Transport Layer

Our goals:

- Understand principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Roadmap

1. Transport-layer services
2. Multiplexing and demultiplexing
3. Connectionless transport: UDP
4. Principles of reliable data transfer
5. Connection-oriented transport: TCP
   a. Segment structure
   b. Reliable data transfer
   c. Flow control
   d. Connection management
6. Principles of congestion control
7. TCP congestion control

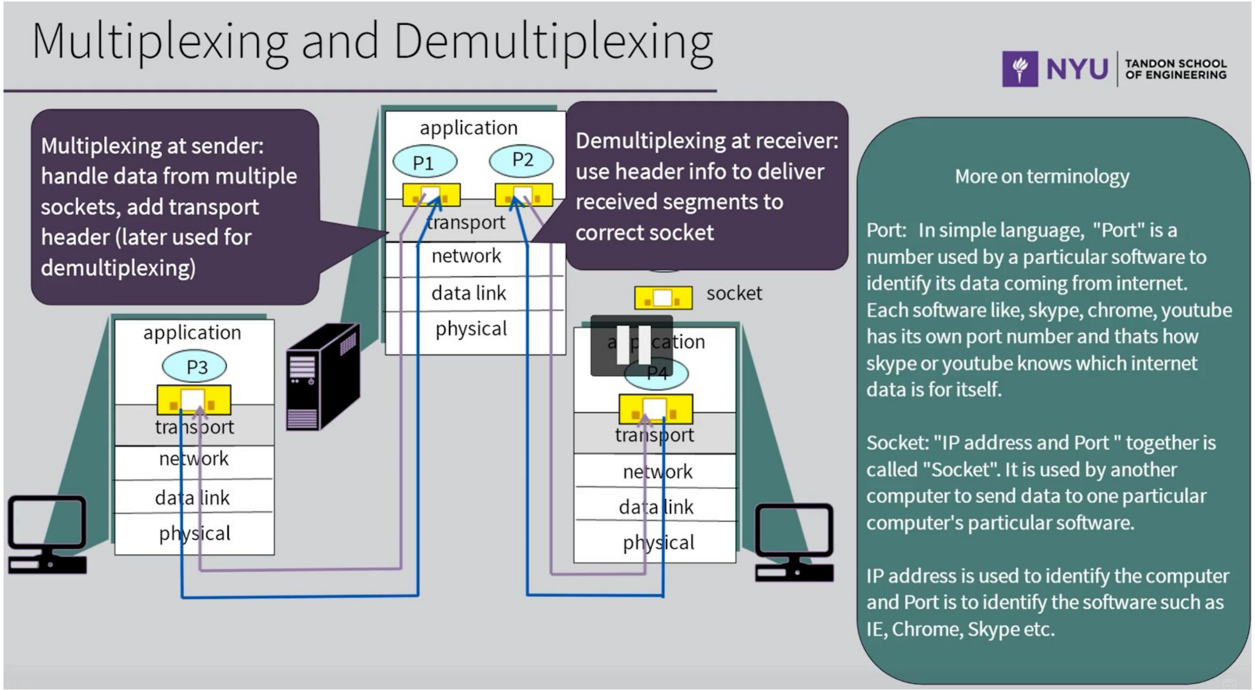
Transport Services and Protocols

- Provide *logical communication* between app processes running on different hosts
- Transport protocols run in end systems
  - Send side: breaks app messages into segments, passes to network layer
  - Receiving side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP

Internet Transport-layer Protocols

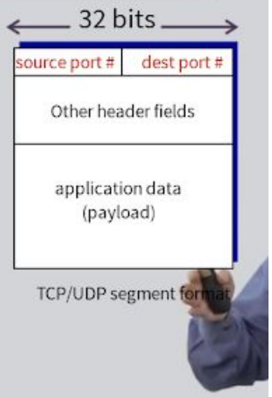
- Reliable, in-order delivery (TCP)
  - Congestion control
  - Flow control
  - Connection setup
- Unreliable, unordered delivery: UDP
  - No-frills extension of “best effort” IP
- Services not available
  - Delay guarantees

- Bandwidth guarantees



How Demultiplexing Works

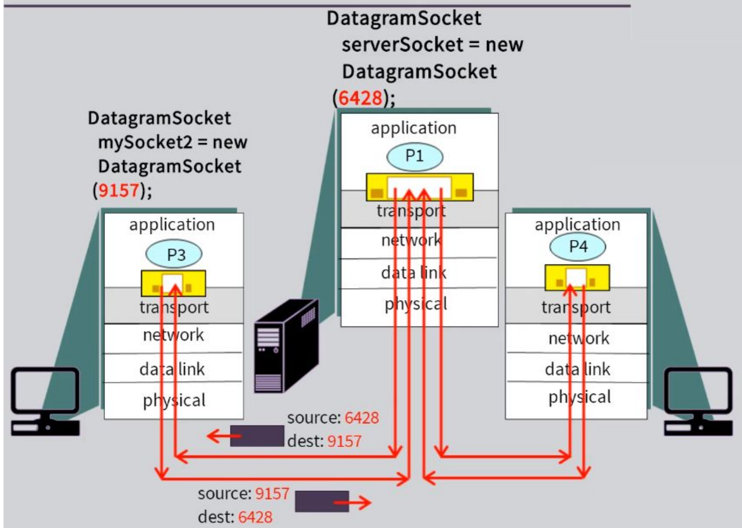
- Host receives IP datagrams
  - Each datagram has source IP address, destination IP address
  - Each datagram carries one transport-layer segment
- Host uses *IP addresses & port numbers* to direct segment to appropriate socket



Connectionless Demultiplexing

- When host receives UDP segment:
  - Checks destination port # in segment
  - Directs UDP segment to socket with that port #
- IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

