Department of
Computer Science and Engineering

University of Dhaka

Title: Implementation of TCP Congestion Control Mechanism: TCP Tahoe and TCP Reno and Their Performance Analysis

CSE 3111: Computer Networking Lab

Batch: 28/3rd Year 1st Semester 2024

Course Instructors

Dr. Ismat Rahman (IsR)
Mr. Jargis Ahmed (JA)
Mr. Palash Roy (PR)

1 Objective(s)

- To gather understanding on the working principle of TCP Tahoe and TCP Reno.
- To implement key mechanisms: slow start, congestion avoidance, fast retransmit, and fast recovery.
- To simulate and visualize the congestion window (cwnd) growth over time.
- To compare the behavior of TCP Tahoe and TCP Reno in the presence of packet loss under different performance metrics.

2 Background Theory

TCP is one of the protocols of the transport layer for network communication. TCP provides reliable, ordered, and error-checked delivery of a stream of bytes between applications running on hosts communicating via an IP network. TCP uses a congestion control mechanism, which manages the data transmission rate between two nodes to prevent a sender from overwhelming a link to a receiver. The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. If a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate; if the sender perceives that there is congestion along the path, then the sender reduces its send rate. Congestion control is different to the Flow control of TCP. Flow control should be done only not to overwhelm a certain receiver for a certain one-way connection. But in contrast, congestion control occurs due to the link carrying the packets in-between them, so that data can be sent in such a way that data do not get loss or dropped due to the congestion occurring in the link between the sender and receiver.

In this lab, we will have a look at the different congestion control algorithms and compare their performances. From the last lab, we know that each host maintains a **receive window, *rwnd*** so that the other host does not overflow it by sending more data than this value, maintaining the following equation:

$$LastByteSent - LastByteAcked \leq rwnd. \tag{1}$$

Similarly, the TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the congestion window. The **congestion window, *cwnd***, imposes a constraint on the rate at which a TCP sender can send traffic into the network. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of ***cwnd*** and ***rwnd***, that is:

$$LastByteSent - LastByteAcked \leq min\{cwnd, rwnd\}. \tag{2}$$

Generally the receive buffer is very large for every host nowadays. So only considering the value of ***cwnd*** will be able to satisfy both the Flow and Congetsoin Controlling mechanisms of TCP. The constraint of this above equality limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender's send rate. To maintain the congestion across the link between a sender and a receiver, TCP maintains the following three guiding principles.

1. A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.
2. An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.
3. *Bandwidth probing:* Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP's strategy for adjusting its transmission rate is to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased. The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then to begins probing again to see if the congestion onset rate has changed.

In addition to the above principles, it is slso needed to be clear that how a link's congestion is detected by any host. A "*loss event*" due to congestion might be detected using a normal ***timeout*** or using a ***triple duplicate ACK***. Whenever a ACK does not return from a receiver to the sender and timeout occurs, sender perceives this as a congestion being detected in the link between them. Also when three duplicated ACKs reach a sender,

this is also considered as a loss of data packet to the receiver from sender, which is also perceived as a detection of data loss from sender to receiver. When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram (containing a TCP segment) to be dropped. The dropped datagram, in turn, results in "*a loss event*" at the sender - either a ***timeout*** or the receipt of ***triple duplicate ACKs*** - which is taken by the sender to be an indication of congestion on the sender-to-receiver path.

Given these three principles and the fact of how a congestion is detected, we will now have a look at actually how the TCP congestion control mechanism operates. Three phases are implemented, suing which this contorlling mechanism works. These are briefly described below.

TCP uses a sliding window flow control protocol. In each TCP segment, the receiver specifies in the receive window field the amount of additional data (in bytes) that it is willing to buffer for the connection. The receiver acknowledges the highest sequence number received in order. It means if segments 1, 2, and 3 are received, an ACK for segment 3 is sent, implying that 1 and 2 were also received. This feature is known as Cumulative Acknowledgment. The sending host can send only up to that amount of data before it must wait for an acknowledgment and window update from the receiving host.

