Transport Layer

Annalise Tarhan

November 30, 2020

1 Suppose Client A initiates a Telnet session with Server S. At about the same time, Client B also initiates a Telnet session with Server S. Provide possible source and destination port numbers for:

The segments sent from A to S. Source: 1024 Dest: 23

The segments sent from B to S. Source: 1024 Dest: 23

The segments sent from S to A. Source: 23 Dest: 1024

The segments sent from S to B. Source: 23 Dest: 1025

If A and B are different hosts, is it possible that the source port number in the segments from A to S is the same as that from B to S? Yes

How about if they are the same host? No

2 Consider Figure 3.5. What are the source and destination port values in the segments flowing from the server back to the clients' processes? What are the IP addresses in the network-layer datagrams carrying the transport-layer segments?

Source port value: 80
Destination port values: 7532 and 26145

Source IP address: B
Destination IP addreesss: A and C

3 What is the 1s complement of the sum of these bytes: 01010011, 01100110, 01110100? Why is it that UDP takes the 1s complement of the sum; why not just use the sum? With the 1s complement scheme, how does the receiver detect errors? Is it possible that a 1-bit error will go undetected? How about a 2-bit error?

01010011+01100110=10111001
10111001+01110100=100101101
(wrap-around) 100101101 $\rightarrow$ 00101110
(1s complement) 00101110 $\rightarrow$ 11010001

UDP takes the 1s complement of the sum on one side and leaves the sum as-is on the other. The sum of the two results should equal zero if no corruption occurred. The choice of which end to calculate the 1s complement and which to leave as-is is arbitrary. A 1-bit error will be detected, because one of the bits in the final sum will be a one instead of a zero, but if there are two corrupted bits at corresponding places, the errors could cancel each other out and go undetected.

4

4.1 What is the 1s complement of the sum of 01011100 and 01100101?

01011100+01100101=11000001
(1s complement) 11000001 $\rightarrow$ 00111110

4.2 What is the 1s complement of the sum of 11011010 and 01100101?

11011010+01100101=1001111110
(wrap-around) 1001111110 $\rightarrow$ 01000000
(1s complement) 01000000 $\rightarrow$ 10111111

4.3 For the bytes in the first part, give an example where one bit is flipped in each of the two bytes and yet the 1s complement doesn't change.

The last digit of each is flipped:
01011101+01100100=11000001
(1s complement) 11000001 $\rightarrow$ 00111110

5 Suppose that the UDP receiver computes the Internet checksum for the received UDP segment and finds that it matches the value carried in the checksum field. Can the receiver be absolutely certain that no bit errors have eoccurred? Explain.

No. As in the previous problem, even numbers of bits can change and not change the result of the checksum, if they are in corresponding places.

6 Show that the receiver in Figure 3.57, when operating with the sender in Figure 3.11, can lead the sender and receiver to enter into a deadlock state, where each is waiting for an event that will never occur.

The system could enter a deadlock state if an ACK packet is corrupted. The sender will detect the corrupted packet and resend the previous packet. The problem is that the receiver will have already moved on to the next sequence number. If the packet, for example, had sequence number 0, the receiver will have sent the soon to be corrupted ACK 0 packet and then moved into the "Wait for 1" state. Since the resent packet will still have sequence number 0 the system is deadlocked. The receiver will send NAKs, the sender will resend the same packet, and neither will progress.

7 In protocol rdt3.0, the ACK packets flowing from the receiver to the sender do not have sequence numbers, although they do have an ACK field with the sequence number of the packet they are acknowledging. Why?

In this system, the ACK packets never contain their own data, as they might in a real system. Since ACK packets without data don't need to be acknowledged, there is no need to be able to identify them. The sequence number of the packet they are acknowledging is enough for them to serve their purpose.

8 Describe the FSM for the receiver side of protocol rdt3.0.

The only change from rdt2.2 to rdt3.0 was adding timers, which only affects sender-side logic. The receiver for rdt3.0 is the same as for rdt2.2. To summarize, there are two states: 'Wait for 0 from below' and 'Wait for 1 from below.' The logic for each is symmetrical. To use 'Wait for 0' as an example, there are two possible transitions, one to 'Wait for 1' when a non-corrupt packet with sequence number 0 is received and one self-transition when a packet that is either corrupt or has sequence number 1 is received. When a good packet is received, it extracts and delivers the data and sends an acknowledgement. When a bad packet is received, it sends an acknowledgement for the previous packet instead.

9 Give a trace of the operation of protocol rdt3.0 when data packets and acknowledgement packets are garbled.

Sender: send pkt0
Receiver: rcv (garbled) pkt0
Receiver: send ACK1
Sender: rcv (garbled) ACK1
Sender: ...
Sender: timeout
Sender send pkt0

10 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. Informally argue why your protocol can communicate correctly over this channel.

Adding the possibility of packet loss and a timeout to protocol rdt2.1 is equivalent to adding a different form of NAK. 'Wait for ACK or NAK' becomes 'Wait for ACK, NAK, or timeout' and another line is added after rdt_rcv(rcvpkt) to include '|| timeout.' The argument for correctness is that a timeout is equivalent to a NAK and that if the system's behavior in that case was correct before, it will also be correct after the inclusion of a timeout. From the receiver's perspective, a timeout caused by a NAK for a garbled packet or a garbled ACK also has the same result as before: a resent packet.

11 Consider the rdt 2.2 receiver and the creation of a new packet in the self-transition in the 'Wait for 0 from below' and the 'Wait for 1 from below' states.

11.1 Would the protocol work correctly if this action were removed from the self-transition in the 'Wait for 1 from below' state?

Yes, it would. The packet created is identical to the sndpkt created in the transition from the previous state. As long as the receiver still holds a reference to the previous packet, it should be able to send a new copy of it.

11.2 What if this event were removed from the self-transition in the 'Wait for 0 from below' state?

The answer is the same.

12 The sender side of rdt3.0 simply ignores all received packets that are either in error or have the wrong value in the acknum field of an acknowledgement packet. Suppose that in such circumstances, rdt3.0 were simply to retransmit the current data packet. Would the protocol still work?

Yes, but it would be inefficient, especially if ACKs were frequently garbled. It would be even worse with premature timeouts because the issue compounds. Every received packet causes the receiver to send another packet, which would trigger the sender to resend the current packet, and the cycle would continue indefinitely.

13 Consider the rdt3.0 protocol. Show that if the network connection between the sender and the receiver can reorder messages, then the alternating-bit protocol will not work correctly.

The protocol fails when two errors happen together. First, an ACK is delayed, causing the sender to time out and resend. That packet is received and acknowledged correctly, and commiunication continues. The second error is a packet with the same sequence number as the delayed ACK is actually lost. If the delayed ACK arrives while the sender is waiting for an ACK for the actually lost packet, it will appear to the sender that all packets have been received and it will move on. The receiver will be waiting for the wrong packet, and they will deadlock. The receiver will keep sending ACKs for the packet it is waiting for and the sender will ignore them as duplicates.

The protocol also fails if it is a message that is delayed. It is resent as usual, acknowledged as usual, and communication continues. Later, the delayed message arrives while the receiver is waiting for a message with the same sequence number. The delayed message takes its place, is acknowleged, and the message is corrupted.

