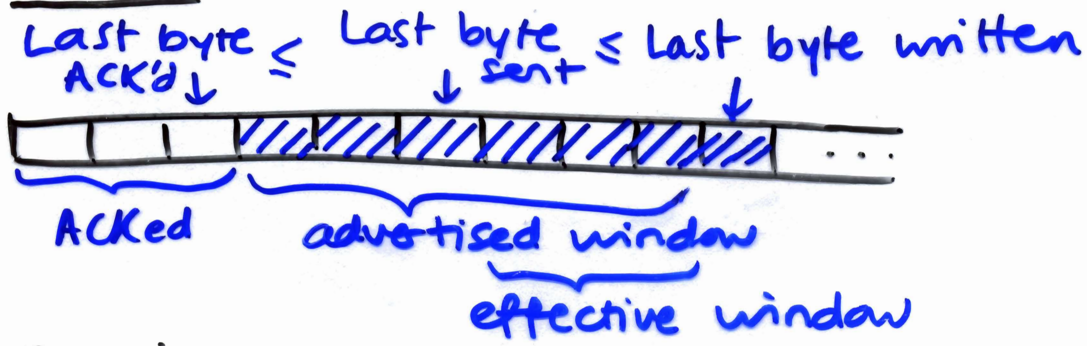
- TCP is a <u>byte-oriented protocol</u>
  - sender writes bytes into a TCP connection
  - receiver reads bytes out of a TCP connection
  - description of service offered to application processes
- TCP on source host buffers enough bytes to fill a reasonably sized packet for transmission
  - TCP packets are called <u>segments</u>
- TCP uses <u>sequence numbers</u> to identify the order of bytes sent by each host
  - ensures in order delivery in case of fragmentation or disordering during transmiss²
- Starting sequence numbers are randomly chosen & exchanged during connection establishment
  - <u>3-way handshake</u> algorithm:
    - client (initiating host) sends a SYN (synchronization) packet (SYN flag set) with sequence no. field SequenceNum = x
    - server (other host) sends a SYN+ACK packet (SYN+ ACK flags set) with Ack = x+1 (next expected seq no.) & SequenceNum = y (return session seq. no.)
    - client sends an ACK packet with Ack = y+1

- random choice of starting seq no. protects against 2 incarnations of the same connection reusing the same sequence no.s too soon
- 32-bit sequence no. space protects against wraparound within the assumed MSL (120 s)
  - wraparound time $= \dfrac{2^{32} \times 8 \text{ bits}}{\text{transmission rate in bps}}$

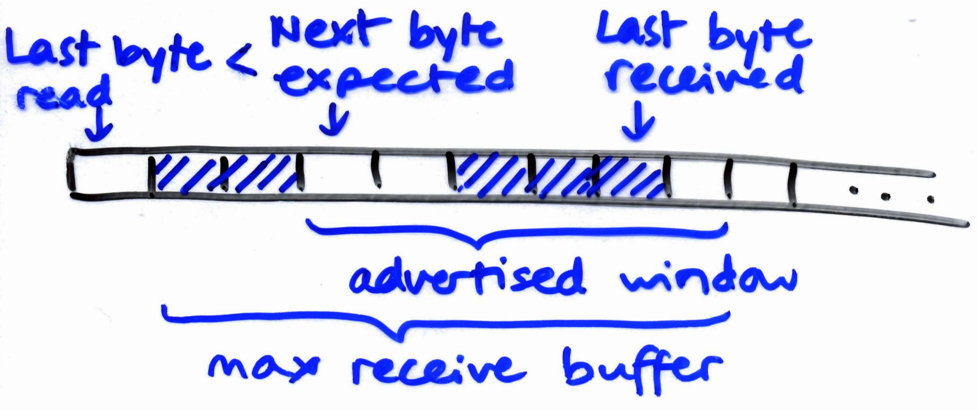
    = 6.4 hours for 1.5 Mbps
      6 min for 100 Mbps

- <u>cumulative ACKs</u>: each ACK specifies the next byte needed – all earlier bytes have been successfully received & can be discarded from the send buffer
  - allows TCP to operate with an unreliable acknowledgment channel
- application on receiving side only reads bytes <u>in order</u>
  - receive buffer holds out of order data, as well as in-order data (i.e. no missing bytes earlier in the stream) that the application has not yet read (if application falls behind)

