1. Consider transferring an enormous file of L bytes from Host A to Host B. Assume an MSS of 1,460 bytes.

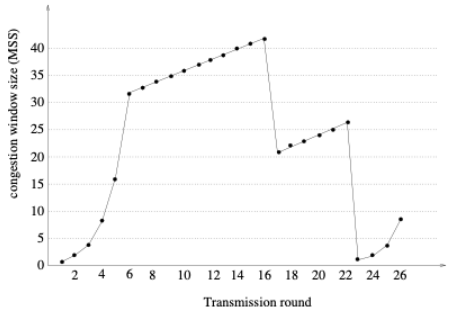
   (a) What is the maximum value of L such that TCP sequence numbers are not exhausted? Recall that the TCP sequence number field has 4 bytes.

   TCP sequence number is in terms of bytes.
   $L = 2^{32} = 4294967296$ bytes.

   (b) For the L you obtain in (a), 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 10 Mbps link. Ignore flow control and congestion control so A can pump out the segments back to back and continuously.

   Each segment has 1460 bytes payload plus 66 bytes overheads.
   $2^{32} \times \frac{1460+66}{1460} \times 8/(10^7) \approx 3591.3$ seconds.

2. Assuming TCP Reno is the protocol experiencing the behavior shown below. Answer the following questions. Provide a short discussion (be brief) justifying your answer.



   (a) Identify the intervals of time when TCP slow start is operating.

   Round 1- 6, and 23 - 26. The congestion window size is exponentially increased during the slowstart stage.

   (b) Identify the intervals of time when TCP congestion avoidance is operating.

   Round 6 - 16, 17 - 22. The congestion window size is linearly increased during the congestion avoidance stage.

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

   By triple duplicate ACK, because the congestion window size is halved after a triple duplicate ACK.

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

   By a timeout, because the congestion window size is reduced to one after a time out.

   (e) What is the initial value of slowstart threshold at the first transmission round?

   32 MSS. When the congestion window is smaller than 32, the window is exponentially increased; when the congestion window exceeds 32, it is linearly increased.

   (f) What is the value of slowstart threshold at the 18th transmission round?

   21 MSS, because the slowstart threshold equals the half of the window size when the packet loss is detected.

   (g) What is the value of slowstart threshold at the 24th transmission round?

   13 MSS, because the slowstart threshold equals the half of the window size when the packet loss is detected.

   (h) During what transmission round is the 70th segment sent?

   7. During the first six round, $1 + 2 + 4 + 8 + 16 + 32 = 63$ segments are sent; during the seventh round, another 33 segments are sent.

   (i) 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 the slowstart threshold?

   4 MSS. The congestion window size and the slowstart threshold are both equal to the half of the window size at 26th round.

3. We note that TCP waits until it has received three duplicate ACKs before performing a fast retransmit. Why do you think the TCP designers chose not to perform a fast retransmit after the first duplicate ACK for a segment is received?

   Since the IP networks do not provide any guarantee reliable and in-order packet delivery services, a single duplicate ACK may be due to packet re-order, or packet duplication, instead of packet loss. Fast retransmission after the first duplicate ACK may not be necessary and may waste network resources (because TCP un-necessarily reduces its cwnd after non-congestion events).

4. Optional question: A TCP sender sends an enormous file over a link between routers A and B, with 10 Mbps. No other traffic share the link with it. Assume that there is no transmission error of the link, and the routers' buffer size is much smaller than the file size. Is it possible that there is no packet lost for the whole file transfer? If so, how?

   Yes, it's possible. Since the sender's window size is the minimal of the congestion window size (cwnd) and the receiver advertised window size (rwnd), we can set the rwnd low enough to avoid overshooting the link bandwidth, so no congestion loss will happen.

1. For a simplified HTML document with URL http://a.b.com/index.html

   <html>
    <img src="image/fig1-small.jpg">
   <img src="http://x.y.com/image/fig2.gif">
   <img src="http://a.b.com/image/fig3.gif">
   </html>

   For a web browser that loads images (JPEG and GIF files) automatically, how many TCP connections will be opened by the browser to display this Web page properly, and:
   (a) non-persistent HTTP connections are used, or
   Ans : 4 connections (1 for the HTML file and 3 for the 3 images).
   (b) persistent HTTP connections with non-pipelining are used, or
   Ans : 2 connections (1 connection to each server).
   (c) persistent HTTP connections with pipelining are used.
   Ans : 2 connections (1 connection to each server).