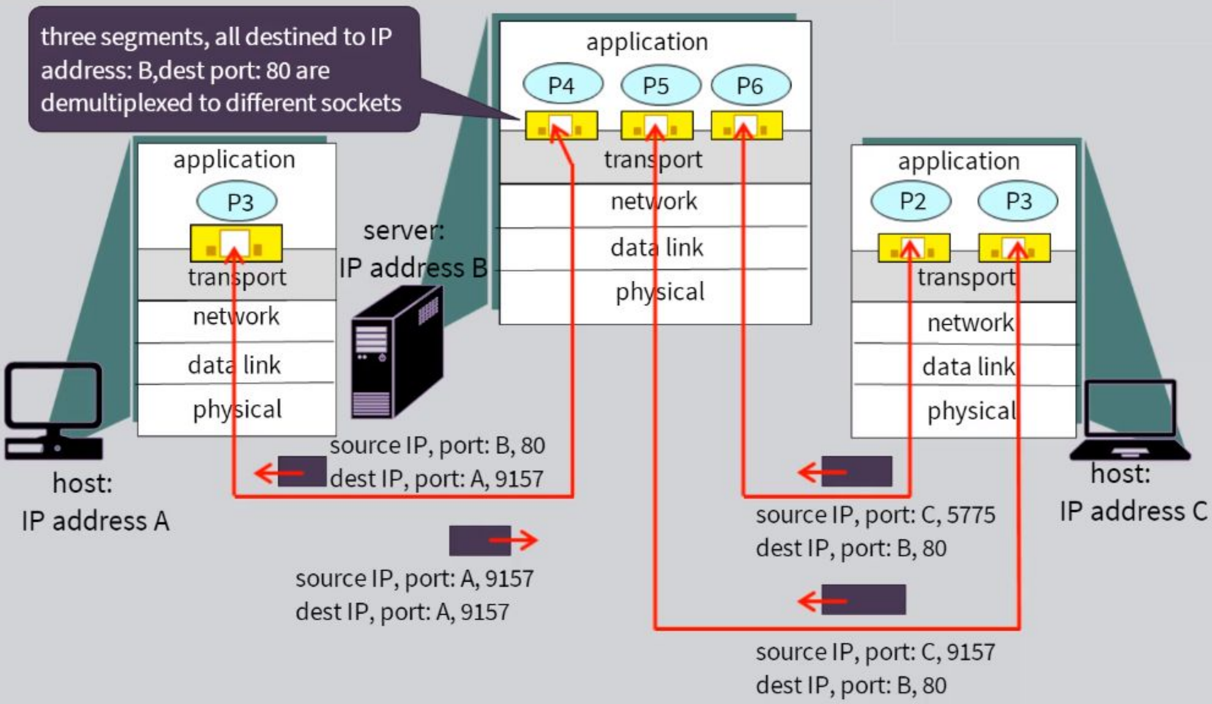
Connectionless Demux: Example



Connection-Oriented Demux

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Server host may support many simultaneous TCP sockets: each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have different socket for each request

Example



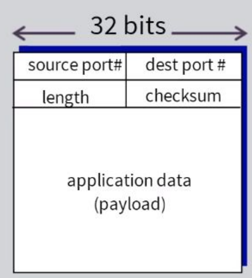
Connectionless Transport: UDP

- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others
- UDP use:
  - Streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
- "Best effort" service
  UDP segments may be:
  - Lost
  - Delivered out-of-order to app

UDP: Segment Header

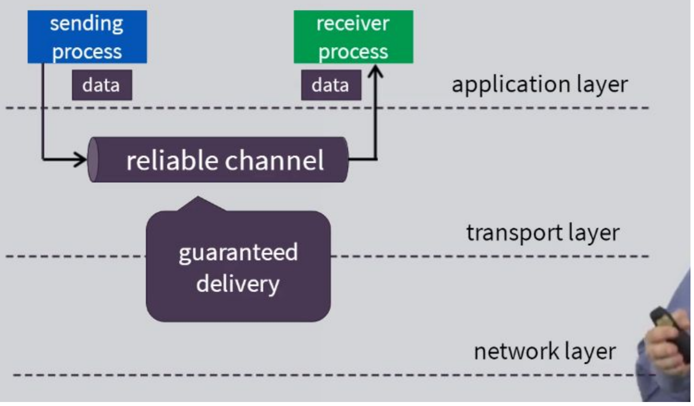
Why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state at sender, receiver
- Small header size
- No congestion control: UDP can blast away as fast as desired

