ECS 152A - Project 1 Report

Nam Nguyen
Richard Ho

February 2026

1 Stop and Wait Protocol

For the stop and wait protocol, we first start by creating our set parameters and helper functions such as `PACKET_SIZE` = 1024 bytes, `SEQ_ID_SIZE` = 4, `MESSAGE_SIZE` = `PACKET_SIZE` − `SEQ_ID_SIZE`, and the `make_packet(0` function which takes in a sequence id and payload and creates a packet with the sequence id appended in front of the payload.

We also have a sender function that takes in data and sends it to the receiver. The way this function works is that it first binds to a port and sets a receiver port. Then starting at sequence id = 0, we want to create a packet with a data chunk starting from our sequence id to sequence id + message size. We will then try to send this packet to the receiver. One of two things will happen here, either the packet times out and the packet is resent, or an acknowledgement is returned from the receiver. In the case that the sender receives an acknowledgement, we want to verify that this acknowledgement matches what we expect from it, which is sequence id + message size. If we successfully verify the acknowledgement, then we can set the next sequence id to be the acknowledgement id and continue to the next packet. This is the basis of the stop and wait protocol: it sends out a packet, and waits until the receiver has successfully received it, only then does it send the next packet. Here are some core variables for the stop and wait protocol:

Variable	Description
`packet_delay_arr`	Array of the packet delays, later used to calculate average per-packet delay
`seq_id`	Current sequence id
`data_chunk`	The data from offset sequence id to sequence id + message size
`ack_id`	The returned acknowledgement number

1.1 Metrics

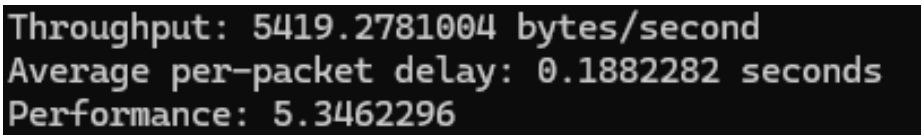


Figure 1: Metrics for the Stop and Wait Protocol

2 Fixed Sliding Window Protocol

Unlike the Stop and Wait protocol, which sends only one packet at a time until receiving the corresponding acknowledgment, the fixed sliding window protocol allows multiple packets to be in flight simultaneously while ensuring in-order packet delivery using cumulative acknowledgments from the receiver. To achieve this, we need a mechanism to properly handle the packet buffer at the sender side. Macros and helper functions for the implementation of this protocol include those introduced in the section 1. Additionally, we also define the macro `WINDOW_SIZE = 100` to indicate the size of the window. Core variables for the implementation of this protocol are given as follows:

Variable	Description
`head`	Index of the earliest unacknowledged packet
`tail`	Index of the next packet allowed to send
`seq_id_head`	Byte offset of the earliest unacknowledged packet
`seq_id_tail`	Byte offset of the next packet allowed to send

Initially, all these variables are set to 0. To adhere to the fixed size of the window range, the following must hold entire time

$$\texttt{tail} - \texttt{head} \leq \texttt{WINDOW\_SIZE}$$

2.1 Transmission Rule

Each packet of size 1024-bytes consists of

- 4-bytes signed big-endian sequence number
- up to 1020 bytes payload

We continuously transmit multiple packets available to be sent in the current window range while the following holds

$$\texttt{tail} < \texttt{head} + \texttt{WINDOW\_SIZE}$$

After each packet gets transmitted, we increment the value of `tail` by 1 and advance `seq_id_tail` by the payload size of the transmitted packet. These updates allow us to expand the window range and keep track of which packet can be sent next.

2.2 Acknowledgement Handling

From the receiver perspective, it buffers out-of-order packets and sends an accumulative acknowledgement, *i.e*, `ack_id` from the receiver means that it has received all bytes up to $\texttt{ack\_id} - 1$ and the offset of the next packet is expected to be `ack_id`

Upon receiving an acknowledgement from the receiver, the sender needs to advance the window while the following holds

$$\texttt{seq\_id\_head} + \texttt{pay\_load\_size} \leq \texttt{ack\_id}$$

To advance the window to the right, we simply increment `seq_id_head` by the size of the corresponding payload at the end of each iteration. This mechanism ensures that we are correctly keeping track of what packets have been received. In other words, we only slide the window forward only when confirmed data has been acknowledged by the receiver.

2.3 Timeout and Retransmission

If we have not received any acknowledgement from the receiver within the defined socket timeout period, we will apply the Go-Back-N strategy to resend lost packets, *i.e.*, we retransmit all packets in the current window range. The number of packet needs to be retransmitted is denoted by $\texttt{tail} - \texttt{head}$, and the retransmission starts at the packet with byte offset `seq_id_head`.

