# Problem 1

Host A sends 6 data segments to Host B, the 3rd segment (sent from A) is lost. This loss is then detected by the protocol-specific means and the protocol invokes recovery actions. In the end, all 6 data segments are correctly received by Host B.

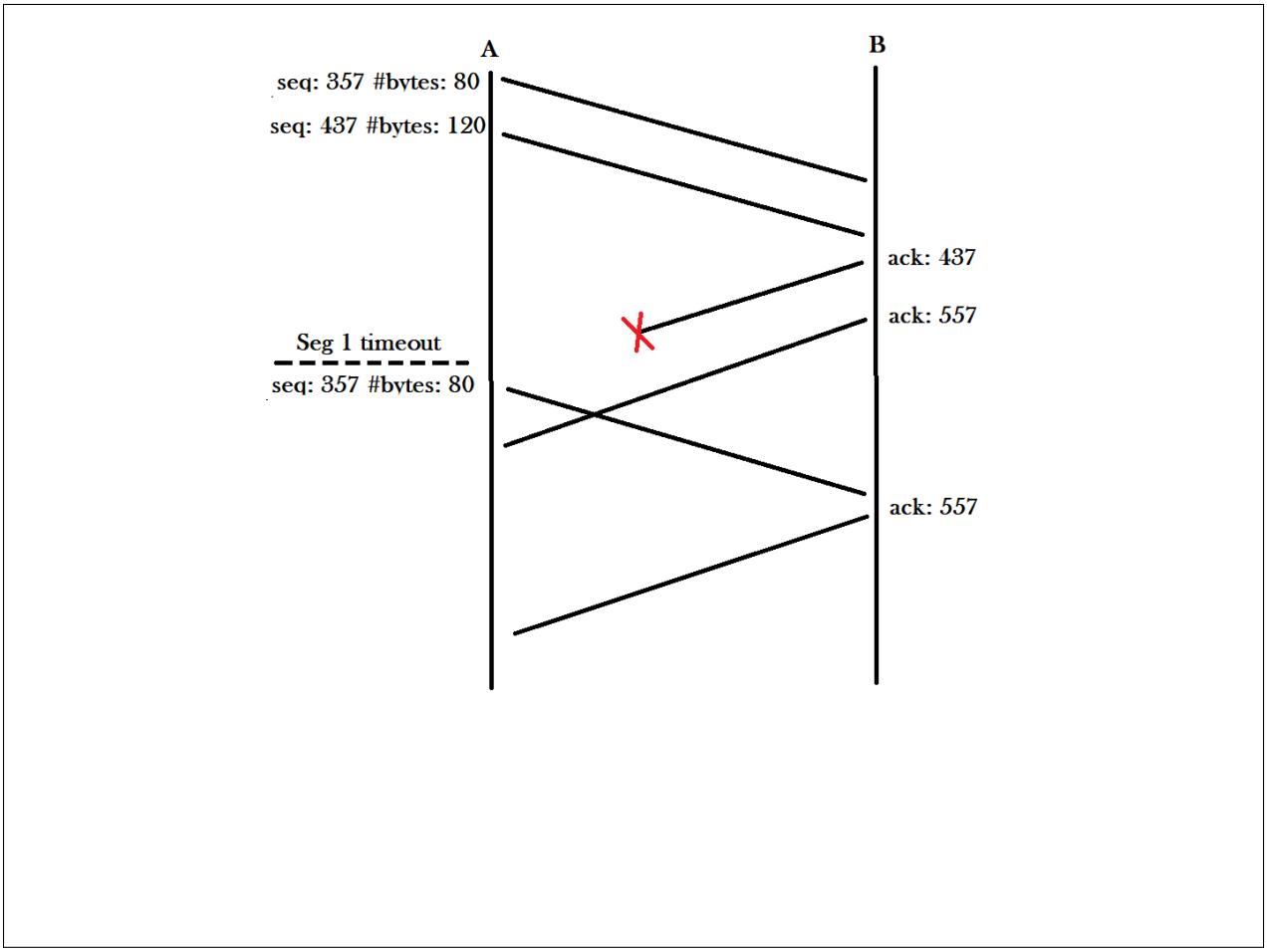
1. If A and B use Go-back-N for the data delivery and every received segment will be acknowledged, what is the total number of segments that Host A sent out (including retransmissions)? And what is the total number ACKs that Host B sent?
2. If A and B use TCP for the data delivery (no delayed ACKs), what is the total number of segments that Host A sent out (including retransmissions)? And what is the total number ACKs that Host B sent?
3. Assume that the retransmission timer for the above two protocols (Go-back-N and TCP) are the same and set to 8*RTT, which protocol finish the data delivery first?

