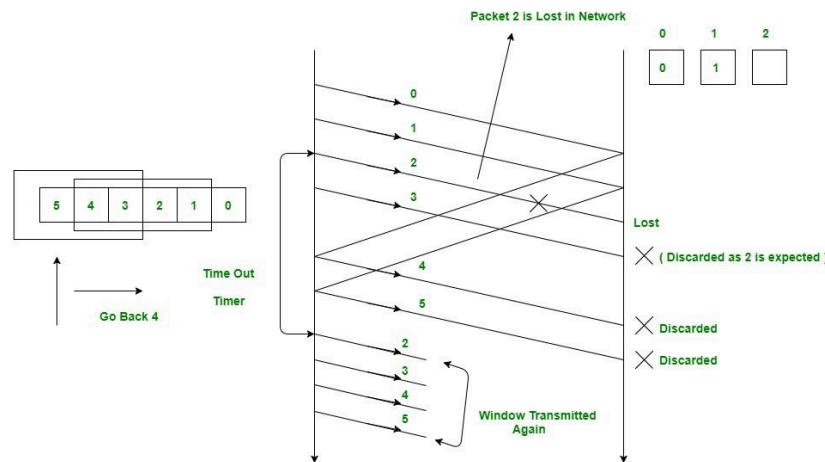


Figure 2.0 Go-Back-N

In our implementation of the Go-Back-N protocol, the receiver drops all corrupted and out-of-order packets. The sender infers that a packet has been lost or corrupted when the base packet in its window times out. Upon detecting this, the sender retransmits all the packets in the window. We found this pipelining method to be effective, as it minimizes overhead on the receiver side while maintaining reliability. Therefore, it seemed like a solid choice for handling flow control.

Lastly, we incorporated congestion control mechanisms to improve the performance of our protocol under higher traffic conditions. We found the TCP Tahoe implementation to be elegant and decided to adopt it for our congestion control. This mechanism uses a congestion window size (cwnd) that adjusts dynamically based on network traffic. The window size increases exponentially during the Slow Start phase until a predefined threshold is reached, after which it grows linearly during the Congestion Avoidance phase. In the event of packet loss, the congestion window is reduced to 1. Our implementation initializes the threshold with half of the current window size, aligning with the behavior of TCP Tahoe (**Figure 3.0**)

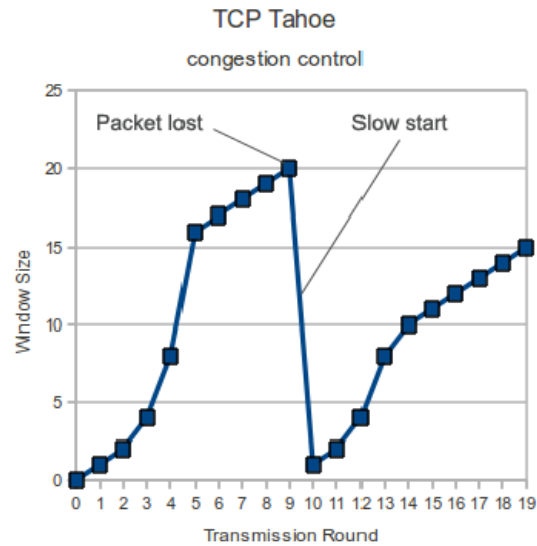


Figure 3.0 TCP Tahoe Sample

Results of the cwnd size changes from our implementation are on Page 14. You can generate similar graphs by simply altering the `LOSS_PROBABILITY` value in our `sender.py` file to be your desired value. With loss probability set to 0, you will get a graph identical to what we have. In other cases, there will be minor differences.

Once the data transmission is complete, we gracefully terminate the connection. The sender initiates the termination by sending a message with the FIN bit set. The receiver responds by acknowledging this with both the FIN and ACK bits set. Upon receiving this response, the sender sends a final ACK to confirm the termination before shutting down. This final ACK is the last message the receiver accepts before closing the connection.

