Department of
Computer Science and Engineering

Implementation of TCP Congestion Control Mechanism: TCP Tahoe and TCP Reno

Computer Networking Lab

CSE 312

Green University of Bangladesh

1 Objective(s)

- To gather knowledge about how the TCP transport protocol controls the congestion of data when a sender or receiver detects a congestion in the link in-between them.
- To implement the TCP Tahoe congestion control mechanism using JAVA.
- To implement TCP Reno using the knowledge of Tahoe.

2 Problem analysis

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. Major internet applications such as the World Wide Web, email, remote administration, and file transfer all rely on TCP. TCP is connection-oriented, and a connection between client and server is established before data can be sent. The server must be listening (passive open) for connection requests from clients before a connection is established. Three-way handshake (active open), retransmission, and error-detection adds to reliability. Thus TCP can maintain various operations to establish perfect communications between a pair of hosts, e.g connection management, error detection, error recovery, congestion control, connection termination, flow control, etc. In this lab, we will have a look at the flow control mechanism of the TCP protocol.

2.1 TCP Congestion Control

2.1.1 Introduction

In data communications, TCP uses a congestion control mechanism which is the process of managing the rate of data transmission 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, that we have observed and implemented in our last lab. 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.

2.1.2 General Mechanism

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 senders send rate. To see this, consider a connection for which loss and packet transmission delays are negligible. Then, roughly, at the beginning of every **RTT**, the constraint permits the sender to send ***cwnd*** bytes of data into the connection; at the end of the **RTT** the sender receives acknowledgments for the data. Thus the senders send rate is roughly $\frac{\mathbf{cwnd}}{\mathbf{RTT}}$ *bytes/sec*. By adjusting the value of ***cwnd***, the sender can therefore adjust the rate at which it sends data into its connection, since a rough average constant **RTT** is followed through the entire application.

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 senders 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, TCPs 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.

2.1.3 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 $\frac{MSS}{RTT}$. For example, if **MSS** = 500 bytes and **RTT** = 200 msec, the resulting initial sending rate is only about **20 kbps**. Since the available bandwidth to the TCP sender may be much larger than $\frac{MSS}{RTT}$, the TCP sender would like to find the amount of available bandwidth quickly. Thus, in the slow-start state, the value of **cwnd** begins at 1 **MSS** and increases by 1 **MSS** every time a transmitted segment is first acknowledged. Figure **??** exemplifies the Slow Start phase.

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.

Now question may arise that when does this Slow Start stops. As we all know 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.

2.1.4 TCP Congestion Avoidance

Congestion Avoidance phase can be started from migration of Slow Start or simply when another congestion is detected by *triple duplicate acknowledgement*. On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered. 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.

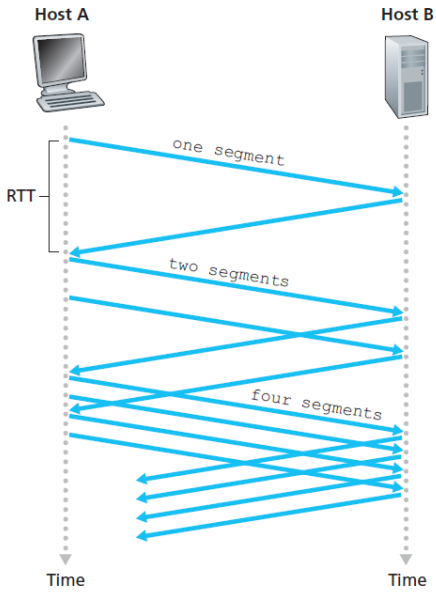


Figure 1: Slow Start phase of Congestion Control

Similar to the Slow Start, we should also consider when this linear increase of 1 **MSS** per **RTT** of Congestion Avoidance stops.

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.

2.1.5 TCP Fast Recovery

Fast Recovery phase starts whenever a congestion is detected. This implies whenever an ACK is not getting back to the sender. When this ACK is again received naturally, if the previous congestion is detected using *triple duplicate acknowledgement*, TCP starts the Congestion Avoidance phase where new starting **cwnd** value equals to a new ***ssthresh*** value that is ultimate equalling to the half of the value of previous **cwnd** value. In contrast, if a *timeout* event occurs for the previous congestion, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of **cwnd** is set to 1 **MSS**, and the value of ***ssthresh*** is set to half the value of **cwnd** when the loss event occurred at the previous congestion.