1. 6 initial transmits, 5 acks, then 4 retransmits, then 4 more acks. 10 segments sent by Host A, 9 acks sent by Host B.
2. 6 initial transmits, 5 acks (3 of which are identical), then 1 retransmit, followed by a single cumulative ack. 7 segments sent by Host A, 6 acks sent by Host B.
3. TCP will finish data delivery first. After receiving the first 2 packets, Host B will send an ack expecting Segment 3. Packet 3 will then be sent, and the RTO will begin. Packets 4-6 will then be sent, resulting in 3 duplicate acks, since Host B is still expecting Segment 3. This will trigger the TCP Fast Retransmit, with an maximum effective RTO of around 3*RTT + $x$ , where $x$ is the delay between sending Packet 3 and 4. This is still less than the RTO of 8*RTT that we would've had to wait for prior to retransmitting with Go-back-N.

# Problem 2

Host A and B are communicating over a TCP connection, and Host B has already received from A all bytes up through byte 356 (Assume the random chosen initial sequence number happens to be 0). Suppose Host A then sends two segments to Host B back-to-back. The first and second segments contain 80 and 120 bytes of data, respectively. In the first segment, the sequence number is 357, the source port number is 30200, and the destination port number is 8080. Host B sends an acknowledgment whenever it receives a segment from Host A. Fill in the blanks for questions (a) – (c) directly; work out the diagram in the box for question (d).

1. In the second segment sent from Host A to B, the sequence number is 437, source port number is 30200, and destination port number is 8080.
2. If the first segment arrives before the second segment, in the acknowledgment of the first arriving segment, the ACK number is 437, the source port number is 8080, and the destination port number is 30200.
3. If the second segment arrives before the first segment, in the acknowledgment of the first arriving segment, the ACK number is 357.
4. Suppose the two segments sent by A arrive in order at B. The first acknowledgment is lost and the second acknowledgment arrives after the first timeout interval. Draw a timing diagram in the box below, showing these segments and all other segments and acknowledgment sent. Assume no additional packet loss. For each segment in your diagram, provide the sequence number and the number of bytes of data; for each acknowledgment that you add, provide the ACK number.



# Problem 3

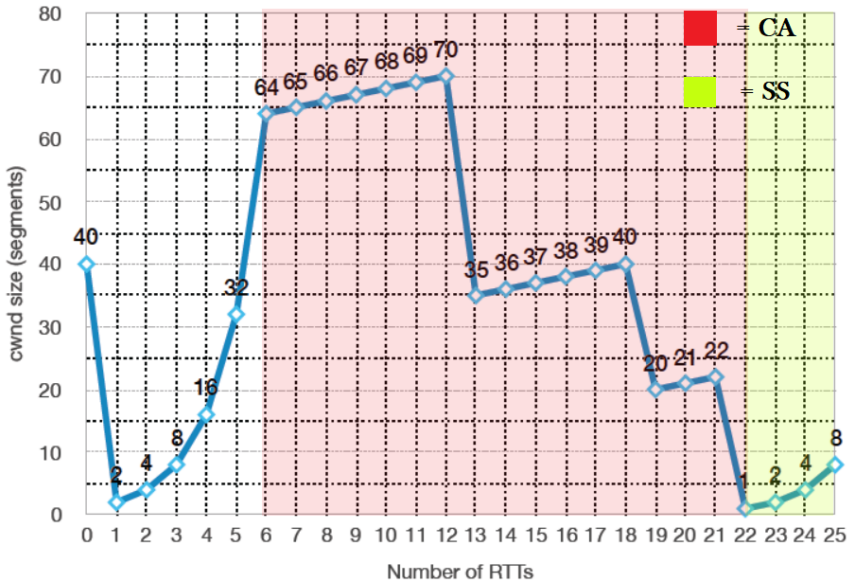



Figure 1: TCP congestion control.

