CSEE 4119 Fall 2022
Homework 3 ANSWER KEY

Assigned: 10/18/2022

Due: Monday 10/31/2022, 11:59pm.

Uploading course materials, including questions from this homework, to sites such as CourseHero, Chegg or Github is academic misconduct at Columbia (see pg 10).

Question 1: Ol' Reliable [28 points, parts a-j]

1. [1 point] In our rdt protocols, why did we need to introduce sequence numbers?

Sequence numbers are required for a receiver to find out whether an arriving packet contains new data or is a retransmission.

+1 for distinguishing new data and retransmission.

2. [2 points] In our rdt protocols, why did we need to introduce timers to the sender? For example, why isn't it straightforward to design the protocol such that the host waits for the other host to notify it that something is wrong?

To handle losses in the channel. If the ACK for a transmitted packet is not received within the duration of the timer for the packet, the packet (or its ACK or NACK) is assumed to have been lost. Hence, the packet is retransmitted. Basically, with packet loss, we will need a timer. But if the receiver knows what to expect , like the sender sending packets with fixed intervals or numbers, the receiver will be able to tell whether something is wrong and remind the sender. In this case you will still need a timer for the receiver. But if the receiver can't know what to expect from the sender, even with a timer for the receiver, the protocol can't work properly. You have to set the timer for the sender.

+1 for handling losses.

+1 for deeper discussion why receiver timer is not sufficient

3. [4 points] Consider a channel that can lose packets but has a maximum delay that is known. Modify protocol rdt2.1 to include sender timeout and retransmit. Explain why your protocol can communicate correctly over this channel. Notice that you can't modify receiver behavior of rdt2.1 which means your receiver will respond with ACK or NAK in exactly the manner described by the rdt2.1 receiver FSM.

Here, we add a timer, whose value is greater than the known round-trip propagation delay. We add a timeout event to the "Wait for ACK or NAK0" and "Wait for ACK or NAK1" states. If the timeout event occurs, the most recently transmitted packet is retransmitted. Let us see why this protocol will still work with the rdt2.1 receiver.

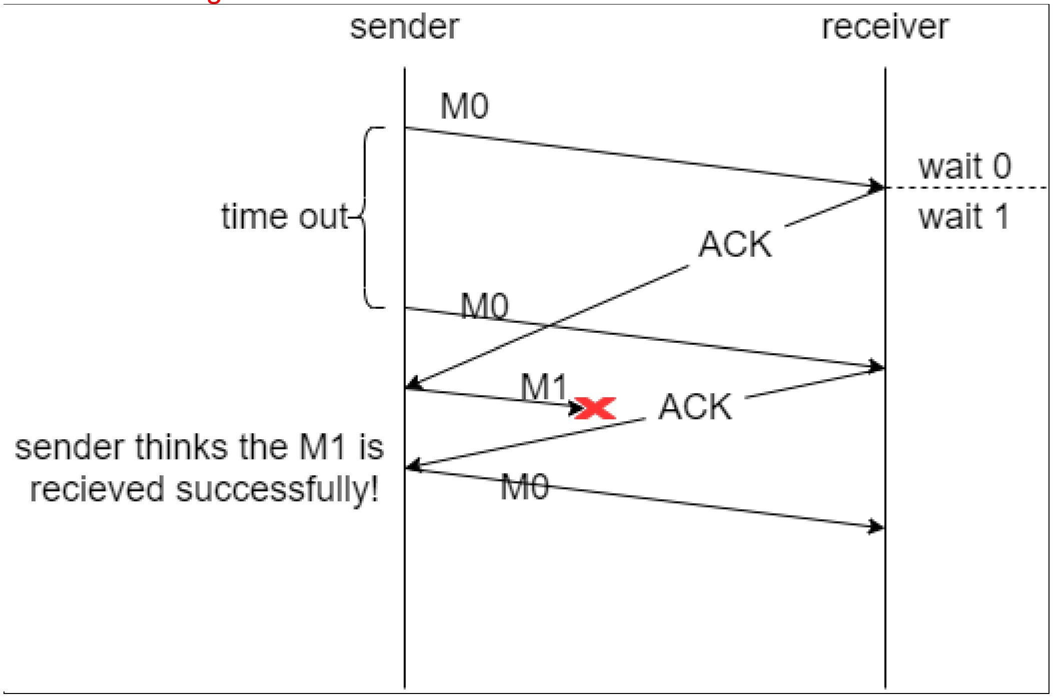
- Suppose the timeout is caused by a lost data packet, i.e., a packet on the sender-to-receiver channel. In this case, the receiver never received the previous transmission and, from the receiver's viewpoint, if the timeout retransmission is received, it looks exactly the same as if the original transmission is being received.

- Suppose now that an ACK is lost. The receiver will eventually retransmit the packet on a timeout. But a retransmission is exactly the same action that if an ACK is garbled. Thus the sender's reaction is the same with a loss, as with a garbled ACK. The rdt 2.1 receiver can already handle the case of a garbled ACK.

+2 for adding timer correctly
+2 for reasonable explanation

4. [4 points] If we don't know the maximum delay, will your protocol in (c) still work? If not, draw a diagram to show a counterexample.

No. See the diagram.

