Lecture 12-13 Congestion Control Exercises ANS

Q1 TCP Congestion Control (15 points)

Assumptions: all TCP values are measured in packets, the sender always has new data to send, RWND is very large, and ACK numbers carry the **next expected packet sequence number**. (Note: explanations are optional, but they may earn you partial credit even if you gave an incorrect answer key.)

Q1.1 (1 point) Immediately after the TCP handshake, suppose you set CWND = 5 packets. In general, not necessarily for this specific flow, why might senders initialize CWND to 5 instead of 1?

ANS: Senders may initialize CWND to 5 instead of 1 to improve performance for short flows. If CWND starts at only 1, then a short flow may spend much of its lifetime waiting for Slow Start to increase the window. A larger initial congestion window allows the sender to transmit more packets immediately, which can reduce completion time for short transfers.

Q1.2 (2 points) Immediately after the TCP handshake, you are in Slow Start mode, with CWND = 5, SSTHRESH = ∞, and the sender's first data packet has sequence number 20. Suppose you receive an ACK with acknowledgment number 21. At this point, which packets are allowed to be in flight?

ANS: The first unacknowledged packet is now 21, because ACK 21 means the receiver has received everything up through packet 20 and is now expecting 21. In Slow Start, each new ACK increases CWND by 1 packet. So **CWND increases from 5 to 6**, and the sender's window allows packets 21 through 26, inclusive, to be in flight.

Q1.3 (2 points) Suppose you then receive an ACK with acknowledgment number 22. At this point, which packets are allowed to be in flight?

ANS: The first unacknowledged packet is now 22, because ACK 22 means the receiver has received everything up through packet 21 and is now expecting 22. This new ACK **increases CWND from 6 to 7** in Slow Start. So the sender's window allows packets 22 through 28, inclusive, to be in flight.

The rest of this question is independent of earlier subparts, and each subpart continues from the previous one.

Some time later, you are in Slow Start mode, with CWND = 14, SSTHRESH = ∞, all packets up to and including 30 sent and acknowledged, packets 31 through 44 sent but not acknowledged, and all packets 45 and later not yet sent. Suppose packet 31 is dropped in transit, while later packets are successfully delivered and cause duplicate ACKs for the missing packet.

Q1.4 (1 point) In this scenario, TCP switches from Slow Start mode to Fast Recovery mode immediately after receiving which ACK?

ANS: TCP switches to Fast Recovery immediately after receiving the **third duplicate ACK with acknowledgment number 31**. Those three duplicate ACKs are triggered by receipt of out-of-order

packets 32, 33, and 34. So the transition occurs immediately after the duplicate ACK 31 caused by arrival of packet 34.

Q1.5 (2 points) What is the value of CWND the instant after TCP switches from Slow Start mode to Fast Recovery mode?

ANS: When Fast Recovery begins, TCP halves CWND from 14 to 7, and set SSTHRESH = 7. It then temporarily inflates the window by 3 packets to account for the three duplicate ACKs already received. Therefore, CWND immediately after entering Fast Recovery is **10**.

Q1.6 (2 points) The receiver receives packets in this order: 32, 33, 34, …, 43, 44, 31 retransmitted. The sender receives the resulting ACKs in the same order, with no timeouts. What is the value of CWND the instant before TCP switches out of Fast Recovery mode?

ANS: At entry to Fast Recovery, CWND = 10.
The ACKs caused by packets 32, 33, and 34 are the three duplicate ACK 31s that already triggered entry into Fast Recovery and were already accounted for.
Each later duplicate ACK 31 caused by packets 35 through 44 increases CWND by 1 packet while in Fast Recovery.
There are 10 such later duplicate ACKs, so CWND grows from 10 to **20** immediately before the new ACK arrives for the retransmitted packet 31.

Q1.7 (2 points) What is the value of CWND the instant after TCP switches out of Fast Recovery mode?

ANS: When TCP exits Fast Recovery on the new non-duplicate ACK, it sets CWND back to SSTHRESH = 7. That value is the halved post-loss window inQ1.5, which is **7**.