A brief visual representation of this FIN-ACK interaction is shown in **Figure 4.0**

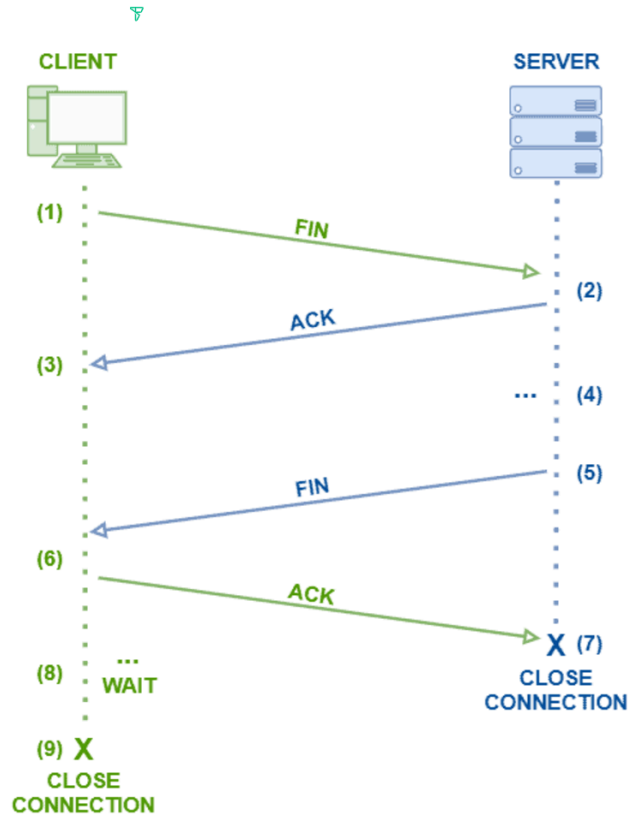


Figure 4.0 FIN Exchange : Terminate Connection

While our implementation closely follows this process, it differs slightly in the termination phase. Instead of the receiver sending two separate messages (one with ACK and another with FIN), we combine both the ACK and FIN bits into a single FIN-ACK message.

To summarize, here are our chosen methodologies

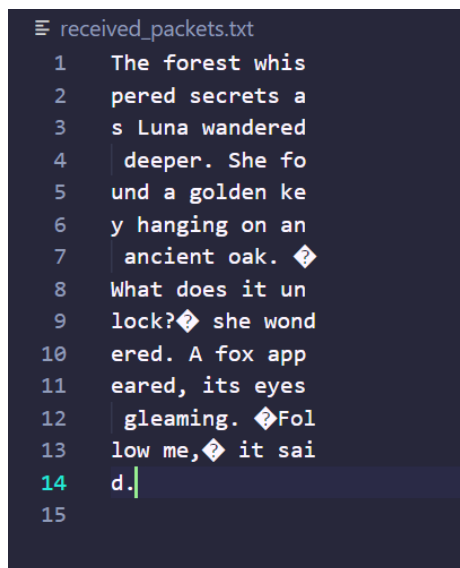
Behavior	Mechanism	Reason
Connection Establishment	Three Way Handshake	Highly secure
Flow Control	Go-Back-N Pipelining	Low overhead and simple to implement
Congestion Control	TCP Tahoe	Elegant and efficient
Termination	FIN-ACK	Secure and efficient

To verify that our protocol functions correctly—specifically, that it **a)** transmits all packets and **b)** processes them in order—we decided to send a piece of text from the sender side, broken into chunks of 15 bytes per packet. Our equivalent of processing a packet and delivering it to the application layer was writing the contents to a file. By the end of the interaction, we expected the file on the receiver's side to be identical in both content and order to the original sender data.

We used the following text as our sender data:

"The forest whispered secrets as Luna wandered deeper. She found a golden key hanging on an ancient oak. 'What does it unlock?' she wondered. A fox appeared, its eyes gleaming. 'Follow me,' it said."

The data was successfully outputted to the receiver's file, which matched the original text in both order and content (**Figure 5.0**)



```
received_packets.txt
1 The forest whis
2 pered secrets a
3 s Luna wandered
4 deeper. She fo
5 und a golden ke
6 y hanging on an
7 ancient oak.
8 What does it un
9 lock? she wond
10 ered. A fox app
11 eared, its eyes
12 gleaming. Fol
13 low me, it sai
14 d.
15
```

Figure 5.0 : Data Received

Implementation Specifics

Our project consists of 5 files in total.

- `header.py` - a file containing the class definition of our custom header
- `sender.py` - the sender side code
- `receiver.py` - the receiver side code
- `data.txt` - the file the sender reads and transmits data from
- `received_packets.txt` - the file the receiver writes received data to

The steps to run this code:

1. Ensure you are in the correct directory
2. Clear `received_packets.txt` of previous transmissions (if any) [optional]
3. Open two terminals
 - a. In the first enter : `python receiver.py`
 - b. In the second one enter : `python sender.py`

The data will then be transmitted. At the end, a graph will appear showing the changes in the congestion window.