14 Consider a reliable data transfer protocol that uses only negative acknowledgements. Suppose the sender sends data only infrequently. Would a NAK-only protocol be preferable to a protocol that uses ACKs? Why? Now suppose the sender has a lot of data to send and the end-to-end connection experiences few losses. In this second case, would a NAK-only protocol be preferable to a protocol that uses ACKs? Why?

The problem with NAK-only in the first scenario is that the receiver would not realize a packet has been lost until the next packet is received. Since packets arrive infrequently, that would cause a long delay. In the second scenario, errors would be recognized and corrected quickly. The benefit of using NAKs instead of ACKs is a lower number of acknowledgment packets, which is significant in the second scenario and insignificant in the first. Therefore, the cost of using NAKs far outweighs the benefit in the first scenario and the reverse is true in the second.

15 Consider the cross-country example in Figure 3.17. How big would the window size have to be for the channel utilization to be greater than 98%? Suppose the size of the packet is 1,500 bytes, including both header fields and data.

The utilization rate of the sender is defined as the fraction of time the sender is actually sending bits into the channel. For simplicity, use the time it takes for a packet to be sent and its acknowledgement received as the unit of time. To calculate, add the round trip time to the transmission time, which is the size of the packet divided by the transmission rate:

Round trip time (RTT): 30 milliseconds
Transmission rate (R): $10^9/1000 = 10^6$ bits per millisecond
Packet size (L): $1500 * 8 = 12000$ bits
Transmission time (L/R): $12000/10^6 = .012$ milliseconds
Total time (RTT + L/R): $30 + .012 = 30.012$ milliseconds

The amount of time the sender spends transmitting depends on the window size, or how many packets it is allowed to send before the first one it sent is acknowledged. Let the window size be N packets. Then the time spent transmitting equals N times the transmission time. The utilization rate, which we set at 98%, equals the time spent transmitting divided by the time to receive the first packet's acknowledgement.

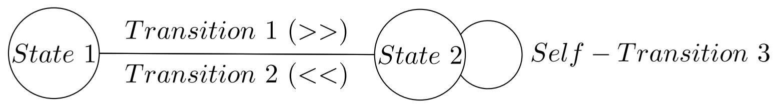
Utilization rate $= (N * L/R)/(RTT + L/R)$
$.98 = .012N/30.012$
$N = 30.012 * .98/.012$
$N = 2450.98$

The minimum window size is approximately 2451 packets.

16 Suppose an application uses rdt 3.0 as its transport layer protocol. As the stop-and-wait protocol has very low channel utilization, the designers of this application let the receiver keep sending back a number of alternating ACK 0 and ACK 1 even if the corresponding data have not arrived at the receiver. Would this application design increase channel utilization? Why? Are there any potential problems with this approach? Explain.

As long as packets were never lost, this would work perfectly. Unfortunately, lost packets would never be recovered

17 Consider two network entities, A and B, which are connected by a perfect bi-directional channel. A and B are to deliver data messages to each other in an alternating manner: First, A must deliver a message to B, then B must deliver a message to A, and so on. If an entity is in a state where it should not attempt to deliver a message to the other side, and there is an event like rdt_send(data) call from above, it can simply be ignored with a call to rdt_unable_to_send(data). Draw a FSM specification for this protocol.



State 1: Wait for call from above
State 2: Wait for call from below

A: Initial state is State 1
B: Initial state is State 2

Transition 1:
rdt_send(data) $\rightarrow$ packet = make_pkt(data); udt_send(data)

Transition 2:
rdt_rcv(packet) $\rightarrow$ extract(packet, data); deliver_data(data)

Self-Transition 3:
rdt_send(data) $\rightarrow$ rdt_unable_to_send(data)

18 In the generic SR protocol, the sender transmits a message as soon as it is available (if it is in the window) without waiting for an acknowledgement. Suppose now that we want an SR protocol that sends messages two at a time. Suppose that the channel may lose messages but will not corrupt or reorder messages. Design an error-control protocol for the unidirectional reliable transfer of messages. Give an FSM description of the sender and receiver. Describe the format of the packets sent between sender and receiver, and vice versa. Give an example showing how the protocol recovers from a lost packet.

This protocol relies on positive acknowledgements, timeouts, and the fact that packets will never be reordered.

SENDER

$$\begin{tikzcd}
State\ 1 \arrow[r, "Transition\ 1"'] & State\ 2 \arrow[d, "Transition\ 2"] \\
State\ 4 \arrow[u, "Transition\ 6"'] \arrow[loop left, "Self-Transition\ 5"] & State\ 3 \arrow[l, "Transition\ 4"'] \arrow[loop right, "Self-Transition\ 3"]
\end{tikzcd}$$

State 1: Wait for message 0 from above
State 2: Wait for message 1 from above
State 3: Wait for ACK0
State 4: Wait for ACK1

Transition 1: rdt_rcv(packet) → extract(packet, data); packet0 = make_pkt(data)

Transition 2: rdt_rcv(packet) → extract(packet, data); packet1 = make_pkt(data); udt_send(packet0); udt_send(packet1); start_timer()

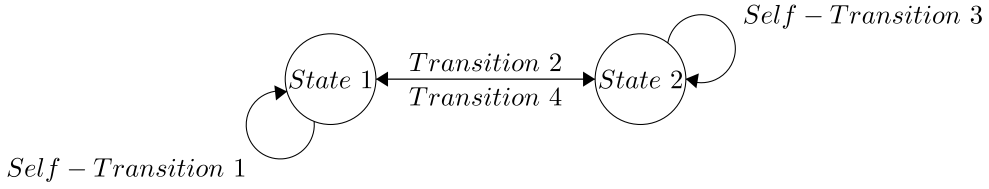
Self-Transition 3: timeout || (udt_rcv(rcvpkt) && hasseqnum(rcvpkt, 1)) → udt_send(packet0); udt_send(packet1)

Transition 4: udt_rcv(packet) && hasseqnum(rcvpkt, 0) → start_timer()

Self-Transition 5: timeout → udt_send(packet1); start_timer()

Transition 6: udt_rcv(packet) && hasseqnum(rcvpkt, 1) → Λ

RECEIVER



State 1: Wait for 0
State 2: Wait for 1

Self-Transition 1: (udt_rcv(rcvpkt) && hasseqnum(rcvpkt, 1)) → sendpkt=make_pkt(ACK, 1); upd_send(sendpkt)

Transition 2: udt_rcv(packet) && hasseqnum(rcvpkt, 0) → extract(rcvpkt, data); deliver_data(data); sendpkt=make_pkt(ACK, 0); udp_send(sendpkt)

Self-Transition 3: (udt_rcv(rcvpkt) && hasseqnum(rcvpkt, 0)) → sendpkt=make_pkt(ACK, 0); upd_send(sendpkt)

Transition 4: udt_rcv(packet) && hasseqnum(rcvpkt, 1) → extract(rcvpkt, data); deliver_data(data); sendpkt=make_pkt(ACK, 1); upd_send(sendpkt)

Packet format:
Sequence number is alternating-bit, 0s and 1s. All other header and data information as usual.