Assume that a TCP connection has been running between hosts A and B for sometime, so that the number of RTTs shown in the above graph are with respect to the time when you started observing the the cwnd value of this connection. Hosts A and B use TCP Reno (with Fast Retransmit and Fast Recovery).

1. **On the graph above**, identify the time periods when TCP slow start is operating.
2. **On the graph above**, identify the time periods when TCP congestion avoidance is operating (AIMD).
3. For each loss event, specify whether it was detected by a triple duplicate ACK or by a timeout
4. For each loss event, indicate the value of the slow start threshold (ssthresh)

1. The first loss event at RTT=0 was neither timeout nor triple ack, since the cwnd size is set to 2, which is not a property of congestion avoidance.

   The second loss event at RTT=12 was a triple ack.

   The third loss event at RTT=18 was a triple ack.

   The fourth loss event at RTT=21 was a timeout.

2. Assuming that this problem refers to the sshthresh value at the time of loss:

   For the first loss event at RTT=12, sshthresh=64.

   For the second loss event at RTT=18, sshthresh=35.

   For the third loss event at RTT=21, sshthresh=20.

# Problem 4

Consider that only a single TCP (Reno) connection uses one 10 Mbps 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 100 msec; and this TCP connection is always in congestion avoidance phase, that is, ignore slow start.

1. What is the maximum window size (in segments) that this TCP connection can achieve?
2. What is the average window size (in segments) and average throughput (in bps) of this TCP connection?

I will not be considering the latency induced by transmission delay in the problem since it does not specify that we should consider that factor.

1. $window_{max} = \frac{10^7 bps * 0.1 sec}{8 * 1500 bits} = 83.33$
   Maximum window size is 83.33 segments.
2. Window size will vary from $window_{max}$ to $window_{max}/2$. Therefore,
   $window_{avg} = window_{max} * 0.75 = 62.5$
   Average window size is 62.5 segments.
   $throughput_{avg} = \frac{62.5 * 8 * 1500 bits}{0.1 sec} = 7.5 Mbps$
   Average throughput is $7.5 * 10^6$ bps

# Problem 5

As we have discussed in the class, a timer is a useful component in various protocol designs: because a communicating end cannot see what is going on either inside the network or at the other end, when needed it sets up an "alarm", and takes some action when the alarm goes off.

1. Does HTTP use any timers? If so, please briefly describe how each is used. If not, please explain why it does not need one.
2. Does DNS use any timers? If so, please briefly describe how each is used. If not, please explain why it does not need one.
3. Does TCP use any timer? If so, please briefly describe how each is used. If not, please explain why it does not need one.
4. Does UDP use any timer? If so, please briefly describe how each is used. If not, please explain why it does not need one.

1. No. HTTP requests/responses are sent through TCP, which handles the reliable delivery of these requests/responses using timers, but HTTP does not use them directly. However there are some HTTP headers that make use of timers, such as using the Cache-Control header to set a TTL for a particular request and avoid successive duplicate computations.
2. DNS does not directly use any timers since it runs atop UDP, but there are timers utilized in DNS Caching resolvers. These are necessary in order to ensure low-latency on duplicate DNS queries sent in succession. Once these caching TTL timers expire, subsequent queries for the same resource have to go through to the authoritative server once more.
3. TCP uses a Retransmission Timer (RTO) in order to detect whether or not a packet sent by the client to the server has been dropped or not. If the client doesn't receive an ACK for a particular packet before the RTO runs out, then the client will retransmit that packet. This RTO is crucial for implementing a reliable transport-level protocol. There are also 3 other timers used in TCP:

   Time-Wait timer - When closing a TCP connection, one side must send a FIN ACK to the other. The reception of this FIN ACK will cause the receiver to close its connection. The sender will then wait for the Time-wait timer to expire prior to closing connection.

   Persistent timer - When a TCP receiver's receive buffer is full, a message will be sent to the sender to stop sending data. However, the sender does not then know when the receiver is ready to receive packets again. By setting the persistent timer, the sender can know when to poll the receiver again.

   Keep-alive timer - If the connection sees no activity by the time this timer expires, then the connection will be closed.
4. UDP does not use any timers. Unlike TCP, UDP is not focused on reliability - it doesn't mind if a few packets are dropped in transmission. As a transport-level protocol, retransmission would be the primary use of a timer if any, so UDP does not need a timer.