2.4 Metrics

```
Throughput: 91912.0029008 bytes/second
Average per-packet delay: 1.0988948 seconds
Performance: 28.2120710
```

Figure 2: Metrics for the Fixed Sliding Window Protocol

2.5 External Resources

None

3 TCP Reno Protocol

The transmission rule of the TCP Reno algorithm is nearly identical to that of Sliding Window as described in 2.1. Unlike the Fixed Sliding Window protocol, where the window is a pre-defined constant, TCP Eeno dynamically adjusts the size of the sender's transmission buffer based on network feedback. For this purpose, TCP Reno maintains the congestion window, namely `cwnd`, to balance throughput and network stability.

The effective sending window is defined as

$$\min(\texttt{cwnd}, \text{rwin})$$

where `rwin` represents advertised flow control window from receiver. Since the receiver does not advertise a receive window, the congestion control is solely relied on `cwnd`.

Variables for TCP Reno protocol implementation include those in 2. We also introduce some additional variables to handle core features of this algorithm as follows

Variable	Description
`cwnd`	Congestion Window (in packets)
`ssthresh`	Slow start threshold
`last_ack`	Byte offset of the last acknowledged packet
`ack_dups`	The number of duplicate acknowledgements
`ca_acked`	The number of accumulative acknowledged packets when in Congestion Avoidance

Initially, we set $\texttt{cwnd} = 1$ and $\texttt{ssthresh} = 64$ while the others are assigned to 0.

3.1 Transmission Rule

As mentioned above, TCP Reno shares the same transmission rule with Fixed Sliding Window, but with a dynamic congestion window. Therefore, the condition for the transmission rule now becomes

$$\texttt{tail} < \texttt{head} + \texttt{cwnd}$$

Similar to the Fixed Sliding Window Protocol, we increment the value of `tail` by 1 and advance `seq_id_tail` by the payload size for each packet. When receiving `ack_id` from the receiver, the value of `cwnd` and `ssthresh` are dynamically adjusted based on each state as described in the following subsections.

3.2 Slow Start

The objective of this phrase is to quickly determine available bandwidth capacity. Upon receiving an acknowledgement from the receiver, we increase the value of `cwnd` by 1 for each new acknowledged packet. Since it is estimated that there are `cwnd` packets acknowledged per RTT in this phrase, we have

$$\texttt{cwnd}_{new} \approx 2 \cdot \texttt{cwnd}_{old}$$

To be more precise, we keep track of the number of accumulated acknowledged packets per RTT in our implementation and increase `cwnd` by that amount. Whenever we observe that $\texttt{cwnd} \geq \texttt{ssthresh}$, we will transition the sender into the Congestion Avoidance state.

3.3 Congestion Avoidance

The objective of this phrase is to cautiously increase the throughput by expanding the congestion window `cwnd` in linear rate. When being in Congestion Avoidance, the sender adjusts the value of `cwnd` as follows

$$\texttt{cwnd} = \texttt{cwnd} + \frac{1}{\texttt{cwnd}} \quad \text{per newly-ACKed packet}$$

In TCP Reno, the sender is expected to receive a window's worth of acknowledgements, *i.e.*, `cwnd` packets for each RTT. This leads to the following net effect formula

$$\texttt{cwnd}_{new} = \texttt{cwnd}_{old} + 1 \quad \text{per RTT}$$

In our implementation, to measure RTT when in Congestion Avoidance, we maintain an ACK accumulator, namely `ca_acked`, and apply the following steps

- Count newly acknowledged packets and store to `ca_acked`
- If `ca_acked` $\geq$ `cwnd`, then increase `cwnd` by 1 and subtract `cwnd` from `ca_acked`

3.4 Fast Retransmit

When receiving three duplicate acknowledgements, we assume that the packet whose byte offset equals `ack_id` gets lost. When this happens, we immediately retransmit the lost packet and adjust the congestion window as follows

$$\texttt{sshthresh} = \max\left(\frac{\texttt{cwnd}}{2}, 1\right)$$

$$\texttt{cwnd} = \texttt{ssthresh} + 3$$

These adjustments implement the multiplicative decrease component of AIMD, and allow the lost packet to be resent without waiting for timeout. This significantly reduces recovery latency.

3.5 Fast Recovery

The objective of this phrase is to prevent the sender from stalling after the Fast Retransmit state. While being in the Fast Recovery, we continue expanding the congestion window upon receiving duplicate acknowledgements. This allows the sender to keep sending new packets instead of stalling when a packet gets lost. Specifically, for each additional duplicate acknowledgement, we adjust the congestion window as follows

$$\texttt{cwnd} = \texttt{cwnd} + 1$$

Once a new acknowledgement has arrived, *i.e.*, `ack_id` > `last_ack`, we can set `cwnd` to its intended value

$$\texttt{cwnd} = \texttt{ssthresh}$$

At this stage, we exit the fast recovery phase and enter congestion avoidance. Notice that `ssthresh` is set to half of `cwnd` in the Fast Retransmit. Therefore, setting `cwnd` = `ssthresh` implements the multiplicative decrease component of AIMD .