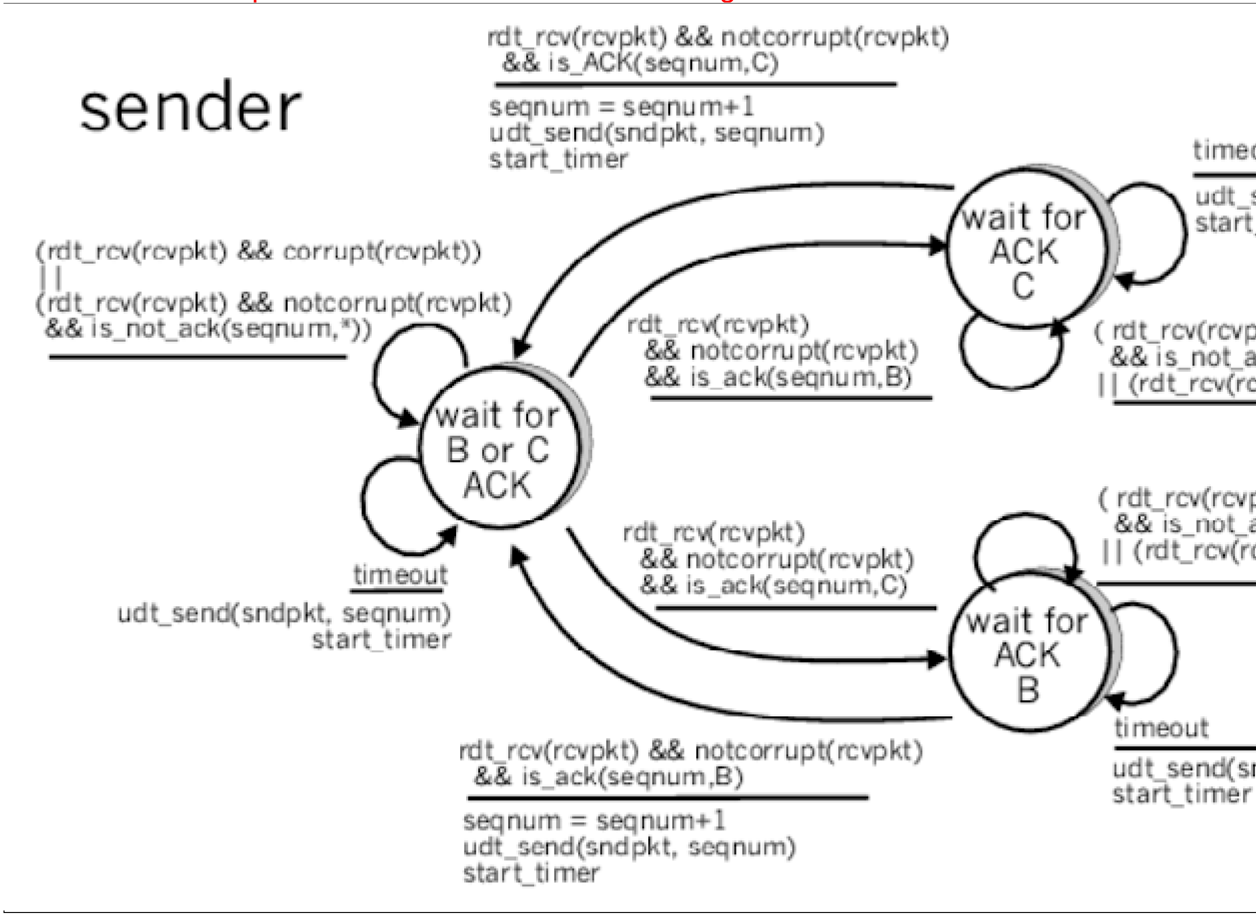


+1 for answering No.
+3 for correct counterexample diagram

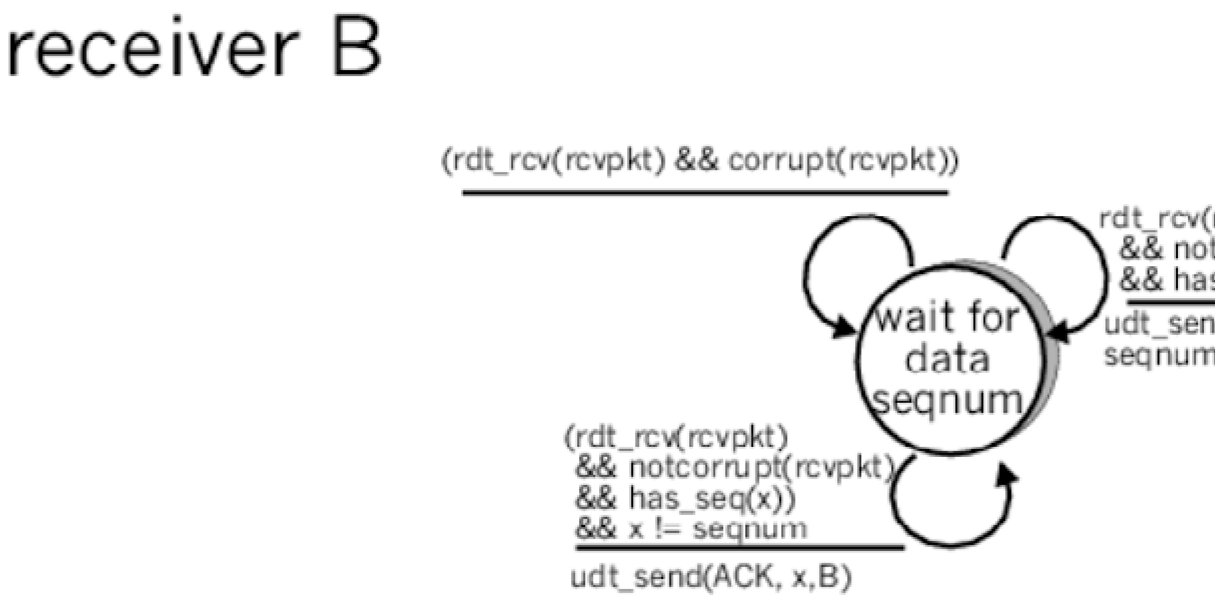
5. [6 points] Consider a scenario in which Host A wants to simultaneously send packets to Hosts B and C. A is connected to B and C via a broadcast channel – a packet sent by A is carried by the channel to both B and C. Suppose that the broadcast channel connecting A, B, and C can independently lose and corrupt packets (and so, for example, a packet sent from A might be correctly received by B, but not by C). Design a stop-and-wait-like error-control protocol for reliably transferring packets from A to B and C, such that A will not get new data from the upper layer until it knows that both B and C have correctly received the current packet. Give FSM descriptions of A and C. (Hint: The FSM for B should be essentially the same for C.) Also, give a description of the packet format(s) used.