Packet loss:
Packet 0 lost: The receiver receives, acknowledges, and discards packet 1. The sender re-sends both packets.
Packet 1 lost: The receiver receives, acknowledges, and delivers packet 0 and waits. The sender times out and re-sends packet 1.
ACK 0 lost: Appears the same as packet 0 being lost, so the sender re-sends both packets. (Shoot! This breaks the protocol. From the receiver's perspective, there was no error and the next pair just happens to be identical to the previous pair. I'm not prepared to do this whole thing over again, but to fix it, you'd need to use a more sophisticated sequence number protocol than alternating-bit so the receiver could detect that the second pair is a repeat.)
ACK 1 lost: Again, this appears to the receiver that packet 1 was lost. The sender times out, resends, and continues once the receiver acknowledges the duplicate packet.

19 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. 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. Also, give a description of the packet format used.

This protocol relies on an appended B or C to identify the sender/receiver, a checksum to verify packet integrity, and a one bit sequence number. Only one bit is needed because from the receivers' perspective, there is only one 'correct' arriving packet at any given time. This model omits the cases of new data arriving from above at inappropriate times and extraneous packets arriving from receivers. In all cases, those would be ignored.

HOST A:
State 1: Wait for message from above
Transition to State 2: rdt_send(data) → packet = make_pkt(seqnum, checksum, data), udt_send(B, packet), udt_send(C, packet), start_timer(B), start_timer(C)

State 2: Wait for ACK from B or C
Self-Transition: rdt_rcv(packet) && from_B(packet) && corrupt(packet) → udt_send(B, packet), start_timer(B)
Self-Transition: rdt_rcv(packet) && from_C(packet) && corrupt(packet) → udt_send(C, packet), start_timer(C)
Self-Transition: timeout(B) → udt_send(B, packet), start_timer(B)
Self-Transition: timeout(C) → udt_send(C, packet), start_timer(C)
Transition to State 3: rdt_rcv(packet) && from_B(packet) && notcorrupt(packet) → kill_timer(B)
Transition to State 4: rdt_rcv(packet) && from_C(packet) && notcorrupt(packet) → kill_timer(C)

State 3: Wait for ACK from C
Self-Transition: rdt_rcv(packet) && from_C(packet) && corrupt(packet) → udt_send(C, packet), start_timer(C)
Self-Transition: timeout(C) → udt_send(C, packet), start_timer(C)
Transition to State 1: rdt_rcv(packet) && from_C(packet) && notcorrupt(packet) → kill_timer(C)

State 4: Wait for ACK from B
Self-Transition: rdt_rcv(packet) && from_B(packet) && corrupt(packet) → udt_send(B, packet), start_timer(B)
Self-Transition: timeout(B) → udt_send(B, packet), start_timer(B)
Transition to State 1: rdt_rcv(packet) && from_B(packet) && notcorrupt(packet) → kill_timer(B)

HOST B/C:
State 1: Wait for message from below
Self-Transition: rdt_rcv(packet) && notcorrupt(packet) && has_expected_seqnum(packet) → extract(packet, data), deliver_data(data), sndpkt = make_packet(B/C, seqnum, checksum), udt_send(sndpkt)
Self-Transition: rdt_rcv(packet) && corrupt(packet) → Λ
Self-Transition: rdt_rcv(packet) && notcorrupt(packet) && wrong_seqnum(packet) → sndpkt = make_packet(B/C, seqnum, checksum), udt_send(sndpkt)

20 Consider a scenario in which Host A and Host B want to send messages to Host C. Hosts A and C are connected by a channel that can lose and corrupt (but not reorder) messages. Hosts B and C are connected by another independent channel with the same properties. The transport layer at Host C should alternate in delivering messages from A and B to the layer above. Design a stop-and-wait-like error-control protocol for reliably transferring packets from A and B to C, with alternating delivery at C. Also, give a description of the packet format used.

To avoid using a buffer, this protocol causes many potential wasted packets to be sent by A and B while C is waiting for a packet from the other sender. Those packets are ignored, time out, and are repeatedly re-sent until C is ready for one and acknowledges it. The exception is if the packet arriving (from either sender) still has the previous seqnum, indicating the ACK was either lost or corrupted. In that case, the packet is acknowledged. The packets include the identity of the sender, an alternating-bit sequence number, and a checksum.

HOST A/B:
State 1: Wait for message from above (ignore messages from below)
Self-Transition: rdt_rcv(packet) $\rightarrow \Lambda$
Transition to State 2: rdt_send(data) $\rightarrow$ packet = make_pkt(A/B, seqnum, checksum, data), udt_send(packet), start_timer()

State 2: Wait for ACK from C
Self-Transition: rdt_send(data) $\rightarrow$ rdt_unable_to_send(data)
Self-Transition: rdt_rcv(packet) && (corrupt(packet) —— wrong_seqnum(packet)) $\rightarrow$ udt_send(packet)
Self-Transition: timeout $\rightarrow$ udt_send(packet)
Transition to State 1: rdt_rcv(packet) && not_corrupt(packet) && correct_seqnum(packet) $\rightarrow$ kill_timer()

HOST C:
State 1: Wait for message from A
Self-Transition: rdt_rcv(packet) && corrupt(packet)$\rightarrow \Lambda$
Self-Transition: rdt_rcv(packet) && from_B(packet) && has_prev_seq_num(packet)

→ make_packet(B, prev_seqnum, checksum), udt_send(sndpkt)
Self-Transition: rdt_rcv(packet) && from_A(packet) && has_prev_seq_num(packet) → make_packet(A, prev_seqnum, checksum), udt_send(sndpkt)
Self-Transition: rdt_rcv(packet) && from_B(packet) → Λ
Transition to State 2: rdt_rcv(packet) && not_corrupt(packet) && from_A(packet) → extract(packet, data), deliver_data(data), sndpkt = make_packet(A, seqnum, checksum), udt_send(sndpkt)

State 2: Wait for message from B
Self-Transition: rdt_rcv(packet) && corrupt(packet)→ Λ
Self-Transition: rdt_rcv(packet) && from_A(packet) && has_prev_seq_num(packet) → make_packet(A, prev_seqnum, checksum), udt_send(sndpkt)
Self-Transition: rdt_rcv(packet) && from_B(packet) && has_prev_seq_num(packet) → make_packet(B, prev_seqnum, checksum), udt_send(sndpkt)
Self-Transition: rdt_rcv(packet) && from_A(packet) → Λ
Transition to State 1: rdt_rcv(packet) && not_corrupt(packet) && from_B(packet) → extract(packet, data), deliver_data(data), sndpkt = make_packet(B, seqnum, checksum), udt_send(sndpkt)

21 Suppose we have two network entities, A and B. B has a supply of data messages that will be sent to A according to the following conventions. When A gets a request from the layer above to get the next data (D) message from B, A must send a request(R) message to B on the A-to-B channel. Only when B receives an R message can it send a data (D) message back to A on the B-to-A channel. A should deliver exactly one copy of each D message to the layer above. R messages can be lost (but not corrupted) in the A-to-B channel; D messages, once sent, are always delivered correctly. The delay along both channels is unknown and variable. Design a protocol that incorporates the appropriate mechanisms to compensate for the loss-prone A-to-B channel and implements message passing to the layer above at entity A.