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

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

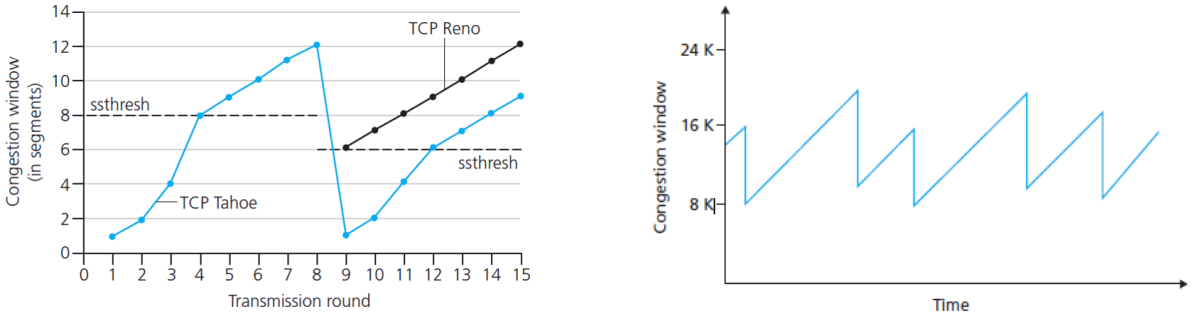
2. **TCP Reno:** This newer version mainly follows the Fast Recovery phase and restarts the sending of data using **cwnd** from half the before-congestion state.

2.1.6 General Overall Control Mechanism

If we consider all the three phases together, TCP congestion control mechanism controls the sending of data through a congestion-prone link using the shown algorithms. Using an FSM diagram, relation and transition among them are clearly shown in Figure 3. This is a general relation, whereas a little variation might be observed due to little difference between TCP Tahoe and TCP Reno.

An example graph is given in Figure 2a, 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.

So all in all following the summarized picture of the Control mechanism, TCP follows a saw-tooth approach for this. Ignoring the initial slow-start period when a connection begins and assuming that losses are indicated by triple duplicate ACKs rather than timeouts, TCPs congestion control consists of linear (additive) increase in **cwnd** of 1 **MSS** per **RTT** and then a halving (multiplicative decrease) of **cwnd** on a triple duplicate ACK event. For this reason, TCP congestion control is often referred to as an additive increase, multiplicative



(a) (b)

Figure 2: (a) An example graph of transmission round vs **cwnd** denoting phases of Congestion Control (b) The general AIMD mechanism of Congestion Control of TCP

decrease (AIMD) form of congestion control. AIMD congestion control gives rise to the"*saw tooth*" behavior shown in Figure 2b. This can also be denoted as "*probing* for bandwidth" where TCP linearly increases its congestion window size (and hence its transmission rate) until a triple duplicate ACK event occurs. It then decreases its congestion window size by a factor of two but then again begins increasing it linearly, probing to see if there is additional available bandwidth. Finally a FSM diagram in Figure 3 shows the transition between the three phases/stages of TCP Congestion Control mechanism in details.

