Pipelined Reliable Transfer Protocol

By:
Kandisa Agarwal and Puja Shah

TABLE OF CONTENTS

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.

<table>
<tr><td colspan="2">Source Port</td><td colspan="4">Destination Port</td></tr>
<tr><td colspan="6">Sequence Number</td></tr>
<tr><td colspan="6">ACK Number</td></tr>
<tr><td>MSS</td><td>Checksum</td><td>Sending Window</td><td>SIN</td><td>ACK</td><td>FIN</td></tr>
<tr><td colspan="6">App Data</td></tr>
</table>

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.

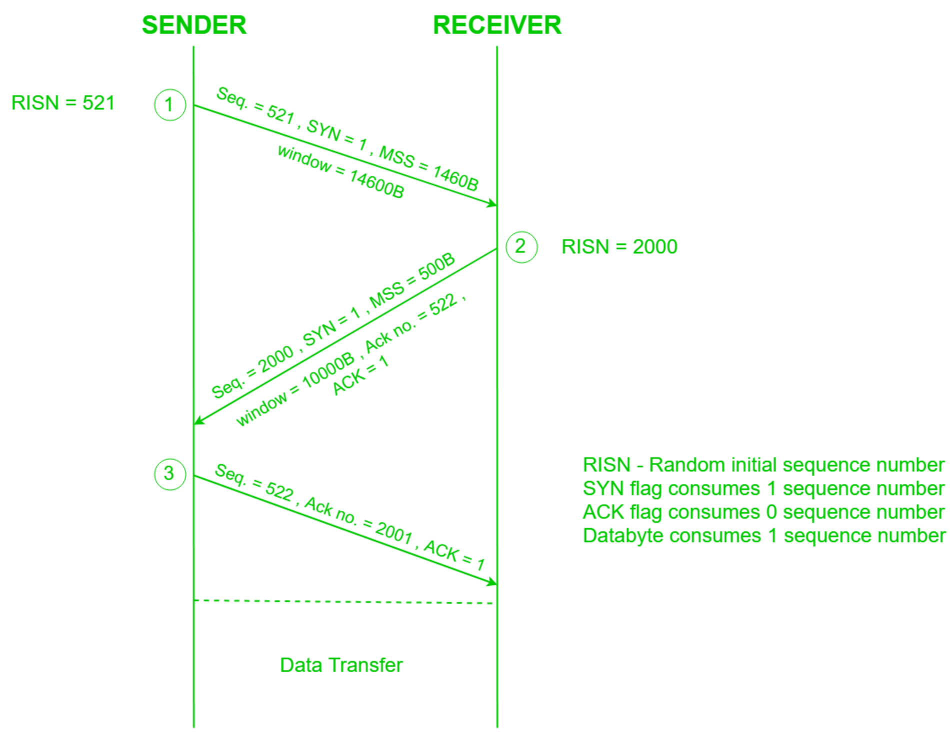


Figure 1.0 3-way Handshake

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**:

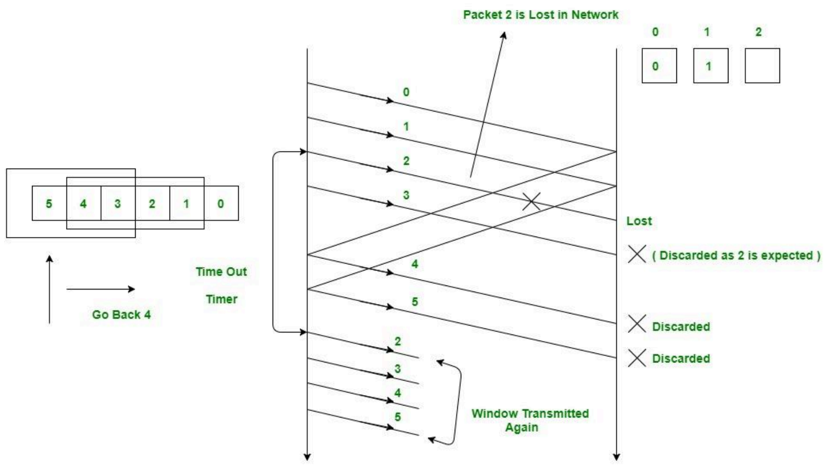


Figure 2.0 Go-Back-N

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)**

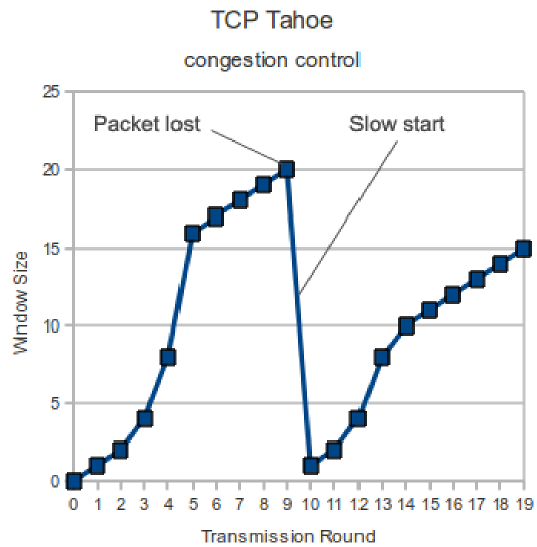


Figure 3.0 TCP Tahoe Sample

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**

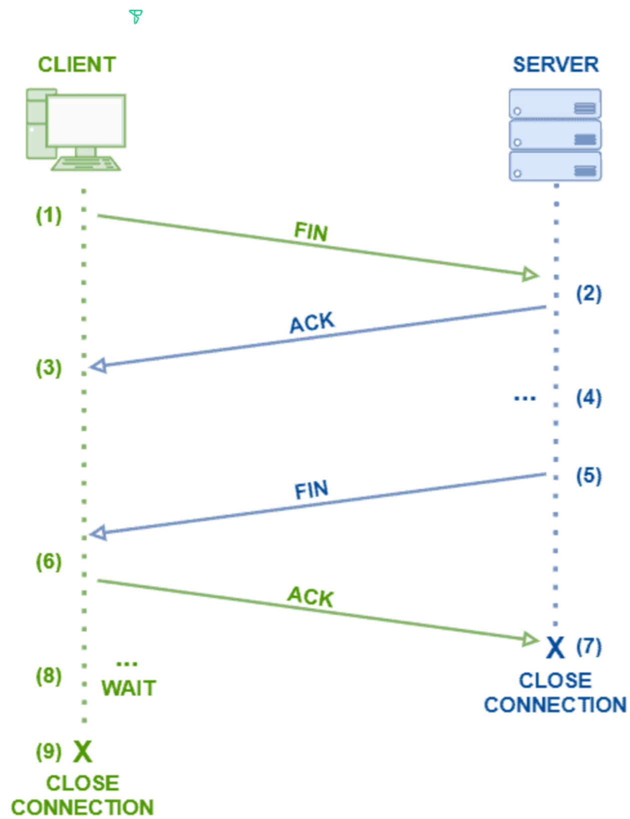


Figure 4.0 FIN Exchange : Terminate Connection

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

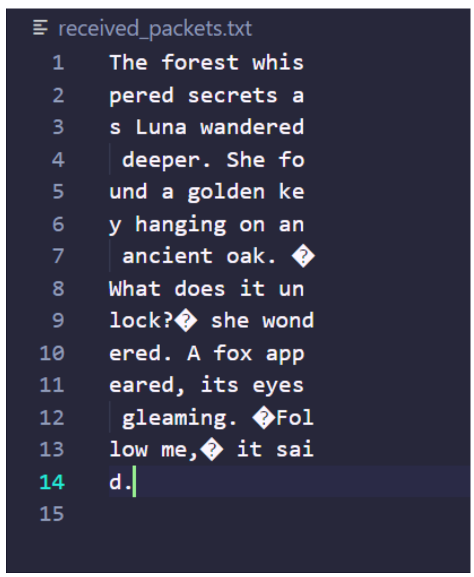
Behavior	Mechanism	Reason
Connection Establishment	Three Way Handshake	Highly secure
Flow Control	Go-Back-N Pipelining	Low overhead and simple to implement
Congestion Control	TCP Tahoe	Elegant and efficient
Termination	FIN-ACK	Secure and efficient

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

```
The forest whis
pered secrets a
s Luna wandered
 deeper. She fo
und a golden ke
y hanging on an
 ancient oak. �
What does it un
lock?� she wond
ered. A fox app
eared, its eyes
 gleaming. �Fol
low me,� it sai
d.

```

Figure 5.0 : Data Received

<u>Implementation Specifics</u>

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.

