ECE361S Lab 4 Instructions:

TCP congestion control (CC)

The TCP congestion control scheme is designed to estimate the connection's round trip time (RTT) and based on that, estimate the time out interval (`timeOut`), adjust the transmission rate (by selecting a proper congestion window (`cwnd`), so that the bandwidth is used efficiently and fairly, while avoiding network congestion. The relations that TCP uses for estimating the timeout interval are as follows:

```
DevRTT = (int)((1-beta)*DevRTT + beta*Math.abs(SampleRTT-EstimatedRTT));
EstimatedRTT = (int)((1-alpha)*EstimatedRTT+alpha*SampleRTT);
timeOut =EstimatedRTT + 4*DevRTT;
```

The TCP congestion control scheme consists of 3 phases:

1. Slow start (doubling the `cwnd` after each successful transmission of the whole window; upon timeout, make `cwnd`=1)
2. Congestion avoidance (increasing `cwnd` linearly once reaching `ssthresh`)
3. Fast Recovery (happens once three duplicate ACKs are received before the timeout; cut the `cwnd` in half and go to congestion avoidance)

You can find more details of these three states in the text book (Section 3.7). Here we will consider a simplified version of the protocol.

Implementing simplified TCP Congestion control

In this lab, you will be implementing a simplified version of TCP congestion control scheme. Similar to previous Lab instructions, you will be interacting with a server provided to you. You should write a client that transmits a given number of packets to the server. The server is simulating a limited bandwidth (which is assumed to be fixed for simplicity). Therefore, you should decide on the proper rate to send the packets. If the rate goes above the allotted server bandwidth, the packets will be dropped and no acknowledgement will be received. If the rate stays very small, network bandwidth is wasted. Your client should be receiving acknowledgements of the transmitted packets. Upon detection of a lost packet (i.e., timeout) the client should retransmit the lost packets and also reduce the transmission rate, according to the TCP protocol.

TCP uses a window size called congestion window (`cwnd`) to adjust the transmission rate. The used scheme is somehow similar to Go-Back-N protocol (ARQ), that you did in the previous lab. However, this time the window size, i.e., `cwnd`, is adjusted based on the available bandwidth.

Note that unlike the ARQ Lab, this time the packets are not lost due to a loss probability. The packets are lost if congestion happens. Congestion happens if the data transmission rate goes above the network bandwidth.

The following are the simplified assumptions:

- Assume packet sizes are fixed = MSS bytes. Since we don't really send data in this scenario, assume that the packet only contains a sequence number value (ranging from `0-255` i.e., MSS=1 byte).
- Assume that the `RTT` is fixed to be `1000` milliseconds, so that `sampleRTT=EstimatedRTT`, and hence, `DevRTT=0`, leading to `TimeOutInterval` =`RTT`. In other words, you do not need to estimate the timeout via the formulas above (estimating the timeout is given to you as an optional task). It is recommended to choose a `TimeOutInterval` value slightly greater than RTT since the processing time and actual underlying network delay is not considered in the simulation on the Server.
- The connection bandwidth is assumed to be fixed to a random value ranging from 4 to 16 MSS/RTT.
- You can avoid considering triple duplicate ACKs. Just care for `timeOut` (as an optional deliverable, you can add that part for more efficiency).
- Assume receiver window (`rwnd`) is so large, so that the amount of unacknowledged data is only limited by the congestion window (`cwnd`).
- Assume that the receiver (server) does not buffer the out of order packets, so that if packet n is not received by the receiver, while n+1, n+2, … are received, the receiver simply discards those packets, hence in effect you are developing Go-Back-N ARQ. In practice, most of the implementations of TCP do the buffering since it is more efficient.
- Due to the simplified conditions, you can send bursts of data (of size `cwnd [MSS]`) and then wait until receiving all the ACKs or a timeout happens. As a result, in slow start state, `cwnd` is doubled once all ACKs are received. In congestion avoidance state, `cwnd` is increased by one once all ACKs are received.
  However, upon receiving the first ACK, similar to Go-Back-N scenario, sender is allowed to send more packets. The `cwnd` is also updated upon receiving each single ACK packet. If the system is in slow start state, `cwnd` should be increased by one (i.e., `cwnd= cwnd + 1` [MSS]) after receiving each ACK. If the system is in congestion avoidance state, cwnd should be increased by `1/cwnd` (i.e., `cwnd = cwnd + 1/cwnd` [MSS]).

Scenario:

- Define a `Socket` variable and connect to the server, similar to previous Labs.
- Ask the user for the number of packets to be sent. You can use `Scanner scanner=new Scanner(System.in)` along with the `scanner.nextInt()` function to read an integer value.
- Name the read integer as `NoPackets` and send it to the server via DataOutputStream writer = new `DataOutputStream(socket.getOutputStream())` along with the `write(int b)` function.
- Initially set `cwnd` to 1.
- Start transmitting the packets; send `cwnd` packets of size MSS bytes (i.e. just the sequence number of the packet, starting form 1).
- Once the `cwnd` packets are sent, wait until you receive all of the sent packets' ACKs, or `timeout` for one of the packets happens. In order to do that you should read ACK packets on a separate thread.

