6. 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 the receiver's ACK advertising a larger window is lost, then a later sender probe will elicit a duplicate of that ACK.

   If responsibility for the lost window-size-change ACK 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 it 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, so if the connection happens to fall idle the receiver may be wasting its time.

9. (a) The advertised window should be large enough to keep the pipe full; delay (RTT) $\times$ bandwidth here is 100 ms $\times$ 100 Mbps = 10 Mb = 1.25 MB of data. This requires 21 bits ($2^{21} = 2,097,152$) for the AdvertisedWindow field. The sequence number field must not wrap around in the maximum segment lifetime. In 60 seconds, 750 MB can be transmitted. 30 bits allows a sequence space of 1024 MB, and so will not wrap in 60 seconds. (If the maximum segment lifetime were not an issue, the sequence number field would still need to be large enough to support twice the maximum window size; see "Finite Sequence Numbers and Sliding Window" in Section 2.5.)

   (b) The bandwidth is straightforward from the hardware; the RTT is also a precise measurement but will be affected by any future change in the size of the network. The MSL is perhaps the least certain value, depending as it does on such things as the size and complexity of the network, and on how long it takes routing loops to be resolved.

16. (a) A would send an ACK to B for the new data. When this arrived at B, however, it would lie outside the range of "acceptable ACKs" and so B would respond with its own current ACK. B's ACK would be acceptable to A, and so the exchanges would stop.

    If B later sent less than 100 bytes of data, then this exchange would be repeated.

    (b) Each end would send an ACK for the new, forged data. However, when received both these ACKs would lie outside the range of "acceptable ACKs" at the other end, and so each of A and B would in turn generate their current ACK in response. These would again be the ACKs for the forged data, and these ACKs would again be out of range, and again the receivers would generate the current ACKs in response. These exchanges would continue indefinitely, until one of the ACKs was lost.

    If A later sent 200 bytes of data to B, B would discard the first 100 bytes as duplicate, and deliver to the application the second 100 bytes. It would acknowledge the entire 200 bytes. This would be a valid ACK for A.

    For more examples of this type of scenario, see Joncheray, L; A Simple Active Attack Against TCP; *Proceedings of the Fifth USENIX UNIX Security Symposium*, June, 1995.

20. (a)

T=0.0	‘a’ sent
T=1.0	‘b’ collected in buffer
T=2.0	‘c’ collected in buffer
T=3.0	‘d’ collected in buffer
T=4.0	‘e’ collected in buffer
T=4.1	ACK of ‘a’ arrives, “bcde” sent
T=5.0	‘f’ collected in buffer
T=6.0	‘g’ collected in buffer
T=7.0	‘h’ collected in buffer
T=8.0	‘i’ collected in buffer
T=8.2	ACK arrives; “fghi” sent

(b) The user would type ahead blindly at times. Characters would be echoed between 4 and 8 seconds late, and echoing would come in chunks of four or so. Such behavior is quite common over telnet connections, even those with much more modest RTTs, but the extent to which this is due to the Nagle algorithm is unclear.

(c) With the Nagle algorithm, the mouse would appear to skip from one spot to another. Without the Nagle algorithm the mouse cursor would move smoothly, but it would display some inertia: it would keep moving for one RTT after the physical mouse were stopped.

29. Here is the table of the updates to the EstRTT, etc statistics. Packet loss is ignored; the SampleRTTs given may be assumed to be from successive singly transmitted segments. Note that the first column, therefore, is simply a row number, *not* a packet number, as packets are sent without updating the statistics when the measurements are ambiguous. Note also that both algorithms calculate the same values for EstimatedRTT; only the TimeOut calculations vary.

					new TimeOut	old TimeOut
	SampleRTT	EstRTT	Dev	diff	EstRTT+4$\times$Dev	2$\times$EstRTT
		1.00	0.10		1.40	2.00
1	5.00	1.50	0.59	4.00	3.85	3.00
2	5.00	1.94	0.95	3.50	5.74	3.88
3	5.00	2.32	1.22	3.06	7.18	4.64
4	5.00	2.66	1.40	2.68	8.25	5.32

New algorithm (TimeOut = EstimatedRTT+ 4$\times$Deviation):

There are a total of three retransmissions, two for packet 1 and one for packet 3.

