ECE 4380/6380: Computer Communications **Spring 2017**
Problem Set 10 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**.

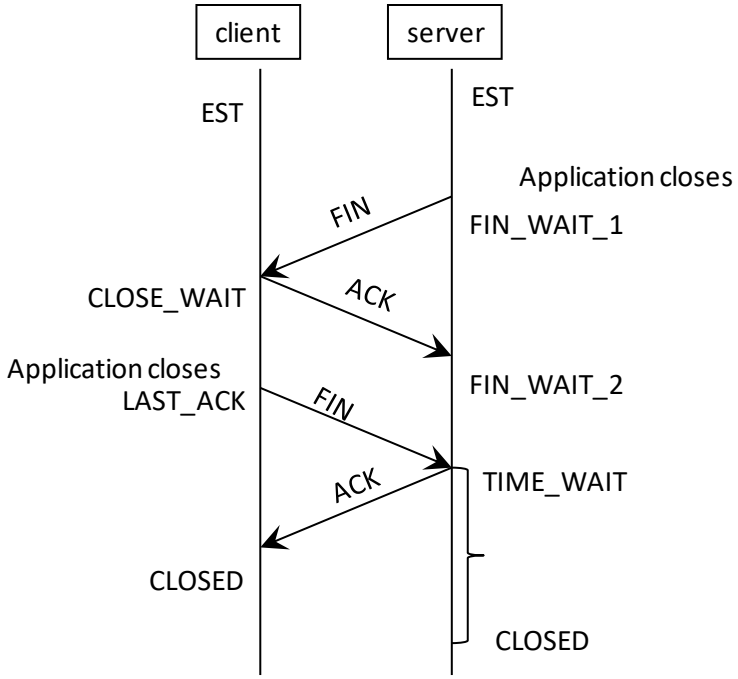
(a) the sequence is:

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

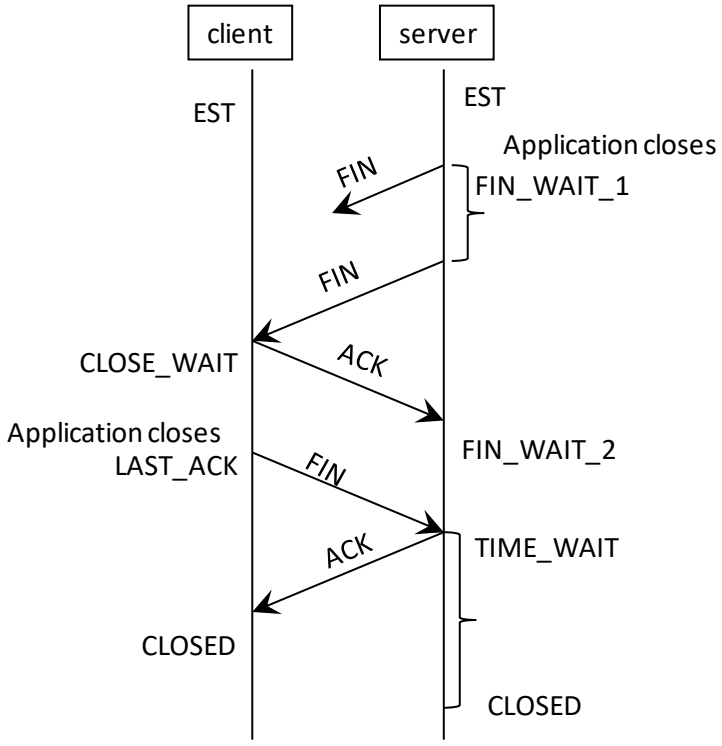
(b) after the client process also executes the close statement:

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.

