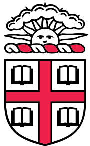
CSCI-1680
Transport Layer 2
Data over TCP

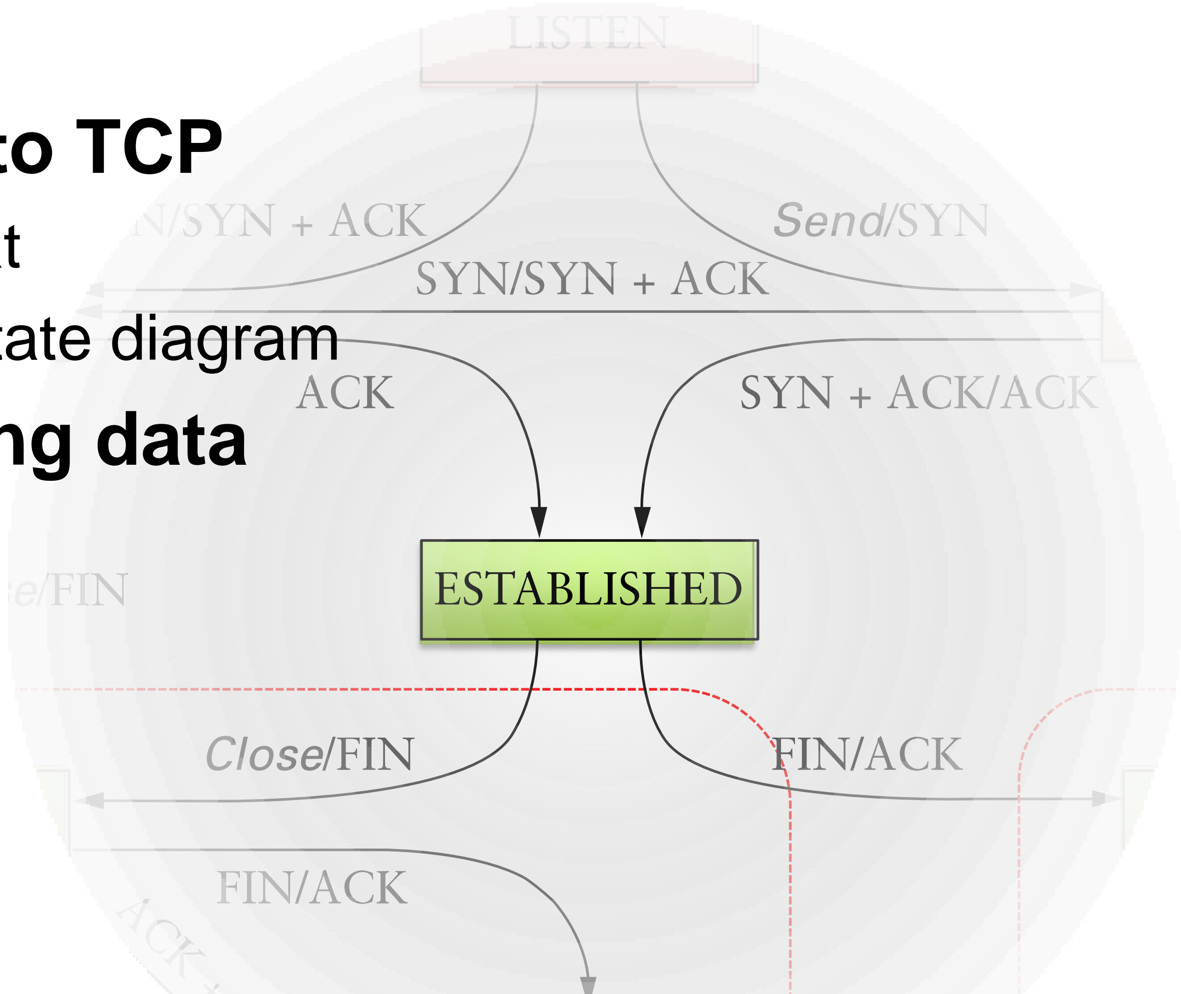
Chen Avin



Based partly on lecture notes by David Mazières, Phil Levis, John Jannotti, Peterson & Davie, Rodrigo Fonseca and “Computer Networking: A Top Down Approach” - 6th edition

Last Class

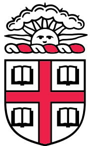
- **Introduction to TCP**
  - Header format
  - Connection state diagram
- **Today: sending data**



TCP reliable data transfer

- **TCP creates rdt service on top of IP’ s unreliable service**
  – pipelined segments
  – cumulative acks
  – single retransmission timer
- **retransmissions triggered by:**
  – timeout events
  – duplicate acks

let’ s initially consider simplified TCP sender:
  – ignore duplicate acks
  – ignore flow control, congestion control



TCP sender events:

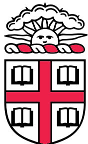
- ***data rcvd from app:***
- **create segment with seq #**
- **seq # is byte-stream number of first data byte in segment**
- **start timer if not already running**
  - think of timer as for oldest unacked segment
  - expiration interval: `TimeOutInterval`

timeout:

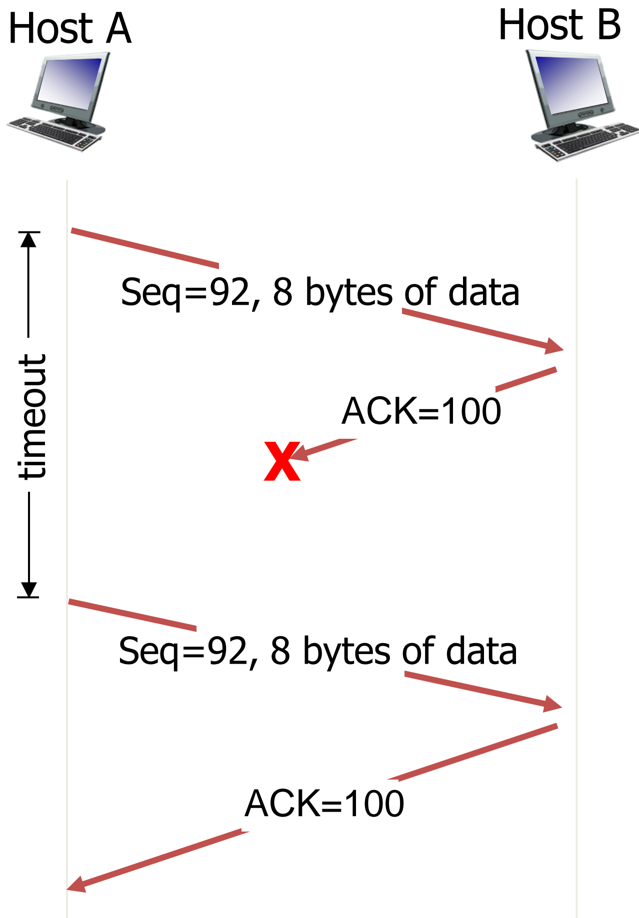
- **retransmit segment that caused timeout**
- **restart timer**

ack rcvd:

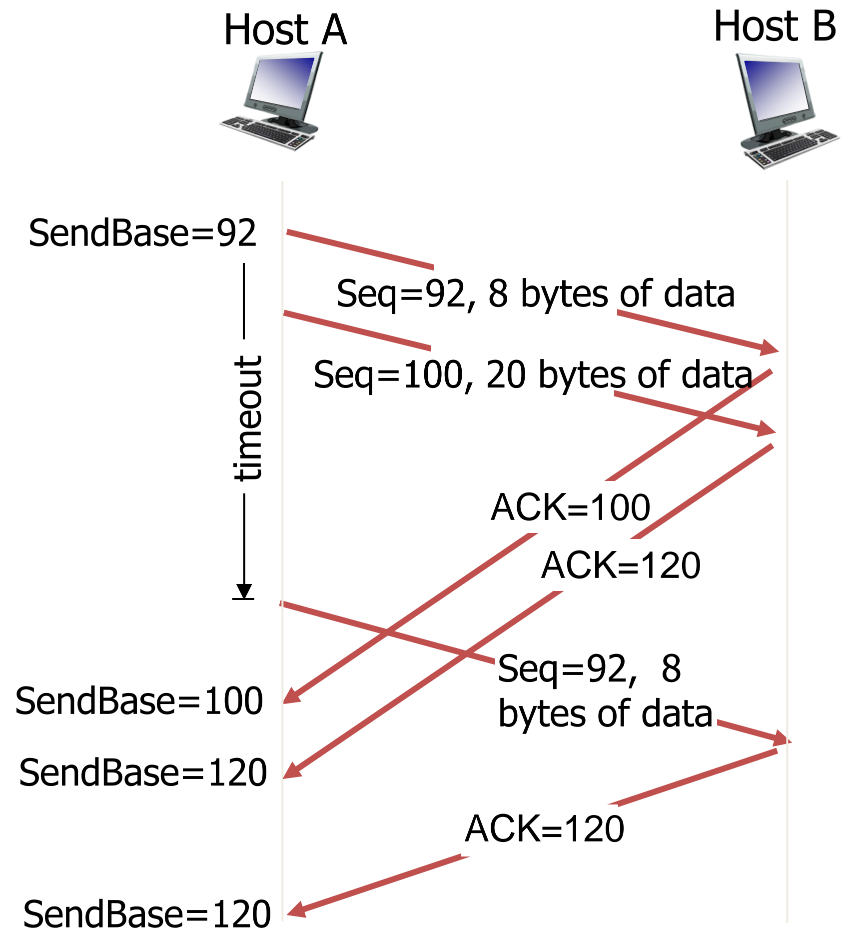
- **if ack acknowledges previously unacked segments**
  – update what is known to be ACKed
  – start timer if there are still unacked segments



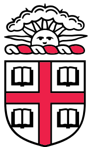
TCP: retransmission scenarios



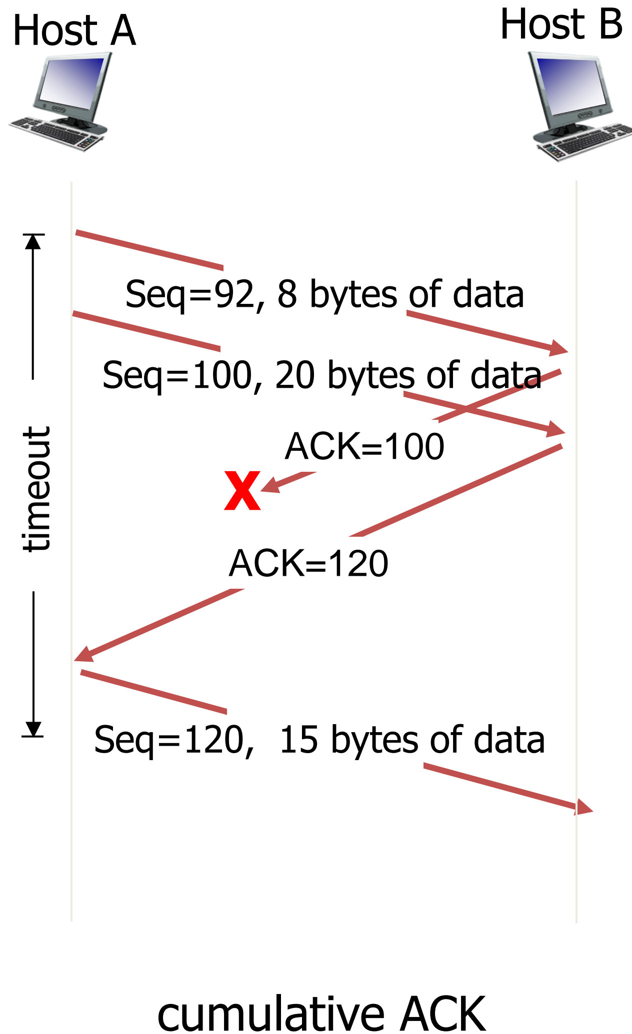
lost ACK scenario



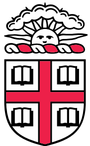
premature timeout



TCP: retransmission scenarios

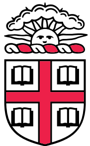


cumulative ACK



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



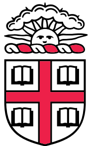
TCP fast retransmit

- **time-out period often relatively long:**
  - long delay before resending lost packet
- **detect lost segments via duplicate ACKs.**
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

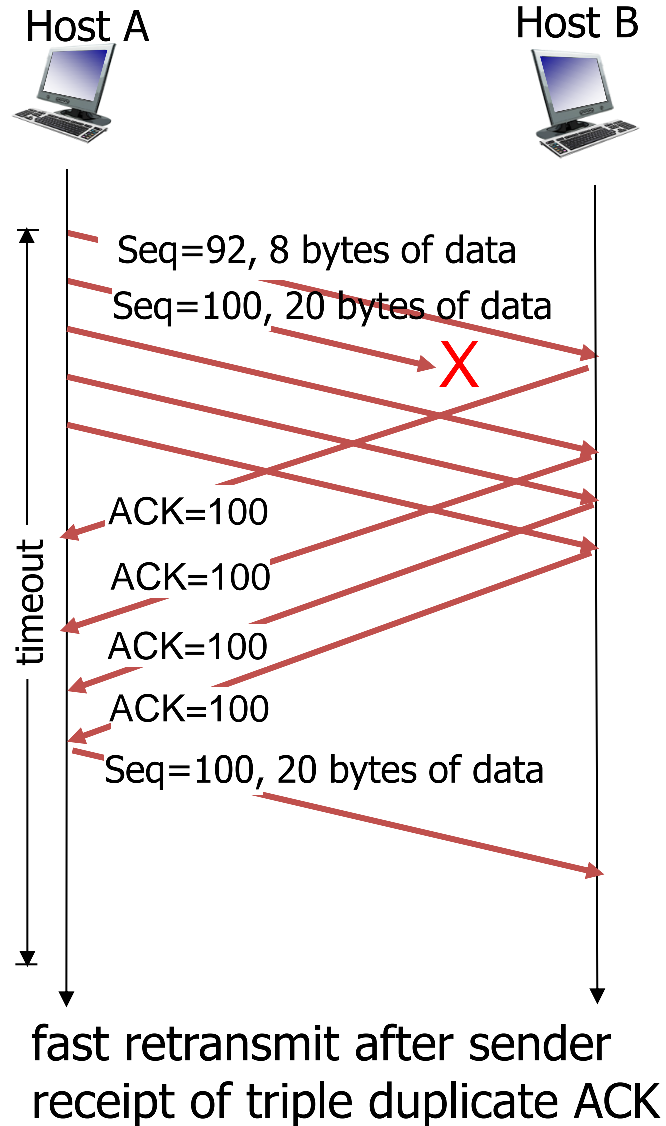
TCP fast retransmit

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

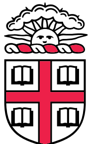
- likely that unacked segment lost, so don't wait for timeout



TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK



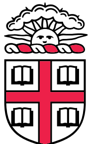
TCP round trip time, timeout

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
```

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

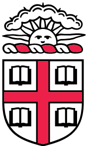
RTT (milliseconds)

350
300
250
200
150
100

1 8 15 22 29 36 43 50 57 64 71 78 85 92 99 106

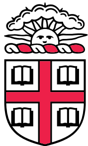
time (seconds)

sampleRTT
EstimatedRTT



Originally

- **$EstRTT = (1 - \alpha) \times EstRTT + \alpha \times SampleRTT$**
- **$Timeout = 2 \times EstRTT$**
- **Problem 1:**
  - in case of retransmission, ack corresponds to which send?
  - Solution: only sample for segments with no retransmission
- **Problem 2:**
  - does not take variance into account: too aggressive when there is more load!



TCP round trip time, timeout

- **timeout interval:** `EstimatedRTT` **plus "safety margin"**
  - large variation in `EstimatedRTT ->` larger safety margin
- **estimate SampleRTT deviation from EstimatedRTT:**

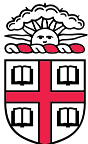
```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|

              (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT (points to EstimatedRTT)

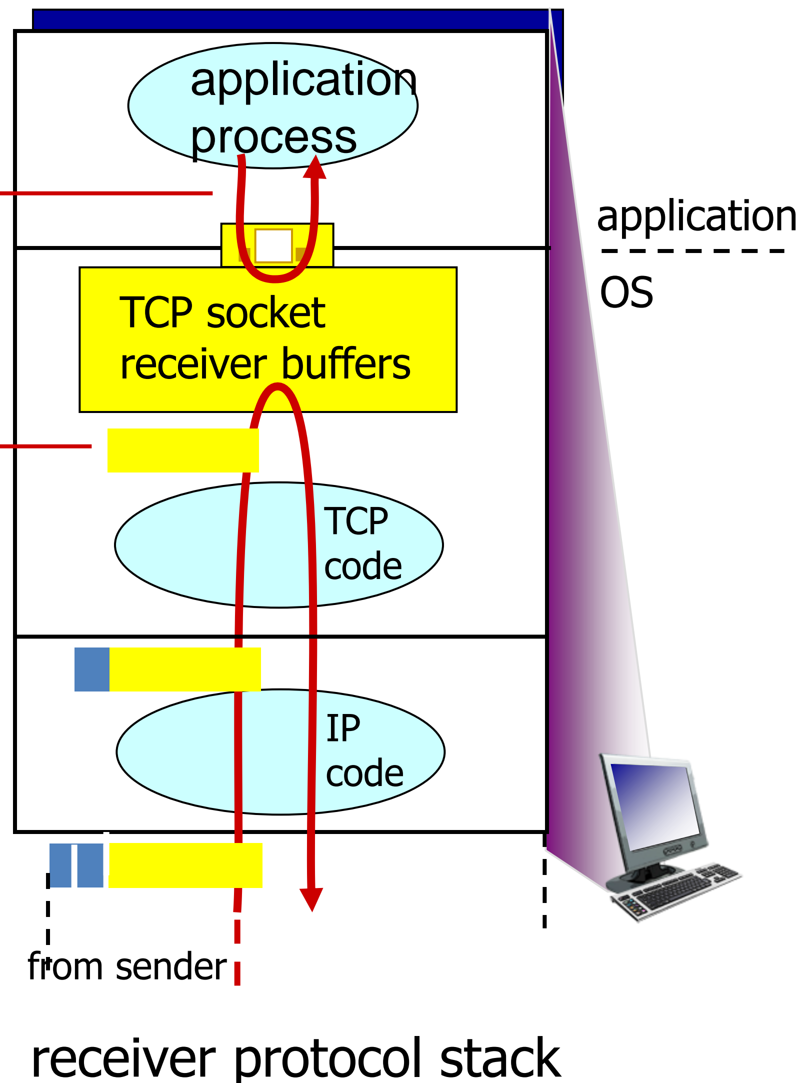
"safety margin" (points to 4*DevRTT)



TCP flow control

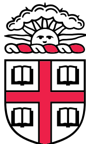
application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



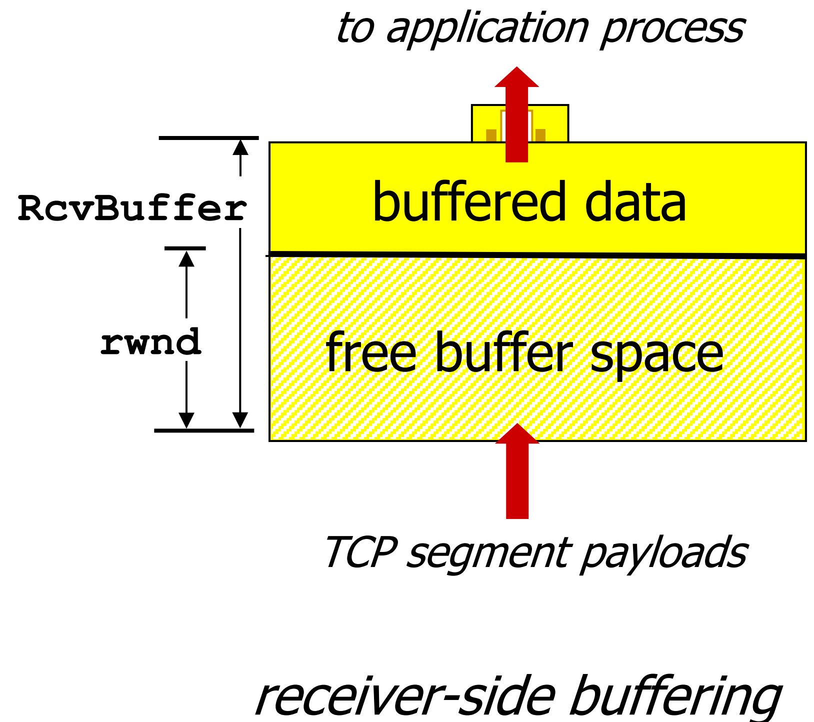
flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

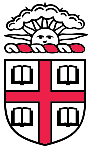


TCP flow control

- **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**

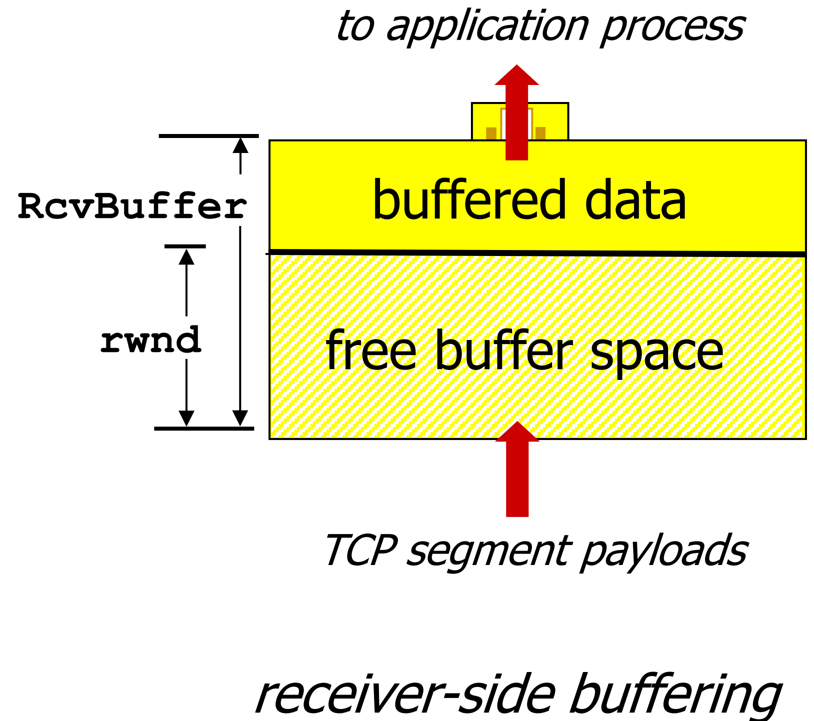


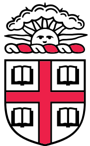
receiver-side buffering



TCP flow control – A problem

- **Advertised window can fall to 0**
  - How?
  - Sender eventually stops sending, blocks application
- **Sender keeps sending 1-byte segments until window comes back > 0**



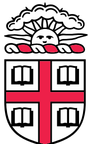


When to Transmit?