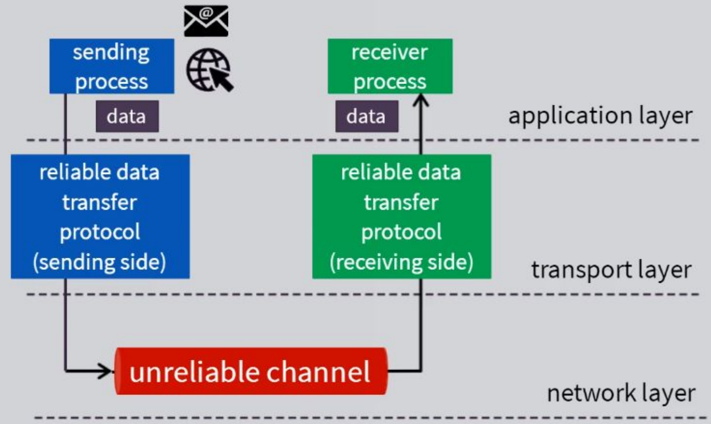


UDP segment format

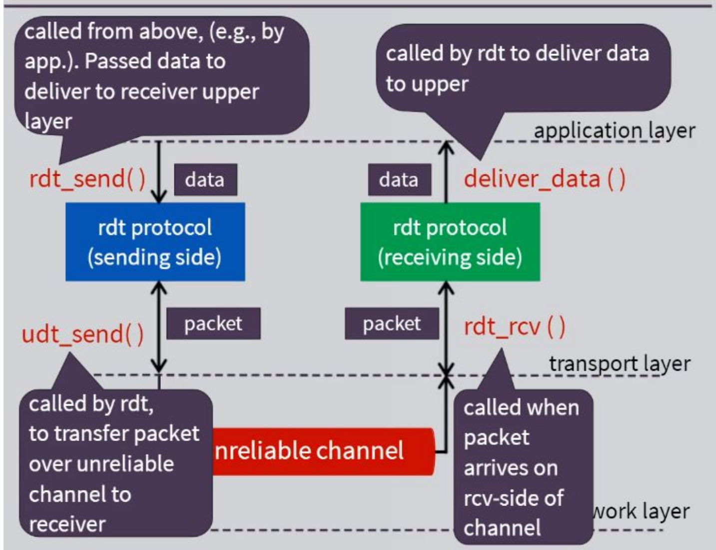
Principles of Reliable Data Transfer



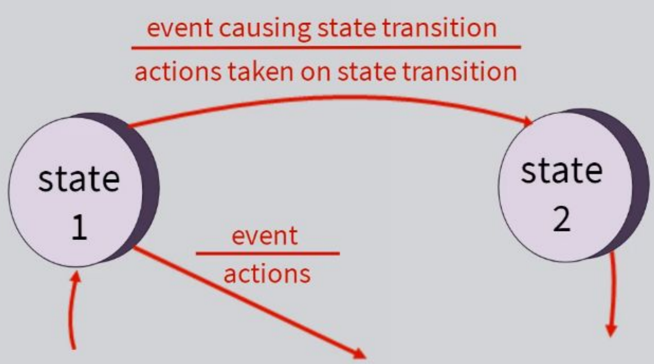
Principles of Reliable Data Transfer



RDT: Getting Started



RDT: Getting Started



RDT1.0

Reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel

RDT2.0: Channel with Bit Errors

- Underlying channel may flip bits in packet
  - Checksum to detect bit
- *The* question: how to recover from errors?
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - Sender retransmits pkt on receipt of NAK

RDT2.0 has a Fatal Flaw!

What happens if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver!
- Can't just retransmit: possible duplicate

Handling duplicates:

- Sender retransmits current pkt if ACK/NAK corrupted
- Sender adds *sequence number* to each pkt
- Receiver discards (doesn't deliver up) duplicate pkt

Stop and wait

- Sender sends one packet, then waits for receiver respond

RDT2.1: Discussion

Sender:

- Seq # added to pkt
- Two seq. #'s (0,1) will suffice.
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must "remember" whether "expected" pkt should have seq # of 0 or 1

Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #
- Note: receiver can *not* know if its last ACK/NAK received OK at sender

RDT2.2: a NAK-free Protocol

- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - Receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

RDT3.0: Channels with Errors and Loss

New assumption:
Underlying channel can also lose packets (data, ACKs)

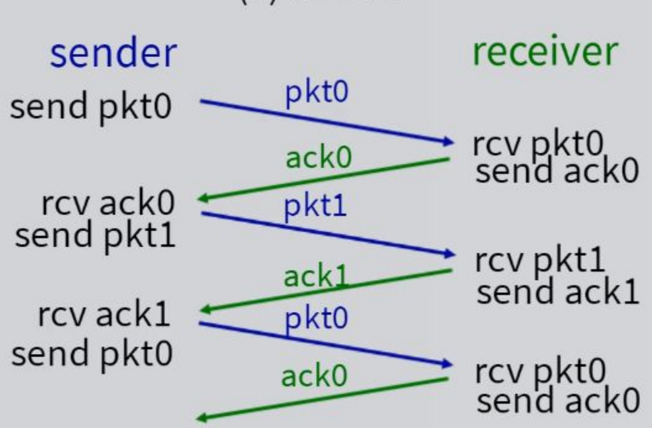
- Checksum, seq. #, ACKs, retransmissions will be of help … but not enough

Approach: Sender waits "reasonable" amount of time for ACK

- Retransmits if no ACK received in this time
- If pkt (or ACK) just delayed (not lost):
  - Retransmission will be duplicate, but seq. #'s already handles this
  - Receiver must specify seq # of pkt being ACKed
- Requires countdown timer