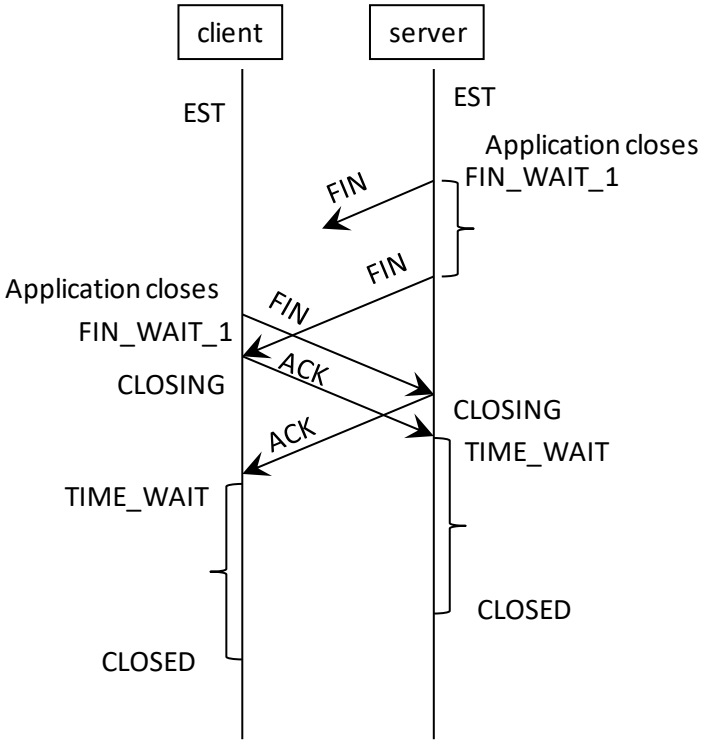


(c) First consider the case that the server times out while in the FIN_WAIT_1 state, retransmit the FIN, and the FIN is received before the client executes a close statement. In this situation the sequence of events will be the same as for part (a) and (b) and is shown below with the timeout.

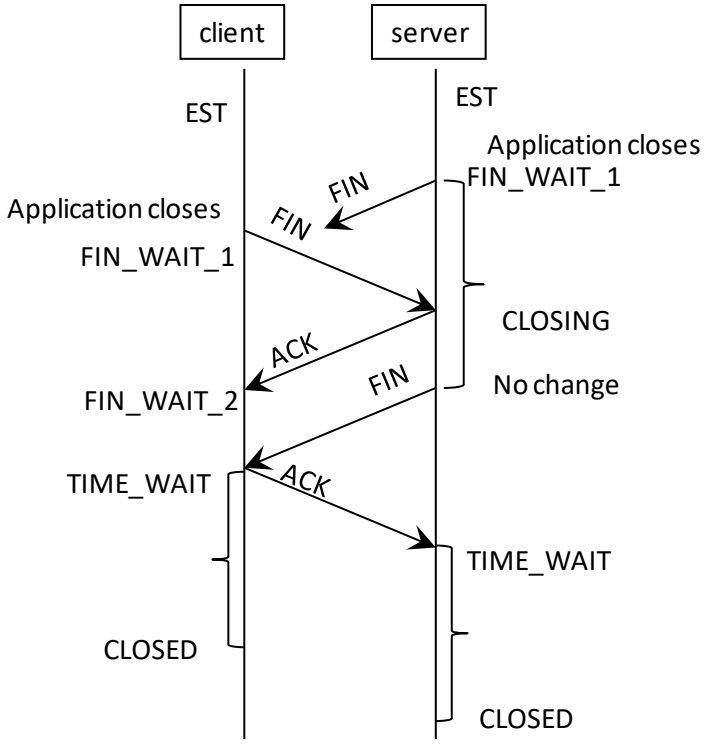


But another possibility is that the client executes a close statement before it receives the retransmission of the FIN from the server. And, the server retransmits the FIN before it receives the client's FIN. The sequence of states is shown below. In this scenario, the server retransmits the FIN

while still in the FIN_WAIT_1 state. Notice now both the client and server enter the TIME_WAIT state.



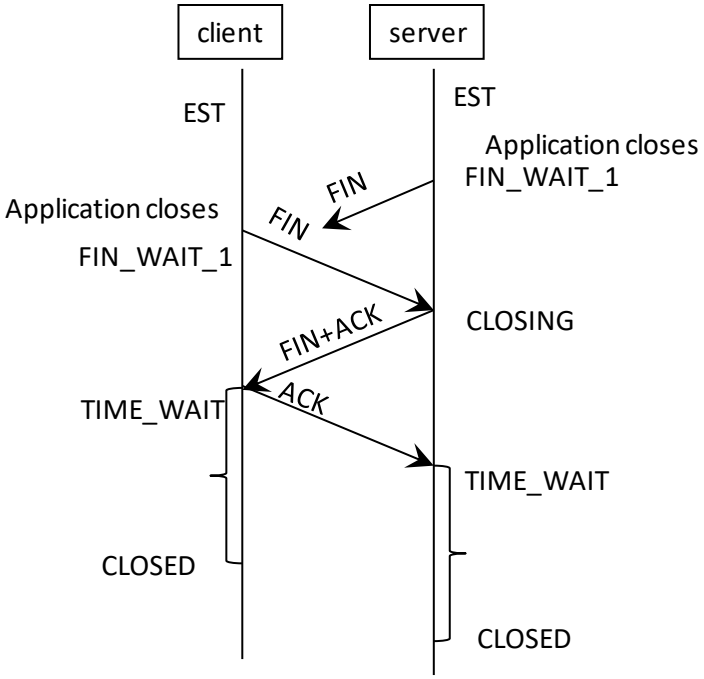
Yet another possibility is that the client executes a close statement before it receives the FIN from the server. And, the server receives the client's FIN before it retransmits its own FIN. The server retransmits the FIN while it is in the CLOSING state.