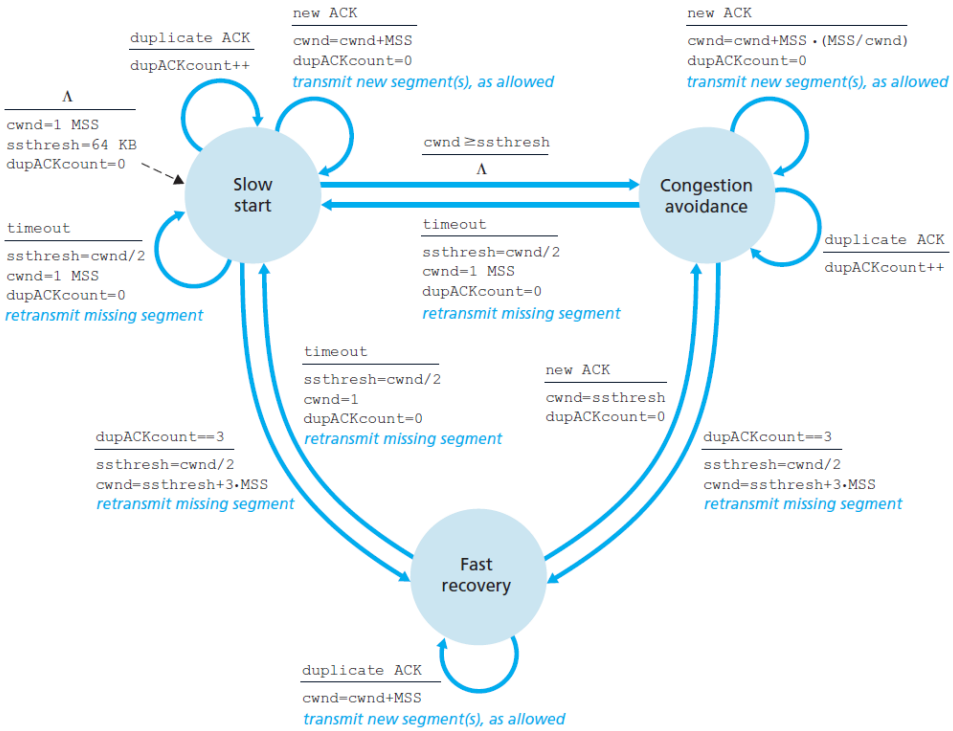


Figure 3: Finite State machine diagram to view relation and transitions among the phases of Congestion Control mechanism of TCP

In this Lab Manual we will be completing and understanding the TCP Tahoe of the TCP Fast Recovery phase from the TCP Congestion Control in its entirety.

3 Procedure: TCP Reno

TCP Reno is a congestion control mechanism used in the Transmission Control Protocol **(TCP)**. It is one of the earliest congestion control algorithms developed for TCP and forms the basis for many subsequent TCP

variants. The TCP Reno algorithm aims to regulate the rate at which data is sent over a network to avoid congestion and ensure fair bandwidth allocation among competing connections.

TCP Reno operates in three main phases: **slow start, congestion avoidance,** and **fast recovery.**

1. **Slow Start Phase:** In the slow start phase, the sender initializes its **congestion window (cwnd)** to a small value, usually one segment. It then exponentially increases the cwnd for each acknowledgment received, effectively doubling its size.

2. **Congestion Avoidance Phase:** Once the cwnd reaches a predefined threshold called **the slow start threshold (ssthresh)**, TCP Tahoe enters the congestion avoidance phase. In this phase, the sender increases the cwnd linearly by one segment for every acknowledgment received. This linear growth strategy helps prevent congestion by gradually increasing the sending rate.

3. **Fast Recovery Phase:** If a packet loss is detected, TCP Reno enters the fast recovery phase. During this phase, the sender assumes that a network congestion event has occurred and takes appropriate actions to recover from it.

   The fast recovery phase involves the returning of the value of **cwnd** to a state of tolerance, once a congestion is detected. As we already know, congestion might be detected using two ways. **TCP Reno** maintains two possible recoveries on the basis of these congestion detections.

   (a) Detection using timeout: Reno recovers by making the **cwnd** value come straight back to 1. The new **ssthresh** value will be half of the previous **cwnd** value immediately before congestion detection. New **Slow Start** phase restarts from here then.

   (b) Detection using triple duplicate acknowledgement: Reno recovers this just a bit differently. The value of **cwnd** is calculated to half. The value of **ssthresh** is also this same value - half of the previous **cwnd** value. New **Congestion Avoidance** phase starts from here.

   After both of these recoveries, **TCP Reno** again resumes sending of data and carries on with a new cycle.

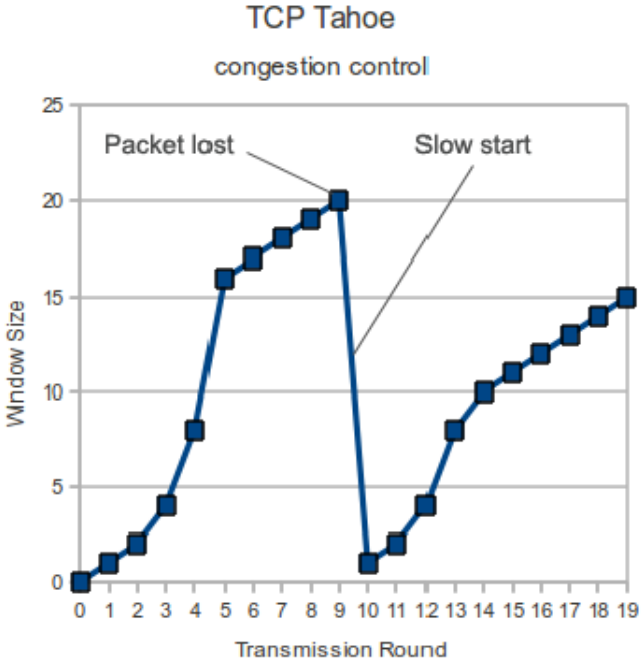


Figure 4: Tahoe TCP

4 Algorithm: TCP Reno

In this section, we observe the algorithm of the **TCP Reno** of the congestion control mechanism along with the mentioned three phases using some formal steps, to visualize this **Reno** technique.