This problem is a variation on the simple stop-and-wait protocol (rdt3.0). Because the channel may lose messages and because the sender may resend a message that one of the receivers has already received (either because of a premature timeout or because the other receiver has yet to receive the data correctly), sequence numbers are needed. As in rdt3.0, a 1-

bit sequence number will suffice here. In addition, for detecting packet corruption, a checksum field is needed.
The FSMs of sender A and receiver B (= receiver C) are shown below. In this problem, the sender state indicates whether the sender has received an ACK from B (only), from C (only) or from neither C nor B. The receiver state indicates which sequence number the receiver is waiting for.


sender
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && is_ACK(seqnum,C)
seqnum = seqnum+1
udt_send(sndpkt, seqnum)
start_timer
(rdt_rcv(rcvpkt) && corrupt(rcvpkt)) || (rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && is_not_ack(seqnum,*))
wait for ACK C
wait for B or C ACK
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && is_ack(seqnum,B)
timeout
udt_send(sndpkt, seqnum)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && is_ack(seqnum,C)
wait for ACK B
timeout
udt_send(s
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && is_ack(seqnum,B)
seqnum = seqnum+1
udt_send(sndpkt, seqnum)
start_timer

Receiver



To get full credits, it needs to provide a description of the packet format. For example,

Source Addr.	Destination Addr.	Sequence # (0 or 1)	ACK (0 or 1)	Checksum	Data

+1 for correct sender states
+0.5 for correct transition from “wait for B or C ACK” to “wait for ACK B” and “C”
+0.5 for correct transition from “wait for ACK B” and “C” to “wait for B or C ACK”
+0.5 for timeout transitions
+0.5 for wrong ack seq# transitions
+3 for fully correct sender FSM
+0.5 for correct receiver states
+0.5 for correct packet transition
+0.5 for wrong seq# packet transition
+0.5 for corrupted packet transition
+2 for fully correct receiver FSM
+0.5 for packet design with minor errors
+1 for correct packet format design

6. [2 points] From class, you should already know the utilization for “stop and wait” mechanism without failures is $\frac{L/R}{RTT + L/R}$, given L is the packet length (ignore the length of ACK packet), R the transmission rate, RTT the round trip time between sender and receiver. Now let’s consider how failure affects utilization. We will consider a modified definition of utilization-under-failures, considering failures -- define utilization-under-failures as the (expected) fraction of time the receiver is (successfully) receiving packets (so packets that are sent but lost are equivalent to idle periods). Suppose each packet has the probability of p to fail, the ACK packet never fails, and the timeout is set to RTT (so after 1 RTT, the sender either receives an ACK or retransmits the packet). Derive the equation of utilization-under-

failures for the stop and wait mechanism with failure. (*Hint: you can try to first derive the average time used to successfully transmit one packet.*)

$$(1-p)\frac{L/R}{RTT+L/R}$$

Since the probability of successfully transmitting the packet is 1-p (see this link for detailed calculation), the average time of successfully transmitting a packet is $\frac{1}{1-p}(RTT+L/R)$. Therefore, the utilization-under-failures is

$$(1-p)\frac{L/R}{RTT+L/R}$$

+0.5 for giving out correct average tries to successfully sending one packet
+1 for giving out correct average transmitting time of one packet
+2 for giving out correct formula

7. [1 point] Now consider the pipelining mechanism. Consider an alternative "transmit-N" protocol that sends *N* packets sequentially before stopping to wait for an ACK. The receiver expects *N* packets and will respond either with an ACK saying that all *N* packets were retrieved successfully or nothing at all (and the sender will timeout after one RTT and retransmit all *N* packets). First, suppose the packets never fail. Find a formula for the utilization without failures in this case.

$\frac{NL/R}{NL/R+RTT}$. Note that all N packets must be received before an ACK is sent. This is different from the example on slide 56 of the reliable data transport slides. In that case, an ACK is sent for each packet and so the next round of transmissions starts after the first ACK. In this case, all N packets must be received before the ACK is sent.
+0.5 for correct formula of the time used for sending out N packets
+1 for correct formula

8. [2 points] Assume each packet fails independently with probability p (but the ACK packet never fails). Find the utilization-under-failures for this "transmit-N" protocol.

$(1-p)^N\frac{NL/R}{NL/R+RTT}$. The average time of successfully transmitting N packets is $\frac{1}{(1-p)^N}(NL/R+RTT)$, therefore the utilization-under-failures is $(1-p)^N\frac{NL/R}{NL/R+RTT}$
+0.5 for correct average tries to successfully sending out N packets
+1 for correct average time for transmitting N packets
+2 for correct final answer

9. [4 points] Numerical calculations. Assume R = 10^6 bps, L = 4000 bits, RTT = 10 ms. Use your results from (f) and (h) to calculate the following.
    1. Assume a failure probability p = 0.01. Calculate the utilization-under-failures for:
        1. [1 point] The rdt 3.0 protocol

        28.29%
        +1 for correct answer based on formula

    2. [1 point] The transmit-N protocol with N = 10
  72.35%
  +1 for correct answer based on formula
2. Consider a more lossy link with failure probability p = 0.3. Calculate the utilization-under-failures for:
    3. [1 point] The rdt 3.0 protocol
  20%
  +1 for correct answer based on formula
    4. [1 point] The transmit-N protocol with N = 10
  2.26%
  +1 for correct answer based on formula

10. [2 points] Does the “transmit-N” protocol make sense for a high-loss medium? Justify your answer (1-2 sentences).

The transmit-N protocol does not make sense for a high-loss medium. Since each individual packet has a chance of failure, any one failure forces the protocol to resend all N packets, the failure probability gets very high and the sender often has to retransmit everything, as seen with these numerical results.