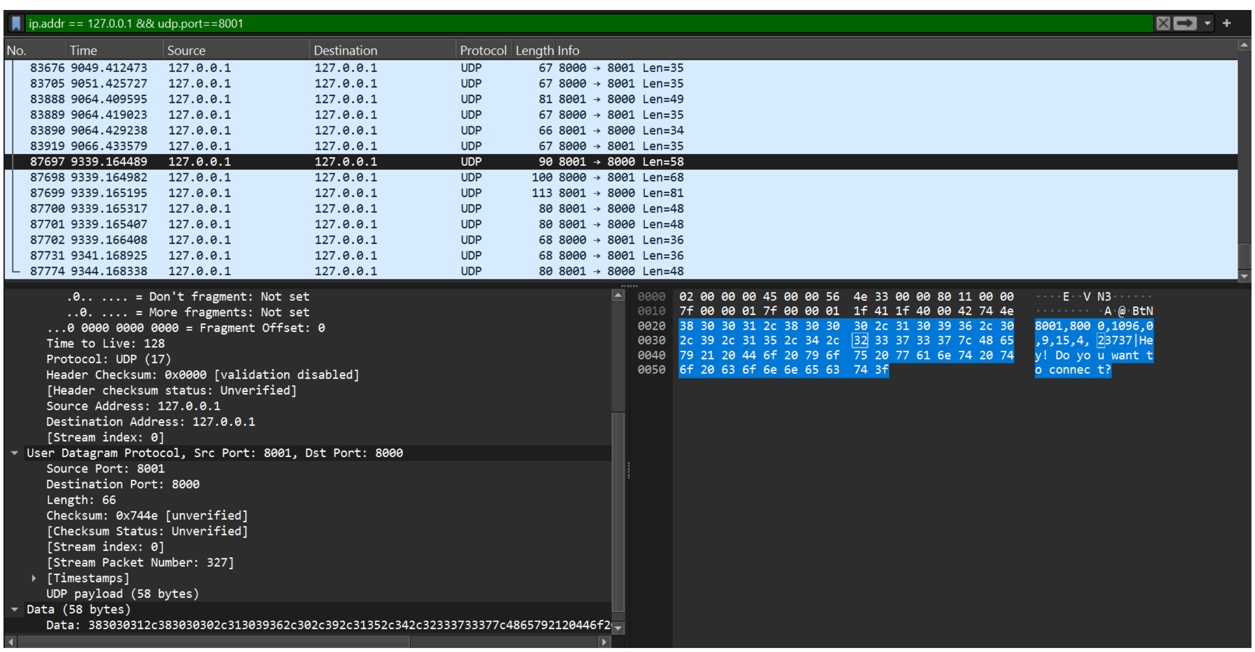
Case for corrupted packet

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.