Interesting Modifications to Explore:

- ★ Modify the **LOSS_PROBABILITY** value in `sender.py` to simulate packet loss. (Note: Increasing the loss probability will result in more retransmissions, which may significantly increase the transmission time, potentially taking several minutes.)
- ★ Adjust the **WINDOW_SIZE** parameter in `sender.py` to experiment with different window sizes.
- ★ Try transmitting different data by editing or replacing the content in `data.txt`.

Simulating Errors

As mentioned earlier, in modern communication systems, data loss, delays and errors are inevitable. Hence, to test the robustness of our transport protocol under such conditions, we simulated these network issues, as expected in the real-world network.

This step of the project involved simulating packet loss and checksum errors to evaluate the performance, flow control and congestion control mechanisms of our protocol which was otherwise set up in a controlled environment.

In the real-world, errors take place in the network link layer, as the data may get corrupted after the sender transmits the data however, the receiver did not receive it in the original form (or did not receive it all). Hence in order to simulate this behavior, we used a randomised mechanism that drops the packets (packet loss) and corrupts the packet by disrupting the checksum after the sender sends the data. This randomisation was controlled using probability of error that we intended for our protocol to handle.

Case for packet loss

1. The sender transmits a packet to the receiver.
2. A random packet gets dropped, simulating packet loss.
3. The receiver does not receive the packet, while the sender keeps expecting an ACK.
4. Since the receiver uses Go-back-N mechanism, it keeps dropping subsequent packets that are out of order, while the sender again continues waiting for their ACKs.
5. Eventually, when the ACKs are not received and timeout occurs, the sender detects loss and performs congestion control.
6. Since we have implemented TCP Tahoe, the congestion window size (cwnd) is reduced to 1 MSS to prevent further congestion.
7. Once that is done, the sender prepares for re-transmission of the lost packet, followed by the subsequently sent packets in the correct sequence order.
8. Now, as the packets are sent without any errors, the receiver processes the data and sends ACK for each packet received.
9. Parallel to this, the congestion window size is adjusted again to resume regular data transmission.

Case for corrupted packet

1. The sender transmits a packet to the receiver.
2. A random packet is selected, and the simulator changes its checksum value by 1, leading to a corrupted bit flip situation.
3. The receiver reads the packet, however, leveraging our protocol's reliable transfer mechanism, the receiver verifies the checksum before it sends an ACK. Since the data received is corrupted, it drops the packet, while the sender keeps expecting an ACK.
4. The rest of the steps are repeated as described above, ensuring a proper flow control and congestion control to handle a corrupted packet.

Traffic Analysis

Three Way Handshake

1. Sender Sends Connection Request (SYN) and waits for SYNACK from the Receiver

The image shows a Wireshark packet capture interface. The top pane displays a list of network packets. The selected packet (No. 87699) is a UDP packet from 127.0.0.1 to 127.0.0.1, port 8001 to 8000, with a length of 66 bytes. The middle pane shows the packet details, including the User Datagram Protocol (UDP) section, which indicates the source port is 8001, destination port is 8000, and the length is 66. The bottom pane shows the raw packet data in hexadecimal and ASCII. The ASCII data is garbled, indicating a corrupted packet.

No.	Time	Source	Destination	Protocol	Length	Info
83676	9049.412473	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83705	9051.425727	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83888	9064.409595	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
83889	9064.419023	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83898	9064.429238	127.0.0.1	127.0.0.1	UDP	66	8001 → 8000 Len=34
83919	9065.433579	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
87699	9339.164469	127.0.0.1	127.0.0.1	UDP	66	8001 → 8000 Len=66
87698	9339.164982	127.0.0.1	127.0.0.1	UDP	100	8000 → 8001 Len=68
87699	9339.165195	127.0.0.1	127.0.0.1	UDP	113	8001 → 8000 Len=81
87700	9339.165317	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48
87701	9339.165407	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48
87702	9339.166408	127.0.0.1	127.0.0.1	UDP	68	8000 → 8001 Len=36
87731	9341.168925	127.0.0.1	127.0.0.1	UDP	68	8000 → 8001 Len=36
87774	9344.168338	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48

Packet 87699 details:

- Ethernet II, Src: VirtualBox (00:00:00:00:00:00), Dst: VirtualBox (00:00:00:00:00:00)
- Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
- User Datagram Protocol, Src Port: 8001, Dst Port: 8000
 - Source Port: 8001
 - Destination Port: 8000
 - Length: 66
 - Checksum: 0x744e [unverified]
 - [Checksum Status: Unverified]
 - [Stream index: 0]
 - [Stream Packet Number: 327]
 - [Timestamps]
 - UDP payload (58 bytes)
 - Data (58 bytes): 383030312c383030302c313039362c302c392c31352c342c32333733377c4865792120446f2

2. Receiver sends SYNACK and waits for ACK

ip.addr == 127.0.0.1 && udp.port==8001

No.	Time	Source	Destination	Protocol	Length	Info
83647	9047.399354	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83676	9049.412473	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83705	9051.425727	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83888	9064.409595	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
83889	9064.419023	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83890	9064.429238	127.0.0.1	127.0.0.1	UDP	66	8001 → 8000 Len=34
83919	9066.433579	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
87697	9339.164489	127.0.0.1	127.0.0.1	UDP	90	8001 → 8000 Len=58
87698	9339.164982	127.0.0.1	127.0.0.1	UDP	100	8000 → 8001 Len=68
87699	9339.165195	127.0.0.1	127.0.0.1	UDP	113	8001 → 8000 Len=81
87700	9339.165317	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48
87701	9339.165407	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48
87702	9339.166408	127.0.0.1	127.0.0.1	UDP	68	8000 → 8001 Len=36
87731	9341.168925	127.0.0.1	127.0.0.1	UDP	68	8000 → 8001 Len=36

.0. = Don't fragment: Not set
 .0. = More fragments: Not set
 ...0 0000 0000 0000 = Fragment Offset: 0
 Time to Live: 128
 Protocol: UDP (17)
 Header Checksum: 0x0000 [validation disabled]
 [Header checksum status: Unverified]
 Source Address: 127.0.0.1
 Destination Address: 127.0.0.1
 [Stream index: 0]
 User Datagram Protocol, Src Port: 8000, Dst Port: 8001
 Source Port: 8000
 Destination Port: 8001
 Length: 76
 Checksum: 0x6ee4 [unverified]
 [Checksum Status: Unverified]
 [Stream index: 0]
 [Stream Packet Number: 328]
 [Timestamps]
 UDP payload (68 bytes)
 Data (68 bytes)
 Data: 383030302c383030312c313636322c313039372c342c31352c362c3631313537c537572652

0000 02 00 00 00 45 00 00 6d 4e 34 00 00 80 11 00 00 ... E... N4
 0010 7f 00 00 01 7f 00 00 01 1f 40 1f 41 00 4c 6e e4 A L n
 0020 38 30 30 30 2c 38 30 30 31 2c 31 36 36 32 2c 31 8000,800 1,1662,1
 0030 30 39 37 2c 34 2c 31 35 2c 36 2c 36 31 31 35 33 897,4,15 ,6,11153
 0040 7c 53 75 72 65 21 20 4a 6f 20 79 6f 75 20 73 74 |Sure! D o you st
 0050 69 6c 6c 20 77 61 6e 74 20 74 6f 20 63 6f 6e 6e ill want to conn
 0060 65 63 74 3f ect?

3. Sender receives SYNACK, goes into established stage and sends an ACK

ip.addr == 127.0.0.1 && udp.port==8001

No.	Time	Source	Destination	Protocol	Length	Info
83647	9047.399354	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83676	9049.412473	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83705	9051.425727	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83888	9064.409595	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
83889	9064.419023	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83890	9064.429238	127.0.0.1	127.0.0.1	UDP	66	8001 → 8000 Len=34
83919	9066.433579	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
87697	9339.164489	127.0.0.1	127.0.0.1	UDP	90	8001 → 8000 Len=58
87698	9339.164982	127.0.0.1	127.0.0.1	UDP	100	8000 → 8001 Len=68
87699	9339.165195	127.0.0.1	127.0.0.1	UDP	113	8001 → 8000 Len=81
87700	9339.165317	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48
87701	9339.165407	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48
87702	9339.166408	127.0.0.1	127.0.0.1	UDP	68	8000 → 8001 Len=36
87731	9341.168925	127.0.0.1	127.0.0.1	UDP	68	8000 → 8001 Len=36