- **Nagle's algorithm**
- **Goal: reduce the overhead of small packets**

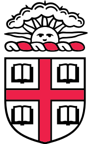
```
If available data and window >= MSS
  Send a MSS segment
else
  If there is unAcked data in flight
      buffer the new data until ACK 
  arrives
  else
      send all the new data now
```

- **Receiver should avoid advertising a window <= MSS after advertising a window of 0**



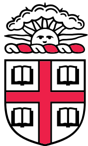
Delayed Acknowledgments

- **Goal: Piggy-back ACKs on data**
  - Delay ACK for 200ms in case application sends data
  - If more data received, immediately ACK second segment
  - Note: never delay duplicate ACKs (if missing a segment)
- **Warning: can interact *very* badly with Nagle**
  - Temporary deadlock
  - Can disable Nagle with TCP_NODELAY
  - Application can also avoid many small writes



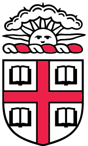
Limitations of Flow Control

- Network may be the bottleneck
- Signal from receiver not enough!
- Sending too fast will cause queue overflows, heavy packet loss
- Flow control provides *correctness*
- Need more for performance: congestion control



Second goal

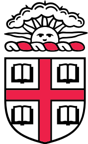
- **We should not send more data than the network can take: *congestion control***



Principles of congestion control

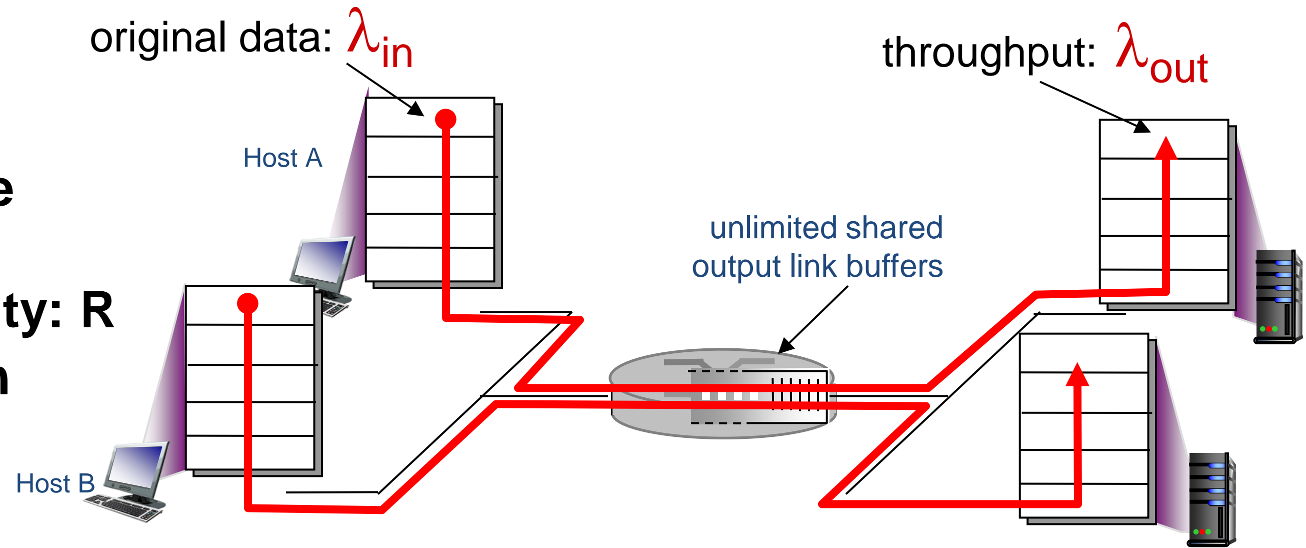
congestion:

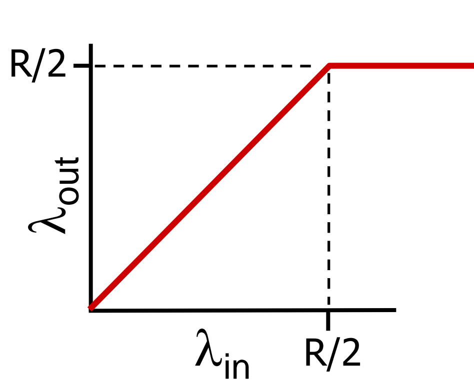
- **informally: "too many sources sending too much data too fast for *network* to handle"**
- **different from flow control!**
- **manifestations:**
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- **a top-10 problem!**

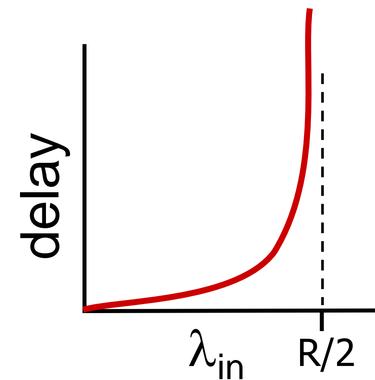


Causes/costs of congestion: scenario 1

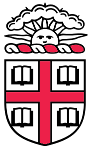
- **two senders, two receivers**
- **one router, infinite buffers**
- **output link capacity: R**
- **no retransmission**







- **maximum per-connection throughput: R/2**
- large delays as arrival rate, λ_{in}, approaches capacity



Causes/costs of congestion: scenario 2

- **one router, *finite* buffers**
- **sender retransmission of timed-out packet**
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

λ_{in} : original data

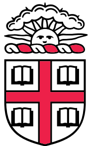
λ'_{in}: original data, *plus* retransmitted data

λ_{out}

Host A

Host B

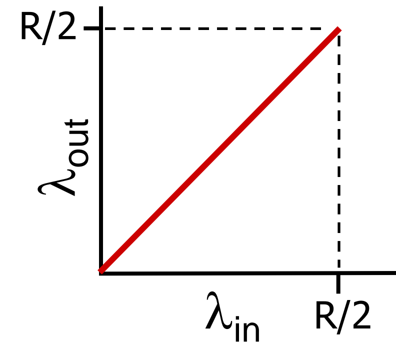
finite shared output link buffers



Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- **sender sends only when router buffers available**



λ_{in} : original data

copy

λ'_{in}: original data, *plus* retransmitted data

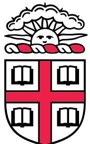
λ_{out}

A

free buffer space!