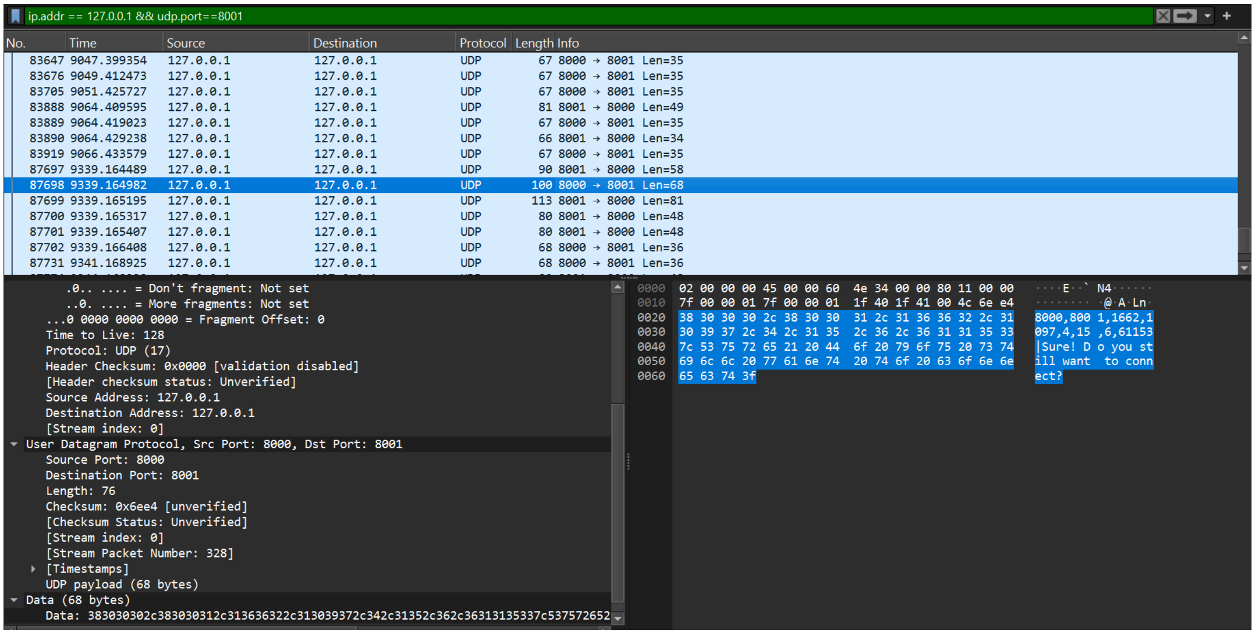
Traffic Analysis

Three Way Handshake

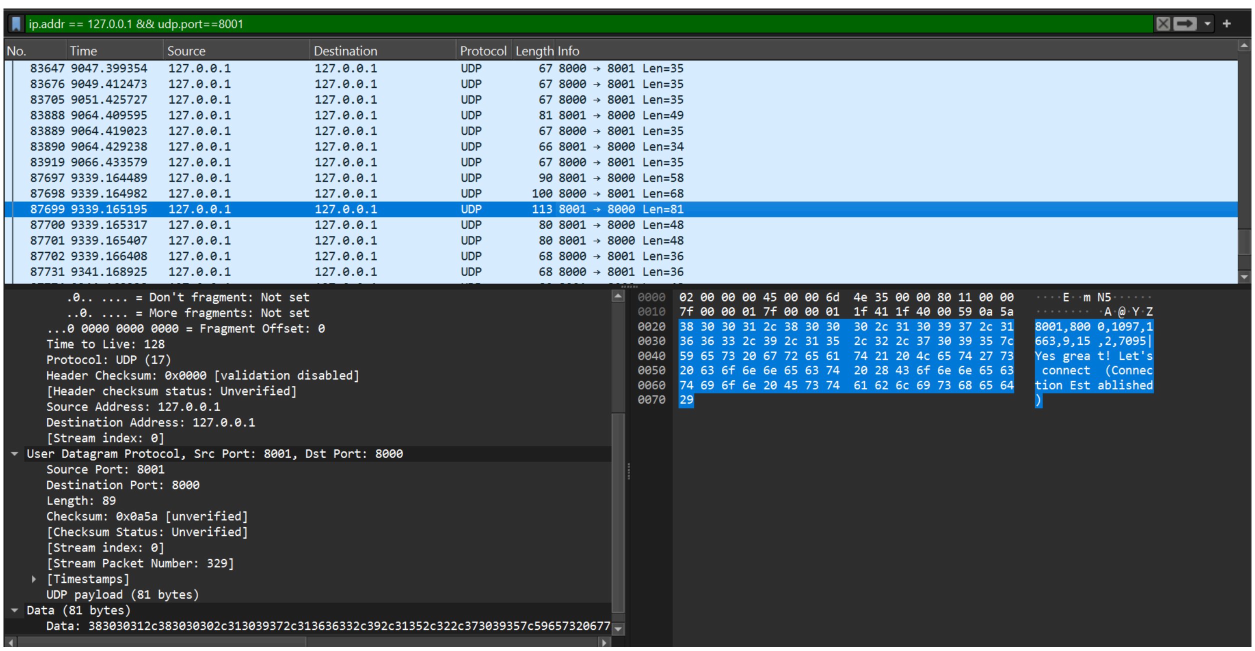
1. Sender Sends Connection Request (SYN) and waits for SYNACK from the Receiver



