Solutions to Study Guide Problems (Mid-Term #2)

CS 352 Internet Technology

Released: November 06, 2022

These are sample solutions to the textbook problems mentioned in the study guide for mid-term 2. Some problems and questions may admit multiple solutions that are correct. Please contact us if you require clarifications on these or other solutions.

You are *strongly encouraged* to try answering the questions by yourself first without looking at these solutions. Only use these solution (sketches) once you've identified where you get stuck, to help guide your thought process and reasoning, and fix any conceptual misunderstandings or gaps.

The questions and solutions on the exam will be different from the ones below. The primary utility of these solutions is to help you gain a deeper conceptual understanding of the material. There is no point in memorizing or reproducing any specific solution below for the exam.

Chapter 3, R3. The source and destination parameters are just reversed on packets flowing in the reverse direction. Hence, the source (destination) port number of packets flowing from B to A is y (x).

Chapter 3, R7. UDP is connectionless and its packets are demultiplexed purely using the (dst IP, dst port) on the packet. Hence, UDP packets from both *A* and *B* will be sent to the same socket on *C*.

The UDP application at *C* can identify the sender of a given message by fetching the address of the segment corresponding to the message that was received. UDP servers typically use the `recvfrom()` system call to achieve this. You can read more about this system call here: `https://linux.die.net/man/2/recvfrom`

Chapter 3, R8. TCP is connection-oriented and its established sockets are demultiplexed using the 4-tuple of (src IP, src port, dst IP, dst port). Hence, TCP packets from the two source endpoints *A* and *B* will go to different sockets. Both sockets can use destination port 80, because the network stack uses the 4-tuple to find the socket, not just the (dst IP, dst port).

(More subtleties can arise: there can be a listening socket also on port 80 which will accept fresh connections coming into the (dst IP, dst port), through a `connect()` call from yet another client endpoint. The demultiplexing rules of TCP make all of these possible to coexist.)

Chapter 3, R14.

(a) False. ACKs are piggybacked on data if there is any data being transmitted in the reverse direction. Otherwise, ACKs will be sent as their own packets.

(b) False. Here, the textbook uses the term `rwnd` to denote the flow control or the receiver-advertised window. Indeed, the size of this window changes over time, based on the time-varying

processing speed of the receiving application and the time-varying socket buffer availability.

(c) (conditionally) True. Suppose at one instant of time, a TCP sender keeps its window size smaller or equal to the latest receiver-advertised window size (which is itself smaller than the receiver's buffer size). Assuming the size of the receiver's allocated memory buffer is constant over time, the sender will not increase the amount of its in-flight data beyond the receiver's socket buffer size.

One exception to this condition is when the receiver has to slash its receiver buffer memory due to memory pressure on the system. In such extreme cases, it is possible that data in flight now is already out of the valid range of bytes that can be held at the receiver socket buffer. A correctly-implemented TCP receiver will not renege on its previously advertised window size by dropping data that is cumulatively ACKed, but may do so by dropping data that was selectively (but not cumulatively) ACKed.

(d) False. TCP uses byte sequence numbers. So the sequence number of the next segment will depend on how many bytes there are in the current segment.

(e) True. This is the window size field.

(g) False. The ACK numbers from A to B are incomparable to the sequence numbers from A to B, because the ACK numbers corresponds to the sequence numbers of data sent from B to A.

Chapter 3, R15.

(a) Recall that TCP uses byte sequence numbers. The amount of application data in the segment is equal to the difference in the starting sequence numbers between two adjacent segments of the connection. So the number of bytes is 110 - 90 = 20 bytes.

(b) Assuming that the TCP receiver has indeed receiver all the data sent before these two segments, the cumulative ACK number will be 90, since that is the first sequence number expected by the receiver in order.

Chapter 3, P1.

(a) Possible source ports p_a include values that may be chosen by the client network stack, from 1 through 65K. The server likely uses a fixed destination port p_s depending on the application being used. Possible values are in the same range as the client, from 1 through 65K.

(b) Possible source ports p_b include values that may be chosen by the client network stack, from 1 through 65K. The server will use the same value as part (a), p_s.

(c) The source and destination ports of a segment from S to A are the reverse of those of segments from A to S. Hence, the source port is the fixed server port (from part (a)) p_s, and the destination port is the client port p_a from part (a).

(d) Similarly to part (c), the source port is p_s and the destination port is p_b from part (b).

(e) Yes, this is possible.

(f) Telnet uses TCP by default (not clear or obvious from our discussions in class, so it's OK if you didn't get this). So, if A and B were the same endpoint, they cannot use the same local port for the two connections to S. Hence, $p_a \neq p_b$.

Chapter 3, P4.

(a) Doing regular binary addition of these two values results in 1100 0001. Since there is no carryout, this is also the 1s complement sum.

(b) Doing regular binary addition of these two values results in 1 0011 1111. There is a carry-out bit of 1, which, when added to the 8-bit residue produces 0100 0000.

(c) We could flip some bit position from the values in part (a) where one operand has a 0 bit and another has a 1 bit. An example: 01011101 and 01100100. The sum would be the same but there are 2 bits flipped with respect to the values in part (a).