3.6 Timeout Handling

A timeout indicates severe congestion, and we have to take conservative actions to stablize the network. More specifically, we do the following

- Set the slow start threshold to half of the current congestion window

$$\texttt{ssthresh} = \max\left(\frac{\texttt{cwnd}}{2}, 1\right)$$

- Reset the congestion window to 1 MSS

$$\texttt{cwnd} = 1$$

- Re-enter the slow start phrase

3.7 Handling Congestion

To illustrate how the congestion window is adjusted during the Fast Retransmit and Fast Recovery phases, we introduce the following example

- Current State: Congestion Avoidance
- `cwnd` = 10
- `ssthresh` = 20
- Packet 101 - 110 are in flight

Now, consider the situation where packet 101 gets lost

1. The packet 102 has been received, the sender receives `ack_id = 101`. We can't send new packets.
2. The packet 103 and 104 has been received, the sender receives `ack_id = 101`. We can't send new packets.

3. The packet 105 has been received, the sender receives `ack_id = 101`. Since we have received 3 duplicate acknowledgements, we enter the Fast Retransmit phrase and resend packet 101. Finally, we adjust the congestion window as follows

$$\texttt{ssthresh} = \frac{\texttt{cwnd}}{2} = 5$$

$$\texttt{cwnd} = \texttt{ssthresh} + 3 = 8$$

We can't still send new packets. At this point, we enter the Fast Recovery

4. The packet 106 has been received, the sender receives `ack_id` = 101. Since we're currently in the Fast Recovery phase and receive a duplicate acknowledgement, we increase the congestion window by 1, *i.e.*,

$$\texttt{cwnd} = \texttt{cwnd} + 1 = 9$$

We can't send new packets.

5. The packet 107 has been received, the sender receives `ack_id` = 101. Since we're currently in the Fast Recovery phase and receive a duplicate acknowledgement, we increase the congestion window by 1, *i.e.*,

$$\texttt{cwnd} = \texttt{cwnd} + 1 = 10$$

We can't send new packets.

6. The packet 108 and 109 has been received, the sender receives `ack_id` = 101. Since we're currently in the Fast Recovery phase and receive a duplicate acknowledgement, we increase the congestion window by 1, *i.e.*,

$$\texttt{cwnd} = \texttt{cwnd} + 1 = 11$$

Now, we can send the packet 111.

7. The packet 110 has been received, the sender receives `ack_id` = 101. Since we're currently in the Fast Recovery phase and receive a duplicate acknowledgement, we increase the congestion window by 1, *i.e.*,

$$\texttt{cwnd} = \texttt{cwnd} + 1 = 12$$

Now, we can send the packet 112.

8. The packet 101 and 111 has been received, the sender receives `ack_id` = 112. We now transition back to the Congestion Avoidance phase since we have received a new acknowledgement. Now, we set the congestion window to its intended size, *i.e.*,

$$\texttt{cwnd} = \texttt{ssthresh} = 5$$

We can now send 4 packets, from 113 to 116.

3.8 Metrics

Figure 3: Metrics for the TCP Reno Protocol

3.9 External Resources

The references for our TCP implementation are [1] and [2]

4 Summary Results

	Stop And Wait	**Fixed Sliding Window**	**TCP Reno**
Average Throughput	5419.2781004	91912.0029008	92522.4197578
Standard Deviation Throughput	110.8673840	4159.5596783	4257.4966927
Average Per-Packet Delay	0.1882282	1.0988948	0.9698473
Standard Deviation Per-Packet Delay	0.003848	0.053693	0.093982
Average Performance	5.3462296	28.2120710	28.4870791
Standard Deviation Performance	0.1093904	1.2769119	1.2735404

References

[1] CS 168 Course Staff. Congestion control implementation. [Online]. Available: https://textbook.cs168.io/transport/cc-implementation.html

[2] F. Fund. (2017, Apr.) Tcp congestion control. [Online]. Available: https://witestlab.poly.edu/blog/tcp-congestion-control-basics/

Submission Page

Include this signed page with your submission

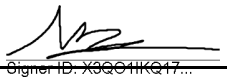
I certify that all submitted work is my own work. I have completed all of the assignments on my own without assistance from others except as indicated by appropriate citation. I have read and understand the university policy on plagiarism and academic dishonesty. I further understand that official sanctions will be imposed if there is any evidence of academic dishonesty in this work. I certify that the above statements are true.

For Group Submissions,

Team Member 1's contributions to this assignment: Nam Nguyen

Short Summary:

Nam implemented the Fixed Sliding Window and TCP Reno Protocol.

Nam Nguyen	 Signer ID: X9QO1IKQ17...	02/12/2026
Full Name	Signature	Date

Team Member 2's contributions to this assignment: Richard Ho

Short Summary:

Richard implemented the stop-and-wait protocol and its corresponding report.

Richard Ho		2/13/2026
Full Name	Signature	Date