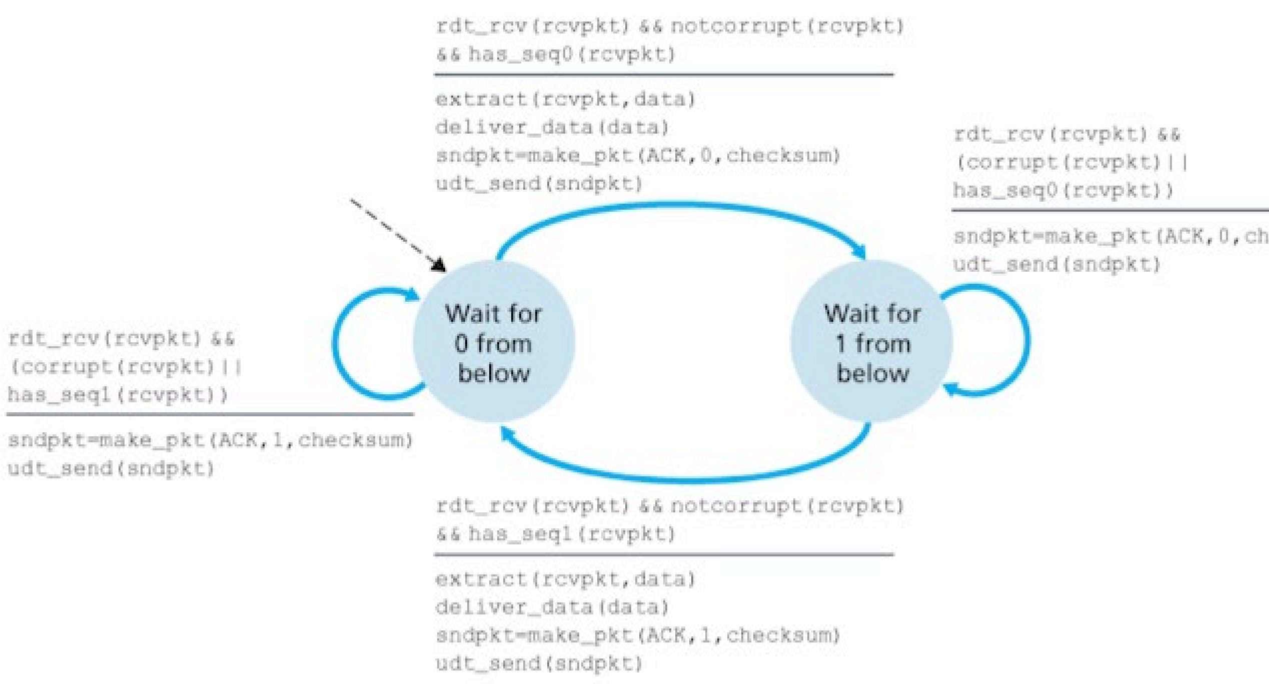
+1 for correct answer, but vague explanation

+2 for correct answer with good explanation

Question 2: Go-Back-N and Selective Repeat [26 points, parts a-c]

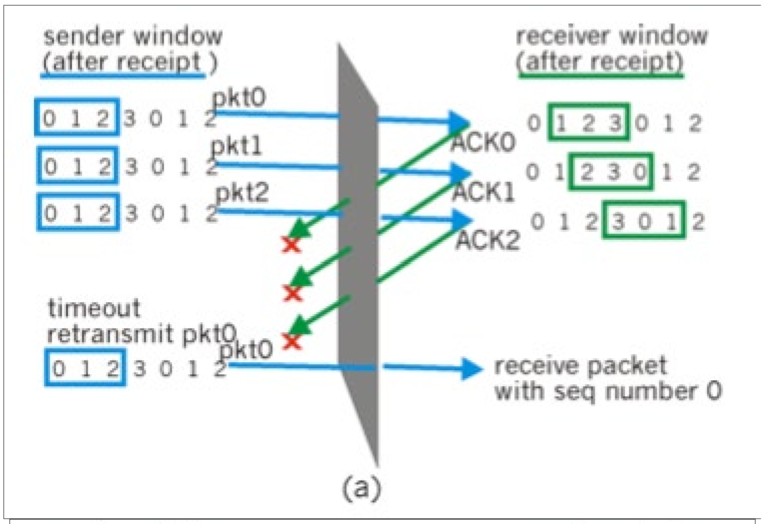
1. [4 points] In the textbook, we already have the FSM for the sender side of rdt3.0. Draw the receiver side FSM of rdt3.0. To get full credit, please include all the conditions and actions in your answer.

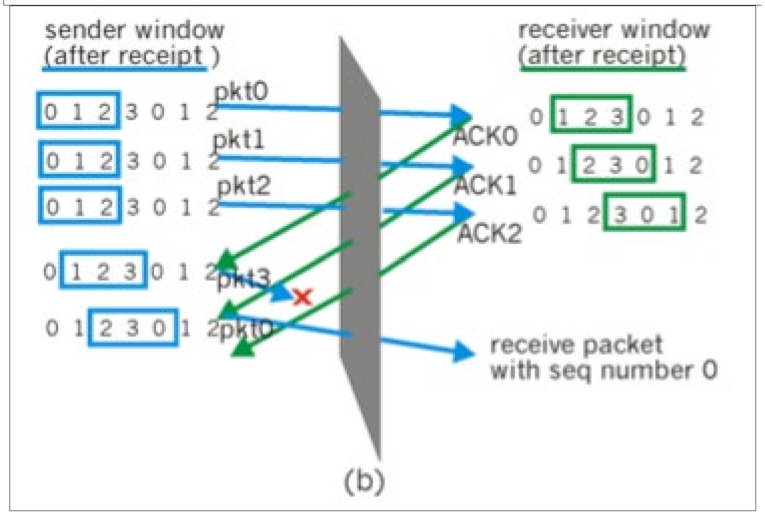
FSM of receiver side, each credit corresponds to a transition.


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,ch
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq1(rcvpkt))
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)

2. [10 points]Consider SR and GBN protocols. Assume that the sequence number space for each protocol is of size n. Relative to this sequence space of size n, what is the largest allowable sender window that can be used safely while allowing the protocol to operate properly and to exchange messages without ambiguity, in the presence of loss and retransmissions for:
    1. [5 points] SR
    2. [5 points] GBN

Note: The example below shows a sender window of size 3, but your answer should be given in terms of the sequence space of size n. Explain each answer in a few sentences.





To avoid both scenarios, we want to avoid a situation where the leading (rightmost) edge of the receiver's window wraps around the sequence number space and overlaps with the trailing (leftmost) edge of the sender's window, as that will cause the receiver to read an old packet as a new one (since they have the same sequence number).