.0. = Don't fragment: Not set
 .0. = More fragments: Not set
 ...0 0000 0000 0000 = Fragment Offset: 0
 Time to Live: 128
 Protocol: UDP (17)
 Header Checksum: 0x0000 [validation disabled]
 [Header checksum status: Unverified]
 Source Address: 127.0.0.1
 Destination Address: 127.0.0.1
 [Stream index: 0]
 User Datagram Protocol, Src Port: 8001, Dst Port: 8000
 Source Port: 8001
 Destination Port: 8000
 Length: 89
 Checksum: 0x0a5a [unverified]
 [Checksum Status: Unverified]
 [Stream index: 0]
 [Stream Packet Number: 329]
 [Timestamps]
 UDP payload (81 bytes)
 Data (81 bytes)
 Data: 383030312c383030302c313039372c313636332c392c31352c322c373039357c59657320672

0000 02 00 00 00 45 00 00 6d 4e 35 00 00 80 11 00 00 ... E... m N5
 0010 7f 00 00 01 7f 00 00 01 1f 41 1f 40 00 59 0a 5a A @ Y Z
 0020 38 30 30 31 2c 38 30 30 30 2c 31 30 39 37 2c 31 8000,800 0,1097,1
 0030 36 36 33 2c 39 2c 31 35 2c 32 2c 37 30 39 35 7c 663,9,15 ,2,70951
 0040 59 65 73 20 67 72 65 61 74 21 20 4c 65 74 27 73 Yes grea t! Let's
 0050 20 63 6f 6e 6e 65 63 74 20 28 43 6f 6e 6e 65 63 connect (Conne
 0060 74 69 6f 6e 20 45 73 74 61 62 6c 69 73 68 65 64 tion Est ablished
 0070 29

4. Receiver receives ACK and goes into the established stage.

Flow Control:

Sender sends data:

ip.addr == 127.0.0.1 && udp.port == 8001

No.	Time	Source	Destination	Protocol	Length	Info
82851	8991.238829	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
82894	8994.235687	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
82895	8994.236211	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
82896	8994.239039	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
82925	8996.243805	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83038	9004.251946	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
83039	9004.256699	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83040	9004.283038	127.0.0.1	127.0.0.1	UDP	78	8001 → 8000 Len=46
83041	9004.292737	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
83042	9004.295362	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
83043	9004.296424	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
83524	9039.327864	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
83525	9039.328639	127.0.0.1	127.0.0.1	UDP	80	8001 → 8000 Len=48
83526	9039.329396	127.0.0.1	127.0.0.1	UDP	78	8001 → 8000 Len=46

..0.. = Don't fragment: Not set
 ..0.. = More fragments: Not set
 ...0 0000 0000 0000 = Fragment Offset: 0
 Time to Live: 128
 Protocol: UDP (17)
 Header Checksum: 0x0000 [validation disabled]
 [Header checksum status: Unverified]
 Source Address: 127.0.0.1
 Destination Address: 127.0.0.1
 [Stream index: 0]
 User Datagram Protocol, Src Port: 8001, Dst Port: 8000
 Source Port: 8001
 Destination Port: 8000
 Length: 55
 Checksum: 0xc3e9 [unverified]
 [Checksum Status: Unverified]
 [Stream index: 0]
 [Stream Packet Number: 299]
 [Timestamps]
 UDP payload (47 bytes)
 Data (47 bytes)
 Data: 383030312c383030302c313134352c3532372c392c31352c302c35333330357c73204c756e6

0000 02 00 00 00 45 00 00 4b 3b 78 00 00 00 11 00 00 ... E..K ;x
 0010 7f 00 00 01 7f 00 00 01 1f 41 1f 40 00 37 c3 e9 @ A
 0020 38 30 30 31 2c 38 30 30 30 2c 31 31 34 35 2c 35 8001,800 0,1145,5
 0030 32 37 2c 39 2c 31 35 2c 30 2c 35 33 33 30 35 7c 27,9,15, 0,53305
 0040 73 20 4c 75 6e 61 20 77 61 6e 64 65 72 65 64 s Luna w andered

Receiver side sends ACK:

ip.addr == 127.0.0.1 && udp.port == 8001

No.	Time	Source	Destination	Protocol	Length	Info
82820	8989.232985	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
82821	8989.233259	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
82822	8989.234278	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
82851	8991.238829	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
82894	8994.235687	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
82895	8994.236211	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
82896	8994.239039	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
82925	8996.243805	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83038	9004.251946	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
83039	9004.256699	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
83040	9004.283038	127.0.0.1	127.0.0.1	UDP	78	8001 → 8000 Len=46
83041	9004.292737	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
83042	9004.295362	127.0.0.1	127.0.0.1	UDP	79	8001 → 8000 Len=47
83043	9004.296424	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49