2.1 TCP Slow Start

When a TCP connection begins, the value of **cwnd** is typically initialized to a small value of 1 **MSS**, resulting in an initial sending rate of roughly $\frac{MSS}{RTT}$. TCP sends the first segment into the network and waits for an acknowledgment. When this acknowledgment arrives, the TCP sender increases the congestion window by one **MSS** and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 **MSS** for each of the acknowledged segments, giving a congestion window of 4 **MSS**, and so on. This process results in a doubling of the sending rate every **RTT**. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase.

this exponential growth of **cwnd** and sending rate must be stopped somewhere, otherwise both host and link might face severe problems. Slow Star might be stopped by following any of the following two ways:

1. When congestion is detected using a *timeout*, Slow Start makes the **cwnd** value to 1. Then the overall Slow Start resumes from the beginning.

2. Slow Start maintains another important variable ***ssthresh***. Whenever the value of **cwnd** reaches this value, Slow Start stops and enters into the Congestion Avoidance phase. After congestion detected in that phase, a new value for ***sstresh*** is defined newly.

2.2 TCP Congestion Avoidance

On entry to the congestion-avoidance state, rather than doubling the value of **cwnd** every **RTT** just like in Slow Start phase, TCP adopts a more conservative approach and increases the value of **cwnd** by just a single **MSS** every **RTT**. A common approach is for the TCP sender to increase **cwnd** by **MSS** bytes ($\frac{MSS}{cwnd}$) whenever a new acknowledgment arrives. Similar to the Slow Start, Congestion Avoidance can be stopped based on the following conditions.

1. When a congestion is detected by *timeout*, the value of **cwnd** is set to 1 **MSS**, and the value of ***ssthresh*** is updated to half the value of **cwnd** when the loss event occurred. After that Slow start phase resumes from the beginning.

2. When congestion detected using *triple duplicate ACK* TCP follows less drastic growth than detected by *timeout*. So TCP halves the value of **cwnd** and records the value of ***ssthresh*** to be half the value of **cwnd**. After this, using the Fast Recovery phase, the Congestion Avoidance phase starts again so that TCP increases **cwnd** value linearly.

2.3 TCP Fast Retransmit

If packet loss is detected due to the reception of three duplicate ACKs, TCP Reno assumes that a packet has been lost and immediately retransmits the lost packet.

2.4 TCP Fast Recovery

Fast recovery is a recommended, but not required, component of the TCP congestion control mechanism. Two versions of fast recovery have been highly implemented from the beginning:

1. **TCP Tahoe:** This version unconditionally cuts its **cwnd** to 1 **MSS** and enters the Slow Start phase after a congestion loss event is detected either by a *timeout* or *triple duplicate ACK*.

2. **TCP Reno:** This newer version mainly follows the Fast Recovery phase. The **cwnd** size is halved and then kept constant until all lost packets are retransmitted. After that, the congestion avoidance phase is entered, and the congestion window size is increased linearly.

2.5 General Overall Control Mechanism

If we consider all three phases together, the TCP congestion control mechanism controls the sending of data through a congestion-prone link using the shown algorithms. An example graph is given in Figure 1, where all phases are displayed, including TCP Tahoe and Reno separately. The threshold is initially equal to 8 **MSS**. For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicate ACK event occurs, just after transmission round 8. Note that the congestion window is 12 **MSS** when this loss event occurs. The value of ***ssthresh*** is then set to $0.5\times$**cwnd** $=$ 6 **MSS**. Under TCP Reno, the congestion window is set to **cwnd** $=$ 6 **MSS** and then grows linearly. Under TCP Tahoe, the congestion window is set to 1 **MSS**, starting Slow Start again and grows exponentially until it reaches the value of new ***ssthresh***, at which point it grows linearly.