/* Initialization: */

Step 1: Begin
Step 2: cwnd = 1 /* Congestion window size */
Step 3: ssthresh = according to the given value by user /* Slow start threshold */
/* Main Loop starting now */

Run():
Step 4: while (true)
Step 5: if (cwnd < ssthresh)
/* Slow Start Phase */
Step 6: cwnd = cwnd * 2
Step 5: else if(cwnd >= ssthresh)
/* Congestion Avoidance Phase */
Step 6: cwnd = cwnd + 1
Step 7: trying to send the data packets using sendPacket() function.

sendPacket():
Step 8: if(data not received by receiver)
Step 9: congestion = true;
Step 10: if(congestion detected by timeout)
Step 11: handleTimeout() function to handle the timeout congestion.
Step 10: else Step 11: handleDuplicate() function to handle the duplicate congestion.
Step 8: else congestion = false;

function handleTimeout():
/* severe congestion. cwnd will start from slow start with value 1 */
Step 12: ssthresh = cwnd / 2;
/* Reduce slow start threshold to half of cwnd */
Step 13: cwnd = 1;
/* Reset cwnd to 1 for slow start */
Step 14: retransmitPacket() function to after recovery;

function handle3AckCongestion():
/* light congestion. cwnd will start from half of the ssthresh value */
Step 15: ssthresh = cwnd / 2;
Step 16: cwnd = ssthresh;
Step 17: retransmitPacket() function to after recovery;

function retransmitPacket():
Step 18: Retransmit the lost packet making congestion = false

function receiveAcknowledgment()
/* Receive the acknowledgment */
Step 19: randomly check whether acknowledgemnt is received by sender randomly