SR: The issue here is that when the sender resends packet 0 after failing to resend it previously, the receiver's window contains sequence # 0 for the next iteration (the n+1th packet), so it reads packet 0 as a new packet instead of the old packet. So, in order to prevent this from happening, the maximum window size must be **n/2** so that at most, both windows will cover up to n sequence numbers, so there will be no overlap with the next iteration of n sequence numbers.

+1 for correct answer (n/2)

+2 for explanation of how overlapping sequence numbers affect SR
+2 for explanation for the solution
GBN: The situation is slightly different for GBN. Because GBN cares about the order in which the packets arrive (i.e. if packet *expectedseqnum* has not been received, it disregards all packets with sequence numbers > *expectedseqnum*), we only need to protect ourselves against the case where the first packet in the sender's window and the receiver's *expectedseqnum* have the same sequence number. This will only happen if the window is of size ≥ n, so the largest allowable window will be **n - 1**.
+1 for correct answer (n-1)
+2 for explanation of how overlapping sequence numbers affect GBN
+2 for explanation of the solution

3. [12 points] Answer True or False and provide a brief explanation for your answer.
    1. [3 points] With the SR protocol, it is possible for the sender to send a packet that falls outside of the receiver's window.

    True. Take the following case:
    1. Sender sends packets 0, 1 to receiver at t = 0s.
    2. Receiver receives packets 0, 1 at t = 1s and sends ACKs 0, 1 to sender. It advances its window to [2, 3].
    3. ACKs do not arrive at the sender, so it resends packets 0, 1.
    4. Receiver receives packets 0, 1, which are outside of its window [2, 3].

   +1 for correct answer
    +2 for correct explanation
    2. [3 points] With the SR protocol, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.

    True. Take the following case:
    1. Sender sends packets 0, 1 to receiver at t = 0s.
    2. Receiver receives packets 0, 1 at t = 1s and sends ACKs 0, 1 to sender.
    3. ACKs are delayed in arriving, so sender times out and resends packets 0, 1 to receiver at t = 3s.
    4. Receiver receives packets 0, 1 again at t = 4s and resends ACKs 0, 1 to sender.
    5. Sender finally receives the delayed ACKs 0, 1 and advances its window to [2, 3].
    6. Sender then receives the new ACKs 0, 1 that were resent by receiver, which are outside of its window.

    +1 for correct answer
    +2 for correct explanation
    3. [3 points] With the GBN protocol, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.

    True. Identical example as in SR.
   +1 for correct answer
    +2 for correct explanation
    4. [3 points] With the GBN protocol, it is possible for the sender to have a window that is further advanced than the receiver's *expected seqnum*. Assume that the maximum window size is what you calculated in part b.

    False. The sender in GBN must receive an ACK from the receiver before advancing its window, so it can only remain on par with or behind the receiver.

+1 for correct answer
+2 for correct explanation

Question 3: TCP Handshake and Congestion Control [22 points, parts a-i]

Assume we are using Reno for TCP congestion control unless otherwise specified.

1. [6 points] Assume no transport layer security protocol is adopted, and the TCP implementation in host A and B always uses 0 as the initial sequence number. If an attacker, host C, cannot eavesdrop or intercept the traffic between host A and host B (i.e. host C is not a man-in-the-middle for host A and B), but host C knows host A and host B will talk to each other, both on port X, can host C still manage to make A believe some of the TCP packets sent by host C are actually from B?

Explain how host C can orchestrate such an attack, and why using randomized initial sequence numbers can significantly lower the success rate of such attacks. When proposing the attack, explicitly state the addresses, ports, sequence number, and acknowledgement number the attacker should use for the spoofed packet(s).
Hint: the attack need not succeed 100% of the time, but if launched repeatedly it should succeed much more often when sequence numbers are not randomized than when they are. The goal of the attack is for C to inject data that A will accept, not to disrupt the communication between B and A.
The attack is possible.
Usually, TCP clients and servers would not send actual data during the 3-way handshake due to how sockets APIs are implemented; therefore, the acknowledgment number that host B need to reply to host A right after the 3-way handshake is always 0 if A uses 0 as the initial sequence number. Thus, host C can keep sending the following spoof TCP packet to host A repetitively:
Source IP: host B
Destination IP: host A
Source Port: X
Destination Port: X
Sequence number: 1
Acknowledgement number: 1
— —
Actual data of n bytes
As long as this spoofed packet arrives at A after A and B just finished a 3-way handshake and before the first actual data packet from B arrives at A, A will mistake the packet as from B. Since host C can keep sending this spoof packet repetitively, this method has a fair chance of succeeding as long as one of the spoofed packets from C has the correct timing.
Using randomized sequence numbers on both sides can effectively mitigate such attacks, because the attack C would not only need to nail the timing, but also correctly guess the sequence number and acknowledgement number that B is going to use. This reduces the attack success rate by $(2^{32})^2$ times, making such attacks almost impossible to carry out in practice.
+1 point for stating the attack is possible
+1 point for stating the timing required for the attack to work (the packet must arrive at A after A and B just finished the 3 way handshake)
+1 point for stating the correct addresses and ports
+1 point for stating the correct sequence number and acknowledgement number

+1 point for stating the attack is no longer possible if we use randomized initial sequence numbers.
+1 point for explaining why the attack is no longer possible after using randomized initial sequence numbers.

2. [2 points] Once the connection is established, the hosts can begin to exchange data. Assume that host A is attempting to download a 100 MB zip file from host B. What two TCP state attributes determine how much of the 100 MB zip file host B places onto the wire at any given time?

The host A's receive window (buffer) size and the current congestion window size (CWND). The minimum of these two determines the maximum amount host B can send at any point in time.
+1 for receive window / buffer size
+1 for congestion window size

3. [1 points] What is the purpose of slow-start at the start of the connection?

Slow start is designed to quickly determine the congestion window size given the connection's bandwidth delay product (BDP) and current cross-traffic.
+1 for purpose of slow start (determine max congestion window size)

4. [1 point] During slow start, how will host B's sending rate change over time in reaction to events?

