**ECE 438/638: Computer Communications** **Spring 2013**
**Problem Set 9 solutions**

1. **TCP transition diagram**. Chapter 5, number 5.

The two-segment-lifetime timeout is needed to address two issues. First, the extra delay before closing the TCP connection and releasing the associated port number provides protection from any old segments for the existing incarnation of the TCP connection from being mistaken for a new incarnation of the same TCP connection (i.e., same use of 4-tuple of port and IP numbers for both the source and destination). Second, the sender of the last ACK is uncertain if the ACK is received, and by waiting it is likely to be able to generate a duplicate ACK if the other side retransmits the final FIN. For the first issue we only need one connection endpoint in TIMEWAIT and the TCP state diagram guarantees that at least one of the two sides will enter the TIMEWAIT state. For the second issue, a host in the LAST ACK state expects to receive the last ACK, rather than send it (the sender of the last ACK is the one that needs to wait to determine if a retransmission is necessary). Thus, the two-segment-lifetime timeout is not needed on the transition from LAST_ACK to CLOSED.

2. **Advertised window**. Chapter 5, number 6

For TCP, the sender periodically probes the receiver if the advertised window is zero. The issue is which side is responsible for retransmissions in the event of a lost packet. The receiver includes the advertised window in the ACKs to the sender. The sender probes the receiver to know when the advertised window becomes greater than 0; if either the probe or the receiver's ACK advertising a larger window is lost, then a later probe by the sender will elicit a duplicate of that ACK.

If responsibility for transmitting an ACK for a change in window-size is shifted from the sender to the receiver, then the receiver would need a timer for managing retransmission of this ACK until the receiver were able to verify that the ACK had been received.

A more serious problem is that the receiver only gets confirmation that the sender has received the ACK when new data arrives from the sender, so if the connection happens to fall idle the receiver may be wasting its time. The problem is how can the sender acknowledge the message from the receiver that the advertised window is now open. The acknowledgment is implicit if the sender has data to send. However, it is not clear how the sender can reply if the sender does not have any data to send.

3. **Sequence number wrap**. Chapter 5, number 8

The sequence number doesn't always begin at 0 for a transfer, but is randomly or clock generated.

4. **TCP transition diagram**.

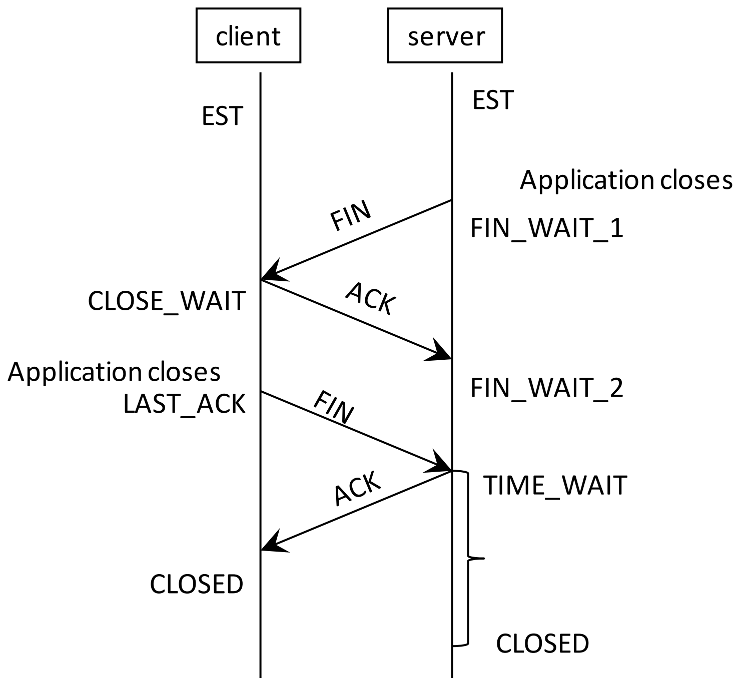
   (a) the sequence is:

   Server moves to FIN_WAIT_1
   Client moves to CLOSE_WAIT
   Server moves to FIN_WAIT_2