The first packet after the change times out at T=1.40, the value of TimeOut at that moment. It is retransmitted, with TimeOut backed off to 2.8. It times out again 4.2 sec after the first transmission, and TimeOut is backed off to 5.6.

At T=5.0 the first ACK arrives and the second packet is sent, using the backed-off TimeOut value of 5.6. This second packet does not time out, so this constitutes an unambiguous RTT measurement, and so timing statistics are updated to those of row 1 above.

When the third packet is sent, with TimeOut=3.85, it times out and is retransmitted. When its ACK arrives the fourth packet is sent, with the backed-off TimeOut value, 2$\times$3.85 = 7.70; the resulting RTT measurement is unambiguous so timing statistics are updated to row 2. When the fifth packet is sent, TimeOut=5.74 and no further timeouts occur.

If we continue the above table to row 9, we get the maximum value for TimeOut, of 10.1, at which point TimeOut decreases toward 5.0.

Original algorithm (TimeOut = 2×EstimatedRTT):

There are five retransmissions: for packets 1, 2, 4, 6, 8.

The first packet times out at T=2.0, and is retransmitted. The ACK arrives before the second timeout, which would have been at T=6.0.

When the second packet is sent, the backed-off TimeOut of 4.0 is used and we time out again. TimeOut is now backed off to 8.0. When the third packet is sent, it thus does not time out; statistics are updated to those of row 1.

The fourth packet is sent with TimeOut=3.0. We time out once, and then transmit the fifth packet without timeout. Statistics are then updated to row 2.

This pattern continues. The sixth packet is sent with TimeOut = 3.88; we again time out once, send the seventh packet without loss, and update to row 3. The eighth packet is sent with TimeOut=4.64; we time out, back off, send packet 9, and update to row 4. Finally the tenth packet does not time out, as TimeOut=2×2.66=5.32 is larger than 5.0.

TimeOut continues to increase monotonically towards 10.0, as EstimatedRTT converges on 5.0.

16. (a) In slow start, the size of the window doubles every RTT. At the end of the ith RTT, the window size is 2^i KB. It will take 10 RTTs before the send window has reached 2^{10} KB = 1 MB.

    (b) After 10 RTTs, 1023 KB = 1 MB − 1 KB has been transferred, and the window size is now 1 MB. Since we have not yet reached the maximum capacity of the network, slow start continues to double the window each RTT, so it takes 4 more RTTs to transfer the remaining 9MB (the amounts transferred during each of these last 4 RTTs are 1 MB, 2 MB, 4 MB, 1 MB; these are all well below the maximum capacity of the link in one RTT of 12.5 MB). Therefore, the file is transferred in 14 RTTs.

    (c) It takes 1.4 seconds (14 RTTs) to send the file. The effective throughput is (10MB / 1.4s) = 7.1MBps = 57.1Mbps. This is only 5.7% of the available link bandwidth.

19. The formula is accurate if each new ACK acknowledges one new MSS-sized segment. However, an ACK can acknowledge either small size packets (smaller than MSS) or cumulatively acknowledge many MSS's worth of data.

Let N = CongestionWindow/MSS, the window size measured in segments. The goal of the original formula was so that after N segments arrived the net increment would be MSS, making the increment for one MSS-sized segment MSS/N. If instead we receive an ACK acknowledging an arbitrary AmountACKed, we should thus expand the window by

$$\begin{aligned}\text{Increment} &= \text{AmountACKed}/N \\ &= (\text{AmountACKed} \times \text{MSS})/\text{CongestionWindow}\end{aligned}$$

31. (a) We lose 1100 ms: we wait 300 ms initially to detect the third duplicate ACK, and then one full 800 ms RTT as the sender waits for the ACK of the retransmitted segment. If the lost packet is sent at T=-800, the lost ACK would have arrived at T=0. The duplicates arrive at T=100, 200, and 300. We retransmit at T=300, and the ACK finally arrives at T=1100.

(b) We lose $1100 - 400 = 700$ ms. As shown in the diagram, the elapsed time before we resume is again 1100 ms but we have had four extra chances to transmit during that interval, for a savings of 400 ms.