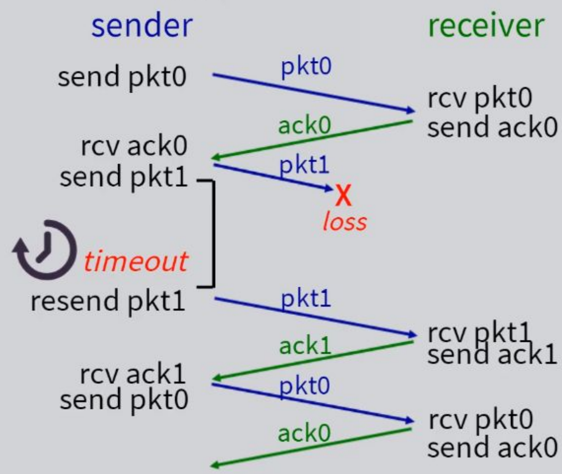
RDT3.0 in Action

(a) no loss



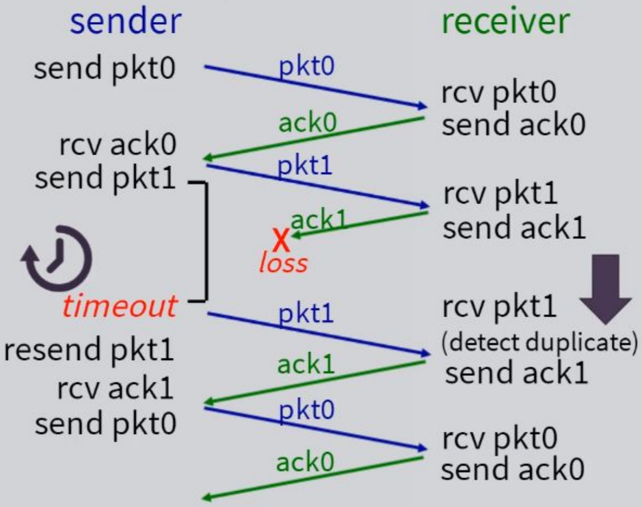
RDT3.0 in Action

(b) packet loss



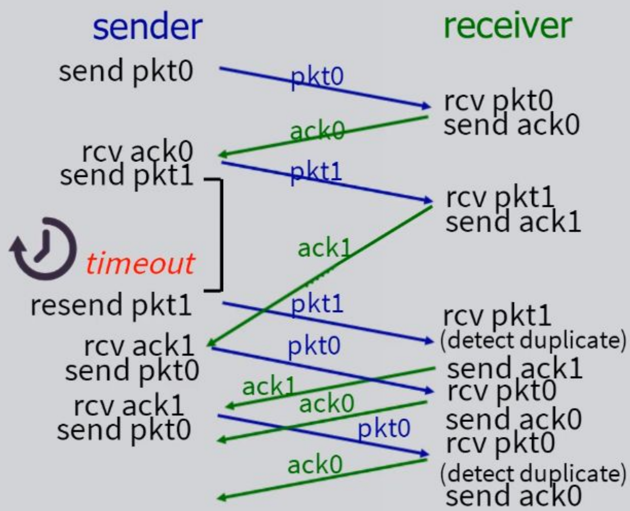
RDT3.0 in Action

(c) ACK loss



RDT3.0 in Action

(d) premature timeout/ delayed ACK



Performance of RDT3.0

- Rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8\ \mu s$$

U_{sender}: *utilization* – fraction of time sender busy sending

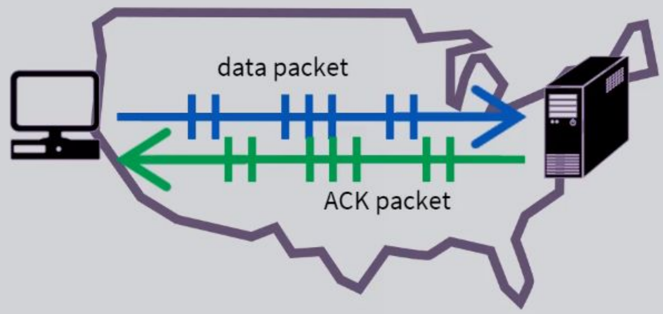
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- If RTT=30 millisecond, 1KB pkt every 30 millisecond: 33kB/sec throughput over 1 Gbps link

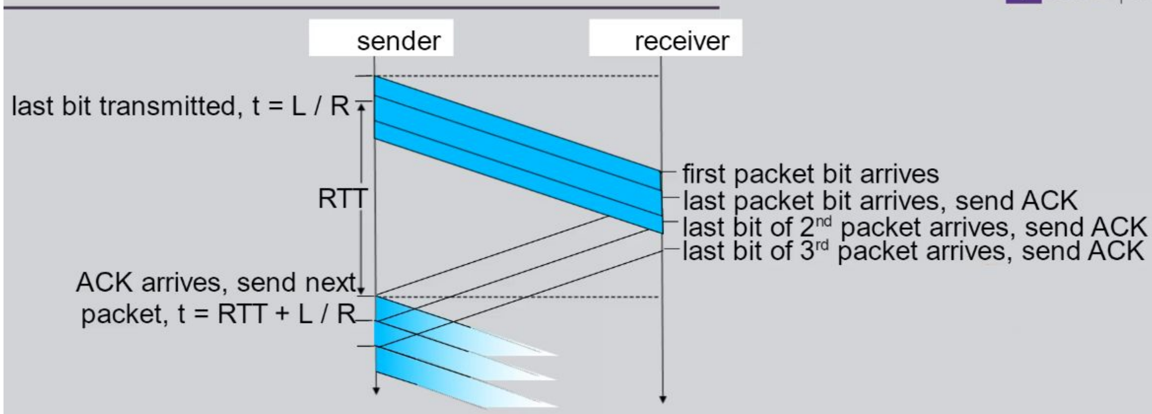
Pipelined Protocols

Two generic form:

- Go-Back-N
- Selective repeat



Pipelining: Increased Utilization



$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined Protocols: Overview

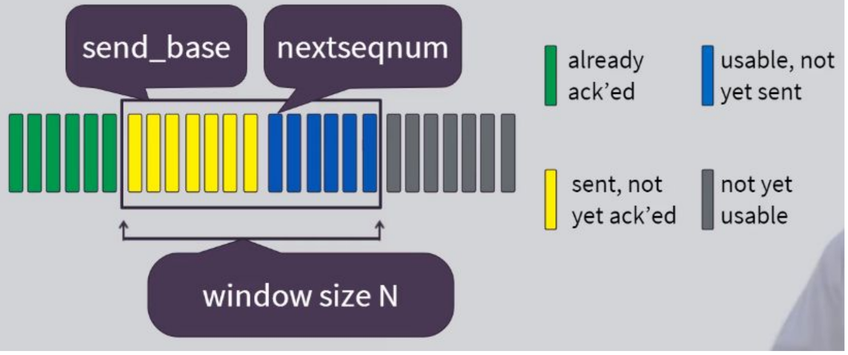
Go-back-N:

- Sender can have up to N unacked packets in pipeline
- Receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- Sender has timer for oldest unacked packet
  - When timer expires, retransmit *all* unacked packets

Selective Repeat:

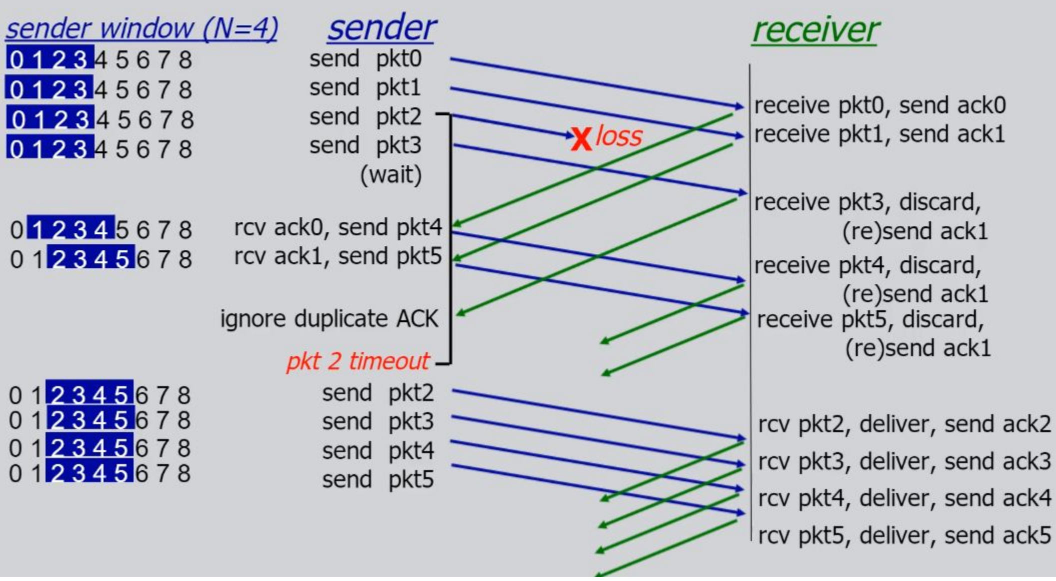
- Sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only that unacked packet