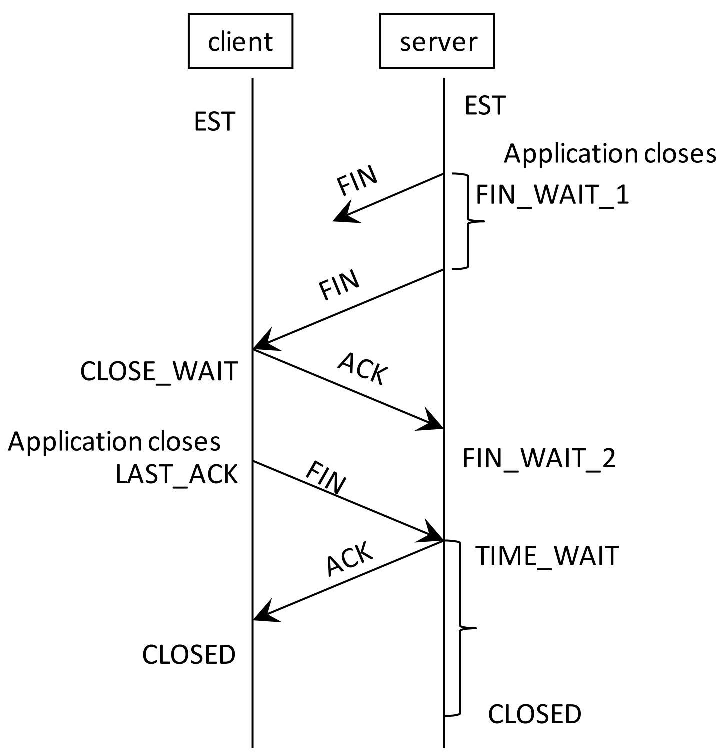
   (b) after the client process also executes the close statement:

   Client moves to LAST_ACK
   Server moves to TIME_WAIT
   Client moves to CLOSED
   Server moves to CLOSED

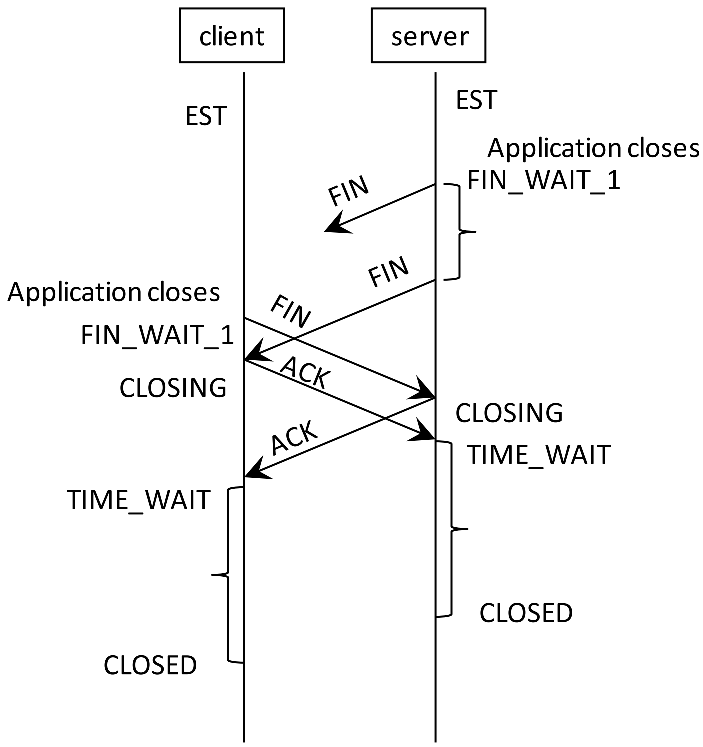
   A figure illustrating the sequence for the server and client is shown in the figure.



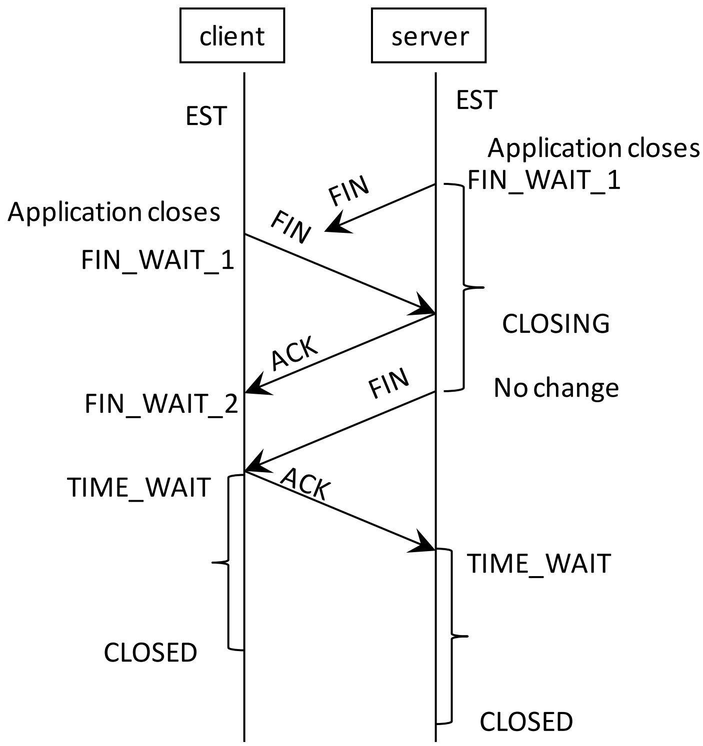

(c) First consider the case that the server times out while in the FIN_WAIT_1 state, retransmit the FIN, and the FIN is received before the client executes a close statement. In this situation the sequence of events will be the same as for part (a) and (b) and is shown below with the timeout.