Host B

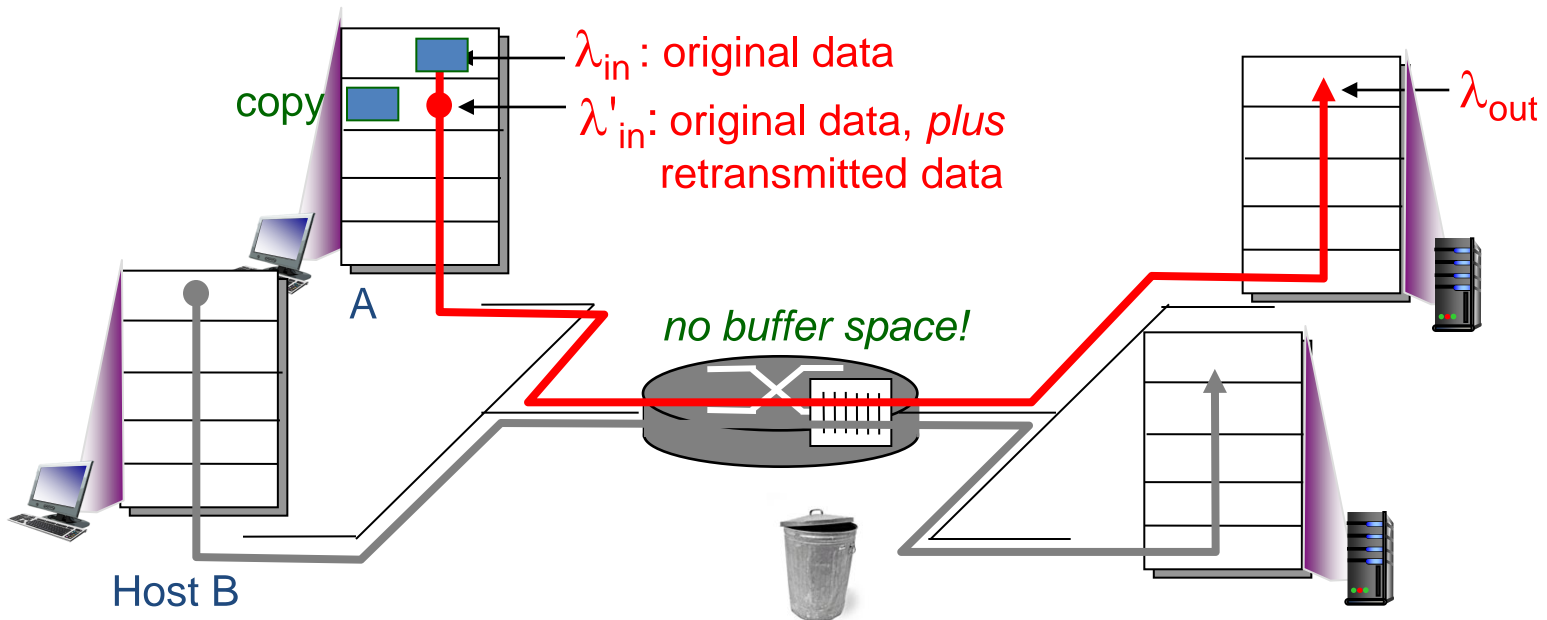
finite shared output link buffers

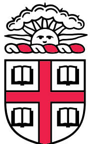


Causes/costs of congestion: scenario 2

Idealization: known loss **packets can be lost, dropped at router due to full buffers**

- **sender only resends if packet *known* to be lost**

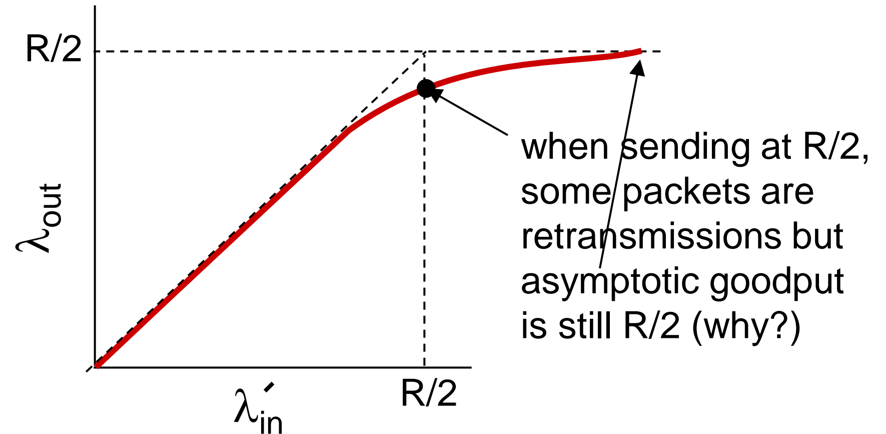


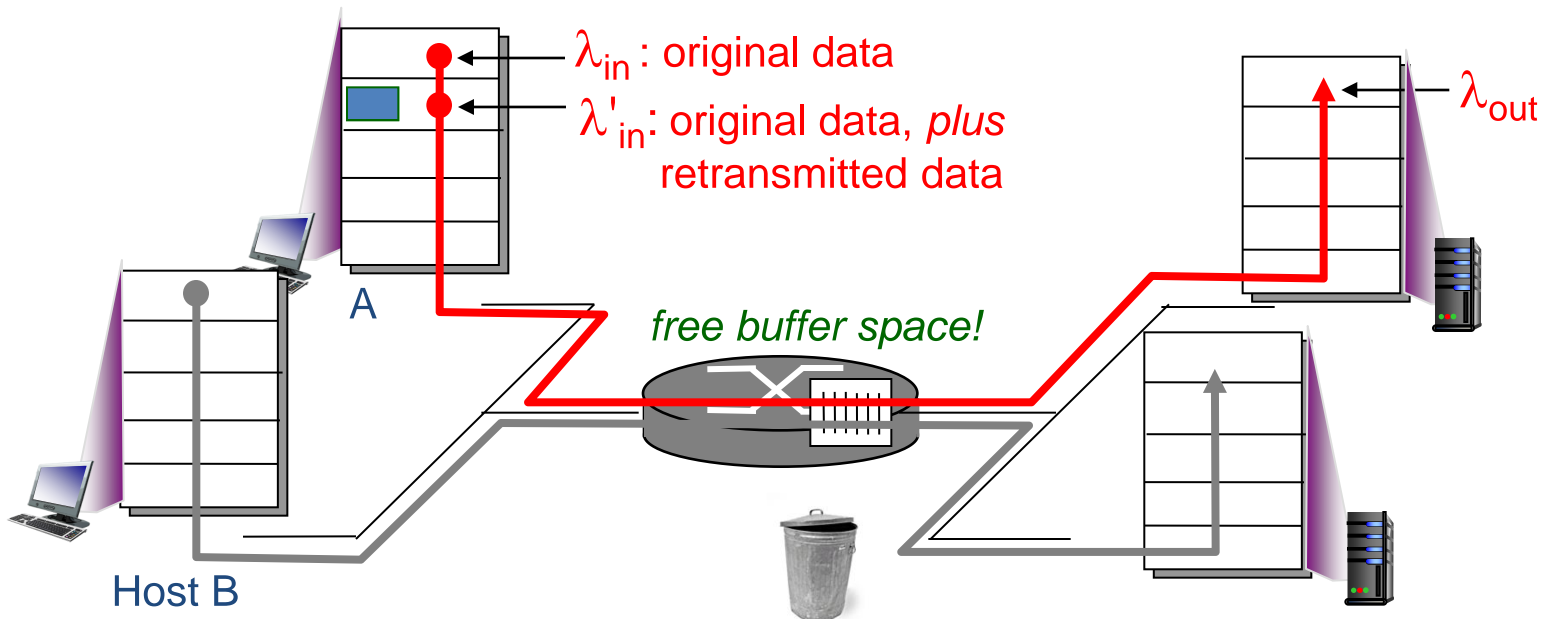


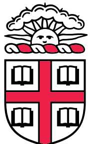
Causes/costs of congestion: scenario 2

Idealization: known loss **packets can be lost, dropped at router due to full buffers**

