Network Layer: logical communication between hosts
Transport Layer: logical communication between processes (services: connection-oriented data stream, reliability, flow control, multiplexing, congestion control, connection setup).
Multiplexing @ sender: handle data from multiple sockets, add transport header.
Demultiplexing @ receiver: use header info to deliver received segments to correct socket.
How demultiplexing works:
Host receives IP datagrams: each datagram source IP address, destination IP address, carries one transport-layer segment, source/destination port number.
UDP demultiplexing: SEND: socket=host-local port #, datagram=des IP addr, des port #; RECV: check des port #, direct to socket w/port #.
TCP demultiplexing: source IP addr & port #, des IP addr & port #.
UDP flaws: may be lost, delivered out-of order, no handshaking between snd/rcv; each UDP segment handled indp.
Why UDP? No connec. Establishment, small header size, no congestion control.
Checksum: detects errors in transmitted segment.
ACK: receiver “pkt OK” → sender **NAK(neg ack):** recv “pkt errors”→ sender → retransmission
Handling Duplicates: sender retransmits pkt if ACK/NAK corrupted → sender adds seq # to pkts → recv discards dup pkts
Pipelined Protocol: sender allows multiple, “in-flight”, yet-to-be-ack pkts (twp forms: go-back-N, selective repeat)

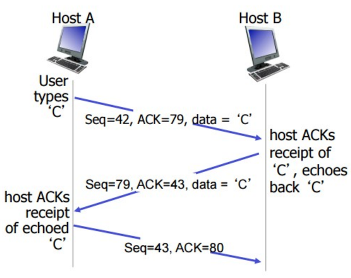
	Go-back-N	**Selective Repeat**
sender	max = N unACK pkts	max = N unACK pckts
receiver	sends **cummulative ACK** (doesn't ACK pck if gap exists)	sends **individual ACK**
sender	timer = oldest unACK pkt (when timer expires: retransmit **all** unACK pkts)	timer = on each unACK pkt (when timer expires: retransmit only that unACK pkt)

TCP seq # & ACKs:

seq # → byte stream “#” of first byte in segment's data(e.g seq=500, 100 bytes → next segment seq=600).
*instead of numbering segments, TCP numbers bytes of transmitted data & uses the # of the first data byte in a segment as that segment's sequence #.

ACK → seq of next byte expected from other side; cumulative ACK.

TCP Seq # & ACKS example:*



simple telnet scenario

****

Computing Timeout Intervals:

Low Fluctuations, Steady graph:

$EstRTT_{new} = (1\text{-}\alpha)*EstRTT_{old} + \alpha*SampleRTT$

where , α is usually 1/8 = 0.125.

High Fluctuations (How to calculate deviations:):

$DevRTTnew = (1 - \beta)\ DevRTTold + \beta\ |SampleRTT - EstRTT|$

where, β is usually ¼ = 0.25.

TimeoutInterval = EstRTT + 4*DevRTT
“est RTT” “safety margin”

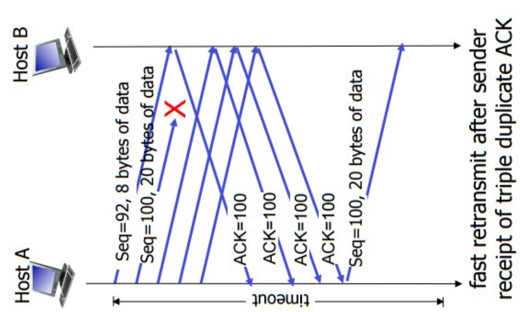
*when a segment timesout and is re-sent, timeout interval = 2 * timeout interval
*when 3 dupACK (indicating out of order), segment is re-sent, even if timer has not expired, this is called **FAST RETRANSMIT**.

****

TCP Fast Retransmit

- time-out period long
- detect lost segments via 3 dupACKs
  (if sender receives 3 dupAcks resend segment w/smallest seq #, w/timing out)

example →



Flow Control: endpoints in TCP prevent a fast sender from "overwhelming" a slow receiver. Each w/**receive window** sizes.
* Receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much too fast.

