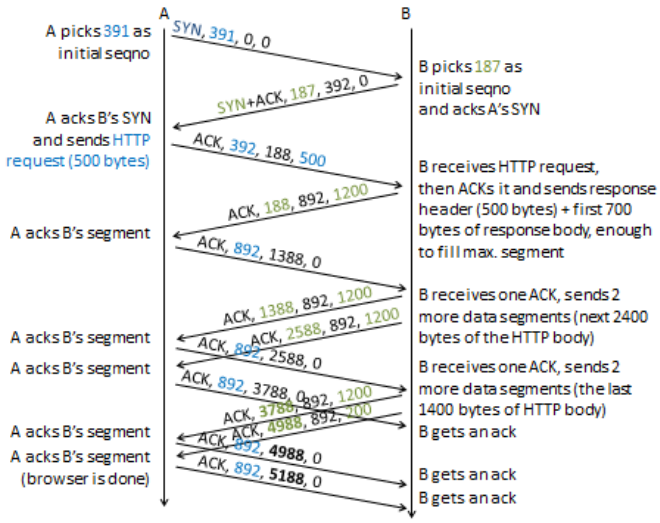
Question 1. Timeline is at right, with some extra explanation. Many other possible arrangements are plausible. This picture assumes many things:



- All packets arrive with no errors (so no packets are discarded due to checksum mismatches).
- A has the HTTP GET request ready to be sent early enough that it can be combined with the last part of the handshake, i.e. with the ACK for B's SYN. Alternatively, A could have sent the ACK, then shortly after that, send the 500 byte payload for the HTTP GET.
- B starts growing its send-window when it gets the first ACK from the data packet, not when it gets the ACK from its SYN packet. Alternatively, B could start growing its send-window on the first ACK, moving some of the packets a little earlier.
- B has the entire the HTTP response (headers + body) ready early enough, so some of the body can be combined into B's first data packet. Alternatively, B could first send 500byte payload packet, then separately send all the packets needed for the body.
- A is not using delayed acks. Alternatively, A would send fewer acks, e.g. every other one. When B gets a delayed ack, B would notice that it counts for 2 packets worth of data, so would send up to 4 new packets.

→ Common mistake: Almost every packet has an ACK. If you are sending a packet, there is no reason to leave out the ACK (it doesn't take up any extra space, the header field would just be empty otherwise), and plenty of reason to send the ACK (e.g. in case some previous ack got lost). The only exception I can think of is the very first SYN packet.

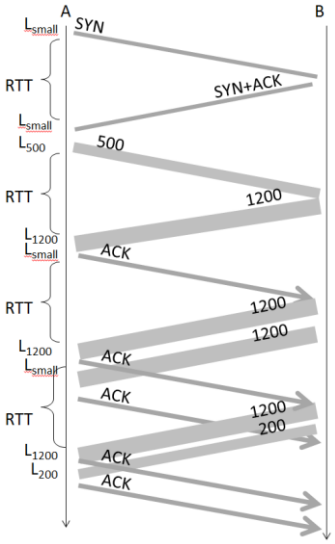
→ Common mistake: seqno counts the *amount of payload*, not the number of packets. Conceptually, each byte of payload data to be sent is assigned a sequence number, and the seqno for a packet is just the seqno for the first byte of the payload. So a packet with seqno=392 and 500 bytes of payload conceptually has the bytes labeled 392, 393, 394, 395, …, 890, and 891. And if there is no payload, the seqno does not increase. However, SYN packets are special: they count as 1 byte, even though they have no payload.

→ Common mistake: the ackno is the number we are next expecting, not the number we just received. That's what "cumulative ack" means. Some of the toy "rdt" protocols in the book didn't use cumulative acks.