Host B's congestion window size will increase by 1 for each ACK that it receives from Host A.
+1 for mentioning that congestion window increases by 1 for each ACK received

5. [2 points] When will host B exit slow start? (If there are multiple possible exit conditions, list all of them)

+1 When the congestion window size exceeds ssthresh.
+1 When the first loss event occurs (can state DUP-ACK or RTO -- both acceptable)
+1 When connection closes.
Full credits are given for any of the two answers.

6. [1 points] During congestion avoidance, how will host B's sending rate change over time (assuming no loss)?

Congestion window size will increase by 1/CWND for each ACK received. In other words, the CWND will increase by 1 MSS for every RTT.
+1 for mentioning that the congestion window increases by 1/CWND for each ACK received OR CWND will increase by 1 MSS for every RTT.

7. [5 points] Assume host A uses TCP to receive a file from host B. Calculate the total time (in milliseconds) it takes for host A to receive all the bytes in the file from host B using the following assumptions and parameters. Show your work.

File size: 1 MB
Link speed: 500 Mbps
RTT: 50 milliseconds
TCP Max Segment Size (number of bytes for data payload): 1460 Bytes
Ethernet Packet Size (including all headers from different layers): 1500 Bytes
Initial CWND for both A and B: 32 * Max Segment Size
Initial ssthresh: Arbitrarily High
Receive Window Size: 8 MB
TCP slow start: enabled
Assume the network will not drop packets unless the amount of data inflight exceeds the bandwidth delay product (BDP). Assume there is no previous or ongoing TCP connection between A and B, no payload data will be sent during TCP 3 way handshake, and the host B will start pushing the file to A as

soon as the TCP 3-way handshake is finished. Include the time to establish TCP connection in your answer. Ignore the network transmission delay.
First, we calculate the bandwidth delay product of the network:
500 Mbps * 0.05 seconds = 3.125 MB, or 2083 packets.
As long as the amount of data inflight does not exceed this number, there will be no packet loss, and our TCP connection will stay in slow start state.
Then, during the slow start state, the amount of packets we receive w.r.t rounds of communication x could be approximated by the following formula:
$f(x) = 32 * 2^{(x-2)}, x \geq 2$
We try to figure out how many rounds of RTT it takes to finish transmitting the 1 MB file, which is 685 segments. So we calculate the antiderivative of f(x):
$\int 32 * 2^{(x-2)} dx = 32 * 2^{(x-2)} / ln(2) + C$
We set $32 * 2^{(x-2)} / ln(2) = 685$ packets to figure out how many RTT it takes for A to receive all the bytes in the file. Solving the equation, we get x = 5.89. Notice that when x=6, the CWND is $32 * 2^{(6-2)} = 512$, which is smaller than our bandwidth delay product of 2083 packets. Therefore, our TCP connection will never experience a loss, and will stay in slow start state until the file transmission finishes. Therefore, the total time it takes to download the file is approximately 5.89 * 50 ms = 294.5 ms.
Alternatively, one may argue that the packet transmission might be bursty in nature, so host A will only receive packets at the end of each RTT round. If so, we can expect the following:
32 total packets received at the end of 2 * RTT (The server will start sending data once it gets the ACK [the 3rd packet from the 3-way handshake] from the client, which is at around 1.5 * RTT. Therefore, the first batch of packets will arrive at the client at 2 * RTT.)
64 + 32 total packets received at the end of 3 * RTT
128 + 64 + 32 total packets received at the end of 4 * RTT
256 + 128 + 64 + 32 total packets received at the end of 5 * RTT
512 + 256 + 128 + 64 + 32 packets received at the end of 6 * RTT
Notice that the amount of packets inflight also does not exceed the BDP. Therefore, we should finish the file transmission at the end of 6 * RTT, so the transmission will take 300ms. We also consider this as a valid answer.
+1: accounting for handshake
+2: accounting for exponential growth starting from 32
+1: correctly calculating how many RTTs it will take based on that growth
+1: correct final answer

8. [2 points] The 32 MSS initial CWND in the previous question is adopted by CDNs such as Akamai today (if you are curious, here's some related reading: Google's argument for increasing the initial window, an argument against it that relates to another question on this homework, and a study of values used by CDNs). Historically, the initial CWND for Linux web servers is usually set to 4 MSS or even 1 MSS. Given your results in the previous question, how would a smaller initial CWND impact initial TCP throughput in today's high-speed (say, 200 Mbps+) connections?

Initial TCP transmission would be slower because it would take more time to get the CWND to the optimal value, and thus bandwidth between the two hosts would be used inefficiently.
+1 if mention slower download / takes longer
+1 if mention due to slow growth of CWND

