        - Cumulative ACK: ACKs all packets up to, including seq # n. On receiving ACK(n): move window forward to begin at n+1 timer for oldest in-flight packet. Timeout(n): retransmit packet n and all higher seq # packets in window.
    - **Go-Back-N: receiver:** ACK-only: always send ACK for correctly received packets so far, with highest in-order seq#
        - May generate duplicate ACKs. Need only remember rcv_base.
        - On receipt of out-of-order packet:
            - Can discard (don't buffer) or buffer: an implementation decision
            - Re-ACK pkt with highest in-order seq #
    - **Selective repeat: the approach:** pipelining. Receiver individually ACKs all correctly received packets and buffers packets, as needed, for in-order delivery to upper layer.
    - Sender: maintains (conceptually) a timer for each unACKed pkt. Timeout: retransmits single unACKed packet assocated with timeout and maintains (conceptually) "window" over N consecutive seq#s. Limits pipelined, "in flight" packets to be within this window.
    - Sender:
        - data from above:
            - if next available seq# in window, send packet.
        - Timeout(n): resend packet n, restart timer.
        - ACK(n): in [sendbase, sendbase+N-1]:
            - mark packet n as received. If n smallest unACKed packet, advance window base to next unACKed seq#.
    - Receiver:
        - packet n in [rcvbase, rcvbase+N-1]
            - send ACK(n).
            - Out-of-order: buffer.
            - In-order: deliver (also deliver buffered, in-order packets),
            - advance window to next not-yet-received packet.
        - Packet n in [rcvbase-N, rcvbase-1]
            - ACK(n)
        - Otherwise:
            - Ignore

TCP Segment Structure:

- **32 Bit Rows**
- **Source Port# 16 bits, dest port # 16 bits**
- **Sequence Number 32 bits:** counting bytes of data into bytestream (not segments!)
- **Acknowledgement number 32 bits:** seq # of next expected byte; A bit: this is an ACK
- **Length (of TCP header), Not used, C, E:** congestion notification, **TCP options, RST, SYN, FIN: Connection management (all together 16 bits)**
- **Receive window 16 bits:** flow control: # bytes receiver willing to accept.
- **Checksum 16 bits, URG data pointer 16 bits**
- **Options (variable length)**
- **Application data (variable length)**

TCP Sequence numbers, ACKs:

- **Sequence numbers:** byte stream "number" of first byte in segment's data.
- **Acknowledgements:** seq # of next byte expected from other side. Cumulative ACK

TCP Sequence cont.

Step	Direction	Flags	Seq	Ack	Data	Description
1	A → B	SYN	1000	—	—	Client opens connection
2	B → A	SYN-ACK	5000	1001	—	Server acknowledges and opens
3	A → B	ACK	1001	5001	—	Connection established
4	A → B	PSH-ACK	1001	5001	"HELLO "	Send 6 bytes
5	B → A	ACK	5001	1007	—	ACK next expected byte 1007
6	A → B	PSH-ACK	1007	5001	"WORLD"	Send 5 bytes
7	B → A	ACK	5001	1012	—	ACK next expected byte 1012

TCP: Retransmission Scenarios:

- Lost ACK scenario: Sender resends unACKed segment.
- Premature timeout: Receiver resends last ACK (cumulative ACK) when sender resends unACKed segments.
    - Cumulative ACK covers for earlier lost ACK
- TCP Fast Transmit: if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq#. Likely that unACKed segment lost, so don't wait for timeout.

TCP Flow Control: used when network layer delivers data faster than application layer removes data from socket buffers.

- Receiver controls sender, so sender wont overflow receiver's buffer by transmitting too much, too fast.
- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header.
    - **RcvBuffer:** size set via socket options (typical default is 4096 bytes). Many OS's auto-adjust RcvBuffer
    - Sender limits amount of unACKed ("in-flight") data to received **rwnd** guarantees receive buffer will not overflow.
    - Receiver advertises rwnd = how much free buffer it has.
    - The sender limits its unACKed data to fit within that window.
    - As the receiver's app reads data, buffer space frees up -> rwnd increases -> sender sends more.
    - This dynamic dance = flow control.
    - It ensures sender speed matches receiver capacity.