- Define a Listener thread (similar to previous lab) to listen to the ACK packets. Once an ACK packet is received, update the number of last received ACK number (`lastAck`) in the main Thread.
- Adjust `cwnd` based on TCP Tahoe, i.e., upon timeout always set `ssthresh` to `cwnd/2`, set `cwnd` to `1`, and go to slow start (increase `cwnd` exponentially until `ssthresh` is reached and increase `cwnd` linearly afterwards).
- Go back to 5 until all of the packets are sent and their ACKs are received.
- Don't forget to close the socket after transmission is done.

Deliverables:

- Calculate the performance of the system, in terms of number of needed RTTs, total transmission time in seconds and average transmission rate. Find the throughput of the system based on the server's given bandwidth (it will be printed at the end of the transmission, once all packets are received successfully and the socket is closed properly)
- Try the above scheme for different `ssthresh` values. Compare the performance of the system for different `ssthresh` values.
- What property should a good `ssthresh` value have?
- What is the throughput of the system in this implementation? Is it different from the theoretical TCP throughput? Explain.

Optional deliverables:

- Implement the TCP Reno version, by detecting the triple-duplicate-ACKs and going to fast recovery phase.
- What is throughput of the system for TCP Reno?
- Instead of using fixed timeouts, estimate them using the given formulas.
- What is the trend of variations of `estimatedRTT`?
- What is the trend of variations of `TimeOut`?

Sample client output:	Sample Server output
$ java CCClient Connected to : localhost:9876 Enter number of packets to be sent to the server [0-127], 0 to Quit: 20 cwnd= 1 sending packet no:1 Received ack no:1 Received ack no:1 last ack:1 # of acks received for cwnd of 1 cwnd= 2 sending packet no:2 sending packet no:3 Received ack no:2 Received ack no:3 last ack:3 # of acks received for cwnd of 2 cwnd= 4	request received. request number: 1 client: /127.0.0.1:52299 connection established: service type:CC_SERVER mode:VERBOSE client id:1 socket: Socket[addr=/127.0.0.1,port=52299,localport=9876] Client Connected ... [23:08:23] 1: Number of packets to be received. from 20 [23:08:23] 1: Received packet#1 [23:08:23] 1: Acknowledging packet #1 [23:08:23] 1: Received packet#1 [23:08:23] 1: Acknowledging packet #1 [23:08:24] 1: Received packet#2 [23:08:24] 1: Acknowledging packet #2 [23:08:24] 1: Received packet#3 [23:08:24] 1: Acknowledging packet #3 [23:08:25] 1: Received packet#4

sending packet no:4 sending packet no:5 sending packet no:6 sending packet no:7 Received ack no:4 Received ack no:5 Received ack no:6 Received ack no:7 last ack:7 # of acks received for cwnd of 4 cwnd= 8 sending packet no:8 sending packet no:9 sending packet no:10 sending packet no:11 sending packet no:12 sending packet no:13 sending packet no:14 sending packet no:15 Received ack no:8 Received ack no:9 Received ack no:10 Received ack no:11 Received ack no:12 Received ack no:13 Received ack no:14 Received ack no:15 last ack:15 # of acks received for cwnd of 8 cwnd= 9 sending packet no:16 sending packet no:17 sending packet no:18 sending packet no:19 sending packet no:20 Received ack no:16 Received ack no:17 Received ack no:18 Received ack no:19 Received ack no:20 last ack:20 # of acks received for cwnd of 9 Total time to send all packets: 5 seconds. Total time in terms of RTT: 5 RTT. 20 out of 20 packets have been sent successfully Quitting...	[23:08:25] 1: Acknowledging packet #4 [23:08:25] 1: Received packet#5 [23:08:25] 1: Acknowledging packet #5 [23:08:25] 1: Received packet#6 [23:08:25] 1: Acknowledging packet #6 [23:08:25] 1: Received packet#7 [23:08:25] 1: Acknowledging packet #7 [23:08:26] 1: Received packet#8 [23:08:26] 1: Acknowledging packet #8 [23:08:26] 1: Received packet#9 [23:08:26] 1: Acknowledging packet #9 [23:08:26] 1: Received packet#10 [23:08:26] 1: Acknowledging packet #10 [23:08:26] 1: Received packet#11 [23:08:26] 1: Acknowledging packet #11 [23:08:26] 1: Received packet#12 [23:08:26] 1: Acknowledging packet #12 [23:08:26] 1: Received packet#13 [23:08:26] 1: Acknowledging packet #13 [23:08:26] 1: Received packet#14 [23:08:26] 1: Acknowledging packet #14 [23:08:26] 1: Received packet#15 [23:08:26] 1: Acknowledging packet #15 [23:08:27] 1: Received packet#16 [23:08:27] 1: Acknowledging packet #16 [23:08:27] 1: Received packet#17 [23:08:27] 1: Acknowledging packet #17 [23:08:27] 1: Received packet#18 [23:08:27] 1: Acknowledging packet #18 [23:08:27] 1: Received packet#19 [23:08:27] 1: Acknowledging packet #19 [23:08:27] 1: Received packet#20 [23:08:27] 1: Acknowledging packet #20 [23:08:28] 20 out of 20 packets have been received successfully [23:08:28] Total bandwidth: 9 MSS/RTT Total number of packets: 20 Total transmission time: 5081 seconds. Average round trip time: 1000 Minimum possible transmission time: 2.2222222222222223 RTT. Expected transmission time using TCP Tahoe (for initial ssthres= 16):5 RTT. Actual transmission time: ~5.081 RTT. Actual throughput: 0.4373592249994533 connection to 1 closed.

Goodluck. ☺