Question 2. Let's assume queuing and congestion in the network isn't significant, so the round-trip propagation delay is always RTT_{min} = 25ms. Since the bandwidth is reasonably high, non-payload packets are small, and there isn't a lot of overlap in sending (TCP didn't have a chance to grow the window very large yet), a very rough estimate of the total time is then just four round trips, plus the transmission time for the HTTP request and response payloads. That is:

$T = 4*RTT_{min}$ + (500bytes+500bytes+3500bytes)/BW = 144ms

Here is a more precise calculation:



The first 2 packets and some of A's ACKs have no payload, so for these the transmission time is:

$L_{small} = N_{TCP} + N_{IP} + N_{NET}$ = 20+20+60 = 100bytes

$T_{small} = L_{small}$ /BW = 0.8 ms

The largest packets have 1200 bytes of payload, so have transmission time:

$L_{1200} = 1200 + N_{TCP} + N_{IP} + N_{NET}$ = 1300bytes

$T_{1200} = L_{1200}$ /BW = 10.4 ms

The first data packet from A and the last data packet from B have 500 bytes of payload, and 200 bytes of payload, so those have transmission times:

$T_{500} = L_{500}$ /BW = 600bytes/BW = 4.8ms

$T_{200} = L_{200}$ /BW = 300bytes/BW = 2.4ms

The total time is:

$T = 4*RTT + 4*L_{small} + L_{500} + 3*L_{1000} + L_{200}$= 141.6ms

An easier way to calculate is:

T = 4*RTT + (total data transmitted, not including overlapping packets)/BW

Question 3. Assuming queues and congestion are negligible, the when the loss rate is L, the average throughput will approach:

R = 1.22*MSS / (RTT * sqrt(L)) = 1.22 * 1200bytes / (25ms * sqrt(L)) = 5856/sqrt(L) bytes/sec

We want this to be close to the bandwidth of the bottleneck link, 1Mbps. Solving, we have:

R = 1Mbps = 125000 bytes/sec = 5856/sqrt(L) bytes/sec

sqrt(L) = 5856/125000

$L = (22400/125000)^2 = 0.219 = 21.9\%$ maximum allowable loss rate.

note: this would be an extraordinarily high loss rate. We can tolerate so many losses because the RTT is so small (so the window grows quickly) and the target throughput is so low (so even a small window will be enough to fill the bottleneck link).

Question 4.

(a) Note: headers do not count in sequence number calculations!

(1) (A → B): Seqno = 5000 Ackno = 8000 // payload 100 bytes

(2) (B → A): Seqno = 8000 Ackno = 5100 // payload 40 bytes

(3) (B → A): Seqno = 8040 Ackno = 5100 // payload 30 bytes

(4) (A → B): Seqno = 5100 Ackno = 8070 // payload 50 bytes

(b) No. A's data wasn't lost (it was B's data that was lost). And as soon as A receives packet (3), A will see the ackno 5100 and know that packet (1) must have been received by B.

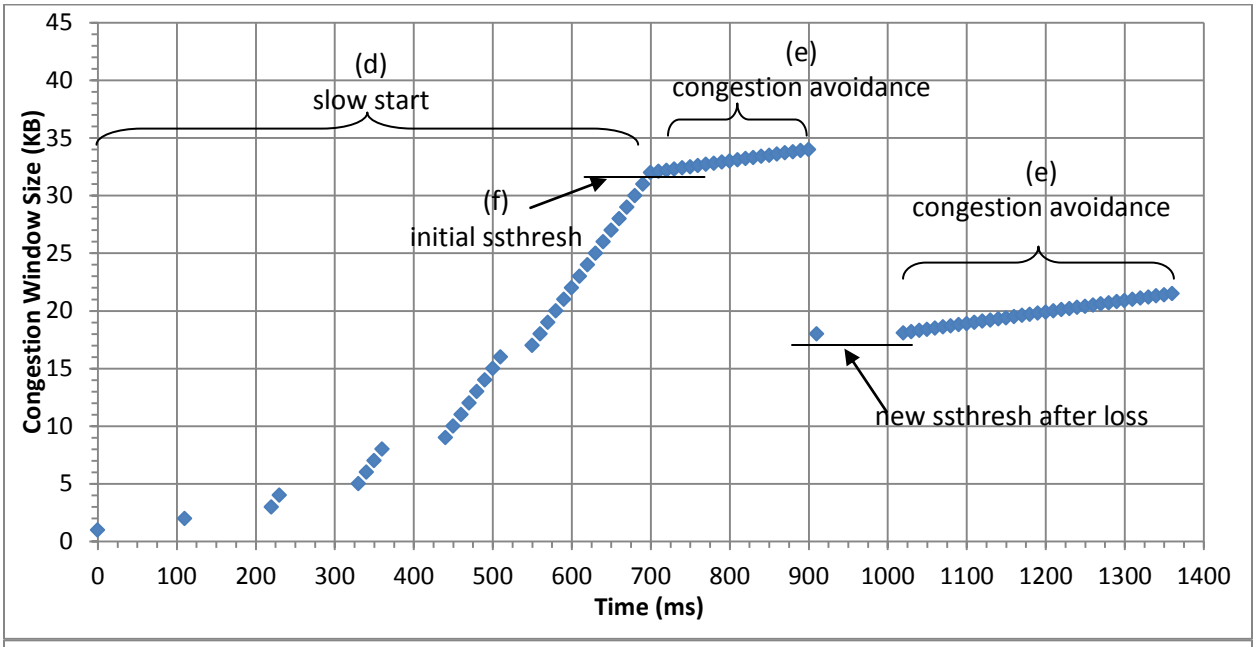
(c) Yes, A can send (4). From A's perspective, it sent 100 bytes in packet 1 (so the congestion window must have been at least 100), then it got an ACK, so it can slide the window right and increase the congestion window. There will be plenty of space in the send window to send 50 bytes.