__Receive Window (rwnd):__
+ two hosts exchange rwnd size at beginning
+ sender guarantees that diff between last seq # & ACK sent never > rwnd

__Principles of Congestion Control__
+"too many sources sending too much data too fast for the network to handle"
+ manifestations:
- lost pkts (buffer overflow at routers)
- long delays (queuing in router buffers)

__Causes of Congestion (from ex in book):__
+ too many senders, too many receivers
+ one router, w/infinite buffers
+ output link capacity: R
+ no retransmission

Perfect knowledge scenario:
+ sender sends only when avail buffer

Known Loss Scenario w/out free buff space:
+ pkts can be lost, dropped @ router bc full buffers
+ sender only resends if packet known to be lost

Known Loss w/free buffer space:
+packets can be lost, dropped at router bc full buffer
+sender only resends if packet known to be lost

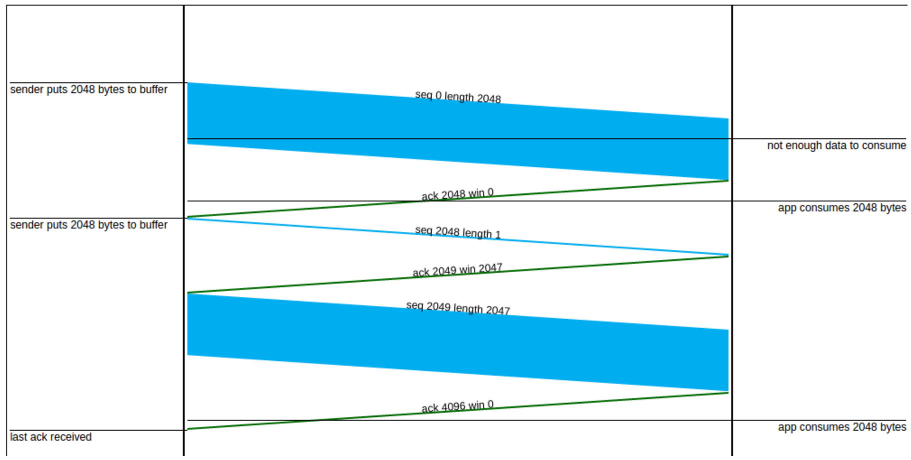
Duplicates (realistic):
+packets can be lost, dropped at router bc full buffer
+ sender times out permanently, sends 2 copies, delivers

__Costs of congestion:__
+more work (retrans) for given "goodput"
+unneeded retrans: link carries multiple copies of pkt
- dec goodput
+when pkt dropped, any "upstream trans capacity used" for that pkt was wasted!

__TCP Congestion Control: add inc/mult dec__
+sender inc trans rate (window size), until loss occurs
- additive inc: inc cwnd by 1 MSS every RTT until loss detected
- multiplicative dec: cut cwnd in half after loss

+**Sending Rate:** roughly: send cwnd bytes, wait RTT for ACKS, then send more bytes ; rate = cwnd/ RTT (bytes/sec)
+**Last byte sent/ACK <= cwnd**

***rwnd Question (assume rwnd=0): How does recv tell sender**
* **to start sending again?**
*Answer: sender sends tiny messages (1 byte each); recv then uses
* ACK messages for these to carry info about rwnd.



__TCP Slow Start__
+ when connection begins, inc rate expon until 1st loss event
- init cwnd = 1 MSS
- double cwnd every RTT
- done by incrementing cwnd per ACK reveived
- **TCP Tahoe, cwnd = 1**
- **TCP Reno, 3 dupACKs, cwnd=cwnd/2**
- Slow Start Treshold

ssthresh = cwnd/2

***3 dupAcks** = ssthresh=cwnd/2 +3

* The rate of growth of cwnd is determined by another variable
* **ssthresh**, the slow-start threshhold. When the value of cwnd is
* than ssthresh, the size of the window grows exponentially
* (doubling). When cwnd exceeds ssthresh, the window size grows
* only linearly.

****

***Congestion control** is different from flow control. It uses observations about network behavior to control the amount of data that is sent. A variable called the "congestion window" (cwnd) is set to the maximum number of bytes that can be transmitted at one time. The congestion window grows when acknowledgements are received in a timely way and shrinks when timeouts occur.
*Seq, length, and ACK fields in the TCP segment. When the receiver acknowledges a segment with seq and len, it send the ACK=seq + len, the number of the next byte expected. (NOTE: this happens only if all seqments up to seq have been received; if there are gaps, then the acknowledgement is for the last segment that was received with no preceding gaps.)