But another possibility is that the client executes a close statement before it receives the retransmission of the FIN from the server. And, the server retransmits the FIN before it receives the client's FIN. The sequence of states is shown below. In this scenario, the server retransmits the FIN while still in the FIN_WAIT_1 state. Notice now both the client and server enter the TIME_WAIT state.



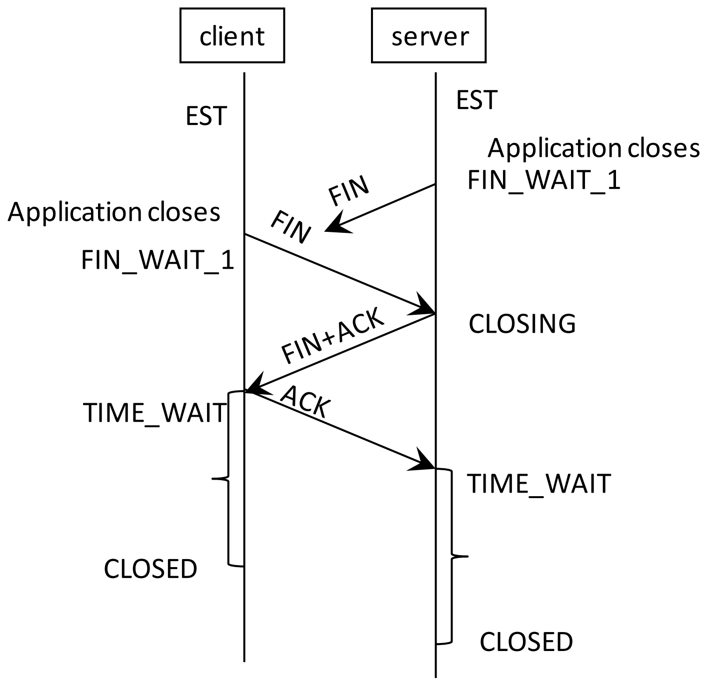

Yet another possibility is that the client executes a close statement before it receives the FIN from the server. And, the server receives the client's FIN <u>before</u> it retransmits its own FIN. The server retransmits the FIN while it is in the CLOSING state.




The sequence of events for the above figure (in which the client executes a close statement before it receives the FIN from the server):

Client moves to FIN_WAIT_1 (and sends FIN to server)
Server moves to CLOSING (it sends an ACK to client)
Client moves to FIN_WAIT_2 (waiting for the retransmission of the FIN from the server)
Eventually the server retransmits the FIN (from the CLOSING state).
Client moves to TIME_WAIT
Server moves to TIME_WAIT
The client and server both wait for two segment lifetimes before moving to the CLOSED state.

However, for the above scenario, a better implementation of TCP recognizes that when the server transmits the first ACK, it can also retransmit the FIN, since both an ACK and FIN can be included in the same TCP header. This gives the following sequence:




Notice the server follows the same sequence of states but the client can move from FIN_WAIT_1 directly to TIME_WAIT.

5. **TCP adaptive timeout**.

As pointed out in "comment 1" you need to continue to check for early timeouts until the timeout value is above 5 seconds. It turns out that segments 1, 2, and 4 are transmitted twice, and the others are transmitted once. The scenario goes as follows