2. Receiver sends SYNACK and waits for ACK



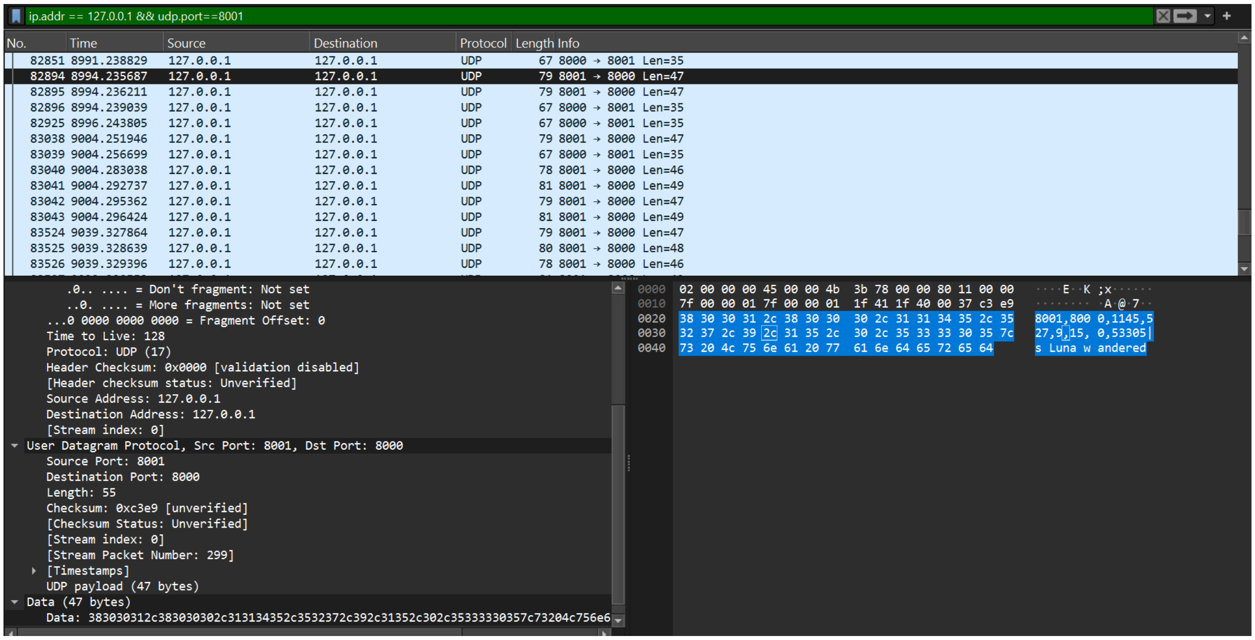
3. Sender receives SYNACK, goes into established stage and sends an ACK



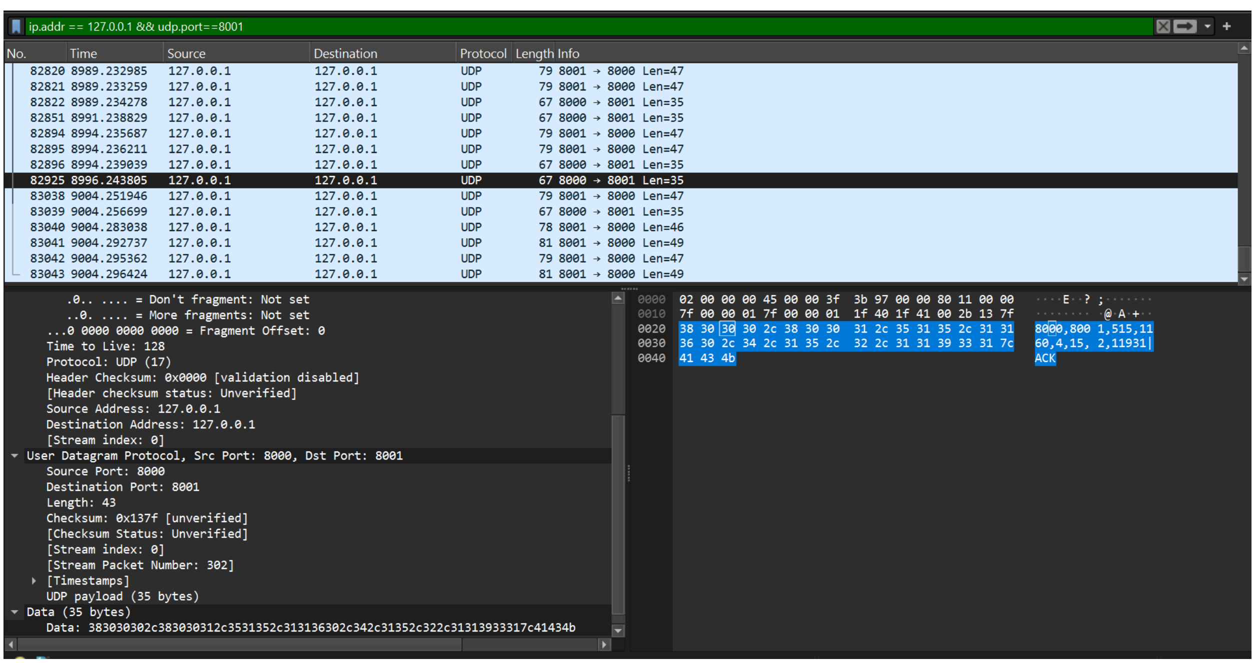
4. Receiver receives ACK and goes into the established stage.