- **sender only resends if packet *known* to be lost**



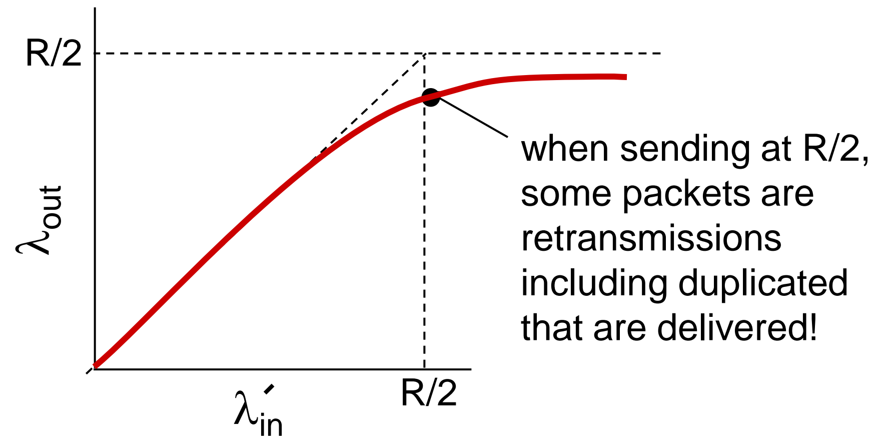




Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



timeout

λ_{in}

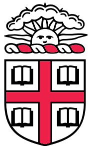
λ'_{in}

λ_{out}

A

free buffer space!

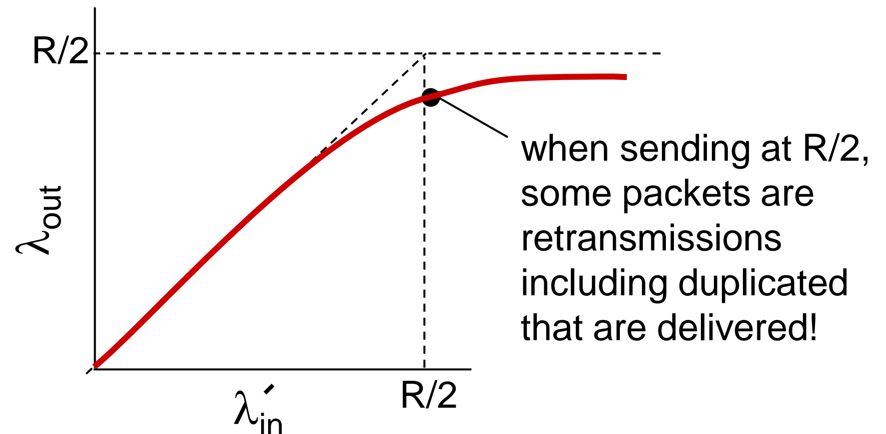
Host B



Causes/costs of congestion: scenario 2

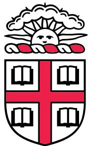
Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

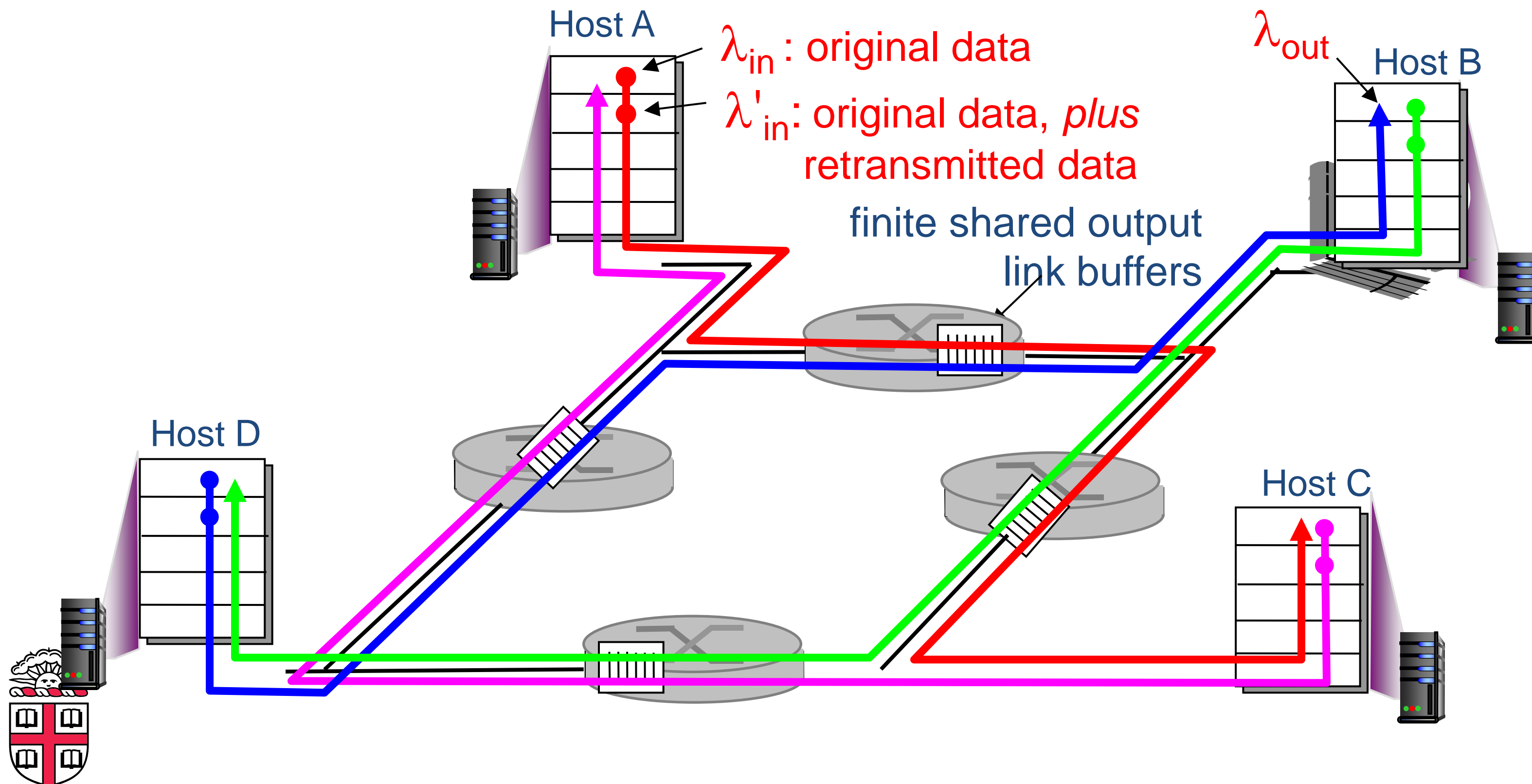


Causes/costs of congestion: scenario 3

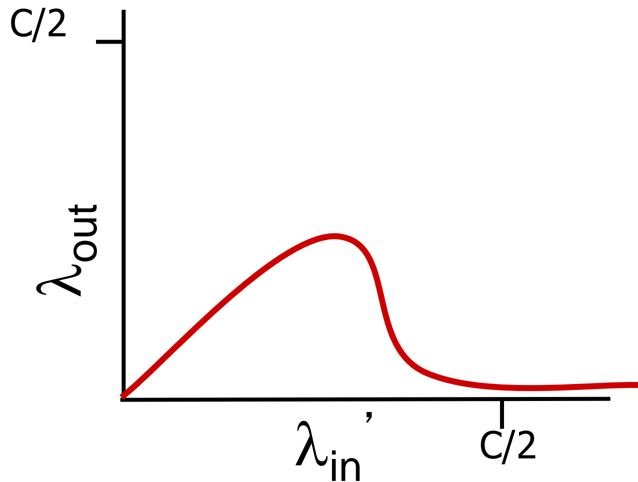
- four senders
- multihop paths
- timeout/retransmit