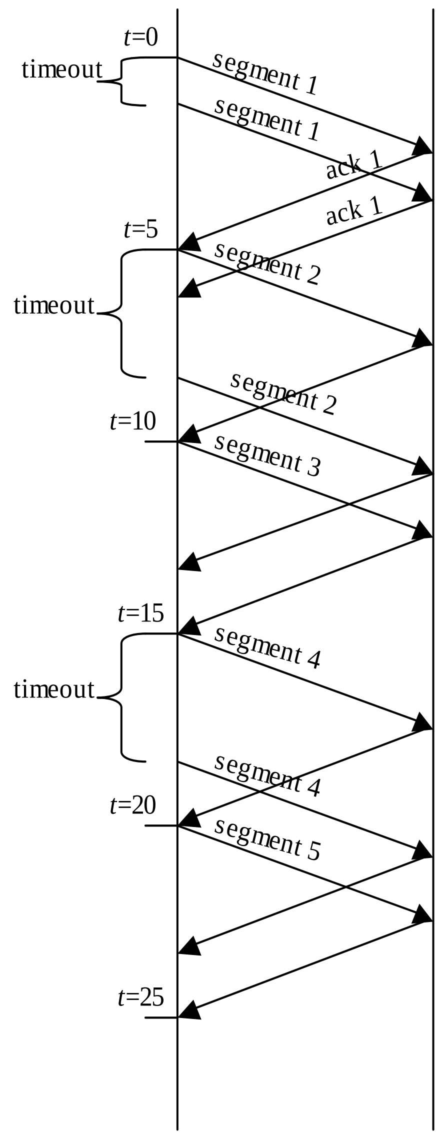
*t* = 0: segment 1 is transmitted with timeout 1.7

*t* = 1.7: segment 1 is retransmitted with timeout 3.4 (double timeout on retransmission). The timeout is scheduled to occur at time 5.1 (= 1.7 + 3.4) but when the ACK at time 5 is received, the timeout is canceled.

*t* = 5: ACK for segment 1 is received (the source does not know if this is the ACK from the transmission at time 0 or at time 1.7, so no sample of the RTT is taken). Segment 2 is transmitted with timeout 3.4. There is no change in the timeout because this is not a retransmission and there is no new estimate of the RTT.

*t* = 8.4: segment 2 is retransmitted with timeout 6.8 (double the timeout on retransmission)

*t* = 10: ACK for segment 2 is received (and pending timeout is canceled). Segment 3 is transmitted with timeout 6.8



(unchanged). For the same reason as given at time *t*=5, there is no sample of the RTT and the timeout is not updated.

*t* = 15: ACK for segment 3 is received. Notice that segment 3 was not retransmitted, so we have a sample RTT (the sample is equal to 5 sec). Updates yield Estimatted RTT 1.59 and deviation 0.619. Segment 4 is transmitted with timeout 4.06. (The timeout is still too short.)

*t* = 19.06: segment 4 is retransmitted with timeout 8.12

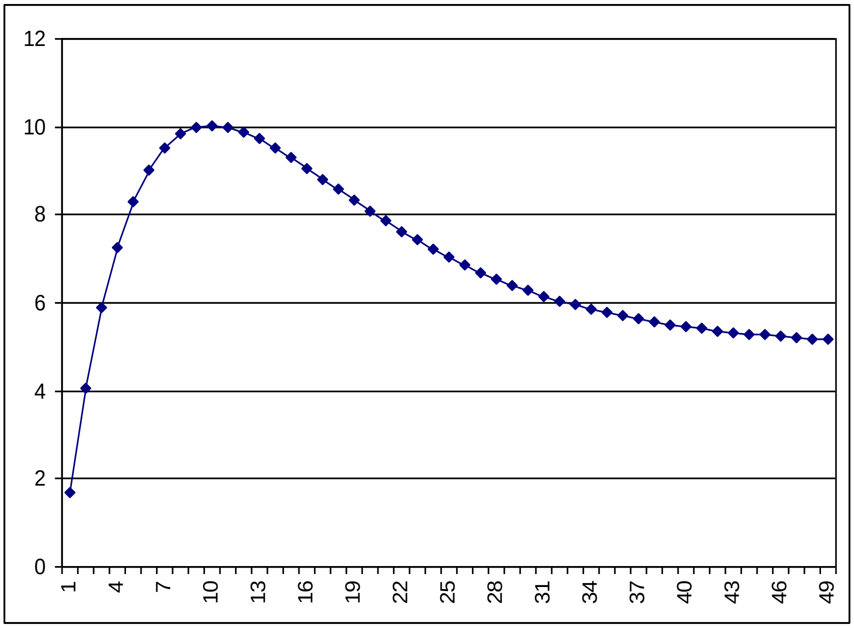
*t* = 20: ACK for segment 4 is received. Segment 5 is transmitted with timeout 8.12

*t* = 25: ACK segment 5 is received. Because this segment was not retransmitted we use the sampled RTT. Updates yield Estimatted RTT 2.01 and deviation 0.968. Segment 6 is transmitted with timeout 5.89. Finally, the timeout is above 5 seconds and unnecessary retransmissions are avoided.