Flow Control:

Sender sends data:

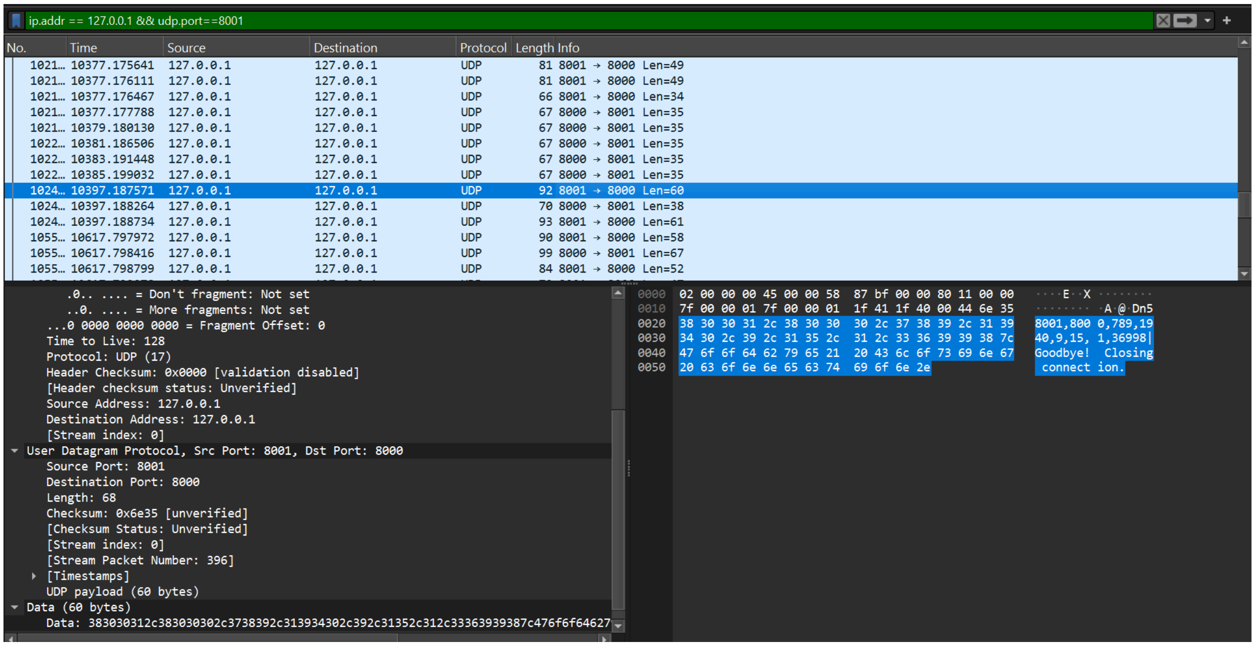


Receiver side sends ACK:

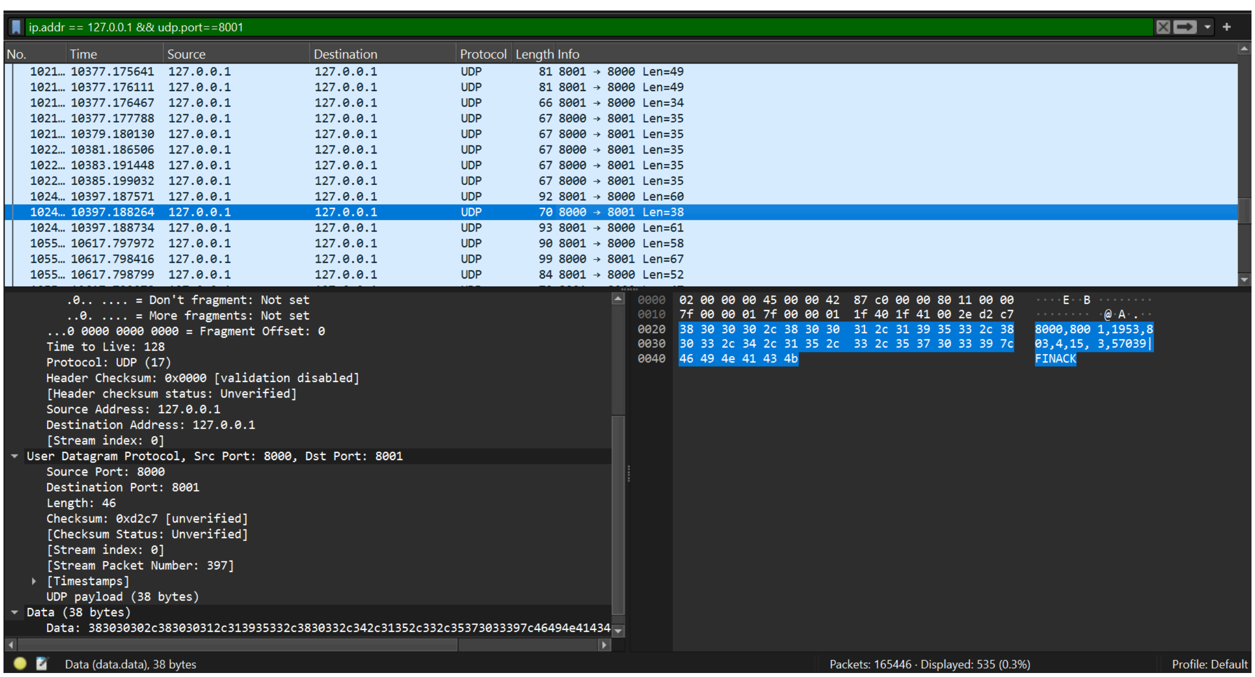


Connection Termination:

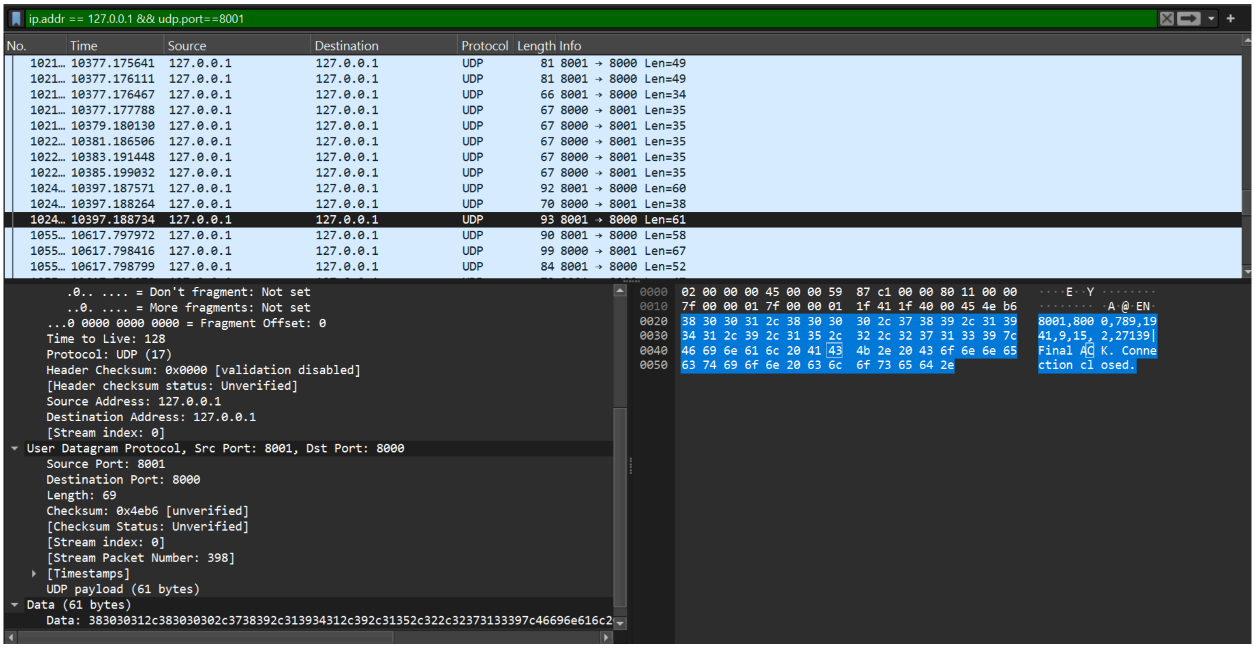
1. Sender transmits FIN to terminate connection.