..0.. = Don't fragment: Not set
 ..0.. = More fragments: Not set
 ...0 0000 0000 0000 = Fragment Offset: 0
 Time to Live: 128
 Protocol: UDP (17)
 Header Checksum: 0x0000 [validation disabled]
 [Header checksum status: Unverified]
 Source Address: 127.0.0.1
 Destination Address: 127.0.0.1
 [Stream index: 0]
 User Datagram Protocol, Src Port: 8000, Dst Port: 8001
 Source Port: 8000
 Destination Port: 8001
 Length: 43
 Checksum: 0x137f [unverified]
 [Checksum Status: Unverified]
 [Stream index: 0]
 [Stream Packet Number: 302]
 [Timestamps]
 UDP payload (35 bytes)
 Data (35 bytes)
 Data: 383030302c383030312c3531352c313136302c342c31352c322c313139333317c41434b

0000 02 00 00 00 45 00 00 3f 3b 97 00 00 00 11 00 00 ... E..? ;
 0010 7f 00 00 01 7f 00 00 01 1f 40 1f 41 00 2b 13 7f @ A
 0020 38 30 30 30 2c 38 30 30 31 2c 35 31 35 2c 31 31 8000,800 1,515,11
 0030 36 30 2c 34 2c 31 35 2c 32 2c 31 31 39 33 31 7c 60,4,15, 2,11931
 0040 41 43 4b ACK

Connection Termination:

1. Sender transmits FIN to terminate connection.

The screenshot shows a Wireshark packet capture of a network connection. The top pane displays a list of packets, with packet 1024 selected. The middle pane shows the details of the selected packet, which is a User Datagram Protocol (UDP) packet. The bottom pane shows the raw data of the packet in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Length	Info
1021..	10377.175641	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
1021..	10377.176111	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
1021..	10377.176467	127.0.0.1	127.0.0.1	UDP	66	8001 → 8000 Len=34
1021..	10377.177788	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1021..	10379.180130	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022..	10381.186506	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022..	10383.191448	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022..	10385.199032	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1024..	10397.187571	127.0.0.1	127.0.0.1	UDP	92	8001 → 8000 Len=60
1024..	10397.188264	127.0.0.1	127.0.0.1	UDP	70	8000 → 8001 Len=38
1024..	10397.188734	127.0.0.1	127.0.0.1	UDP	93	8001 → 8000 Len=61
1055..	10617.797972	127.0.0.1	127.0.0.1	UDP	90	8001 → 8000 Len=58
1055..	10617.798416	127.0.0.1	127.0.0.1	UDP	99	8000 → 8001 Len=67
1055..	10617.798799	127.0.0.1	127.0.0.1	UDP	84	8001 → 8000 Len=52

Details of packet 1024 (UDP):

- Header Checksum: 0x0000 [validation disabled]
- [Header checksum status: Unverified]
- Source Address: 127.0.0.1
- Destination Address: 127.0.0.1
- [Stream index: 0]
- User Datagram Protocol, Src Port: 8001, Dst Port: 8000
- Source Port: 8001
- Destination Port: 8000
- Length: 68
- Checksum: 0x6e35 [unverified]
- [Checksum Status: Unverified]
- [Stream index: 0]
- [Stream Packet Number: 396]
- [Timestamps]
- UDP payload (60 bytes)
- Data (60 bytes)
- Data: 383030312c383030302c3738392c313934302c392c31352c312c33363939387c476f664627

Raw data (hex): 0000 02 00 00 00 45 00 00 58 87 bf 00 00 80 11 00 00
 0010 7f 00 00 01 7f 00 00 01 1f 41 1f 40 00 44 6e 35
 0020 38 30 30 31 2c 38 30 30 30 2c 37 38 39 2c 31 39
 0030 34 30 2c 39 2c 31 35 2c 31 2c 33 36 39 39 38 7c
 0040 47 6f 6f 64 62 79 65 21 20 43 6c 6f 73 69 6e 67
 0050 20 63 6f 6e 6e 65 63 74 69 6f 6e 2e