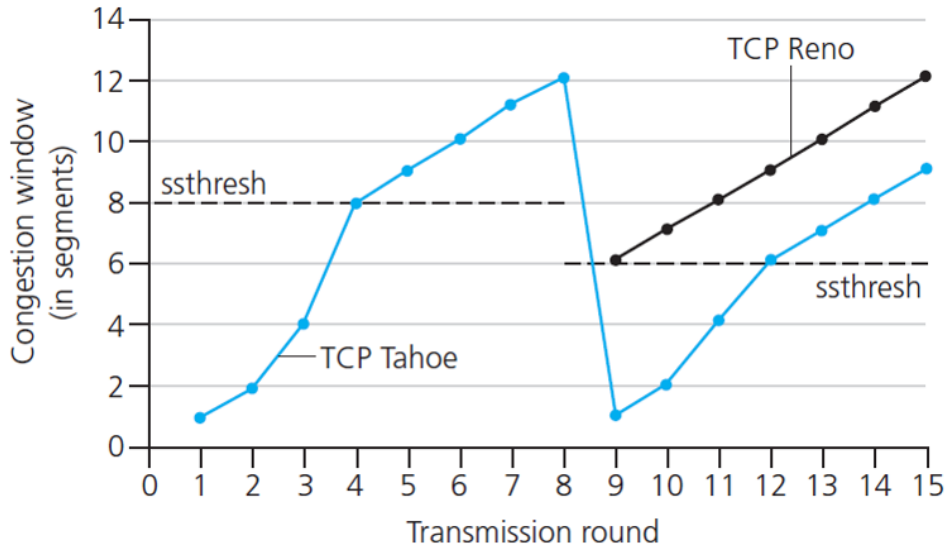

Figure 1: An example graph of transmission round vs **cwnd** denoting phases of Congestion Control

3 Lab Task: Please implement yourself and show the output to the instructor to next lab)

The implementation of TCP Tahoe and TCP Reno should include the following:

- Slow start: Increase the congestion window size exponentially until the threshold is reached.
- Congestion avoidance: Increase the congestion window size linearly after the threshold is reached.
- Fast retransmit: Retransmit lost packets immediately after receiving three duplicate ACKs.
- Fast recovery: Keep the congestion window size constant after retransmitting lost packets.

3.1 Problem analysis

3.1.1 Sender (Client Side) Algorithm Steps:

The detailed step-by-step procedure of the client-side connections is given below.

1. Initialize Parameters: cwnd = 1 (congestion window), ssthresh = 8 (slow start threshold), dupACKcount = 0, lastACK = "", Select TCP mode: TAHOE or RENO
2. For each transmission round (1 to N):
    - Log current cwnd, ssthresh, and round number.
    - Create cwnd number of packets labeled uniquely (pkt_round_index).
    - Send all packets to the server in one message.
3. Receive ACKs from Server:
    - For each packet sent, receive one ACK.
    - If ACK is the same as the previous ACK, **Increment dupACKcount**.
    - If dupACKcount == 3:
        - Fast Retransmit, Set ssthresh = cwnd / 2
        - If mode is RENO:
            - Set cwnd = ssthresh
            - Stay in congestion avoidance (fast recovery).
        - If mode is TAHOE:
            - Set cwnd = 1 (reset)
            - Go back to slow start.
        - Set dupACKcount = 0
        - Go to next round.
    - If ACK is new:
        - Reset dupACKcount = 1
        - Set lastACK = currentACK
4. If No Packet Loss (No Fast Retransmit):
    - If cwnd < ssthresh: (Slow Start), then, set cwnd = cwnd * 2 (exponential growth)
    - If cwnd >= ssthresh (Congestion Avoidance), then, set cwnd = cwnd + 1 (linear growth)
5. Repeat for the next round until N rounds or termination condition.
6. Close connection and exit.

3.1.2 Server Side (Receiver) Algorithm Steps:

The detailed step-by-step procedure of the server-side connections is given below.

1. Start server socket and wait for client connection.
2. For each incoming connection:
    - Read message from client.
    - Parse the incoming packet list (comma-separated values).
3. Simulate Packet Loss:
    - Randomly choose one packet to simulate as "lost" (lossIndex).
4. Send ACKs (For each received packet):
    - If packet is not lost, then, Send ACK as "ACK:<packet_id>"
    - If packet is lost:
        - If it's the first packet, send "ACK:NA" three times.
        - Else, Send 3 duplicate ACKs for the previous packet: "ACK:<previous_packet_id>"
5. Go back to step 2 for next transmission round.
6. Close the socket after communication is complete.