Once A gets a request from above to get the next data message, it sends repeated requests at a fixed interval until the data message is received. B sends the data message as soon as it receives the first request messages and ignores all further messages with the same sequence number. An alternating-bit sequence is sufficient.

HOST A:
State 1: Wait for message from above
Transition to State 2: rdt_send(data) → packet = make_pkt(seqnum), udt_send(packet), start_timer()

State 2: Wait for data message
Self-Transition: timeout → udt_send(packet), start_timer()
Transition to State 1: rdt_rcv(packet) → kill_timer(), extract(packet, data), deliver_data(data)

HOST B:

State 1: Wait for request
Self-Transition: rdt_rcv(packet) && has_prev_seq_num(packet) → Λ
Self-Transition: rdt_rcv(packet) && has_correct_seq_num(packet) → udt_send(data_message)

22 Consider the GBN protocol with a sender window size of 4 and a sequence number range of 1,024. Suppose that at time t, the next in-order packet that the receiver is expecting has a sequence number of k. Assume that the medium does not reorder messages.

22.1 What are the possible sets of sequence numbers inside the sender's window at time t?

The possible sets must be consecutive subsets of {(k-4)*, ... , (k+4)*}, either be empty or contain $k-1$ or k, and have no more than four elements. The asterisk indicates mod 1024.
{(k-4)*, (k-3)*, (k-2)*, (k-1)*}, {(k-3)*, (k-2)*, (k-1)*, k}, {(k-2)*, (k-1)*, k, (k+1)}, {(k-1)*, k, (k+1)*, (k+2)*}, { k, (k+1)*, (k+2)*, (k+3)*}
{(k-3)*, (k-2)*, (k-1)*}, {(k-2)*, (k-1)*, k}, {(k-1)*, k, (k+1)*}, {k, (k+1)*, (k+2)*}
{(k-2)*, (k-1)*}, {(k-1)*, k}, {k, (k+1)*}
{(k-1)*}, {k}
∅

22.2 What are all possible values of the ACK field in all possible messages currently propagating back to the sender at time t?

{(k-4)*, (k-3)*, (k-2)*, (k-1)*}
It is a given that the medium does not reorder messages, and because the receiver has received (k-1)*, the sender must have received the ACK for (k-5)*. Even if packets were lost and resent, the receiver doesn't acknowledge out of order packets. This prevents the case of the first packet being lost, the lot being resent, and the sender moving on after only the first packet is acknowledged. In that case, the sender could start sending new packets while old re-ACKs are still in the pipeline. As it is, there is perfect ordering.

23 Consider the GBN and SR protocols. Suppose the sequence number space is of size k. What is the largest allowable sender window that will avoid the occurrence of problems such as that in Figure 3.27 for each of these prototcols?

GBN: k-1
SR: floor(k/2)

In Go-Back-N, the receiver doesn't acknowledge out of order packets, causing the sender to re-send all of them beginning with the lost one. As long as there is no potential for overlap for that one missing packet, the protocol will succeed. That only requires a single sequence number outside of the window. Selective Repeat, on the other hand, is unconstrained by the requirement to repeat all packets after a lost one. This means that each sequence number in one window must be distinguishable from each sequence number in the previous window, necessitating a window size of at most half of the sequence number space size.

24 True or False

24.1 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. If an acknowledgement was severely delayed, it could trigger the sender to re-send the packet, which would then be re-acknowledged. The first ACK to arrive would cause the window to shift, and the second to arrive would fall outside the shifted window.

24.2 With GBN, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.

True. The previous scenario would have the same result.

24.3 The alternating-bit protocol is the same as the SR protocol with a sender and receiver window size of 1.

True. Since the window size is 1, the current sequence number only needs to be distinguishable from the immediately previous and the immediately following numbers, which in alternating-bit it is.

24.4 The alternating-bit protocol is the same as the GBN protocol with a sender and receiver window size of 1.

True. Going back N with a window size of 1 just means re-sending the one packet in the window, as is protocol under alternating-bit.

25 We have said that an application may choose UDP for a transport protocol because UDP offers finer application control than TCP of what data is sent in a segment and when.

25.1 Why does an application have more control of what data is sent in a segment?

TCP's header takes up 20 bytes, while UDP's is only 8 bytes. Since the maximum segment size is fixed, that gives the application 12 more bytes per packet. TCP also grabs bytes from a connection stream, potentially splitting or concatenating individual messages. UDP encapsulate exactly what the application gives it.

25.2 Why does an application have more control on when the segment is sent?

TCP has a congestion-control mechanism that benefits the network at the expense of individual sending rates. UDP, on the other hand, sends messages at the rate the application tells it to, regardless of the health of the network.

26 Consider transferring an enormous file of L bytes from Host A to Host B. Assume an MSS of 536 bytes.

26.1 What is the maximum value of L such that TCP sequence numbers are not exhausted? Recall that the TCP sequence number field has 4 bytes.

The MSS is the maximum amount of application-layer data in the segment, not including TCP headers. Also, sequence numbers are not incremented by one, but by the number of bytes in the segment. Therefore, the MSS is irrelevant.
$L \leq 2^{8*4} = 4.29 * 10^9$ bytes, or 4.29 gigabytes

26.2 For the obtained L, find how long it takes to transmit the file. Assume that a total of 66 bytes of transport, network, and data-link header are added to each segment before the resulting packet is sent out over a 155 Mbps link. Ignore flow control and congestion control so A can pump out the segments back to back and continuously.

Number of messages (N): $N = L/MSS = 4294967296/536 = 8012999$
Size of each message (S): $S = MSS + Header = 536 + 66 = 602$ bytes
Total size (T): $T = N * S = 8012999 * 602 = 4823825398$ bytes or 38590 megabits
Transmission time: $T/Rate = 38590/155 = 249$ seconds

27 Hosts A and B are communicating over a TCP connection, and Host B has already received from A all bytes up through byte 126. Suppose Host A then sends two segments to Host B back-to-back. The first and second segments contain 80 and 40 bytes of data, respectively. In the first segment, the sequence number is 127, the source port number is 302, and the destination port number is 80. Host B sends an acknowledgement whenever it receives a segment from Host A.

27.1 In the second segment sent from Host A to B, what are the sequence number, source port number, and destination port number?

Sequence number: 207
Source port number: 302
Destination port number: 80

27.2 If the first segment arrives before the second segment, in the acknowledgement of the first arriving segment, what is the acknowledgement number, the source port number, and the destination port number?

Acknowledgement number: 207
Source port number: 80
Destination port number: 302

27.3 If the second segment arrives before the first segment, in the acknowledgement of the first arriving segment, what is the acknowledgement number?

127

27.4 Suppose the two segments sent by A arrive in order at B. The first acknowledgement is lost and the second acknowledgement arrives after the first timeout interval. Describe the scenario, including these segments and all other segments and acknowledgements sent, assuming no additional packet loss. For each packet, include the sequence number and the number of bytes of data. For each acknowledgement, include the acknowledgement number.