*t* = 30: ACK segment 6 is received. Updates yield Estimatted RTT 2.39 and deviation 1.22. Segment 7 is transmitted with timeout 7.27

*t* = 35: ACK segment 7 is received. Updates yield Estimatted RTT 2.71 and deviation 1.39. Segment 8 is transmitted with timeout 8.29

For your interest, as additional samples of the RTT are available, the new values for the timeout are shown in the graph. We see that the timeout increases for the first 9 samples, but eventually converges to 5 seconds.




| update | sampleRTT | EstimatedRTT | Deviation | timeout |
|---|---|---|---|---|
| 0 | | 1.1 | 0.15 | 1.7 |
| 1 | 5 | 1.5875 | 0.61875 | 4.0625 |
| 2 | 5 | 2.0140625 | 0.96796875 | 5.8859375 |
| 3 | 5 | 2.387304688 | 1.220214844 | 7.268164063 |
| 4 | 5 | 2.713891602 | 1.394274902 | 8.290991211 |
| 5 | 5 | 2.999655151 | 1.505754089 | 9.022671509 |

| | | | | |
|---|---|---|---|---|
| 6 | 5 | 3.249698257 | 1.567577934 | 9.520009995 |
| 7 | 5 | 3.468485975 | 1.59041841 | 9.830159616 |
| 8 | 5 | 3.659925228 | 1.583055362 | 9.992146677 |
| 9 | 5 | 3.827434575 | 1.552682788 | 10.03816573 |
| 10 | 5 | 3.974005253 | 1.505168118 | 9.994677725 |
| 11 | 5 | 4.102254596 | 1.445271447 | 9.883340383 |
| 12 | 5 | 4.214472772 | 1.376830691 | 9.721795537 |
| 13 | 5 | 4.312663675 | 1.302917758 | 9.524334709 |
| 14 | 5 | 4.398580716 | 1.225970079 | 9.302461032 |
| 15 | 5 | 4.473758126 | 1.14790123 | 9.065363045 |

6. **Weighted fair queueing**

(a)

| Start time | Service counters | | | Packet served | Queued packets | Comments |
|---|---|---|---|---|---|---|
| | A | B | C | | | |
| 0 | 0 | 0 | 0 | A1 | ∅ | |
| 8 | 8 | 0 | 0 | -- | ∅ | No packets so service counters should be reset |
| 10 | 0 | 0 | 0 | A2 | B1 | Tie so serve in alphanumeric order |
| 18 | 8 | 0 | 0 | B1 | ∅ | Smin = 0 for flow B |
| 38 | 8 | 5 | 5 | B2 | ∅ | queues for flows A and C empty, Smin = 5 |
| 58 | 8 | 10 | 5 | C1 | A3, B3, C2, C3 | |
| 68 | 8 | 10 | 15 | A3 | A4, B3, C2, C3 | |
| 76 | 16 | 10 | 15 | B3 | A4, C2, C3 | |
| 96 | 16 | 15 | 15 | C2 | A4, C3, C4 | Smin = 15 |
| 106 | 16 | 16 | 25 | A4 | A5, C3, C4 | Smin = 16, flow B looses credit |
| 114 | 24 | 24 | 25 | A5 | C3, C4 | Smin = 24, flow B empty |
| 122 | 32 | 25 | 25 | C3 | C4 | Smin = 25, flow B looses credit |
| 132 | 35 | 35 | 35 | C4 | ∅ | Smin = 35 |
| 142 | 0 | 0 | 0 | ∅ | ∅ | |

(b). Comparing the queueing delay for the packets from flow B, only the packet B3 is served at a different time. With fair queueing (and no weights) B3 starts transmission at time 94, but with WFQ (and a larger weight for flow B), packet B3 starts transmission at time 76.

Notice that we could use a different rule to decide which packet to service. Instead of just picking the flow with the lowest value for *S*, we could use the current value of *S* plus the value that is added to *S* if its next packet is served. For example, at time 10 there are packets in the queue from flows A and B. With the basic rule we pick the packet from flow A because there is a tie in the service counters (they are both 0) and we just pick the flows in alphanumeric order. However, if we used the packet size (adjusted by the weights) we would pick B1 (with a value of 5) over A2 (with a value of 8).

In the long run, either approach for selecting packets will allocate the capacity fairly. But, the rule that considers the current amount of service plus the (weighted) size of the next packet will result in lower queueing delays.