Q1.8 (2 points) At the instant before TCP switches out of Fast Recovery mode, which packets have been sent out? State the largest packet sequence number that has been sent.

ANS: Just before leaving Fast Recovery, the left edge of the window is still packet 31, because the sender has not yet received a new ACK advancing past the loss.
At that moment, CWND = 20, so the sender's window spans packets 31 through 50, inclusive.
Therefore, the largest packet sequence number that has been sent is **50**.

Q1.9 (1 point) After TCP switches out of Fast Recovery mode, what congestion-control mode is TCP in?

ANS: After Fast Recovery ends, TCP enters **Congestion Avoidance**.
It does not return to Slow Start, because this loss was detected by duplicate ACKs rather than by a timeout.

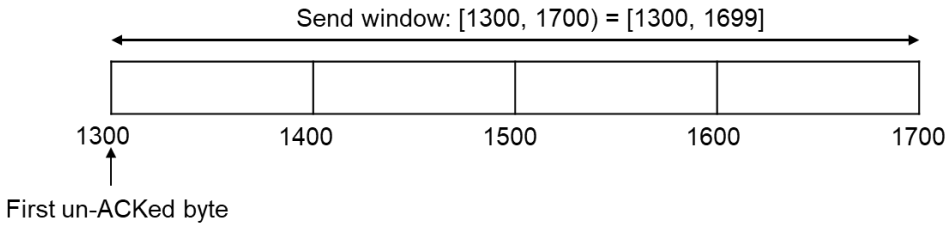
Q2 TCP (10 points) **(Note: This question uses Byte numbers instead of packet sequence numbers.)** Assume a TCP sender has an **MSS of 100 bytes** and a **window size of 400 bytes. There is no congestion control, so the window size is constant.** The sender has just received ACK 1000, meaning the receiver has received all bytes up to byte 999 and is now expecting byte 1000. The timeout interval is constant and is denoted by RTO. Unless otherwise stated, all transmitted segments experience an RTT. Assume segments are never fragmented or reordered, and that losses occur only when explicitly stated. **Each segment carries MSS = 100 bytes** and is identified by the sequence number equal to the number of its first byte. Thus, the sender may transmit segments starting at bytes 1000, 1100, 1200, and so on. Assume TCP keeps a single timer for the segment containing the first unacknowledged byte, and when the first unacknowledged byte advances, the timer becomes associated with the new left edge of the window. All subparts are independent.

Q2.1 (1 point) Immediately after the sender processes **ACK 1000**, which byte is the first unacknowledged byte?

ANS: The first unacknowledged byte is **1000**.

Q2.2 (2 points) At the moment the sender transmits the segment whose first byte is **1600**, what is the highest ACK number the sender must already have received?

ANS: 1300. With a 400-byte window, transmitting the segment starting at byte 1600 means the send window is [1300,1700), so the first unacknowledged byte must be **1300**, which means the sender must already have received **ACK 1300**.

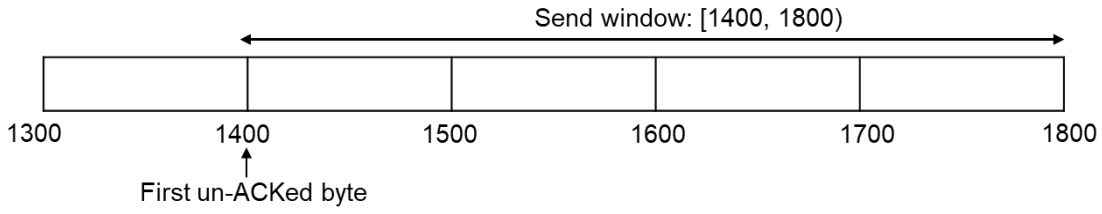


Q2.3 (4 points) Assume that the only lost segment is the one whose first byte is **1400**.

(a) How does the sender detect the loss?