In this scenario, the lost acknowledgement wouldn't have even caused a hiccup if the second ACK had arrived before the timeout, since TCP uses cumulative acknowledgements. Since it did, the packet was re-sent and re-acknowledged, even though the sender was notified implicitly that the first packet had arrived the first time by the second (cumulative) ACK.

Packet 1 sent: (seq:127, byt:80)
Packet 2 sent: (seq 207, byt:40)
Packet 1 received, ACK 1 sent: (ack:207) XXX
Packet 2 received, ACK 2 sent: (ack:247)
TIMEOUT
Packet 1 re-sent: (seq:127, byt:80)
ACK 2 received: (ack 247)
Packet 1 re-received, ACK 1 re-sent: (ack:247)
ACK 1 received

28 Hosts A and B are directly connected with a 100 Mbps link. There is one TCP connection between the two hosts, and Host A is using it to send Host B an enormous file. Host A can send its application data into its TCP socket at a rate as high as 120 Mbps but Host B can read out of its TCP receive buffer at a maximum rate of 50 Mbps. Describe the effect of TCP flow control.

TCP's flow-control service eliminates the possibility of the sender overflowing the receiver's buffer. It does this by having the sender maintain a receive window variable, which indicates how much buffer space the receiver has available and is updated by the rwnd value in acknowledgement headers.

The bottleneck in this connection is the rate at which Host B is reading out of its receive buffer. The flow-control service prevents Host A from overflowing the buffer, stopping when the buffer is full, and when its one-byte test segments are acknowledged with packets that contain nonzero rwnd (receive window) values in their headers, Host A will continue sending again until the buffer fills up.

29 SYN cookies

29.1 Why is it necessary for the server to use a special initial sequence number in the SYNACK?

The server's initial sequence number is a SYN cookie, a complicated hash function of source and destination IP addresses and port numbers, as well as a secret number. This is necessary to protect against SYN flood attacks. If the server used an ordinary randomly generated sequence number, it would have to store the value. That allocation of resources at the initial stage of the connection is what would expose it to SYN flood attacks.

29.2 Suppose an attacker knows that a target host uses SYN cookies. Can the attacker create half-open or fully open connections by simply sending an ACK packet to the target? Why or why not?

No. SYN cookies work by sending the client an initial sequence number calculated using a function and secret number known only to the server. Without a correct ISN, the client's ACK packet would be discarded and do no damage.

29.3 Suppose an attacker collects a large amount of initial sequence numbers sent by the server. Can the attacker cause the server to create many fully open connections by sending ACKs with those initial sequence numbers? Why?

No. The sequence numbers are only valid for the host that originally received them. An attacker would have to have access to each of those systems and would only be able to open one connection from each.

30 Consider the network shown in scenario 2 in section 3.6.1. Suppose both sending hosts A and B have some fixed timeout values.

30.1 Argue that increasing the size of the finite buffer of the router might possibly decrease the throughput (λ_{out}).

Increasing the size of the buffer would decrease lost packets due to buffer overflow at the expense of longer delays. If the longer delay significantly increased the number of unnecessary re-transmissions, it could have a negative effect on throughput that outweighed the improvement of fewer dropped packets.

30.2 Now suppose both hosts dynamically adjust their timeout values based on the buffering delay at the router. Would increasing the buffer size help to increase the throughput? Why?

Yes. The increased buffer size would prevent buffer overflows, leading to fewer retransmissions and increased throughput.

31 Suppose that the five measured *SampleRTT* values are 106 ms, 120 ms, 140 ms, 90 ms, and 115 ms.

31.1 Compute *EstimatedRTT* after each of these *SampleRTT* values is obtained, using a value of $\alpha = 0.125$ and assuming that the value of *EstimatedRTT* was 100 ms just before the first of these five samples were obtained.

EstimatedRTT = (1-α)*EstimatedRTT+α*SampleRTT
$(1 - 0.125) * 100 + 0.125 * 106 = 100.75$
$(1 - 0.125) * 100.75 + 0.125 * 120 = 103.16$
$(1 - 0.125) * 103.16 + 0.125 * 140 = 107.77$
$(1 - 0.125) * 107.77 + 0.125 * 90 = 105.55$
$(1 - 0.125) * 105.55 + 0.125 * 115 = 106.73$

31.2 Compute the *DevRTT* after each sample is obtained, assuming a value of $\beta = 0.25$ and assuming the value of *DevRTT* was 5 ms just before the first of these five samples was obtained.

According to RFC 6298, the EstimatedRTT value used in this calculation should be the old value, not the newly calculated one.

DevRTT = (1-β)*DevRTT+$\beta *$ |SampleRTT-EstimatedRTT|
$(1 - 0.25) * 5 + 0.25 * |106 - 100| = 5.25$
$(1 - 0.25) * 5.25 + 0.25 * |120 - 100.75| = 8.75$
$(1 - 0.25) * 8.75 + 0.25 * |140 - 103.16| = 15.77$
$(1 - 0.25) * 15.77 + 0.25 * |90 - 107.77| = 16.27$
$(1 - 0.25) * 16.27 + 0.25 * |115 - 105.55| = 14.57$

31.3 Compute the TCP *TimeoutInterval* after each of these samples is obtained.

TimeoutInterval=EstimatedRTT+4*DevRTT
100.75+4*5.25=121.75
103.16+4*8.75=138.16
107.77+4*15.77=170.85
105.55+4*16.27=170.63
106.73+4*14.57=165.01

32 Consider the TCP procedure for estimating RTT. Suppose that $\alpha = 0.1$. Let $SampleRTT_1$ be the most recent sample RTT, $SampleRTT_2$ be the next most recent, and so on.

32.1 For a given TCP connection, suppose four acknowledgements have been returned with corresponding sample RTTs. Express $EstimatedRTT$ in terms of the four sample RTTs.

EstimatedRTT =
$\alpha(1-\alpha)^3 * SampleRTT_4$
$+ \alpha(1-\alpha)^2 * SampleRTT_3$
$+ \alpha(1-\alpha) * SampleRTT_2$
$+ \alpha * SampleRTT_1$

32.2 Generalize your formula for n sample RTTs.

EstimatedRTT $= \sum_{i=1}^{n} \alpha * (1-\alpha)^{n-1} * SampleRTT_n$

32.3 For the previous formula, let n approach infinity. Comment on why this averaging procedure is called an exponential moving average.

$\lim_{n\to\infty} EstimatedRTT = \sum_{i=1}^{n} \alpha * (1-\alpha)^{n-1} * SampleRTT_n$

An exponential moving average applies weighting factors which decrease exponentially. Each iteration of $EstimatedRTT$ multiplies the weight of each term by $(1-\alpha)$ which reduces it exponentially.

33 Why do you think TCP avoids measuring the $SampleRTT$ for retransmitted segments?

There is no indicator on retransmitted packets or their acknowledgements that mark them as such, so calculating the RTT for either requires the sender to guess which copy of the packet an ACK corresponds to. An incorrect guess could significantly change the calculated value.

34 What is the relationship between the variable *SendBase* and the variable *LastByteRcvd*?

SendBase is a state variable maintained by the sender that corresponds to the sequence number of the oldest unacknowledged byte. *LastByteRcvd* is maintained by the receiver and represents the number of the last byte in the data stream that has arrived from the network and been placed in the receive buffer. The difference between them is the number of bytes whose ACKs have been sent but not received.