For your information, here is the packet service order if we use the service count plus weighted packet size in selecting packets. Notice in this example the queueing delay for flow B's packets is even lower.

| Start time | Service counters | | | Packet served | Queued packets | Comments |
|---|---|---|---|---|---|---|
| | A | B | C | | | |
| 0 | 0 | 0 | 0 | A1 | ∅ | |
| 8 | 8 | 0 | 0 | -- | ∅ | |
| 10 | 0 | 0 | 0 | B1 | A2 | Pick B1 (5) over A2 (8) |
| 30 | 0 | 5 | 0 | A2 | B2 | A2 (8), B2 (10), Smin = 0 |
| 38 | 8 | 5 | 5 | B2 | ∅ | queues for flows A and C empty, Smin = 5 |
| 58 | 8 | 10 | 5 | B3 | A3, C1, C2, C3 | A3 (16), B3 (15), C2 (15). Tie so pick alphanumeric |
| 78 | 8 | 15 | 5 | C1 | A3, A4, C2, C3 | A3 (16), C1 (15), Smin = 5 |
| 88 | 8 | 15 | 15 | A3 | A4, C2, C3 | A3 (16), C2 (25), Smin = 8 |
| 96 | 16 | 15 | 15 | A4 | C2, C3, C4 | A4 (24), C2 (25) Smin = 15 |
| 104 | 24 | 15 | 15 | C2 | A5, C3, C4 | Smin = 15, A5(32), C2(25) |
| 114 | 24 | 24 | 25 | A5 | C3, C4 | Smin = 24 A5(32), C3(35) |
| 122 | 32 | 25 | 25 | C3 | C4 | |
| 132 | 35 | 35 | 35 | C4 | ∅ | |
| 142 | 0 | 0 | 0 | ∅ | ∅ | |

7. **Additive increase – multiplicative decrease**.
The following packets are lost: 9, 25, 30, 38, 50.

The window size is initially 1; when we get the first ACK it increases to 2. At the beginning of the second RTT we send packets 2 and 3. When we get their ACKs we increase the window size to 3 and send packets 4, 5 and 6. When these ACKs arrive the window size becomes 4. Now, at the beginning of the fourth RTT, we send packets 7, 8, 9, and 10; by hypothesis packet 9 is lost. So, at the end of the fourth RTT we have a timeout and the window size is reduced to 4/2 = 2.

We have the following pattern:

| RTT | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Packets sent | 1 | 2-3 | 4-6 | 7-10 | 9-10 | 11-13 | 14-17 | 18-22 | 23-28 |
| Window size | 1 | 2 | 3 | 4 | 2 | 3 | 4 | 5 | 6 |
| Packet lost | | | | 9 | | | | | 25 |
| Ack received | 1 | 3 | 6 | 8 | 10 | 13 | 17 | 22 | 24 |

| RTT | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|
| Packets sent | 25-27 | 29-32 (28 not resent) | 30-31 | 33-35 (32 not resent) | 36-39 | 38-39 | 40-42 | 43-46 | 47-51 | 50-51 |
| Window size | 3 | 4 | 2 | 3 | 4 | 2 | 3 | 4 | 5 | 2 (or 3 your choice) |
| Packet lost | | 30 | | | 38 | | | | 50 | |
| Ack received | 28 * | 29 | 32 | 35 | 37 | 39 | 42 | 46 | 49 | 51 |

Note in the 10-th RTT, the source sent packets 25-27. The destination already has packets 26, 27, and 28, and is just waiting for 25. When the destination gets packets 26 and 27 it discards them as duplicates. Since the destination has now received all packets through 28, it sends the cumulative ack 28 to the source.

Note in the 11-th RTT, the destination receives packets 31, and 32. It is missing packet 30, so it sends an ack with number 29. For your information, consider a different alternative. If all four packets (29-32) were lost the destination gets nothing in this round trip time and does not send any acknowledgement (the destination does not know if the source is done sending packets or if packets were lost, so the destination must wait for the source). In this situation, the source must time out and retransmit.

(A very important note: this problem has considered a simplified congestion control algorithm using AIMD only, and ignores time outs. Also, we consider one ACK per round trip time instead of an ack for each packet. TCP is more complicated because it uses AIMD, slow start, potentially a separate ack for each packet, fast retransmit and fast recovery, etc.)

8. **Adapting the Congestion Window**.
In this problem we consider a simple network and examine the throughput under three scenarios. In the first scenario, you are given the optimal congestion window size and timeout value. The other two scenarios consider adapting the congestion window using slow start and two different mechanisms to trigger retransmissions.