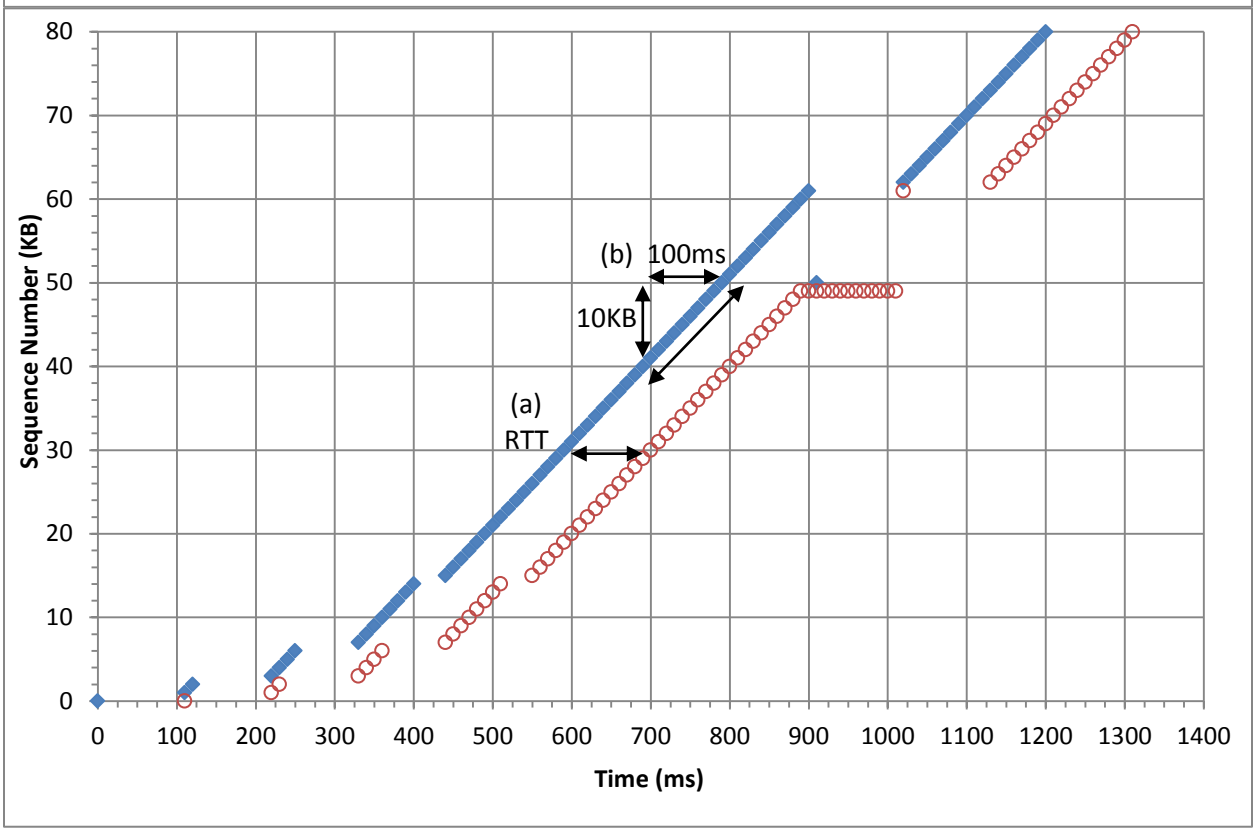
→ This assumes the send window isn't reduced or limited for other reasons (e.g. if the receive window was too small, or if there was a timeout, etc.).