3.2 Sample Output

The sample output of TCP Tahoe Congestion control Mode is as follows
TCP Tahoe Mode

```
The server started on port 5000
Client connected: /127.0.0.1
== TCP TAHOE Mode ==
Round 1: cwnd = 1, ssthresh = 8
Sent packets: pkt1
Received: ACK:pkt1
Slow Start: cwnd -> 2

Round 2: cwnd = 2, ssthresh = 8
Sent packets: pkt2, pkt3
Received: ACK:pkt2
Received: ACK:pkt3
Slow Start: cwnd -> 4

Round 3: cwnd = 4, ssthresh = 8
Sent packets: pkt4, pkt5, pkt6, pkt7
Received: ACK:pkt4
Received: ACK:pkt5
Received: ACK:pkt5
Received: ACK:pkt5
==> 3 Duplicate ACKs: Fast Retransmit triggered.
TCP TAHOE Reset: cwnd -> 1

Round 4: cwnd = 1, ssthresh = 2
Sent packets: pkt5
Received: ACK:pkt5
Slow Start: cwnd -> 2

Round 5: cwnd = 2, ssthresh = 2
Sent packets: pkt6, pkt7
Received: ACK:pkt6
Received: ACK:pkt7
Congestion Avoidance: cwnd -> 3

Round 6: cwnd = 3, ssthresh = 2
Sent packets: pkt8, pkt9, pkt10
Received: ACK:pkt8
Received: ACK:pkt9
Received: ACK:pkt10
Congestion Avoidance: cwnd -> 4

...
Client disconnected.
```

The sample output of TCP Reno Congestion control Mode is as follows

TCP Reno Mode

```
The server started on port 5000
Client connected: /127.0.0.1
== TCP RENO Mode ==
Round 1: cwnd = 1, ssthresh = 8
Sent packets: pkt0
Received: ACK: pkt0
Slow Start: cwnd -> 2

Round 2: cwnd = 2, ssthresh = 8
Sent packets: pkt1, pkt2
Received: ACK:pkt1
Received: ACK:pkt2
Slow Start: cwnd -> 4

Round 3: cwnd = 4, ssthresh = 8
Sent packets: pkt3, pkt4, pkt5, pkt6
Received: ACK:pkt3
Received: ACK:pkt4
Received: ACK:pkt5
Received: ACK:pkt6
==> 3 Duplicate ACKs: Fast Retransmit triggered.
TCP RENO Fast Recovery: cwnd -> 2


Round 4: cwnd = 2, ssthresh = 2
Sent packets: pkt4, pkt5
Received: ACK:pkt4
Received: ACK:pkt5
Congestion Avoidance: cwnd -> 3

Round 5: cwnd = 3, ssthresh = 2
Sent packets: 6, pkt7, pkt8
Received: ACK:pkt6
Received: ACK:pkt7
Received: ACK:pkt8
Congestion Avoidance: cwnd -> 4

Round 6: cwnd = 4, ssthresh = 2
Sent packets: 9, pkt10, pkt11, pkt12
Received: ACK:pkt9
Received: ACK:pkt10
Received: ACK:pkt11
Received: ACK:pkt12
Congestion Avoidance: cwnd -> 5

...
Client disconnected.
```

4 Submission as a Lab Report

Run simulations with TCP Tahoe and TCP Reno and collect performance metrics such as throughput, packet loss rate, and round-trip time (RTT). Compare the results obtained in TCP Tahoe and TCP Reno Algorithms. Analyze the differences and similarities between the two algorithms and provide a graphical analysis on trans-

mission round (x-axis) vs congestion window (y-axis) for both of these algorithms.

The report must documents the design and implementation of the algorithms, as well as the performance comparison under different network conditions between TCP Tahoe and TCP Reno.

5 Policy

Copying from the Internet, classmates, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.