35 What is the relationship between the variable *LastByteRcvd* and the variable y?

The variable y represents the ACK value on the segment. It will be either one greater than *LastByteRcvd* if it is the most recent ACK sent or less than *LastByteRcvd* if other packets have been received by the receiver since it was sent.

36 Why do you think the TCP designers chose to perform a fast retransmit after three duplicate ACKs instead of after the first one?

The cost of unnecessary retransmits can be high, so avoiding them is a key consideration. Every reordered packet causes a duplicate ACK, and the cost of an unnecessary retransmission for every reordered packet is too high.

37 Compare GBN, SR, and TCP. Assume that the timeout values for all three protocols are sufficiently long such that 5 consecutive data segments and their corresponding ACKs can be received (if not lost in the channel) by the receiving host (Host B) and the sending host (Host A) respectively. Suppose Host A sends 5 data segments to Host B, and the second segment is lost. In the end, all 5 data segments have been correctly received by Host B.

37.1 How many segments has Host A sent in total and how many ACKs has Host B sent in total? What are their sequence numbers? Answer this question for all three protocols.

These GBN and SR protocols acknowledge packets by echoing the segment number back to the sender. Acknowledge packet 0 with ACK 0.

GBN
Host A sends 9: 0, 1, 2, 3, 4, 1, 2, 3, 4
Host B sends 8: 0, 0, 0, 0, 1, 2, 3, 4

SR
Host A sends 6: 0, 1, 2, 3, 4, 1
Host B sends 5: 0, 2, 3, 4, 1

TCP's acknowledgement segments use the next-expected byte number. These hypothetical packets are one byte each, so acknowledge packet 0 with ACK 1.

TCP: Host A sends 6: 0, 1, 2, 3, 4, 1; Host B sends 5: 1, 1, 1, 1, 6

37.2 If the timeout values for all three protocols are much longer than 5 RTT, then which protocol successfully delivers all five data segments in the shortest time interval?

TCP. Because of fast retransmit, the three duplicate acknowledgements will trigger the missing segment to be reset before it times out.

38 In our description of TCP, the value of the threshold, $ssthresh$, is set as $ssthresh = cwnd/2$ in several places and the $ssthresh$ value is referred to as being set to half the window size when a loss event occurred. Must the rate at which the sender is sending when the loss event occurred be approximately equal to $cwnd$ segments per RTT? If the answer is no, suggest a different manner in which $ssthresh$ should be set.

At any given time, the sender's send rate is roughly $cwnd/RTT$ bytes per second. However, the size of the congestion window could have changed between the time the doomed packet was sent and the time its loss was noticed.

39 Consider figure 3.46. If λ'_{in} increases beyond $R/2$, can λ_{out} increase beyond $R/3$? If λ'_{in} increases beyond $R/2$, can λ_{out} increase beyond $R/4$ under the assumption that a packet will be forwarded twice on average from the router to the receiver? Explain.

No in both cases. The upper bound on throughput is $R/2$ because of the bidirectional nature of the connection. When λ'_{in} approaches and passes $R/2$, queuing delays increase, packets are lost, and throughput actually decreases.

40 Consider figure 3.58. Assume TCP Reno is the protocol experiencing the behavior shown above.

40.1 Identify the intervals of time when TCP slow start is operating.

(1,6) and (23,26)
Growth is exponential

40.2 Identify the intervals of time when TCP congestion avoidance is operating.

(6,16) and (17,22)
Growth is linear

40.3 After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

Triple duplicate ACK
cwnd cut in half, not set to 1 MSS.

40.4 After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

Timeout
cwnd set to 1 MSS.

40.5 What is the initial value of *ssthresh* at the first transmission round?

1 MSS

40.6 What is the value of *ssthresh* at the 18th transmission round?

21 MSS after being cut in half from 42

40.7 What is the value of *ssthresh* at the 24th transmission round?

14 MSS after being cut in half from 29

40.8 During what transmission round is the 70th segment sent?

7th

40.9 Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the congestion window size and of *ssthresh*?

$cwnd = 7$ MSS, $ssthresh = 4$ MSS

40.10 Suppose TCP Tahoe is used and assume that triple duplicate ACKs are received at the 16th round. What are the *ssthresh* and the congestion window size at the 19th round?

$ssthresh = 21$ MSS, $cwnd = 4$ MSS

40.11 Again suppose TCP Tahoe is used and that there is a timeout event at the 22nd round. How many packets have been sent out from the 17th round till the 22nd round, inclusive?

52 packets
1 packet at round 17, doubling until it hits *ssthresh* at round 22 and sends 21.

41 Refer to figure 3.55. Suppose that instead of a multiplicative decrease, TCP decreased the window size by a constant amount. Would the resulting AIAD algorithm converge to an equal share algorithm?

No, it wouldn't. Because the constant amount is not proportional to the window size, a participant with a larger share would retain a larger share, and a participant that began with a smaller share would never get a fair share of the bandwidth.

42 Doubling the timeout interval after a timeout event is a form of congestion control. Why does TCP need a window-based congestion-control mechanism in addition to this doubling-timeout-interval mechanism?

The window-based mechanism benefits the entire network by throttling sending rates. The timeout-interval mechanism doesn't change the sending rate, so it doesn't help to reduce congestion.

43 Host A is sending an enormous file to Host B over a TCP connection. Over this connection there is never any packet loss and the timers never expire. Denote the transmission rate of the link connecting Host A to the internet by R bps. Suppose that the process in Host A is capable of sending data into its TCP socket at a rate S bps, where $S = 10R$. Further suppose that the TCP receive buffer is large enough to hold the entire file, and the send buffer can hold only one percent of the file. What would prevent the process in Host A from continuously passing data to its TCP socket at rate S bps? TCP flow control? TCP congestion control? Or something else?

Flow control prevents the sender from overflowing the receiver's receive buffer, which isn't relevant in this scenario. Congestion control would only kick in when packets were dropped, which is impossible on this link. The bottleneck is the transmission rate of the link. Packets can only be sent into the network at a rate of R bps, even as the application tries to fill the send buffer at a rate of S bps. This will cause the buffer to fill up and stop accepting more data from the application.

44 Consider sending a large file from one host to another over a TCP connection that has no loss.

44.1 Suppose TCP uses AIMD for its congestion control without slow start. Assuming *cwnd* increases by 1 MSS every time a batch of ACKs is received and assuming approximately constant round-trip times, how long does it take for *cwnd* to increase from 6 MSS to 12 MSS?

6 RTT

44.2 What is the average throughput (in terms of MSS and RTT) for this connection up through time = 6RTT?

21 MSS are sent in the first 6 RTT, so the throughput is 21/6 = 3.5 MSS/RTT

45 Recall the macroscopic description of TCP throughput. In the period of time from when the connection's rate varies from $W/(2RTT)$ to W/RTT, only one packet is lost (at the very end of the period.)

45.1 Show that the loss rate is equal to $L = \frac{1}{3/8W^2+3/4W}$.

Since only one packet was lost during that interval, calculating the loss rate is equivalent to calculating the number of packets that were sent.