A simplified model for a network is considered. Suppose that between nodes A and B there is a router R. The link from A to R has an infinite bandwidth (i.e., packets are not delayed). However, the link from R to B introduces a transmission delay of 1 second per packet (e.g., if 2 packets arrive at R it takes 1 second to transmit the first packet, and the second packet waits for 1 second before being transmitted during the next second). To simplify the model, assume that acknowledgments from B through R back to A are sent instantaneously. Further assume that router R has queue size of one, in addition to the packet it is sending. At each second, the sender A first processes any arriving ACK's and then responds to any timeouts. Assume the application process at source A has an infinite backlog of packets to send to destination B, and that packets are numbered 1, 2, 3, …

Assume that the congestion window is fixed at one packet, and the timeout time set for each packet sent is 1 second. Note that at time 0, source A sends 1 packet because its congestion window is fixed at one, the packet is immediately transmitted by the router R (and the router's queue is empty). At time 1 the packet has completely arrived at the destination B, and an ACK is instantaneously returned to the source A. Source A cancels the timer for packet 1 and sends packet 2. Complete the table below. (Hint: this part is trivial, and the purpose is to just get you familiar with the table structure.)

| Time | A receives ACK for packet # | Size of A's congestion window | A sends packet # | R sends packet # | R queues packets |
|---|---|---|---|---|---|
| 0 | - | 1 | 1 | 1 | ∅ |
| 1 | 1 | 1 | 2 | 2 | ∅ |
| 2 | 2 | 1 | 3 | 3 | ∅ |
| 3 | 3 | 1 | 4 | 4 | ∅ |
| 4 | 4 | 1 | 5 | 5 | ∅ |
| 5 | 5 | 1 | 6 | 6 | ∅ |
| 6 | 6 | 1 | 7 | 7 | ∅ |
| 7 | 7 | 1 | 8 | 8 | ∅ |

(Comment on solution: this is just a very simple version of stop-and-wait. Note that this is the best window size for this problem because the link from R to B can only send 1 packet per second.)

As you found above, the network works well if the window size and timeout values are known. The problem is that TCP does not know the best window size or timeout value. For this part of the problem assume the network is the same and that the timeout value is fixed at 2 seconds (i.e., we will not adapt the timeout value). Now A sends data packets to B over a TCP connection, using slow start. The TCP buffer sizes at both A and B are arbitrarily large, so ignore the advertised window size. For the remainder of this problem we will assume that when TCP encounters a timeout it reverts to stop-and-wait as the outstanding lost packets in the existing window at the time of the timeout get retransmitted one at a time. The slow start phase does not begin again until all the packets in the existing window are acknowledged. Also, once one timeout and retransmission is pending, subsequent timeouts of later packets are suppressed until the earlier acknowledgment is received. That is, make a list of timeouts that occur and process them one at a time.

Fill in the table below. Note that to fill in the table you will need to keep track of some additional information. For example, you will need to keep track of when timers are set and if they expire, the list of packets that are in the window at both the source and destination, which packets are dropped, etc.

| Time | A recvs ACK # | Size of A's congestion window | A sends packet # | R sends packet # | R queues packets |
|---|---|---|---|---|---|
| 0 | - | 1 | 1 | 1 | ∅ |
| 1 | 1 | 2 [2, 3] | 2, 3 | 2 | 3 |
| 2 | 2 | 3 [3, 4, 5] | 4, 5 | 3 | 4 (5 lost) |
| 3 | 3 | 4 [4, 5, 6, 7] | 6, 7 | 4 | 6 (7 lost) |
| 4 | 4 (timeout packet 5) | 1 | 5 | 6 | 5 |
| 5 | 4 | 1 | Nothing waiting for ack 5 | 5 | ∅ |
| 6 | 6 | 1 | 7 | 7 | ∅ |
| 7 | 7 | 1 [8] | 8 | 8 | |
| 8 | 8 | 2 [9, 10] | 9, 10 | 9 | 10 |

Comment: The most important point is that in slow start, each time an ACK for a packet is received the congestion window is increased by 1 packet. This, of course, only applies when the packet is transmitted and not lost. If there is a retransmission then the ack for the retransmission does not count in increasing the congestion window size.

So, at time 2 when the ack is received for packet 2 the congestion window is immediately increased to 3. Packets 3, 4, and 5 are eligible to be transmitted but packet 3 is already outstanding. So, node *A* transmits packets 4 and 5. Note that packet 5 is lost because the router can only queue 1 packet while it is waiting to transmit packet 3.

At time 3 the source node does not know that there is any problem. It keeps increasing its window and transmitting more packets.

At time 4 the source first notices a problem. Packet 5 was transmitted by the source at time 2, so it expects that ack at time 4. Because there is a timeout at time 4 for packet 5, the source immediately stops using a sliding window and reverts to stop-and-wait. Furthermore, it must use stop-and-wait until all the outstanding packets are acknowledged (that is until packets 5, 6, and 7 are accounted for). So at time 4 the source retransmits packet 5. Note that the router is a layer-3 devise so it does not understand TCP headers. Thus, the router does not know that this is a retransmission or just some other packet. So, there is no way for the router to “flush” the queue or cancel packets in transmission. This would violate the layering approach because the router would have to understand TCP and the TCP headers to be able to try anything this complicated.