TCP 3-way handshake:

Step	Sender	Flags	Seq	Ack	Description
1	Client → Server	SYN	1000	—	Client initiates connection
2	Server → Client	SYN, ACK	5000	1001	Server acknowledges and starts its own seq
3	Client → Server	ACK	1001	5001	Client acknowledges, connection established

- It establishes initial sequence numbers for reliability. It ensures both sides are ready to send and receive. After it completes, the connection entered the **ESTABLISHED** state and data can flow.
- **Why not just two:** because both sides need to confirm two things:
    - That they can send data (their own SYN)
    - That they can receive data (ACK the other's SYN)
    - A 2-step handshake would leave ambiguity – one side could think a connection is open while the other doesn't. The 3-step process guarantees mutual agreement.
    - Duplicate data possible

Congestion Control: "too many sources sending too much data too fast for network to handle.

- Cost = congestion effect (delay/loss), Lambda = sending rate, cost = f(lambda)
- Long delays (queueing in router buffers)
- Packet loss (buffer overflow at routers)
- Scenario 1: one router, infinite buffers. Input, output link capacity: R. Two flows, no retransmission needed.
    - But even if each host tries to send faster, the shared link can't push more than R total – throughput saturates. Main cost is delay.

- Scenario 2: fine buffers -> packet drops, dropped packets -> retransmissions. Retransmissions -> wasted bandwidth and lower throughput. This is the 2nd cost of congestion (after delay).
  - Finite buffer capacity, but sender has perfect knowledge of which packets are dropped. So only retransmits packets that are known to be lost.
  - Wasted work – retransmissions that consume capacity but don't increase throughput
- Scenario 3: Fine buffers, with retransmissions, and duplicates, but there are multiple routers and multiple simultaneous flows.
  - Key effect: Global unfairness & collapse. Some flows starve others, and total throughput of the network drops toward zero.
- Causes/cost of Congestion: insights
  - Throughput can never exceed capacity
  - Delay increases as capacity approached
  - Loss/retransmission decreases effective throughput
  - Un-needed duplicates further decrease effective throughput
  - Upstream transmission capacity/buffering wasted for packets lost downstream.
- Approaches:
  - End-end congestion control: no feedback from network. Congestion is inferred from observed loss, delay. Handled by TCP.
  - Network-assisted congestion control: routers provide direct feedback to sending/receiving hosts with flows passing through congested router. May indicate congestion level or explicitly set sending rate. TCP ECN, ATM, DECbit protocols.
- **TCP Congestion control: AIMD:** senders can increase sending rate until packet loss (congestion) occurs. Then decrease sending rate on loss event.
  - **Additive increase:** increase sending rate by 1 max segment size every RTT until loss detected
  - **Multiplicative decrease:** cut sending rate in half at each loss event.
  - **AIMD:** a distributed, asynchronous algorithm – has been show to: optimize congested flow rates network wide! Have desirable stability properties.
  - **congestion window (CWND):** maintained by TCP sender. It tells sender how much data its allowed to have sent but not yet ACKed by the receiver.
    - **TCP Rate = cwnd/RTT**
    - Dynamically adjusted in response to observed network congestion (implementing TCP congestion control) LastByteSent – LastByteAcked <= cwnd.
    - TCP Slow start: when connection begins increase rate exponentially until first loss event: initially cwnd = 1mss, double cwnd every RTT
    - **Congestion avoidance:** when cwnd gets to ½ of its value before timeout switches from exponential increase to linear.
- **TPC Cubic:**
  - Wmax: sending rate at which congestion loss was detected
  - Congestion state of bottleneck link probably (?) hasn't change much
  - After cutting rate/window in half on loss, initially ramp to Wmax faster, but then approach Wmax more slowly.
- **Quic: Quick UDP Internet Connections:** application layer protocol on top of UDP. Increases performance of HTTP. Deployed on many Google Servers, apps (chrome, mobile YouTube app). **One handshake.**
  - Error and congestion control like TCP's.
  - Connection establishment: reliability, congestion control, authentication, encryption, state established in one RTT. Multiple app level streams multiplex over single QUIC connection.