$\frac{W}{2} + (\frac{W}{2}+1) + (\frac{W}{2}+2) + ... + (\frac{W}{2}+\frac{W}{2})$

$\sum_{i=0}^{\frac{W}{2}} \frac{W}{2} + i$

$(\frac{W}{2}+1)(\frac{W}{2}) + \sum_{i=0}^{\frac{W}{2}} i$

$(\frac{W}{2}+1)(\frac{W}{2}) + (\frac{W}{2})(\frac{W}{2}+1)/2$

$\frac{W^2}{4} + \frac{W}{2} + \frac{W^2}{8} + \frac{W}{4}$

$\frac{3W^2}{8} + \frac{3W}{4}$

45.2 Use the previous result to show that if a connection has loss rate L, then its average rate is approximately $\frac{1.22MSS}{RTT\sqrt{L}}$

According to the text, the average throughput of a connection is $0.75W/RTT$ where W is the maximum window size. So, solve for W.

$L = 1/(\frac{3W^2}{8} + \frac{3W}{4})$
$L(\frac{3W^2}{8} + \frac{3W}{4}) = 1$
$3LW^2 + 6LW - 8 = 0$

By the quadratic equation, and because W must be positive,
$W = \frac{-6L+\sqrt{36L^2+96L}}{6L}$ MSS

Therefore, the throughput equals
$\frac{-6L+\sqrt{36L^2+96L}}{8L}$ MSS/RTT

However, this can be simplified. For large W, the linear term becomes insignificant. Ignoring that term and recalculating,
$L = 1/(\frac{3W^2}{8})$
$W^2 = \frac{8}{3L}$
$W = \sqrt{\frac{8}{3L}}$

Applying the average throughput formula,
$0.75 * \sqrt{\frac{8}{3L}}$
$\sqrt{\frac{9}{16}} * \sqrt{\frac{8}{3L}}$
$\sqrt{\frac{3/2}{L}}$

$\frac{1.22}{\sqrt{L}}$ MSS/RTT

46 Consider that a single TCP (Reno) connection uses one 10Mbps link which does not buffer any data. Suppose that this link is the only congested link between the sending and receiving hosts. Assume that the TCP sender has a huge file to send to the receiver, and the receiver's receive buffer is much larger than the congestion window. We also make the following assumptions: each TCP segment size is 1,500 bytes; the two-way propagation delay of this connection is 150 msec; and this TCP connection is always in congestion avoidance phase.

46.1 What is the maximum window size that this TCP connection can achieve?

The window size is the number of packets that can be sent per round trip time, which is calculated by adding the transmission time to the two-way propagation delay. The transmission time is calculated by dividing the segment size by the speed of the link.
Transmission time = 1500 $bytes * \frac{8*10^{-6}\ Mb}{byte} * \frac{1}{10Mbps} * \frac{1000msec}{sec}$ = 1.2 msec
RTT = 1.2 + 150 = 151.2 msec

Window size = 10 $Mbps * \frac{byte}{8*10^{-6}\ Mb} * \frac{segment}{1500\ bytes} * \frac{sec}{1000\ msec} * 151.2\ msec =$
126 segments

46.2 What is the average window size and average throughput of this TCP connection?

Because this connection is always in congestion avoidance phase, it ranges from a maximum of 126 segments to a minimum of 63 segments after being cut in half, and increases linearly. Averaging the two gives an average window size of 94.5 segments.

The average throughput of a connection is W/RTT where W is the average window size.
94.5 $segments * \frac{1500\ bytes}{segment} * \frac{Mb}{125000\ byte} * \frac{1}{151.2\ msec} * \frac{1000\ msec}{sec}$ = 7.5 Mbps

46.3 How long would it take for this TCP connection to reach its maximum window again after recovering from a packet loss?

Because the window increases by one MSS per RTT, it will take W/2*RTT seconds to reach W from W/2. 63*.1512 = 9.53 seconds

47 Consider the scenario described in the previous problem. Suppose that the 10Mbps link can buffer a finite number of segments. Argue that in order for the link to always be busy sending data, we would like to choose a buffer size that is at least the product of the link speed C and the two-way propagation delay between the sender and the receiver.

The product of the link speed and the two way propagation delay equals the bandwidth delay product, or the total capacity of the link. A buffer size equal to that capacity allows the link to fully utilize the available bandwidth, up to the total amount, as soon as it becomes available.

48 Repeat problem 46, but replacing the 10 Mbps link with a 10 Gbps link. You will notice that it takes a very long time for the congestion window size to reach its maximum window size after recovering from a packet loss. Sketch a solution to solve this problem.

Maximum window size: Link speed / Round trip time
Link speed: 10 Gbps = $1.25 * 10^9$ bytes per second
RTT: Transmission time + Propagation delay
Transmission time: 1500 $bytes * \frac{sec}{1.25*10^9\ bytes} * \frac{1000\ msec}{sec}$ = 0.0012 msec
RTT = 0.0012 + 150 = 150 msec
$W = \frac{1.25*10^9\ bytes}{sec} * \frac{sec}{1000\ msec} * 150\ msec * \frac{segment}{1500\ bytes} = 125,000$ segments

Average window size: $0.75 * 125,000 = 93750$ segments

Average throughput: Average window size / RTT

$93750 \; segments * \frac{1500 \; bytes}{segment} * \frac{8*10^{-9} \; Gb}{byte} * \frac{RTT}{.15 \; sec} = 7.5$ Gbps

Time to reach maximum window size after being cut:
W/2*RTT = 62,500*.150= 9,375 seconds, or 2.6 hours
This absurdly long time could be shortened by scaling the additive factor to the speed of the link. Since the 10 Gbps link is 1000 times faster, add 1000 MSS to the window each time instead of just one.

49 Let T, measured by RTT, denote the time interval that a TCP connection takes to increase its congestion window size from $W/2$ to W, where W is the maximum congestion window size. Argue that T is a function of TCP's average throughput.

W is a function of the link speed and RTT, and T is a function of W and RTT. Since the link speed is a constant, T is really a function of RTT. The average throughput is also a function of W and RTT and really only a function of RTT by the same argument. Since both vary according to the same variable, each can be expressed in terms of the other, so T can be expressed as a function of TCP's average throughput.

50 Consider a simplified TCP's AIMD algorithm where the congestion window size is measured in number of segments, not in bytes. In additive increase, the congestion window size increases by one segment in each RTT. In multiplicative decrese, the congestion window size decreases by half. Suppose that two TCP connections, C_1 and C_2 share a single congested link of speed 30 segments per second. Assume that both C_1 and C_2 are in the congestion avoidance phase. Connection C_1's RTT is 50 msec and connection C_2's RTT is 100 msec. Assume that when the data rate in the link exceeds the link's speed, all TCP connections experience data segment loss.

50.1 If both C_1 and C_2 at time t_0 have a congestion window of 10 segments, what are their congestion window sizes after 1000 msec?

At t=1000, C_1's window size will be 20, C_2's will be 12, and a packet will be lost during the current trip.

50.2 In the long run, will these two connections get the same share of the bandwidth of the congested link?

No, they will not, because their RTTs are different.