2. Receives sends a FIN ACK and terminates connection



3. Sender sends ACK and closes the connection

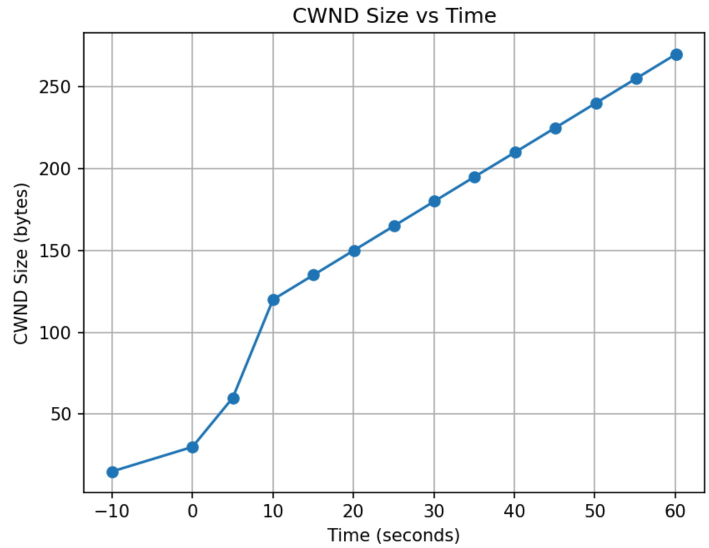


4. Receiver receives ACK and closes connection.

Congestion Control Visual Representation

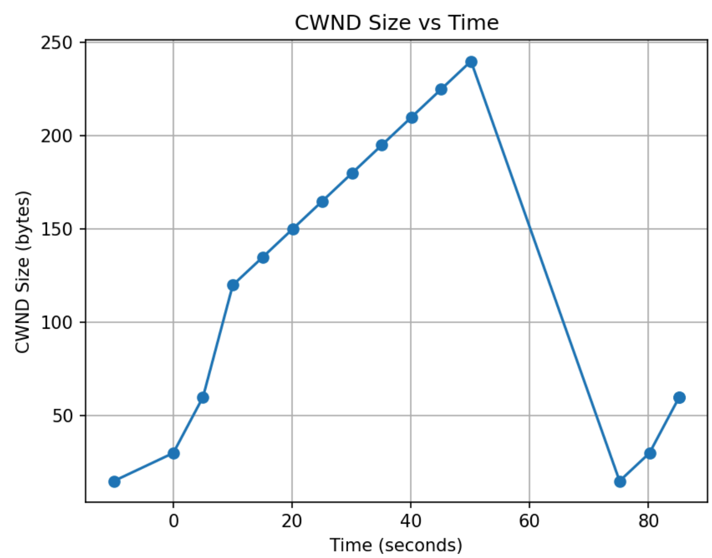
In order to prevent congestion through the link, it is crucial to control the transmission rate of the data from the sender to the receiver. As we discussed, in the real-world network, we experience losses, hence using our simulation, we can see our congestion window size varying through the following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

1. No Loss



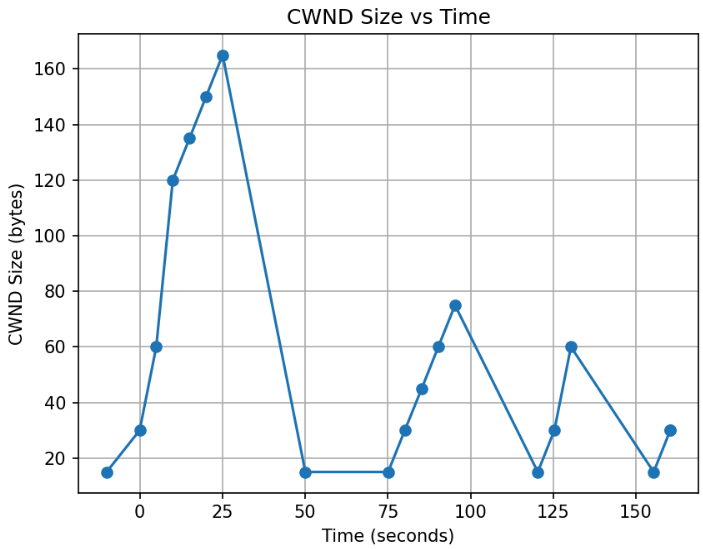
We observe that the window size increases exponentially until the Slow Start Threshold (ssthresh) is reached and then it increases linearly i.e. Transition from Slow Start to Congestion Avoidance.

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 ssthresh 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.