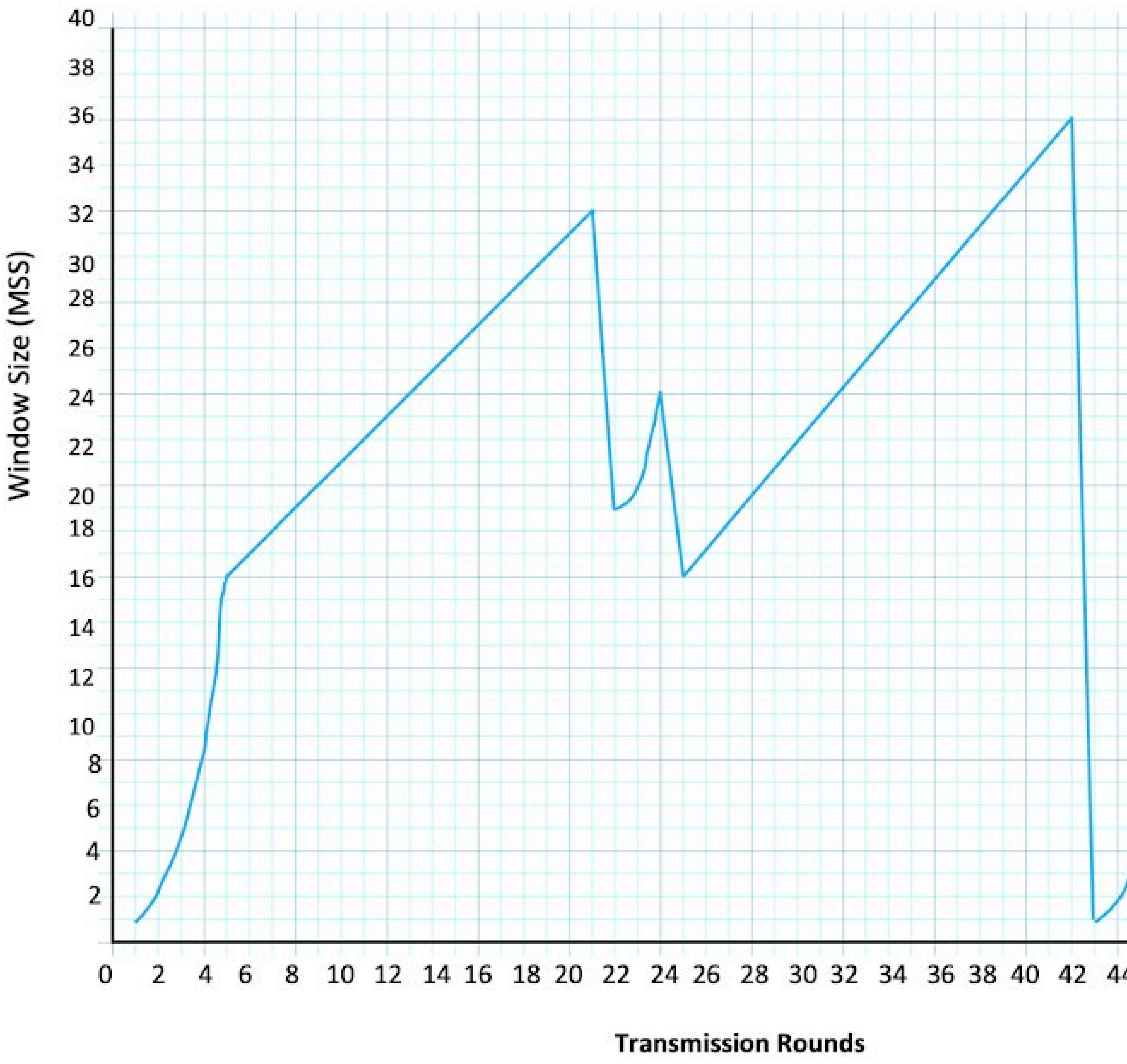
9. [2 points] Two signals TCP Reno uses to detect loss are timeouts and triple-duplicate ACKs. Why does it react differently to a *timeout* and a triple *duplicate ACK*?

Dup-Ack implies that a loss might have occurred, but some of the segments after the lost segment(s) still arrive at the receiver's side, so the congestion is relatively mild. Timeout implies that all of the unacked segments are likely lost, so the congestion is severe and extreme measures such as resetting cwnd have to be taken.

+1 for pointing out in the case of dup-ack, some of the packets after the potentially lost packet can still arrive, so the congestion is relatively minor.
+1 for pointing out timeout implies all unacked segments are likely lost, so the congestion is severe.

Question 4: Keeping Tracks of ACKs [20 points, parts a-k]

Given below is a MSS vs Transmission round graph for a subset of a TCP Reno connection- the connection is active during intervals before and after the shown time period, so the connection establishment and termination is not shown. Answer the following based on the graph.



The slope from round 25-42 was drawn slightly off from what we intended -- it was meant to be 1. In terms of understanding TCP behavior during that period, please pretend that the slope is 1, but [for part (g)] use the data points closest to the line that we drew: (40, 34), (41, 35), (42, 36).

1. [2 points] Identify time intervals where TCP is operating in congestion-avoidance mode. Specify each interval by giving (starting round, ending round).

(5,21) [(6,22) also accepted] and (25 to 42)[(26,42) also accepted]

+1 for each window

2. [1 point] What is the approximate *threshold (ssthresh)* of the congestion window size value at the start of the first transmission round?

16 MSS

+1 for size

3. [2 points] Describe what event(s) trigger what happens at Transmission Round **21**. Does this event definitely indicate that one (or more) of the sender's packets was lost?

Indicates a case of three duplicate ACKS, where not all of the sent packets were received in order. Does not definitely indicate a loss. It could be due to reordering.
+1 for mentioning duplicate ACKs
+1 for No.

4. [1 point] Name the event which occurs at Transmission Round **42**.

Timeout
+1 for Timeout.

5. [3 points] Does this event definitely indicate that one (or more) of the sender's packets was lost? If not, please describe the other possible explanations.

No, it does not definitely indicate a packet loss. The timeout may have been due to an unusually high RTT for a packet sent in the previous window or a lost ACK from the receiver.
+1 for stating No.
+2 if valid reason stated

6. [1 point] What is the *threshold(ssthresh)* value in Transmission Round **23?**

ssthresh=16
+1 for mentioning DUP ACKS.
+1 for ssthresh = 16

7. [4 points] If the RTT is 100 ms, how long (after round **42**) does it take for the current congestion window size to reach that of the ssthresh immediately before round **41**? Assume no duplicate ACKs or packet loss incurs after round **42**. ***Edit: The slope from round 25-42 was drawn slightly off from what we intended -- it was meant to be 1. In terms of understanding TCP behavior during that period, please pretend that the slope is 1, but [for part (g)] use the data points closest to the line that we drew: (40, 34), (41, 35), (42, 36).***

Old threshold value = 16 MSS - identified by looking at the size of the window immediately after receiving a dupAck at Round 19.
Size of window before loss = 36 MSS
New threshold = Size@loss/2 = 18 MSS
After round 42 (not included) Grows exponentially till 16 MSS = 4 RTTs.
Thus, total time taken = 4*100ms = 400ms
+1 for identifying new threshold.
+1 for identifying old threshold
+1 to calculate size for exponential growth
+1 to calculate time.

8. [1point] What events are causing the exponential increase in window size between Transmission Round **22 to 24?**.

Receiver continues to receive DUP ACKS while in fast recovery state.
+1 for mentioning DUP ACKS.

9. [2 points] Suppose the highest ACK received up through time (round) 23 is 20000 MSS. Assume in rounds 21-23 the sender sent as much data as its congestion window allowed. What is the highest possible SEQ value for a segment sent in round 24 (just before the line in the graph drops)? Explain/justify your answer.