At time 5, there is a duplicate ack but note that the source cannot transmit another packet. The source is using stop-and-wait so it is waiting for the ack for packet 5 and it cannot transmit another packet until it gets the ack for 5. Thus, the source cannot transmit anything at time 5.

At time 6, the ack for packet 5 is received. Notice that the ack number is 6 because the receiver has already gotten packet 6 and stored it in its buffer. So, it sends one cumulative ack equal to 6 indicating it has received all packets up through 6. Packet 7 is an outstanding packet from before so we next must use stop-and-wait with packet 7.

At time 7, the ack for packet 7 is received and we can now quit stop-and-wait and return to slow start. Slow start always begins with a window size of 1. Note we do not increment the window size when the ack for packet 7 is received because packet 7 was retransmitted. Only increment the window size when a packet is sent with no retransmissions.

At time 8, the congestion window is increased to 2 (and the same problem starts again).

For the above part, you examined how the window size changes and the order packets are transmitted given a fixed timeout. For this part of the problem, we will look at how duplicate ACK’s trigger retransmissions instead of timeouts. So, changes from the previous part are:

- The router has buffer space for three packets instead of just one.
- Fast retransmit is done the first time that a second duplicate ACK is received (that is, the third ACK of the same packet). Fast retransmit was not considered in the previous part. Also, ignore fast recovery; when a packet is lost let the window size be one.
- The timeout interval is infinite (that is, ignore all timeout timers as they are not needed for this example).

| Time | A recvs ACK # | Size of A’s congestion window | A sends packet # | R sends packet # | R queues packets |
|---|---|---|---|---|---|
| 0 | - | 1 | 1 | 1 | ∅, ∅, ∅ |
| 1 | 1 | 2 [2-3] | 2, 3 | 2 | 3 |
| 2 | 2 | 3 [3-5] | 4, 5 | 3 | 4, 5 |
| 3 | 3 | 4 [4-7] | 6, 7 | 4 | 5, 6, 7 |
| 4 | 4 | 5 [5-9] | 8, 9 | 5 | 6, 7, 8 (9 lost) |
| 5 | 5 | 6 [6-11] | 10, 11 | 6 | 7, 8,10 (11 lost) |
| 6 | 6 | 7 [7-13] | 12, 13 | 7 | 8, 10, 12 (13 lost) |
| 7 | 7 | 8 [8-15] | 14, 15 | 8 | 10, 12, 14 (15 lost) |
| 8 | 8 | 9 [9-17] | 16, 17 | 10 | 12, 14, 16 (17 lost) |
| 9 | 8 | Send nothing. Window does not change and all packets are outstanding | | 12 | 14, 16 |
| 10 | 8 | Fast retransmit. Start stop-and-wait for packets 9 through 17 | 9 | 14 | 16, 9 |

| | | | | | |
|---|---|---|---|---|---|
| 11 | 8 | Still stop-and-wait | Nothing | 16 | 9 |
| 12 | 8 | Still waiting | Nothing | 9 | ∅ |
| 13 | 10 | Cumulative ack shows 9 and 10 received so 11 is next | 11 | 11 | ∅ |
| 14 | 12 | | 13 | 13 | ∅ |
| 15 | 14 | | 15 | 15 | ∅ |
| 16 | 16 | | 17 | 17 | ∅ |
| 17 | 17 | 1 [18] slow start | 18 | 18 | ∅ |
| 18 | 18 | 2 [19-20] | 19, 20 | 19 | 20 |

9. **TCP trace**. Slow start is active up to about 0.5 sec on startup. At that time a packet is sent that is lost; this loss results in a coarse-grained timeout at $t$ =1.9. At that point slow start is again invoked, but this time TCP changes to the linear-increase phase of congestion avoidance before the congestion window gets large enough to trigger losses. The exact transition time is difficult to see in the diagram; it occurs sometime around $t$=2.4. At $t$=5.3 another packet is sent that is lost. This time the loss is detected at $t$=5.5 by fast retransmit; this TCP feature is the one not present in Figure 6.11 of the text, as all lost packets there result in timeouts. Because the congestion window size then drops to 1, we can infer that fast recovery was not in effect; instead, slow start opens the congestion window to half its previous value and then linear increase takes over. The transition between these two phases is shown more sharply here, at $t$ = 5.7.
10. **RED**. Chapter 6, number 37.
    Only when the *average* queue length exceeds MaxThreshold are packets automatically dropped. If the average queue length is less than MaxThreshold, incoming packets may be queued even if the real queue length becomes larger than MaxThreshold. The router must be able to handle this possibility.