(d) The new packet is (4') (A → B): Seqno = __5100_____ Ackno = _8000_____

A is using ackno 8000, because it's still waiting for the missing 8000 packet from B.

Question 5.


Congestion Window Size (KB)
Time (ms)
(d)
slow start
(e)
congestion avoidance
(f)
initial ssthresh
(e)
congestion avoidance
new ssthresh after loss


Sequence Number (KB)
Time (ms)
(b) 100ms
10KB
(a)
RTT

(a) Approximate RTT is about 100ms. Example: Around t=600ms, sequence number 30000 is sent. The ACK for that sequence number arrives around t=700ms.

(b) At their steepest rate, the sequence numbers being sent are going up by about 10KB every 100ms, so the sender's rate seems to be about 10KB/100ms = 100KB/sec.

If this question had asked about the bandwidth delay product (BDP), we might have instead look at the sender window in the first graph, which maxes out at about 34KB right before the loss near t=900ms. If we assume that that loss is due to congestion (i.e. TCP going over 100% of the BDP), then the round trip bandwidth delay would be about 34KB. However, it isn't obvious whether the lost packet really is due to congestion – it may have been a corrupt packet (discarded after the receiver examined the checksum) or just a random loss by some lower layer.

(c) The bottleneck is probably near the sender. We can tell because ACKs always arrive at nearly the exact same rate at which data packets are sent. If the bottleneck were near the receiver or in the middle of the network, when A sends a burst of packets in some short time (steep slope), this burst would be spread out later at the bottleneck link before arriving at B, so the ACKS would also be more spread out in time (lesser slope). But at no point (on the second graph) is the slope for ACKs less than the slope for the data packets.

(d) Slow start is from t=0ms up to about t=700ms, when it reaches sstresh. It might look linear long before then, but that’s just because the sender can’t send fast enough to make the acks come back any quicker.

(e) Congestion avoidance is from about t=700ms to t=900ms, then again from t=1200ms onwards.

(f) The initial ssthresh is about 32KB.

(g) Around t=900ms, the sender starts receiving duplicate ACKs. Because so many duplicate ACKs arrive, we can tell that the receiver is still getting packets – so just one single packet must have been lost or corrupted. When the third duplicate ACK arrives, the sender resends the (presumably lost) packet mentioned in the duplicate ACKs, it shrinks its send window by about half, then it starts to wait. (It doesn't send more packets yet, probably because it now has too many outstanding packets than would be allowed by the now-smaller send window.) After about 100ms (one RTT), we get an ACK indicating that the receiver has all of the packets we have sent, which again confirms that there must have been just one missing/corrupt packet.

(h) Tahoe uses **congestion avoidance** (ssthresh) and **fast retransmit** (detecting 3x-dup acks). Reno uses those too, and adds **fast recovery** (after a 3x-duplicate ack, set the congestion window to the new value of sstresh, that is, to half the old congestion window). **Vegas** adds a whole new retransmission mechanism (which we didn't discuss), in part based on the idea (which we did discuss) that a slowly growing RTT indicates growing congestion. The graph shows congestion avoidance, fast retransmit, and fast recovery. So it can't be Tahoe. It could be either **Reno** or possibly Vegas, or any of the many newer variations.