2. Receiver sends a FIN ACK and terminates connection

The screenshot shows a Wireshark packet capture of a network connection. The top pane displays a list of packets, with packet 1024 selected. The middle pane shows the details of the selected packet, which is a User Datagram Protocol (UDP) packet. The bottom pane shows the raw data of the packet in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Length	Info
1021..	10377.175641	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
1021..	10377.176111	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
1021..	10377.176467	127.0.0.1	127.0.0.1	UDP	66	8001 → 8000 Len=34
1021..	10377.177788	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1021..	10379.180130	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022..	10381.186506	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022..	10383.191448	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022..	10385.199032	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1024..	10397.187571	127.0.0.1	127.0.0.1	UDP	92	8001 → 8000 Len=60
1024..	10397.188264	127.0.0.1	127.0.0.1	UDP	70	8000 → 8001 Len=38
1024..	10397.188734	127.0.0.1	127.0.0.1	UDP	93	8001 → 8000 Len=61
1055..	10617.797972	127.0.0.1	127.0.0.1	UDP	90	8001 → 8000 Len=58
1055..	10617.798416	127.0.0.1	127.0.0.1	UDP	99	8000 → 8001 Len=67
1055..	10617.798799	127.0.0.1	127.0.0.1	UDP	84	8001 → 8000 Len=52

Details of packet 1024 (UDP):

- Header Checksum: 0x0000 [validation disabled]
- [Header checksum status: Unverified]
- Source Address: 127.0.0.1
- Destination Address: 127.0.0.1
- [Stream index: 0]
- User Datagram Protocol, Src Port: 8000, Dst Port: 8001
- Source Port: 8000
- Destination Port: 8001
- Length: 46
- Checksum: 0xd2c7 [unverified]
- [Checksum Status: Unverified]
- [Stream index: 0]
- [Stream Packet Number: 397]
- [Timestamps]
- UDP payload (38 bytes)
- Data (38 bytes)
- Data: 383030302c383030312c313935332c3830332c342c31352c332c35373033397c46494e41434

Raw data (hex): 0000 02 00 00 00 45 00 00 42 87 c0 00 00 80 11 00 00
 0010 7f 00 00 01 7f 00 00 01 1f 40 1f 41 00 2e d2 c7
 0020 38 30 30 30 2c 38 30 30 31 2c 31 39 35 33 2c 38
 0030 30 33 2c 34 2c 31 35 2c 33 2c 35 37 30 33 39 7c
 0040 46 49 4e 41 43 4b

3. Sender sends ACK and closes the connection

No.	Time	Source	Destination	Protocol	Length	Info
1021	10377.175641	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
1021	10377.176111	127.0.0.1	127.0.0.1	UDP	81	8001 → 8000 Len=49
1021	10377.176467	127.0.0.1	127.0.0.1	UDP	66	8001 → 8000 Len=34
1021	10377.177788	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1021	10379.180130	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022	10381.186506	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022	10383.191448	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1022	10385.199032	127.0.0.1	127.0.0.1	UDP	67	8000 → 8001 Len=35
1024	10397.187571	127.0.0.1	127.0.0.1	UDP	92	8001 → 8000 Len=60
1024	10397.188264	127.0.0.1	127.0.0.1	UDP	70	8000 → 8001 Len=38
1024	10397.188734	127.0.0.1	127.0.0.1	UDP	93	8001 → 8000 Len=61
1055	10617.797972	127.0.0.1	127.0.0.1	UDP	90	8001 → 8000 Len=58
1055	10617.798416	127.0.0.1	127.0.0.1	UDP	99	8000 → 8001 Len=67
1055	10617.798799	127.0.0.1	127.0.0.1	UDP	84	8001 → 8000 Len=52

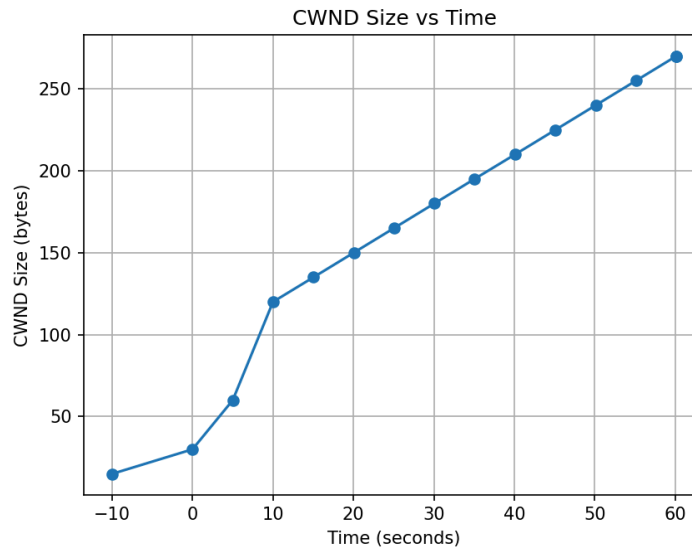
... .. = Don't fragment: Not set
 = More fragments: Not set
 ...0 0000 0000 0000 = Fragment Offset: 0
 Time to Live: 128
 Protocol: UDP (17)
 Header Checksum: 0x0000 [validation disabled]
 [Header checksum status: Unverified]
 Source Address: 127.0.0.1
 Destination Address: 127.0.0.1
 [Stream index: 0]
 User Datagram Protocol, Src Port: 8001, Dst Port: 8000
 Source Port: 8001
 Destination Port: 8000
 Length: 69
 Checksum: 0x4eb6 [unverified]
 [Checksum Status: Unverified]
 [Stream index: 0]
 [Stream Packet Number: 398]
 [Timestamps]
 UDP payload (61 bytes)
 Data (61 bytes)
 Data: 383030312c383030302c3738392c313934312c392c31352c322c32373133397c46696e616c2