51 Consider the network described in the previous problem. Now suppose that the two TCP connections have the same RTT of 100 msec. Suppose that at time t_0, C_1's congestion window size is 15 segments but C_2's congestion window size is 10 segments.

51.1 What are their congestion window sizes after 2200 msec?

C_1's has just been cut to 8 and C_2's has been cut to 7.

51.2 In the long run, will these two connection get about the same share of the bandwidth of the congested link?

Yes. This is AIMD, and they have the same MSS and RTT, so they will eventually get a roughly equal share.

51.3 We say that the two connections are synchronized, if both connections reach their maximum window sizes at the same time and reach their minimum window sizes at the same time. In the long run, will these two connections get synchronized eventually? If so, what are their maximum window sizes?

Yes, in the long run the two connections will synchronize, and when they do, their maximum window sizes will be 15 segments each.

51.4 Will this synchronization help to improve the utilization of the shared link? Why? Sketch some idea to break this synchronization.

No, it won't. They only very briefly fully utilize the bandwidth and are immediately throttled. The problem is that all connections experience segment loss at the same time. If the connection instead allowed one to continue increasing after the other was throttled, it could approach full utilization. This could be achieved by using a buffer.

52 Consider a modification to TCP's congestion control algorithm. Instead of additive increase, we can use a multiplicative increase. A TCP sender increases its window size by a small positive constant a where $0 < a < 1$ whenever it receives a valid ACK. Find the functional relationship between loss rate L and maximum congestion window W. Argue that for this modified TCP, regardless of TCP's average throughput, a TCP connection always spends the same amount of time to increase its congestion window size from $W/2$ to W.

In this scenario, each RTT after the inital round sends the same number of segments before plus a fraction, a, of one for each ACK it receives. In other words, it sends $1 + a$ packets on the subsequent round for each packet in the previous one. Let $1 + a = \alpha$. Then on RTT n after the initial round of $W/2$ segments, $\alpha^n * W/2$ packets are sent.

The loss rate is equal to the inverse of the total number of packets sent between the initial round and the round where the full window size is realized, inclusive. I couldn't find a way to check this, but this is my shot in the dark:

First, find n such that $\alpha^n * W/2 = W$. This is the number of rounds before the congestion window reaches its maximum.
$\alpha^n = 2$
$n = log_\alpha 2$

Then, calculate how many packets were sent:
$\sum_{n=0}^{log_\alpha 2} \alpha^n * W/2$
$W/2 * \sum_{n=0}^{log_\alpha 2} \alpha^n$

Use the formula $\sum_{n=0}^{N-1} r^n = \frac{1-r^N}{1-r}$

$W/2 * \sum_{n=0}^{log_\alpha 2} \alpha^n$
$W/2 * \frac{1-\alpha^{log_\alpha 2-1}}{1-\alpha}$

The loss rate is equal to the inverse of the number of packets sent.

$L = \frac{2(1-\alpha)}{W(1-\alpha^{log_\alpha 2-1})}$

The number of rounds, and therefore the time, it takes for the sender to reach its maximum congestion window is constant with respect to n and the RTT, so it always spends the same amount of time to increase its congestion window size.

53 In our discussion of TCP futures, we noted that to achieve a throughput of 10 Gbps, TCP could only tolerate a segment loss probability of $2 * 10^{-10}$. Show the derivation of $2 * 10^{-10}$ for the given RTT and MSS values. If TCP needed to support a 100 Gbps connection, what would the tolerable loss be?

RTT = 100 ms = .1 seconds
MSS = 1500 bytes = .000012 Gb

Throughput $= 1.22 * \frac{MSS}{RTT\sqrt{L}}$
$10\ Gbps = 1.22 * \frac{.000012\ Gb}{.1\ sec * \sqrt{L}}$
$10 = .0001464/\sqrt{L}$
$\sqrt{L} = .00001464$
$L = 2.14 * 10^{-10}$

A similar calculation for an average throughput of 100 Gbps:
$100\ Gbps = 1.22 * \frac{.000012\ Gb}{.1\ sec\sqrt{L}}$
$100 = .0001464/\sqrt{L}$
$\sqrt{L} = .000001464$
$L = 2.14 * 10^{-12}$

54 In our discussion of TCP congestion control, we implicitly assumed that the TCP sender always had data to send. Consider now the case that the TCP sender sends a large amount of data and then goes idle at t_1. TCP remains idle for a relatively long period of time and then wants to send more data at t_2. What are the advantages and disadvantages of having TCP use the *cwnd* and *ssthresh* values from t_1 when starting to send data at t_2? What alternative would you recommend? Why?

The advantage of skipping slow start would be that the sender could start sending at a higher rate immediately. If the network is no more congested that it was at t_2, this will work well. In the not unlikely case that the network is more congested, packets will be lost quickly and the sender will need to recover. If the network is significantly more congested, even halving the previous *cwnd* value might not be enough to find an appropriate rate, so it would need to repeat the process until finds a window that is small enough.

Compared to the alternative of simply starting over, this is inefficient. Slow-start is not actually that slow, and it guarantees an appropriate rate is reached quickly. So, use the standard TCP slow-start protocol.

55 Investigate whether either UDP or TCP provides a degree of end-point authentication.

55.1 Consider a server that receives a request within a UDP packet and responds to that request within a UDP packet (for example, as done by a DNS server). If a client with IP address spoofs its address with address Y, where will the server send its response?

Y

55.2 Suppose a server receives a SYN with IP source address Y, and after responding with a SYNACK, receives an ACK with IP source address Y with the correct acknowledgement number. Assuming the server chooses a random initial sequence nunber and there is no "man-in-the-middle," can the server be certain that the client is indeed at Y and not at some other address X that is spoofing Y?

Yes. The SYNACK was sent to address Y, so in order for X to be able to send a convincing spoofed packet, it would have to be able to access messages received by Y in order to obtain the correct acknowledgement number.

56 Consider the delay introduced by the TCP slow-start phase and a client and Web server directly connected by one link of rate R. Suppose the client wants to retrieve an object whose size is exactly 15 S, where S is the maximum segment size. Denote the round-trip time between client and server as RTT, a constant. Ignoring protocol headers, determine the time to retrieve the object (including TCP connection establishment) in the following cases.

The time to establish the TCP connection is 1 RTT. After the connection is established, the sender will send the first segment, then two, then four, then the final eight as the previous rounds of segments are acknowledged. This takes RTT (establish connection) + S/R (first segment) + RTT (send/ack first segment) + 2*S/R + RTT + 4*S/R + RTT + 8*S/R + RTT.

Delay = 5 RTT + 15 S/R = 5(S/R + RTT) + 10*S/R.

56.1 4*S/R > S/R+RTT > 2*S/R

Upper bound: S/R+RTT = 4*S/R
5*4*S/R+10*S/R = 30*S/R
Lower bound: S/R+RTT = 2*S/R
5*2*S/R+10*S/R = 20*S/R

20*S/R < Delay < 30*S/R

56.2 S/R + RTT > 4*S/R

Lower bound: S/R + RTT = 4*S/R
5*4*S/R+10*S/R = 30*S/R

Delay > 30*S/R

56.3 S/R > RTT

Upper bound: RTT = S/R
5*S/R+15*S/R = 20*S/R

Delay < 20*S/R