The highest value is 20000 MSS. The sender sent a full window in round 21 and did not increase the window from there, so the first packet must have been lost. So it has a full window (32 MSS) of outstanding packets. The congestion window never grows above 32 MSS during fast recovery, and so it is not allowed to send additional packets.

10. [2 points] What is the highest possible SEQ value for a segment sent in round 25 (just after the line in the graph drops)? Explain/justify your answer.

At round=25, we are in the congestion avoidance phase, which means the sender receives a new ACK at round=24 (to exit Fast Recovery).
The new ACK could have the highest possible number of 20032MSS: In round=21, the sender sent 32 packets, and the highest possible value for them is 200031MSS (since ACK=20000MSS indicates that 20000 MSS was lost, and it could have been the first of the 32 packets). The new ACK indicates that the lost packet was received, and the receiver could have (in round=21) already received all packets 20001MSS...200031MSS out-of-order and buffered them. So the new ACK could be as high as ACK 20031MSS.
Since the new ACK is 20031MSS, at round=25, cwnd=16, the sender's window is [20032MSS, 20047MSS]. Therefore, the highest SEQ value sent in round 25 is 20047MSS.

Credit to Xindi Xu for the writeup.
+1 for mentioning increase of 16 MSS in SEQ num
+1 for correct reason “because in congestion avoidance the window grows to the full window size”

11. [1 point] Describe what event(s) trigger what happens at Transmission Round **24**.

+1 for mentioning New ACK for the outstanding packet that was triple-duplicate-ACKed at t = 21

Question 4: Q5 - QUIC and BBR [24 points, parts a-f]

1. [4 points] Describe the “bufferbloat” problem. Why would bufferbloat cause problems for loss-based congestion control protocols like TCP Reno?

The bufferbloat problem occurs when there are excessively large and frequently full buffers inside the network.
Bufferbloat damages loss-based congestion control algorithms because large buffers cause excessive congestion to happen before actual packet loss. This leads to loss-based algorithms filling these deep buffers repeatedly and causing excessive queuing delay.
+2 for correct description of bufferbloat
+2 for mentioning LBAs filling deep buffers and causing queueing delays

2. [4 points] On p282, the textbook mentions that "The authors of QUIC [Langley 2017] stress that this means that changes can be made to QUIC at 'application-update timescales,' that is, much faster than TCP or UDP update timescales." Why can't TCP or UDP be updated as quickly as applications? Why can Google implement and iterate on its own transport protocols like QUIC so efficiently?

Google’s QUIC and TCP implementations (sometimes) use the BBR (https://groups.google.com/forum/#!forum/bbr-dev) congestion control algorithm. Read the reference (or others online) and answer the following questions. Be sure to cite your sources.

Companies like Google control services like Chrome and Youtube, so it’s a lot easier for them to change these applications and the custom protocols they may use. Since QUIC is built on top of preexisting UDP, google can update and iterate on it without relying on updates to UDP.

It’s difficult to get the different manufacturers of Operating Systems like Linux, Microsoft, and Apple to agree on changing their respective operating system modules that implement transport layer protocols in a timely manner.
+2 for correct reasoning about Google’s quickness in developing its own protocols
+2 for reasoning about different manufacturers of OSs and difficulties of unanimous change

3. [4 points] In steady state, as we discussed in class, TCP Reno decides what rate to send based on additive-increase, multiplicative-decrease of the congestion window, increasing until loss and then decreasing. In steady-state, what two main factors does BBR use to decide the rate to send at? (You should consider the simpler v1 of BBR from 2016/2017, which avoids complexities that have been added since).

BBR uses bottleneck bandwidth and RTT to decide its sending rate.

+2 for mentioning bandwidth
+2 for mentioning RTT

4. [4 points] How does this BBR approach avoid the disadvantages of loss-based congestion control from your earlier answer about the problems caused by bufferbloat?

Maximum throughput and minimum delay correlates positively with bottleneck bandwidth.
By using maximum throughput as a factor in deciding the sending rate, BBR tries to identify (and is often successful) the optimum usage of bandwidth by throttling rates to that point by slowly increasing the data rate until max RTT is reached. This probing behaviour results in maintaining a steady state sending rate close to the bottleneck bandwidth. (Less conservative than Reno)
+1 for stating bottleneck bandwidth
+1 for stating minimizing delay
+1 for how BBR avoids back-off because of spurious loss unlike Reno (Reno is too aggressive)
+1 for how BBR avoids being too conservative- probing operations, and maintain steady state bandwidth

5. [4 points] How does BBR avoid disadvantages of delay-based schemes like TCP Vegas? Answer the question in terms of fairness in conjunction with other schemes.

Delay based mechanisms throttle the connections before actual packet loss. For example, when coupled with Reno, Vegas would reduce the sending rate earlier than Reno, which in turn will reduce its throughput. At the same time, since the traffic has gone down, Reno increases its sending rate. Eventually, Reno dominates the connection. Overall, delay based mechanisms are too conservative when compared to loss based connections.
BBR overcomes this by projecting the optimal data rate and RTT in advance, and can thus push for a larger window than Vegas. This ensures that the bandwidth is close to optimal at all times. Thus, it is able to compete with highly aggressive schemes like Reno, leading to a fairer network distribution.
+1 for stating Loss based dominates over delay based schemes
+1 for reasoning the above-comparing the disadvantages of Vegas over Reno using a specific situation.
+2 for how BBR solves

6. [4 points] Google is currently developing and testing their new version of BBR, BBRv2. How does BBRV2 incorporate help from the network layer routers into its model (Hint: is there a way for routers to inform the transport layer that there is congestion)? How does this new

information further BBR as true congestion as opposed to loss-based algorithm?

BBRv2 has incorporated signals from the network layer via explicit congestion notification (ECN) from routers. This occurs when the routers set a bit on the IP header indicating congestion. This furthers BBR as a true congestion based algorithm because a router may set this bit instead of dropping the packet to signal congestion.

+2 for identifying ECN

+2 for reasoning about how ECN furthers BBR as a true congestion control algorithm