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

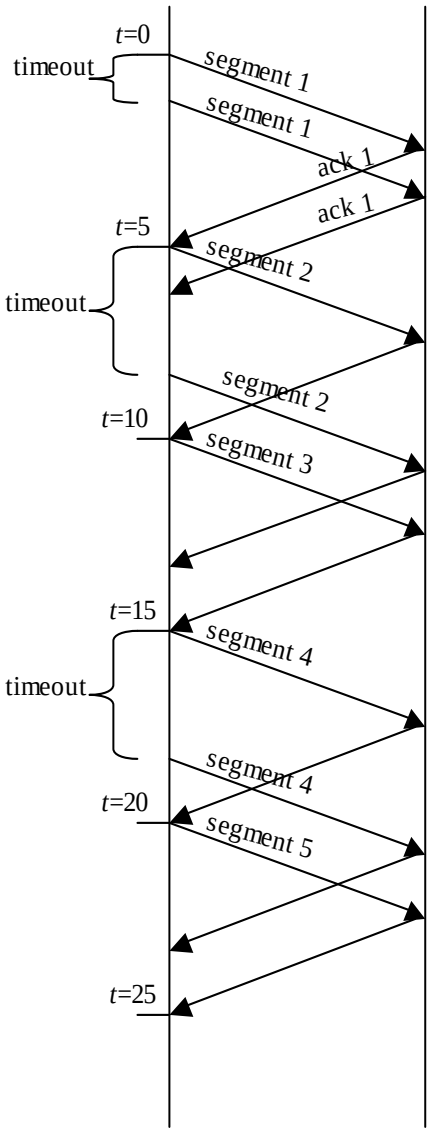
Client moves to FIN_WAIT_1 (and sends FIN to server)
Server moves to CLOSING (it sends an ACK to client)
Client moves to FIN_WAIT_2 (waiting for the retransmission of the FIN from the server)
Eventually the server retransmits the FIN (from the CLOSING state).
Client moves to TIME_WAIT
Server moves to 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

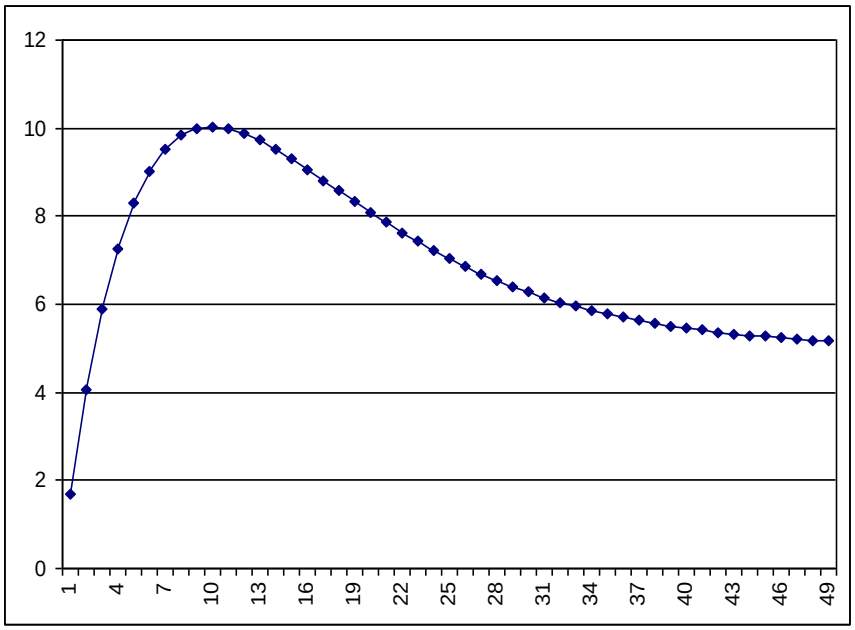
t = 20: ACK for segment 4 is received. Segment 5 is transmitted with timeout 8.12

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