Chapter 3, P22.

(a) Suppose x is an integer such that $k-4 \leq x \leq k$. Then, the sender's window contains the sequence numbers $x \ (mod\ 1024), x+1 \ (mod\ 1024), x+2 \ (mod\ 1024), x+3 \ (mod\ 1024)$, where $a \ (mod\ b)$ is the remainder upon dividing a by divisor b. We need this remainder operation because we need the sequence numbers to fall in the range $[0, 1023]$. The reasoning behind the range provided for the starting sequence number x is as follows:

The receiver is expecting sequence k; the sender may or may not have received all the ACKs that the receiver sent out to slide the receiver window to this position. The sender may be lagging the receiver by an entire window (hence starting its window at $k-4 \ (mod\ 1024)$), or synchronized with the receiver (hence starting its window at k). The sender cannot lag the receiver by a gap larger than the window size because the receiver would not have received sequence $k-1 \ (mod\ 1024)$. The sender's window cannot start ahead of k because the receiver hasn't yet ACKed sequence k.

(b) There may be ACKs in flight for at most 4 sequence numbers, $k-4 \ (mod\ 1024), k-3 \ (mod\ 1024), k-2 \ (mod\ 1024), k-1 \ (mod\ 1024)$. The reasoning follows from part (a).

Chapter 3, P25.

Having more control of data in each packet: TCP is a stream-oriented protocol; individual `send()` calls may not translate to the data being sent in a single packet. The TCP stack is free to fragment the data into multiple packets or combine data from multiple `send()` calls into a single packet. On the other hand, UDP `sendto()` calls allow the application to put a specific message into a single UDP packet (assuming the message is small enough to fit in a single packet).

Having more control on when segment is sent: The TCP stack may delay the transmission of a packet when the window does not allow it, e.g., when the flow control window or congestion control window restricts transmission of the provided data. UDP does not have any notions of flow or congestion control. The UDP `sendto()` call is a signal to the network stack to try and transmit the packet at the earliest opportunity.

Chapter 3, P26.

(a) The number of bytes L is such that the 32-bit sequence number field does not roll over. That is, $L = 2^{32}$.

(b) Each packet being sent out contains an MSS (536 bytes) worth of application data, along with 66 bytes of headers. Hence, each packet has a size of 536 + 66 = 602 bytes. The number of packets is $L/536$. Hence, the amount of time it takes to move this data over a 155 Mbit/s link is $\frac{(L/536) * 602 * 8}{155 * 10^6}$. If we plug in the value of L from part (a), this comes out to 248 seconds. This is the time it takes to roll over the entire TCP sequence number space on a 155 Mbit/s link. (On faster links, like 10 Gbit/s, it may happen much faster, just a few seconds!)

Chapter 3, P36. It is desirable to wait for a few duplicate acknowledgments so that any packet that was transmitted out of order may still be able to reach the receiver. Waiting for more dup

ACKs prevents the sender from misconstruing an instance of packet reordering as packet loss.

Chapter 3, P40.

(a) 0 – 6, 23 – 26

(b) 6 – 16, 17 – 23

(c) triple duplicate ACK. Observe the multiplicative decrease to half the congestion window size.

(d) timeout. Observe the drop in the congestion window size to a low value, 1.

(e) 32. Observe the point where slow start turns to congestion avoidance at the 6th transmission round.

(f) Approximately 24. Observe the point where there is a multiplicative decrease at the 17th transmission round.

(g) The `ssthresh` is half of the congestion window at transmission round 22, right before the timeout. Approximately 14.5.

(h) 7th transmission round. We need to add up the number of segments sent each transmission round up to the point where the total gets to 70. Up to the 6th transmission round, the sender has transmitted 1 + 2 + 4 + 8 + 16 + 32 = 63 segments. In the 7th transmission round, the sender sends 33 segments. The 70th segment of the data is part of this transmission.

(i) 4 segments, using the value of `ssthresh` from part (g) and the behavior of fast recovery.

Chapter 3, P43. The TCP sender will be unable to send data because it will be put to sleep by the operating system. Since the application can write data into the send socket buffer at speed $10 * R$, but that buffer only drains at rate R, the send socket buffer will fill up eventually. Instead of dropping any data that the application writes, a subsequent `send()` from the application at this point will just put the application to sleep. Note that `send()` is a blocking system call that passes control to the operating system from the application. If there is no space in the sender socket buffer, the application will never be woken up until buffer space is available.

Chapter 3, P44.

(a) 7 round-trip times. The `cwnd` values of each corresponding RTT are: 6, 7, 8, 9, 10, 11, 12.

(b) The average throughput is the total data divided by the total time. The total amount of data is 6 + 7 + ... + 11 MSS, which is 51 MSS. This transmission occurred over a period of 6 RTTs. The throughput is 8.5 MSS/RTT. There is also a simpler method to compute this: due to the linear growth in the `cwnd`, the average throughput corresponds to the halfway point in the `cwnd` evolution over time: that is, (6 + 11) / 2 = 8.5 MSS/RTT.

Chapter 3, P46.