5 Implementation in Java

```
1
2  package tcpcongestioncontrol;
3  import java.util.*;
4
5  public class TCPCongestionControl
6  {
7      // This code is designed for Congestion Control mechanism of TCP Reno.
           Change this code, if you want TCP Tahoe.
8      // In Reno, when congestion detected using Triple Dup Ack, cwnd value
           becomes half only.
9      // In Tahoe, cwnd becomes 1 for when congestion detected both using Triple
           Dup Ack and using Timeout.
```

```
10
11      private int cwnd;
12      private int ssthresh;
13      private int rtt;
14      private boolean congestion;
15
16      public TCPCongestionControl(int init_ssthresh)
17      {
18          cwnd = 1;
19          ssthresh = init_ssthresh;
20          congestion = false;
21          rtt = 0;
22      }
23
24      public void run()
25      {
26          System.out.println("Connected to the Server... ...");
27          System.out.println("Enter the length of your data: ");
28          Scanner scan = new Scanner(System.in);
29          int len = scan.nextInt();
30          int dataSeqNum = 0;
31          System.out.println("Your data is started to be sent ... ");
32
33          while(dataSeqNum < len)
34          {
35              this.rtt++;
36              System.out.println(); System.out.println();
37              System.out.println("Data sending in RTT number "+this.rtt);
38              System.out.println("----------------------------");
39              System.out.println("previous cwnd size: "+ cwnd);
40              System.out.println("updated ssthresh value: " + ssthresh);
41              if (!congestion)
42              {
43                  if (cwnd < ssthresh)
44                  {
45                      // Slow Start Phase
46                      // Exponentially increase of cwnd
47                      cwnd = cwnd * 2;
48                      System.out.println("...SS phase running...");
49                  }
50                  else if(cwnd >= ssthresh)
51                  {
52                      // Congestion Avoidance Phase
53                      // Linearly increase of cwnd
54                      cwnd = cwnd + 1;
55                      System.out.println("...CA phase running...");
56                  }
57              }
58              System.out.println("updated cwnd size: "+ cwnd);
59
60              sendPacket(dataSeqNum);
61
62              dataSeqNum = dataSeqNum + cwnd;
63          }
64          System.out.println("\n\nYour data sending is completed. No more data to
                send."
65                  + "\nCongestion Control mechanism concludes.\nIt took "+this.rtt
                        --+" transmission rounds to send the whole data.");
```

```
66      }
67
68      public void sendPacket(int dataSeqNum) // A function to deal with the
            congestion part
69      {
70          System.out.println("Data from " + (dataSeqNum+1) + " - " + (dataSeqNum+
                cwnd)+" is being sent now... ...\n\n");
71          if(!receiveAcknowledgment())
72          {
73              congestion = true;
74              System.out.println("... but wait ! congestion has been detected !");
75              if(timeout())
76              {
77                  handleTimeoutCongestion();
78              }
79              else
80              {
81                  handle3DupAckCongestion();
82              }
83          }
84          else
85          {
86              congestion = false;
87          }
88      }
89
90      public boolean receiveAcknowledgment() // A function to generate congestion
            randomly
91      {
92          //returns true if no congestion and data is received by receiver.
93          //returns false if congestion occurred and data is not received by
                receiver.
94          Random ack = new Random();
95          return ack.nextBoolean();
96      }
97
98      public boolean timeout()  // A function to decide randonmly if congestion is
             detected using timeout or duplicateAck
99      {
100         //returns true if congestion is detected using timeout.
101         Random rttRandom = new Random();
102         return rttRandom.nextBoolean();
103     }
104
105     public void handleTimeoutCongestion() // Function to handle when congestion
            by timeout
106     {
107         // severe congestion. cwnd will start from slow start with value 1
108         System.out.println("\n\nTimeout occured.Handling Timeout based
                congestion: cwnd value will become 1.");
109         ssthresh = cwnd / 2;
110         if (ssthresh==0) ssthresh = 1; // making ssthresh 1, if it comes as zero
                .
111         cwnd = 1;
112
113         retransmitPacket();
114         // we have to make the congestion variable false, since we have
                recovered using the handling process here.
```

```
115     }
116
117     public void handle3DupAckCongestion() // Function to handle when congestion
            by duplicateAck
118     {
119         // light congestion. cwnd will start from half of the ssthresh value
120         // This is the main difference between TCP Reno and TCP Tahoe.
121
122         System.out.println("\n\nHandling Triple Dup Ack based congestion: cwnd
                value will be halved.");
123         ssthresh = cwnd / 2;
124         if (ssthresh==0) ssthresh = 1; // making ssthresh 1, if it comes as zero
                .
125         cwnd = ssthresh;
126
127         retransmitPacket();
128         // we have to make the congestion variable false, since we ahve
                recovered using the handling process here.
129     }
130
131     public void retransmitPacket() // Function to know that congestion has been
            recovered.
132     {
133         congestion = false;
134         System.out.println("\nRetransmiting the lost packet now after handling.\
                n");
135     }
136
137     public static void main(String[] args)
138     {
139         Scanner scn = new Scanner(System.in);
140         System.out.println("Please input the initial ssthresh value: ");
141         int ssthresh = scn.nextInt();
142
143         TCPCongestionControl reno = new TCPCongestionControl(ssthresh);
144         reno.run();
145     }
146 }
```

6 Discussion & Conclusion

In summary, TCP uses a comprehensive and elaborate mechanism to control the flow of data from a host to another once a congestion of a link is detected. Three phases help the hosts to exponentially increase, then linear increase or a fast decrease of the congestion window to maintain the rate of sent data accordingly.

Based on the focused objectives to understand about TCP and its congestion control mechanism, the additional thinking exercises and Lab tasks as viva from the instructors and Lab reports will make the student more confident towards the fulfilment of the objectives.

7 Lab Tasks

- Implement TCP Tahoe in java.

  TCP Tahoe: This is another type of Fast Recovery of the Control Mechanism. The primary goal of TCP Tahoe is to manage network congestion by dynamically adjusting the sending rate of data packets based

on the perceived level of congestion in the network. It achieves this using exaclty the same mentality of TCP Reno, jus ta bit differently in the last step.

1. **Slow Start:** In this phase, the sender gradually increases the transmission rate exponentially by doubling the congestion window size with each successful acknowledgment received.
2. **Congestion Avoidance:** Once the slow-start phase reaches a certain threshold, the congestion window is increased linearly, incrementing it by a fixed amount for every round-trip time (RTT) that passes without congestion being detected.
3. **Fast Recovery:** TCP Tahoe employs the fast recovery mechanism. When a packet loss is detected, whether congestion is detected using timeout or triple duplicate acknowledgement, **TCP Tahoe reduces the congestion window to exactly 1**. It then enters the fast recovery state, allowing the sender to continue transmitting packets and retransmit the missing packet(s) using Slow Start phase.

- Implement using a full working algorithm and code using JAVA language.

8 Lab Exercise (Submit as a report)

Please construct a Lab report of a scenario, where using examples and figure both the implementations of Flow and Congestion Controls could be demonstrated, using the same programming code of JAVA.

9 Policy

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