4. Receiver receives ACK and closes connection.

Congestion Control Visual Representation

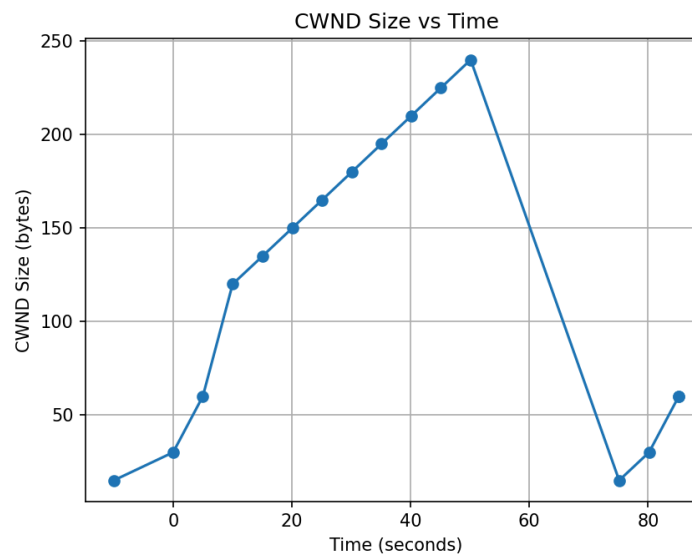
In order to prevent congestion through the link, it is crucial to control the transmission rate of the data from the sender to the receiver. As we discussed, in the real-world network, we experience losses, hence using our simulation, we can see our congestion window size varying through the following graphs. We will demonstrate it for the following scenarios:

1. No Loss



We observe that the window size increases exponentially until the Slow Start Threshold (sssthresh) is reached and then it increases linearly i.e. Transition from Slow Start to Congestion Avoidance.

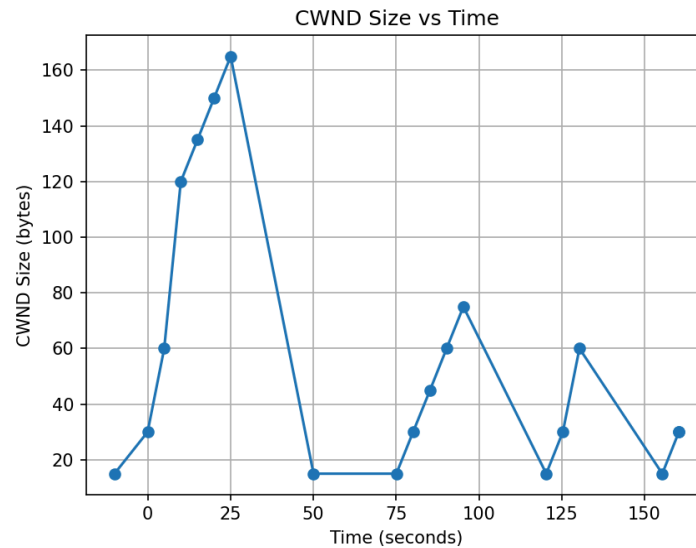
2. Loss with 5% Probability:



We observe as we do not encounter a loss for the first 50 seconds of transmission, we see an exponential growth until we reach our sssthresh and then switch to Congestion Avoidance to grow linearly. However,

as soon as we have a loss/error, we set the Congestion Window back to its initial value by implementing the TCP Tahoe mechanism, and it continues growing similarly while there is no loss/error.

3. Loss with 20% probability:



In this case, we observe multiple loss/error scenarios and can get a better idea of how the use of TCP Slow Start, TCP Tahoe and Congestion Avoidance, we can control the traffic of data being sent by the sender.