(a) Since the buffer size is 0, the TCP sender will drop packets when the congestion window size exceeds the bandwidth-delay product (BDP). That is, the maximum value of `cwnd` before a packet loss is (10 Mbit/s) * (150 ms) / (1500 bytes * 8 bits/byte) = 125 segments.

(b) Assuming there is only linear increase in the congestion window, following from the reasoning in part (b) of P44 in this chapter, the average window size of this connection is (1 + 125)/2 $\approx$ 65 segments. The average throughput is (65 segments * 1500 bytes/segment * 8 bits/byte) / (150 ms) = 5.2 Mbit/s.

(c) Since the window increases by 1 segment per round-trip time, it takes 124 round-trip times

before the window will reach the maximum value, i.e., 124 * 150 ms = 18.6 seconds.

Chapter 3, P47. To always keep the link busy, the sender *on average* must have at least *BDP* data in flight. (From the lecture slides on bandwidth-delay product, if `cwnd` $\leq BDP$, the link will not be fully used.) The buffer size B must be such that the average of the maximum congestion window size $BDP + B$ and the min window size (i.e., 1 segment) is equal to *BDP*. That is, $B \approx BDP$. This is sometimes called the *router buffer-sizing rule.* A single TCP connection requires a buffer of size *BDP* to keep the bottleneck link fully used.

Chapter 3, P50.

(We will ignore the link capacity provided in this question and assume that it is a very large value, since the link rate of 30 segments per second is very small compared to the rates achieved by the window sizes provided in part (a) of this question.)

(a) C_1 ramps up its congestion window faster because it has a smaller RTT. By t = 1000 ms, C_1 has had (1000/50) = 20 round-trip times, hence its `cwnd` is 10 + 20 = 30 segments. On the other hand, C_2 has had (1000/100) = 10 round-trip times, hence its `cwnd` is 10 + 10 = 20 segments.

(b) No. Unfortunately, in a given duration of time, the windows of the two connections grows at different rates. Specifically, the increment during the additive increase phase for connection C_1 is twice that of connection C_2. Multiplicative decrease does not really remedy this unfairness, since the windows will continue to grow at unequal rates. This is the phenomenon of *RTT unfairness* in TCP: when one connection has a smaller RTT, it has a competitive advantage against a connection that has a larger RTT since the first connection gets feedback to adapt (increase) its window faster.

Chapter 3, P56.

We will assume there are no queues or packet losses since there is only one link that directly connects the client to the server. Further, suppose it takes one RTT to establish a TCP connection between the client and server. Hence, in all cases, we need to add one RTT of time to the total time it takes to actually transmit the file data.

S/R is the time it takes to push all the bits of one packet into the link. $S/R + RTT$ is the time it takes to transmit one packet over the link and receive an ACKnowledgment, including both propagation and transmission delays. If $S/R + RTT = n \cdot S/R$ for some $n \geq 2$, then the server can push $n-1$ additional packets into the link by the time the ACK for the first packet arrives. If the server did keep n packets in flight, the link is fully used. Indeed n is the largest window size attainable by the server (in packets).

However, if $S/R + RTT < 2 \cdot S/R$, the server can't even push one additional packet by the time the ACK for the first one arrives. That is, the data transmission rate is severely limited by the bottleneck link rate.

(a) For this case, we are given that $4 \geq n \geq 2$. That is, slow start will send one packet in the first RTT of data transmission, then two packets (having accumulated $3S$ worth of data at the client), but in the third RTT, the server will be limited to sending just n packets per RTT, since $n \leq 4$. Hence, the number of further RTTs required to transmit the remaining $12S$ of data is $(12/n)$ RTTs, where $n = \frac{S/R+RTT}{S/R}$. The total time is then 1 RTT (connection establishment) + 2 RTTs (slow start) + $\frac{12 \cdot (S/R+RTT)}{S/R}$ RTTs. Note that $3 \leq 12/n \leq 6$, so the transfer will complete between 6 and 9 RTTs.

(b) We are given $n > 4$. Hence, the server can accumulate $7S$ worth of data at the client by

implementing slow start doubling over three RTTs ($1S + 2S + 4S$). The remaining $8S$ data can be transmitted over $8/n$ RTTs. So the total time is 1 RTT (connection establishment) + 3 RTTs (slow start) + $\frac{8 \cdot (S/R + RTT)}{S/R}$ RTTs. Note that $8/n$ is at most 2, so the transfer will complete in at most 6 RTTs.

(c) If $RTT < S/R$, then $S/R + RTT < 2 \cdot S/R$, that is, $n < 2$. In the first RTT of actual data transmission, the server pushes out S data, whose ACK returns in time $S/R + RTT$. Slow start will increase the server's window to $2S$. At any window size that is larger than S, the time it takes to push two or more packets (belonging to one window) through the link is larger than the time it takes for the ACK of the first packet of that window to return to the server. Hence, starting from the point where the server's window size is $\geq 2S$, the link will always be fully in use, independent of how large the window grows. Hence, the total time is 1 RTT (connection establishment) + $(S/R + RTT)$ (slow start with window size $1S$) + $14S/R$ (the time it takes to retrieve the remainder of the file $14S$ at link rate R).