- Timeout
- 3 duplicate ACKs
- The sender does not detect the loss

ANS: 3 duplicate ACKs. Since later in-window segments can still arrive, the receiver keeps sending duplicate cumulative ACKs for byte **1400**, and after three duplicates the sender retransmits the missing segment.



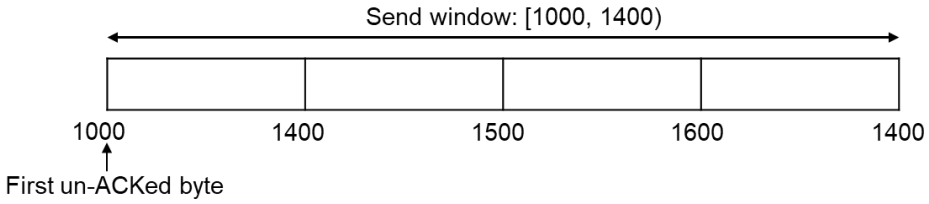
(b) After the retransmitted segment starting at byte **1400** is received successfully, what ACK number does the receiver send?

ANS: 1800. The receiver can now cumulatively acknowledge all bytes through **1799**, so it sends **ACK 1800**, meaning byte **1800** is the next expected byte.

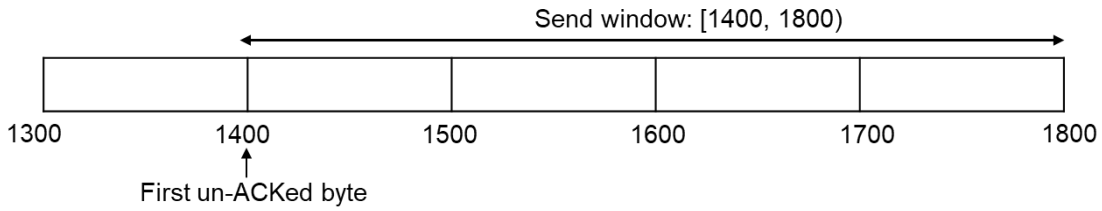
Q8.4 (3 points) Assume every segment whose first byte is **1400 or larger** is dropped. When does the timer for byte 1400 start, and when does it expire?

ANS: The timer for byte **1400** starts when **ACK 1400** is received, because at that moment byte **1400** becomes the first unacknowledged byte and the timer shifts to the new left edge of the window. Therefore, the timer for byte 1400 starts at $t_{1300} + RTT$, which is equivalently t_{1700}, the time the sender transmits the segment starting at byte 1700; and it expires at $t_{1300} + RTT + RTO$, equivalently $t_{1700} + RTO$.

Detailed explanations:
The timer for byte 1400 starts when **byte 1400 becomes the first unacknowledged byte**, because TCP keeps a **single timer for the segment at the left edge of the window**.
Initially, after ACK 1000, the sender window is [1000,1400), so the left edge is 1000.



When the segment starting at byte 1300 is sent at time t_{1300} and experiences RTT R, its cumulative acknowledgment returns at $t_{1300} + R$.
That acknowledgment is **ACK 1400**, since the receiver has then received all bytes through 1399 and is now expecting byte 1400 next. At that instant, byte 1400 becomes the first unacknowledged byte, so the timer for byte 1400 starts at $t_{1300} + R$.
At the same instant, the send window slides to [1400,1800). Within the new window, the valid 100-byte segment starts are 1400, 1500, 1600, 1700. So 1700 is the new rightmost segment that was not allowed before, but becomes allowed when the window slides right. So the sender can immediately send the segment whose first byte is **1700**. Hence, $t_{1300} + R$ is equivalently t_{1700}.



Since the timeout interval is **RTO**, the timer expires **RTO** time units later, at $t_{1300} + R + RTO$, equivalently $t_{1700} + RTO$. (Think of the 400-byte window as holding exactly four 100-byte slots. When the leftmost slot, 1300–1399, gets acknowledged, it drops out, and a new slot opens on the right for 1700–1799.)