Go-Back-N: Sender



GBN in Action

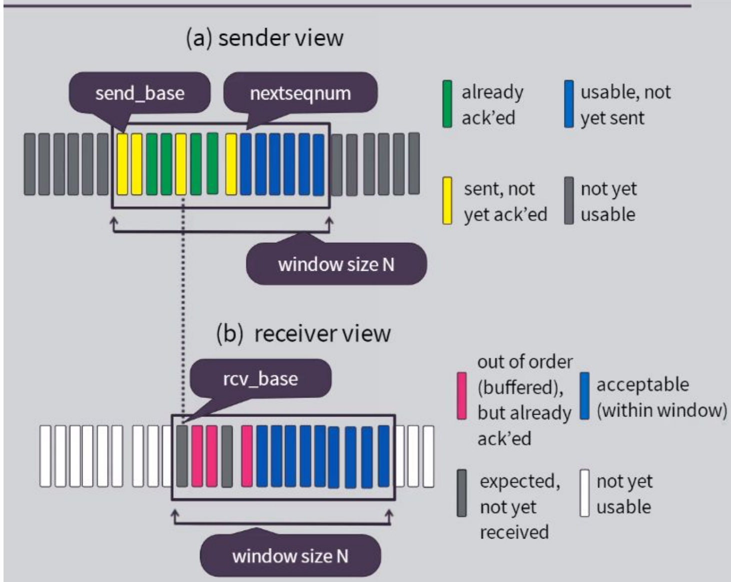
NYU TANDON SCHOOL OF ENGINEERING



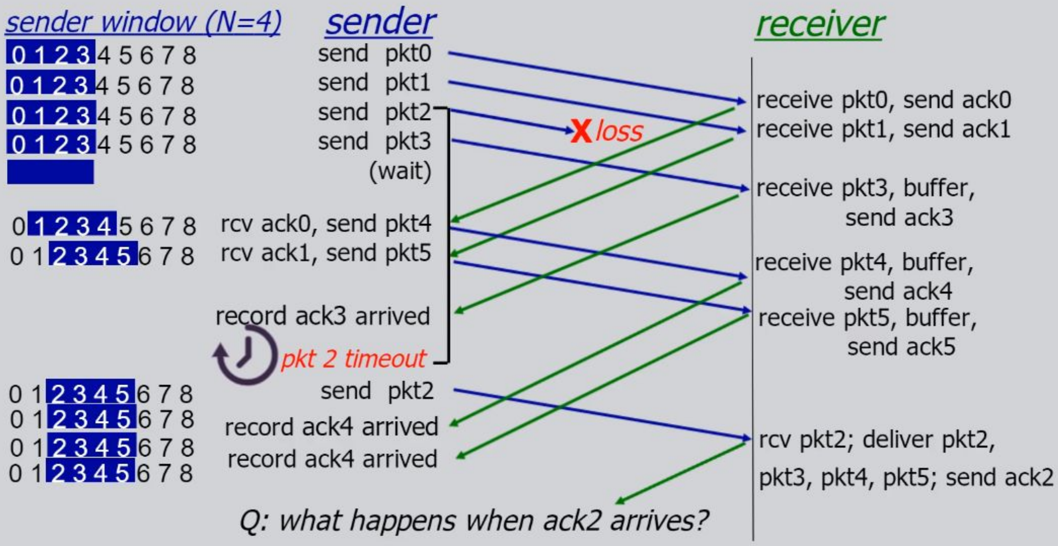
Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - *N* consecutive seq #'s
  - limits seq #s of sent, unACKed pkts

Selective Repeat



Selective Repeat in Action

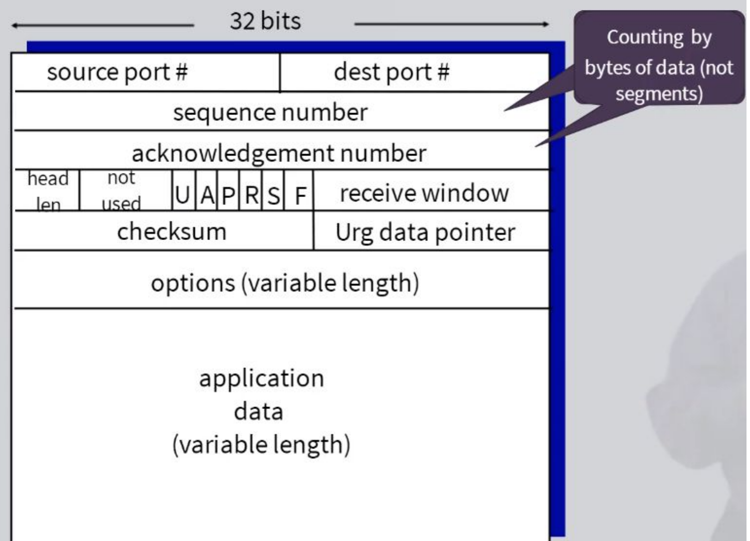


TCP: Overview

RFCs: 793,1122,1323, 2018, 2581

- Point-to-point:
  - One sender, one receiver
- Reliable, in-order *byte steam:*
  - No "message boundaries"
- Pipelined:
  - TCP congestion and flow control set window size
- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-oriented:
  - Handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver

TCP Segment Structure



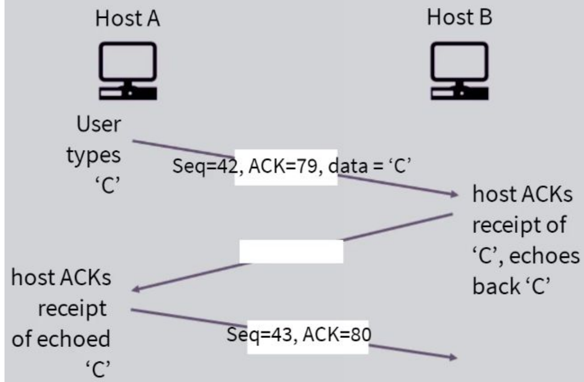
TCP Seq. Numbers, ACKs

Sequence numbers:
- Byte stream “number” of first byte in segment’s data

acknowledgements:
- Seq # of next byte expected from other side
- Cumulative ACK

TCP Seq. Numbers, ACKs

simple telnet scenario



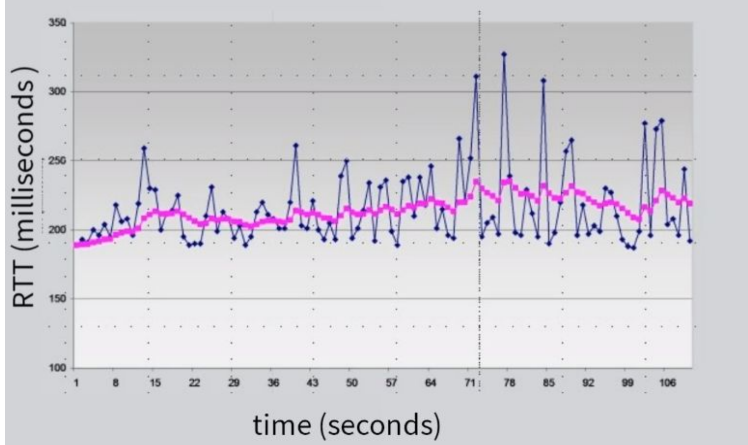
TCP Round Trip Time, Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?
- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT “smoother”
  - average several *recent* measurements, not just current **SampleRTT**