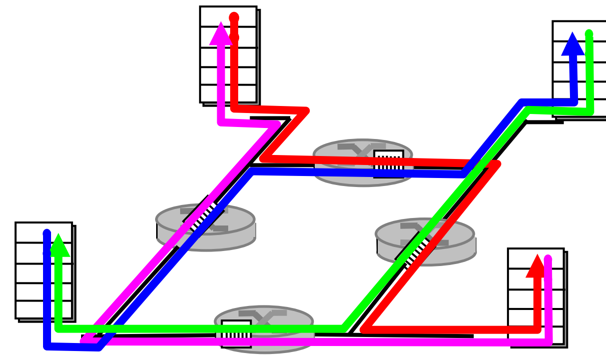
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



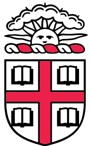
Causes/costs of congestion: scenario 3





another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!



Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
- single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
- explicit rate sender should send at

Why Packet Switching and Not VC ?

- We use packet switching because it makes efficient use of the links. Therefore, buffers in the routers are frequently occupied.
- If buffers are always empty, delay is low, but our usage of the network is low.
- If buffers are always occupied, delay is high, but we are using the network more efficiently.
- So how much congestion is too much?

Why Packet Switching and Not VC ?

- IP layer doesn't provide explicit feedback to end systems
- TCP implements host-based, feedback-based, window-based congestion control.
- TCP sources attempt to determine how much capacity is available
- TCP sends packets, then reacts to observable events (loss).

TCP Congestion Control - main points

- **TCP sources detect congestion and, distributively reduce the rate at which they transmit.**
- **The rate is controlled using the TCP window size.**
- **TCP achieves high throughput by encouraging high delay.**
- **TCP sources change the sending rate by modifying the window size:**

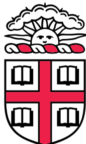
Window = min{Advertized window, Congestion Window}

Receiver ("rwnd")   Transmitter ("cwnd")

- **In other words, send at the rate of the slowest component: network or receiver.**

A Short History of TCP

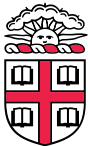
- **1974: 3-way handshake**
- **1978: IP and TCP split**
- **1983: January 1st, ARPAnet switches to TCP/IP**
- **1984: Nagle predicts congestion collapses**
- **1986: Internet begins to suffer congestion collapses**
  - LBL to Berkeley drops from 32Kbps to 40bps
- **1987/8: Van Jacobson fixes TCP, publishes seminal paper*: (TCP Tahoe)**
- **1990: Fast transmit and fast recovery added (TCP Reno)**



* Van Jacobson. Congestion avoidance and control. SIGCOMM ’88

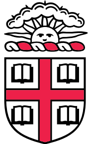
Congestion Collapse
Nagle, rfc896, 1984

- **Mid 1980's. Problem with the protocol *implementations*, not the protocol!**
- **What was happening?**
  – Load on the network → buffers at routers fill up → round trip time increases
- **If close to capacity, and, e.g., a large flow arrives suddenly…**
  – RTT estimates become too short
  – Lots of retransmissions → increase in queue size
  – Eventually many drops happen (full queues)
  – Fraction of useful packets (not copies) decreases



TCP Congestion Control

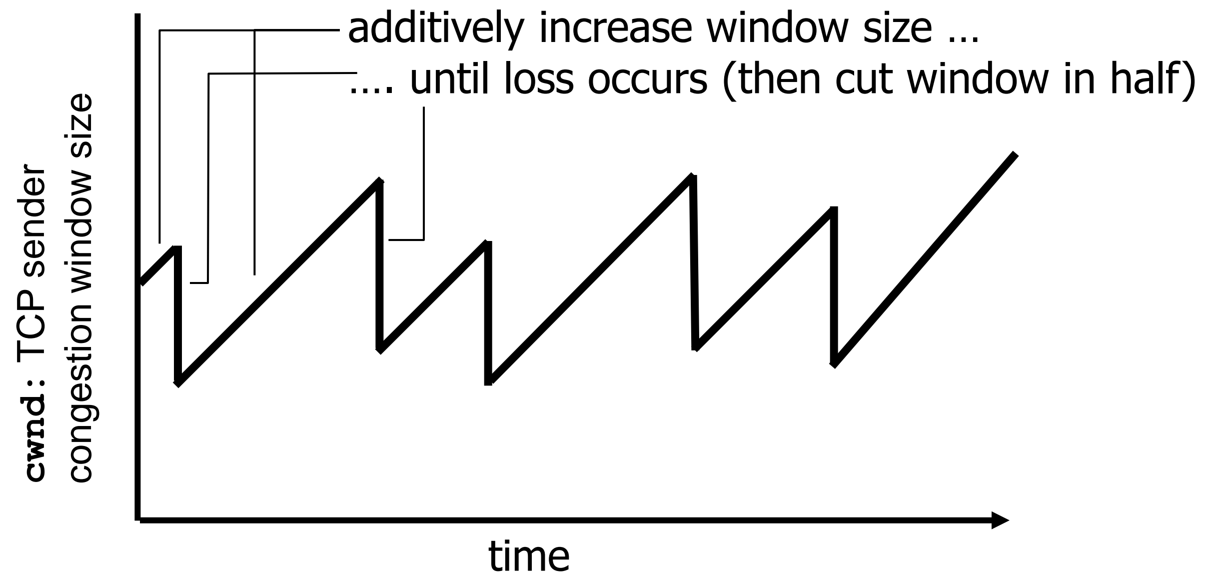
- **3 Key Challenges**
  - Determining the available capacity in the first place
  - Adjusting to changes in the available capacity
  - Sharing capacity between flows
- **Idea**
  - Each source determines network capacity for itself
  - Rate is determined by window size
  - Uses implicit feedback (drops, delay)
  - ACKs pace transmission (self-clocking)

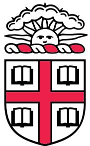


TCP congestion control: additive increase multiplicative decrease

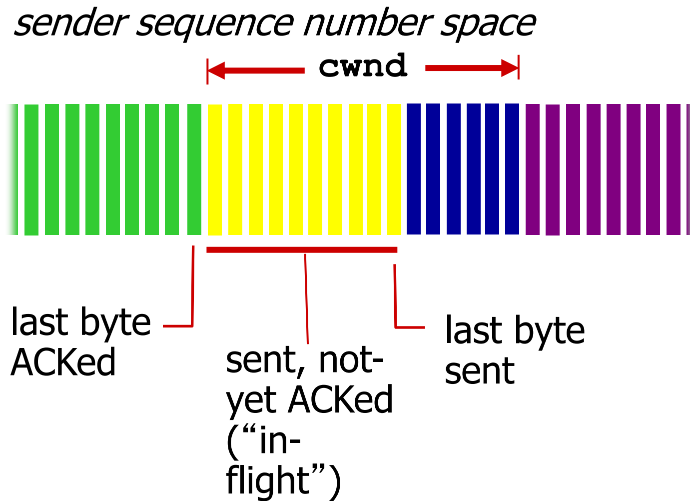
- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth





TCP Congestion Control: details



- **sender limits transmission:**

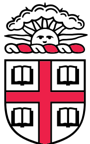
`LastByteSent-` `LastByteAcked` ≤ `cwnd`