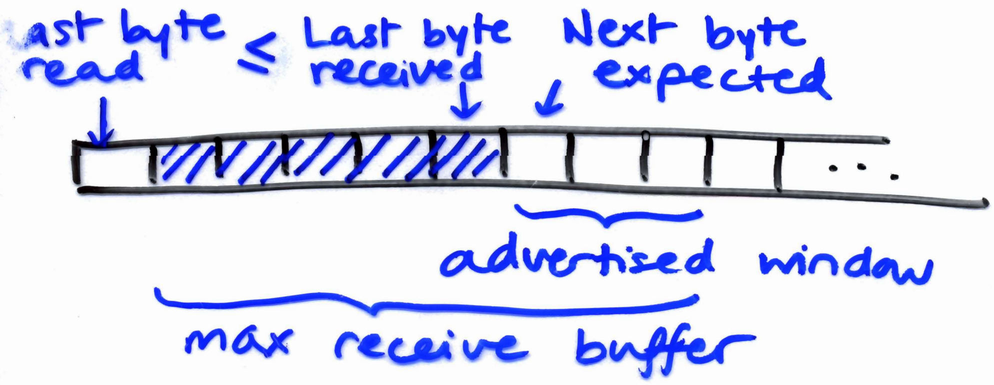
## Sender



## Receiver



out of order data arrivals



in order data arrivals

- flow control: prevent sender from overrunning receiver buffer
  - receiver sends advertised window $W_{max}$ with each ACK (amount of data, starting from NextByteExpected, that it can buffer)
  - $\boxed{W_{max} = \text{MaxRcvBuffer} - (\text{NextByteExp'd} - 1 - \text{LastByteRead})}$
  - If application is reading data as fast as it arrives, $W_{max} = \text{MaxRcvBuffer}$, o/w $W_{max}$ ↓s

- advertised window $W_{max}$ gives the max amount of unacknowledged data that can be outstanding
- effective window is the remaining amount of data that can be transmitted before having to wait for the next ACK

$$\boxed{\text{EffectiveWindow} = W_{max} - \underbrace{(\text{LastByteSent} - \text{LastByteAck'd})}_{\text{current outstanding data}}}$$

- TCP also prevents sending process from overflowing send buffer by blocking it if necessary to ensure

$$\boxed{\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}}$$

- To keep the pipe full, buffers & $W_{max}$ need to be as large as delay × bandwidth product
  - 16-bit $W_{max}$ field → can advertise $2^{16}$ B = 64 KB (insufficient for a 10 Mbps cross-continent connection with 100ms RTT → delay × bandwidth product $0.1 \times \frac{10^7}{8}$ B = 122 KB)
  - window scale option : used during connection establishment (3-way handshake) to specify the no. of bits to left shift the $W_{max}$ field

- when a receiver advertises a window size of 0, the sender cannot send any more data
  - sender starts a <u>persist</u> timer
  - when the timer expires the TCP sender transmits a small segment, to trigger an ACK response from the receiver containing the new window size

  → receive side can be as simple as possible; does not need to initiate any activity (smart sender / dumb receiver design principle)

## TCP segment size

- small segments are less efficient since each segment incurs $\geq 40$ bytes overhead ($\geq 20$ bytes TCP header, 20 bytes IPv4, 40 bytes IPv6 header)
- each network type has a maximum transmission unit (MTU), i.e. the largest IP datagram it can carry in a frame
- TCP has a maximum segment size (MSS) parameter which can be set during connection establishment
  - would ideally want to set MSS as large as possible without resulting in fragmentat^n at the IP level (when a link with an MTU smaller than the IP datagram size is encountered)
  - path MTU discovery can be used to determine appropriate MSS, or sender can set MSS equal to the MTU of the first hop, or minimum MTU for IP networks can be used as MSS
- TCP does not have a minimum segment size
  - TCP supports a push operation that the sending process can use to cause buffered bytes to be sent

- <u>silly window syndrome</u> (SWS)
  - experienced by early naive implementations of basic sliding window flow control
  - if receiving process removes data more slowly than it arrives, the receive buffer builds up & the advertised window ↓s
  - can think of the advertised windows as empty containers sent by the receiver, which the sender can fill with data to transmit to the receiver
  - if the sender fills & transmits small containers immediately instead of waiting to combine them, & if the receiver sends small empty containers as soon as they become available, we end up with lots of tiny inefficient segments
  - <u>receiver SWS avoidance</u>
    - restrict receiver from advertising windows less than MSS or ½ buffer size, whichever is smaller (advertise zero window instead)
  - <u>sender SWS avoidance</u>
    - Nagle's algorithm:
      - push small segments only if no unacknowledged data outstanding
      - can be turned off with TCP_NODELAY option