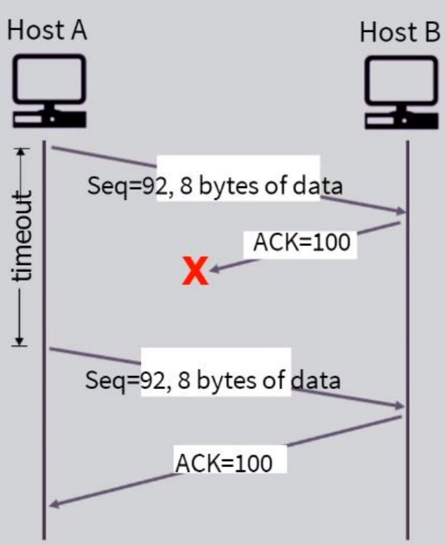
TCP Round Trip Time, Timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
typical value: a = 0.125
```



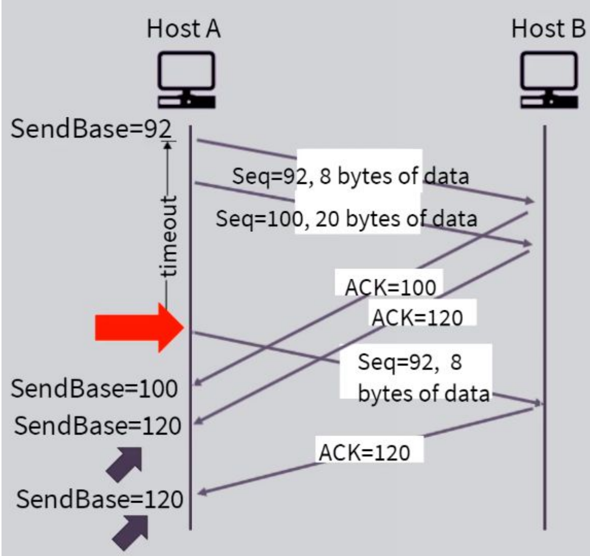
TCP: Retransmission Scenarios

lost ACK scenario



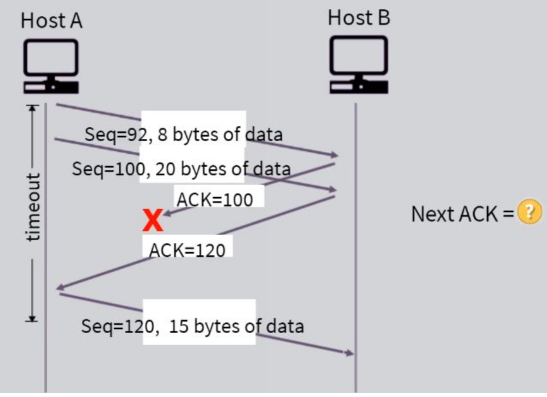
TCP: Retransmission Scenarios

premature timeout



TCP: Retransmission Scenarios

cumulative ACK



TCP ACK Generation [RFC 1122, RFC 2581]

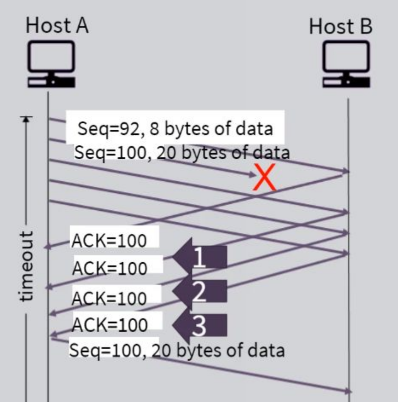
event at receiver	TCP receiver action
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP Fast Retransmit

If sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- Likely that unacked segment lost, so don't wait for timeout

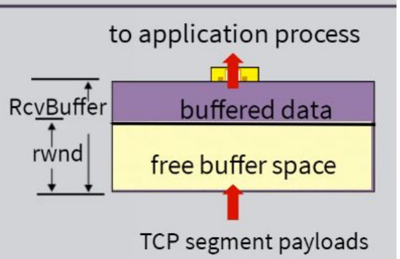
TCP Fast Retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP Flow Control

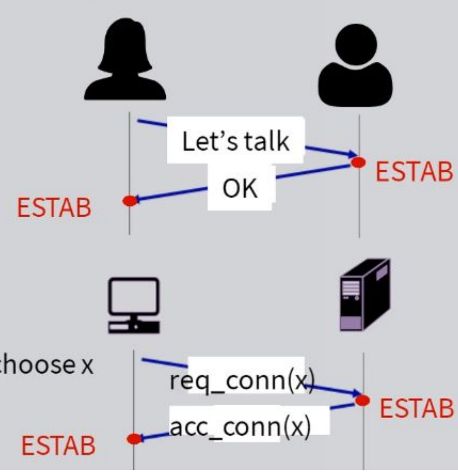
- Receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - Many operating systems autoadjust **RcvBuffer**
- Sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- Guarantees receive buffer will not overflow

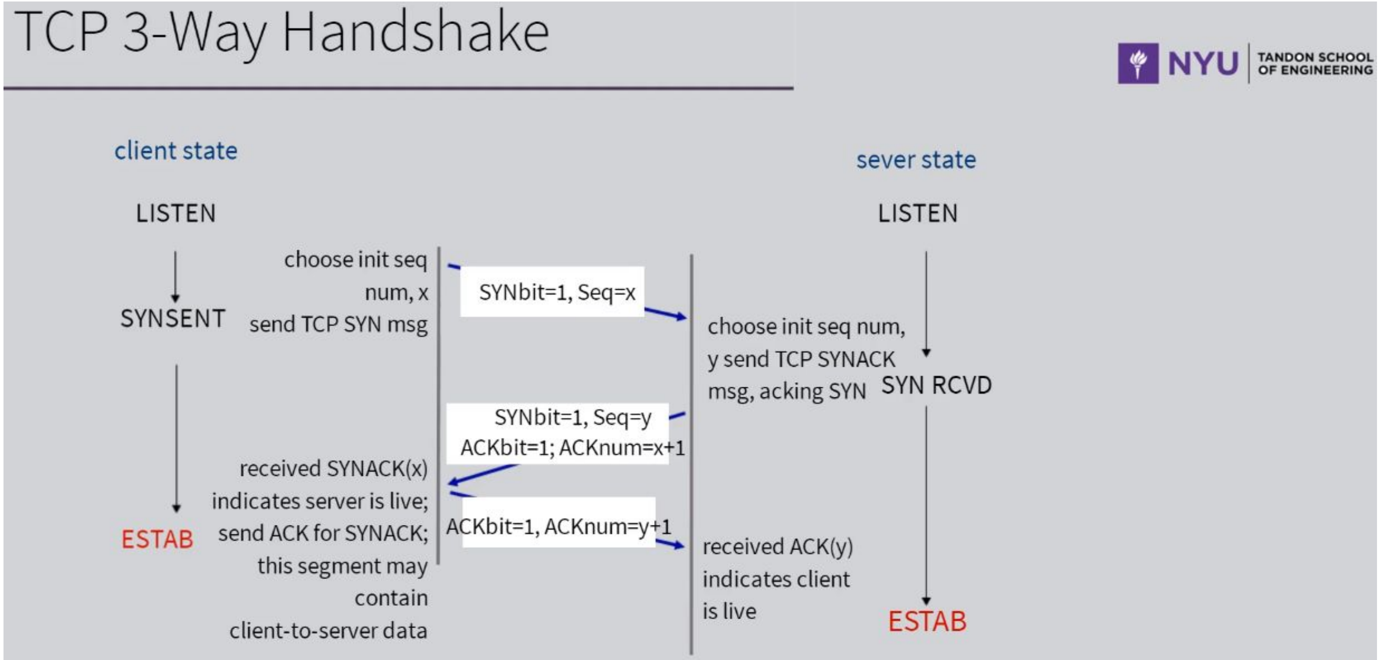


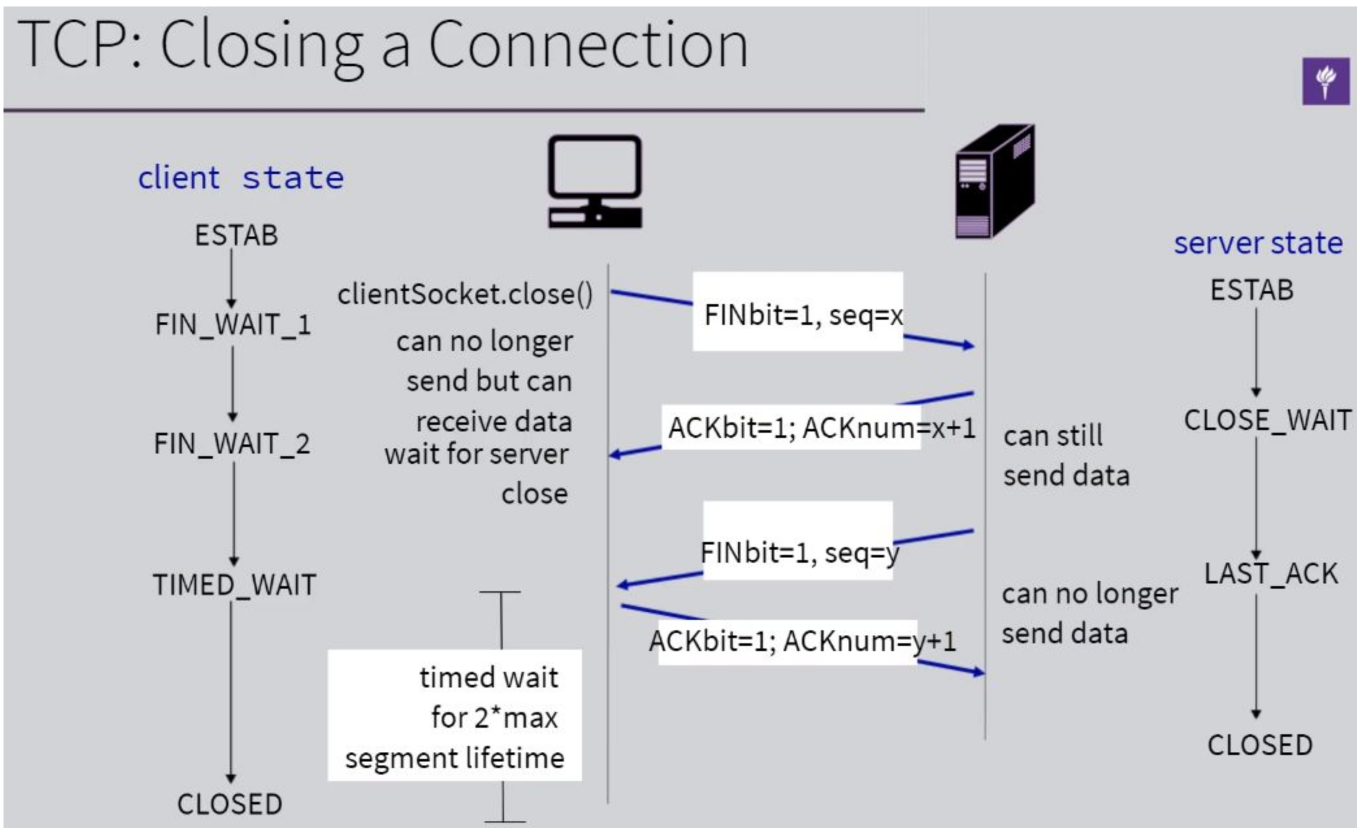
receiver-side buffering

Agreeing to Establish a Connection

2-way handshake:




TCP 3-Way Handshake
NYU TANDON SCHOOL OF ENGINEERING
client state
sever state
LISTEN
LISTEN
SYNSENT
choose init seq num, x send TCP SYN msg
SYNbit=1, Seq=x
choose init seq num, y send TCP SYNACK msg, acking SYN
SYN RCVD
SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1
received SYNACK(x) indicates server is live; send ACK for SYNACK; this segment may contain client-to-server data