2. For a simplified HTML document with URL http://a.b.com/index.html

   <html>
    <img src="image/fig1-small.jpg">
   <img src="http://x.y.com/image/fig2.gif">
   <img src="http://a.b.com/image/fig3.gif">
   </html> For a web browser that loads images (JPEG and GIF files) automatically, how long (in round-trip time) does it take to display this Web page properly (DNS overhead is omitted), if all files are small enough to be accommodated in one packet, and:
   (i) non-persistent HTTP connections are used, or
   (ii) persistent HTTP connections with non-pipelining are used, or
   (iii) persistent HTTP connections with pipelining are used.

   The web-browser may or may not use parallel TCP connections to speed up the download speed. We consider the following cases separately.

   i) nonpersistent, no parallel TCP:

   For each TCP connection, the first round-trip time is used to exchange two hand-shake packets. The ACK to the 2nd handshake packet can be piggy-backed in the request from the client. So, it takes two RTTs for each object and the HTML file.
   $\rightarrow$ totally 8 RTTs

   ii) nonpersistent, with parallel TCP connections:

   2RTTs for HTML plus 2 RTTs for the 3 objects requested with parallel TCP connections
   $\rightarrow$ totally 4 RTTs

   iii) persistent, non-pipeline, no parallel TCP:

   2RTTs for HTML. 2 RTTs for 2 objects in the same server, 2 RTTs for the object in the other server
   $\rightarrow$ 6 RTTs

   persistent, non-pipeline, with parallel TCP connections to different servers:

   2 RTTs for HTML. 2 RTTs for 2 objects in the same server; at the same time, 2 RTTs for the object in the other server
   $\rightarrow$ 4 RTTs

iv) persistent, pipeline, no parallel TCP:

2 RTTs for HTML. 1 RTT for 2 objects in the same server, 2 RTTs for the object in the other server
→5 RTTs

persistent, pipeline, parallel TCP connections to different servers:

2 RTTs for HTML. 1 RTT for 2 objects in the same server; at the same time, 2 RTTs for the object in the other server
→4 RTTs

Question 1: Let 01111110 be the FLAG. To transmit the flowing bit string 011110000111110001111110 at the data link layer, what is the string transmitted after bit stuffing?

Solution.

Bit stuffing is used in data transmission to ensure proper synchronization and to distinguish between data bits and control bits. In this context, bit stuffing is employed to prevent the occurrence of consecutive sequences that might be mistaken for control flags, such as the FLAG sequence 01111110. When transmitting the bit string 011110000111110001111110, bit stuffing ensures that the transmitted string contains the necessary additional bits to avoid consecutive sequences of 1s. In this case, after bit stuffing, the transmitted string becomes 0111100001111100001111010, where an extra 0 is inserted after every sequence of five consecutive 1s, ensuring that the FLAG sequence is not inadvertently detected within the data.

0111100001111100001111010, where reds are consecutive "1", after 5 consecutive "1" there's a blue marked 0.

Question 2: The following character encoding is used in a data link protocol:

A: 01111000 B: 01111100 FLAG: 01111110 ESC: 11100000

Show the bit sequence transmitted (in binary) for the four-character frame: "A B ESC FLAG" when FLAG bytes with byte stuffing is used.

Solution.

ESC characters used in byte stuffing are distinct from control characters and are specifically employed to ensure that some of the bit sequences and flags within the data frame are correctly interpreted as part of the transmitted data, not as control sequences or characters by the receiver. Here, the four-character frame is part of the bit sequence which includes a byte stuffing ESC. When we are transmitting the bit sequence, we are making sure that the receiver understands that the byte stuffing ESC symbol is part of the data sequence itself, but not part of the control sequences. For that, we are adding a control sequenced ESC before the byte stuffing ESC. We are also adding the control sequenced ESC before FLAG.

Here the bit sequence is A B ESC FLAG &
The transmitted sequence is A B **ESC** ESC **ESC** FLAG.

Following the character encoding used in the data-link protocol, the transmitted bit sequence which includes the data frame is:

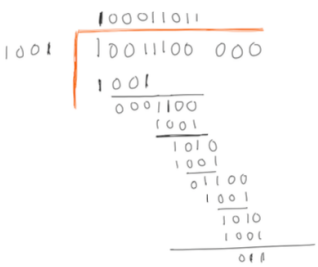
01111000 011111100 11100000 11100000 11100000 01111110

(Concatenated) 011110000111111001110000011100000111000000111110

Question 4: A bit stream 10011100 is transmitted using the standard CRC method. The generator polynomial is $x^3 + 1$.

a) Show the actual bit string transmitted.