- **`cwnd` is dynamic, function of perceived network congestion**

TCP sending rate:

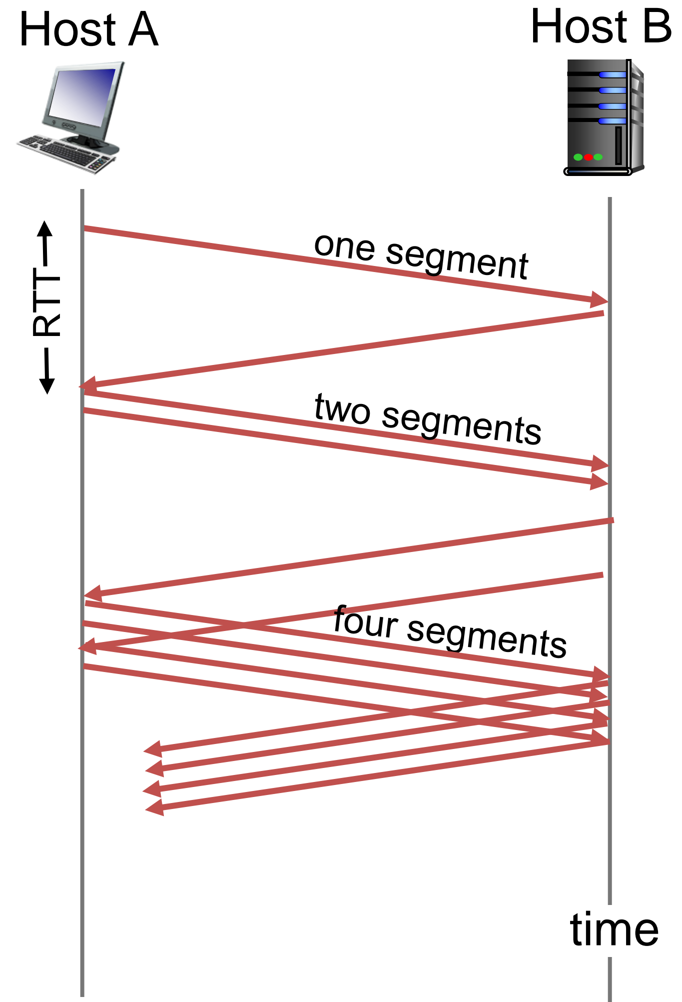
- ***roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes**

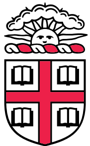
$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$



TCP Slow Start

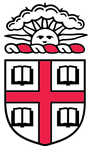
- **when connection begins, increase rate exponentially until first loss event:**
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- ***summary:*** **initial rate is slow but ramps up exponentially fast**





TCP: detecting, reacting to loss

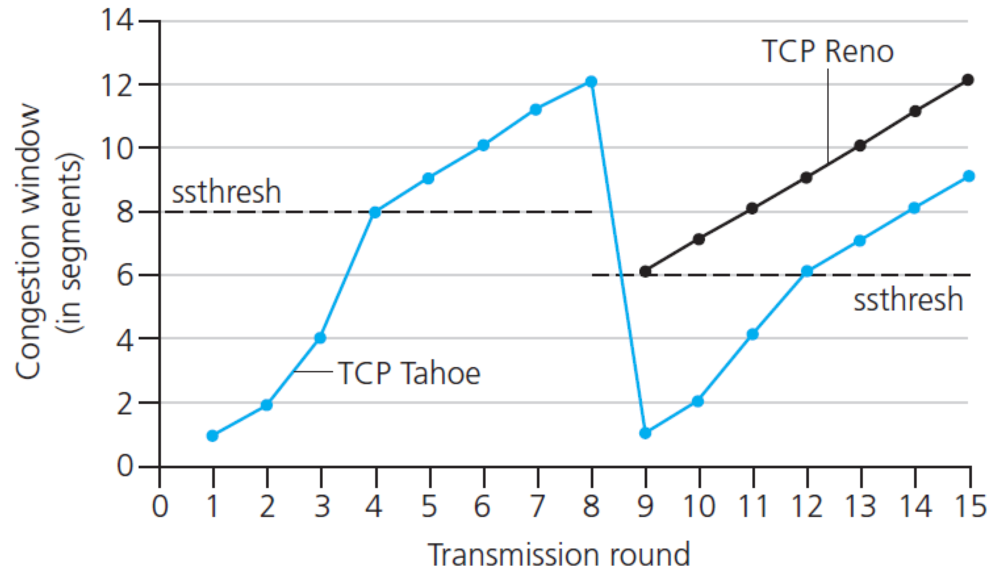
- **loss indicated by timeout:**
  - `cwnd` set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- **loss indicated by 3 duplicate ACKs: TCP RENO**
  - dup ACKs indicate network capable of delivering some segments
  - `cwnd` is cut in half window then grows linearly
- **TCP Tahoe always sets `cwnd` to 1 (timeout or 3 duplicate acks)**



TCP: switching from slow start to CA

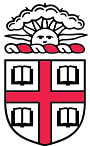
Q: when should the exponential increase switch to linear?

A: when `cwnd` gets to 1/2 of its value before timeout.



Implementation:

- variable `ssthresh`
- on loss event, `ssthresh` is set to 1/2 of `cwnd` just before loss event



Summary: TCP Congestion Control

slow start

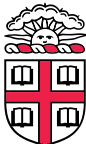
- Λ / cwnd = 1 MSS, ssthresh = 64 KB, dupACKcount = 0 (initial transition into slow start)
- duplicate ACK / dupACKcount++
- new ACK / cwnd = cwnd+MSS, dupACKcount = 0, *transmit new segment(s), as allowed* (New ACK!)
- timeout / ssthresh = cwnd/2, cwnd = 1 MSS, dupACKcount = 0, *retransmit missing segment*
- cwnd ≥ ssthresh / Λ (to congestion avoidance)
- dupACKcount == 3 / ssthresh= cwnd/2, cwnd = ssthresh + 3, *retransmit missing segment* (to fast recovery)

congestion avoidance

- new ACK / cwnd = cwnd + MSS • (MSS/cwnd), dupACKcount = 0, *transmit new segment(s), as allowed* (New ACK!)
- duplicate ACK / dupACKcount++
- timeout / ssthresh = cwnd/2, cwnd = 1 MSS, dupACKcount = 0, *retransmit missing segment* (to slow start)
- dupACKcount == 3 / ssthresh= cwnd/2, cwnd = ssthresh + 3, *retransmit missing segment* (to fast recovery)

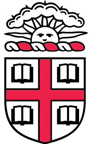
fast recovery

- duplicate ACK / cwnd = cwnd + MSS, *transmit new segment(s), as allowed*
- New ACK / cwnd = ssthresh, dupACKcount = 0 (New ACK!) (to congestion avoidance)
- timeout / ssthresh = cwnd/2, cwnd = 1, dupACKcount = 0, *retransmit missing segment* (to slow start)



3 Challenges Revisited

- **Determining the available capacity in the first place**
  - Exponential increase in congestion window
- **Adjusting to changes in the available capacity**
  - Slow probing, AIMD
- **Sharing capacity between flows**
  - AIMD
- **Detecting Congestion**
  - Timeout based on RTT
  - Triple duplicate acknowledgments
- **Fast retransmit/Fast recovery**
  - Reduces slow starts, timeouts



Next Class

- More Congestion Control fun
- Cheating on TCP
- TCP on extreme conditions
- TCP Friendliness
- TCP Future