ESTAB
ACKbit=1, ACKnum=y+1
received ACK(y) indicates client is live
ESTAB

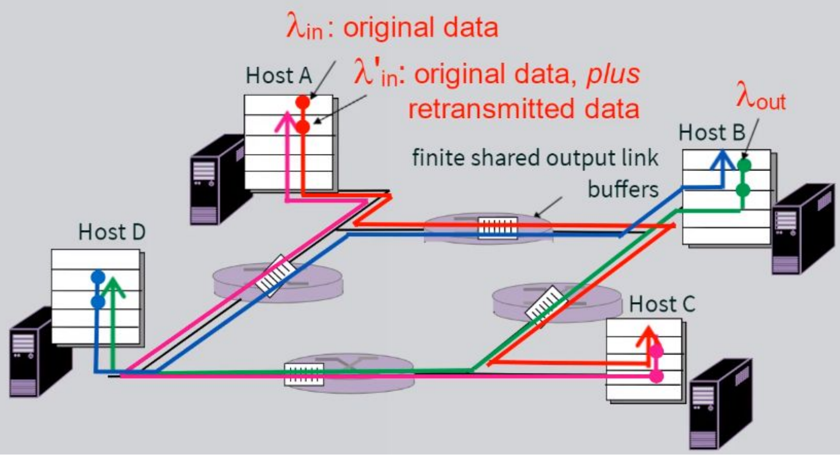

TCP: Closing a Connection
client state
server state
ESTAB
ESTAB
clientSocket.close()
FIN_WAIT_1
FINbit=1, seq=x
can no longer send but can receive data
CLOSE_WAIT
ACKbit=1; ACKnum=x+1
FIN_WAIT_2
wait for server close
can still send data
FINbit=1, seq=y
LAST_ACK
TIMED_WAIT
can no longer send data
ACKbit=1; ACKnum=y+1
timed wait for 2*max segment lifetime
CLOSED
CLOSED

Principles of Congestion Control

Congestion:

- Informally: "too many sources sending too much data too fast for network to handle"
- Different from flow control!
- Manifestations:
- Lost packets (buffer overflow at routers)
- Long delays (queueing in router buffers)

Congestion: Scenario



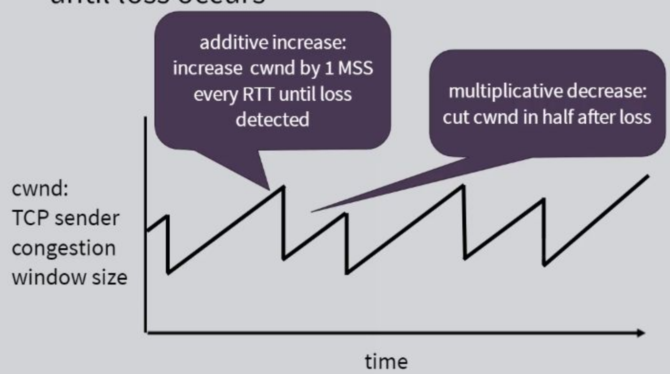
Congestion: Scenario

Another "cost"of congestion:
when packet dropped, any "upstream transmission capacity used for that packet was wasted!

TCP Congestion Control

Additive Increase Multiplicative Decrease (AIMD)

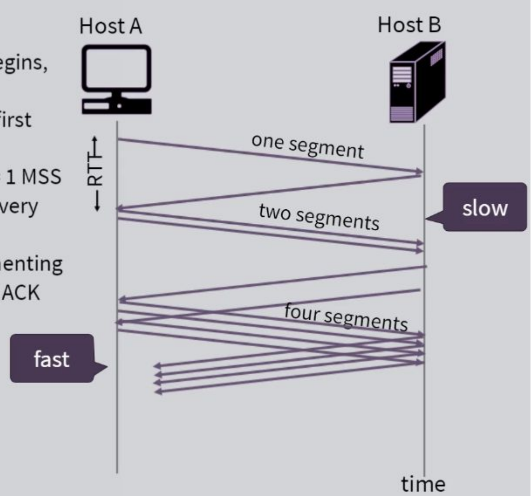
- *Approach: S*ender increases transmission rate (window size), probing for usable bandwidth, until loss occurs



TCP Congestion Control

TCP Slow Start

- When connection begins, increase rate exponentially until first loss event:
  - Initially **cwnd** = 1 MSS
  - Double **cwnd** every RTT
  - Done by incrementing **cwnd** for every ACK received



Summary

- Principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Instantiation, implementation in the Internet
  - UDP
  - TCP