Solution.



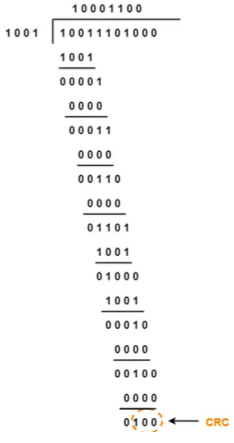
Dividing 10011100 by $x^3 + 1$ which is an equivalent of $1x^3 + 0x^2 + 0x^1 + 1$ or, 1001

We get the CRC as **011**.

Therefore, the bit string transmitted is **10011100011**.

b) Suppose one bit is inverted during transmission. Show that this error is detected at the receiver's end.

Solution.



Let's assume that in the transmitted bit sequence (10011100) the first bit from the right gets inverted (10011101). Dividing it by 1001 will give us a CRC of **0100**.

We know that for the transmission to be detected we use the CRC method where we use Cyclic Redundancy Check (CRC) to validate our transmitted bit sequence. This involves appending a checksum to the transmitted data based on polynomial division. At the receiver's end, the received data is divided by the same polynomial. If the remainder is zero, the transmission is deemed error-free.

From our worked-out example, we can easily see that our CRC is not 0. This is how the error is detected on the receiver's end.

Please Turn Over

Question 3: An 8-bit byte with binary value 10101100 is to be encoded using Hamming code. What is the binary value after encoding? [Hint: use 4 check bits]

Solution.

Given, the binary value is 10101100, which is an 8-bit binary value. We know 8-bit sequence can contain 4 check bits (2^0=0, 2^1=2, 2^2=4, 2^3=8). We can also check if the number of check bits (r) and data bits (m) follows the relation mentioned in the textbook as:

$m+r+1 \leq 2^r$
or, $8+4+1 \leq 2^4$
or, $13 \leq 16$ *which satisfies the condition for the number of checkbits = number of parity bits.*

Bit Position	1	2	3	4	5	6	7	8	9	10	11	12
Binary Before Encoding	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100

From the Binary Values before encoding, we get this table:

Seq.	Binary Table				Check Bit Positions	XOR
1	0	0	0	1	Check bit positions for 1: 3, 5, 7, 9, 11	0
2	0	0	1	0	Check bit positions for 2: 3, 6, 7, 10, 11	1
3	0	0	1	1		
4	0	1	0	0	Check bit positions for 4: 5, 6, 7, 12	0
5	0	1	0	1		
6	0	1	1	0		
7	0	1	1	1		
8	1	0	0	0	Check bit positions for 8: 9, 10, 11, 12	1
9	1	0	0	1		
10	1	0	1	0		
11	1	0	1	1		
12	1	1	0	0		

From this stipulation, we have:

Bit Position	1	2	3	4	5	6	7	8	9	10	11	12
Binary Before Encoding	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
Data Bits			1		0	1	0		1	1	0	0
Check Bits	0	1		1				0				
Binary After Encoding	0	1	1	1	0	1	0	0	1	1	0	0

5. With the 2-D parity method, a block of bits with n rows and k columns uses horizontal and vertical parity bits for error correction or detection. During the class, we have shown that 2-D parity method can correct single bit errors.

(a) Whether the 2-D method can detect double errors? Triple errors?

Solution.

The 2-D parity method can detect double errors as well as triple errors with some exceptions.

- If two bits are inverted in the same row, the parity checks for that row and column will fail, indicating an error.
- If two bits are inverted in the same column, and both bits are opposite to each other (one is a 1, other is a 0) then the chance of getting undetected arises.
- If three bits are inverted, chances of detection are highly probable with some exceptions.

Answer. 2D parity method can detect both double and triple inversions.

(b) Find an example of a pattern of six errors that cannot be detected using 2-D parity method.

Solution.

<>	<>	<>	<>	<>
<>	1	1	0	<>
<>	1	0	1	<>
<>	0	1	1	<>
<>	<>	<>	<>	<>

In this data sequence, <> signifies negligible bits. Errors in these consecutive rows and columns will not be detectable since the resultant number of odds and even numbers will be equal to the number before the inversion of the bits.