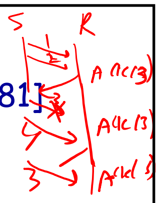
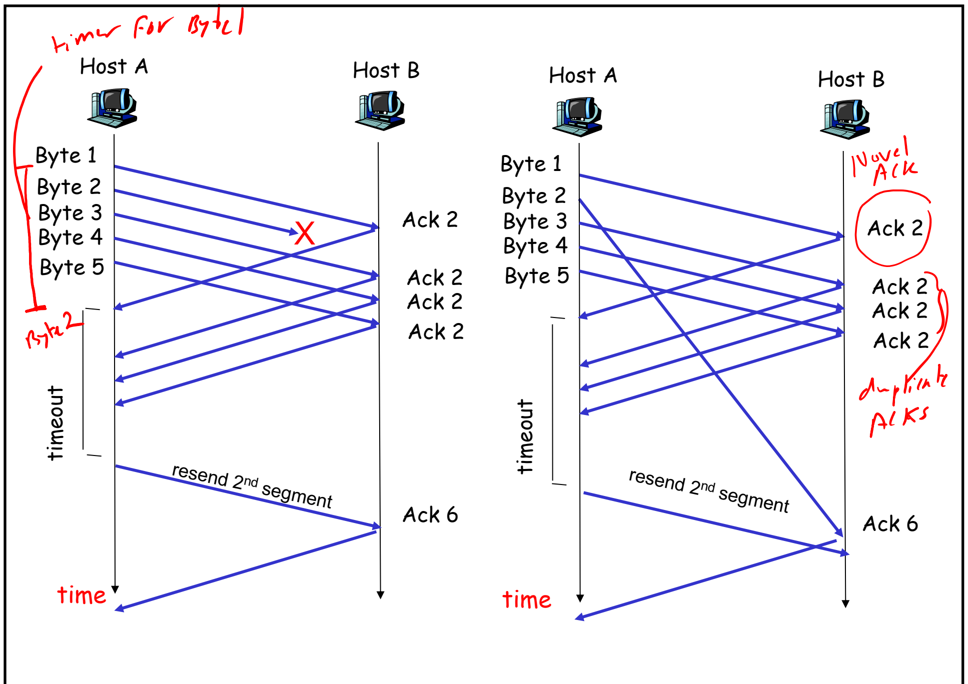
Lecture 17

- Sections 3.5.4, 3.5.5, 3.5.6, 3.6.1 and 3.6.2
  - Connection-oriented transport: TCP
    - reliable data transfer
    - flow control
    - TCP connection management
  - Principles of congestion control
    - causes and cost of congestion

TCP ACK generation [RFC 1122, RFC 2581]



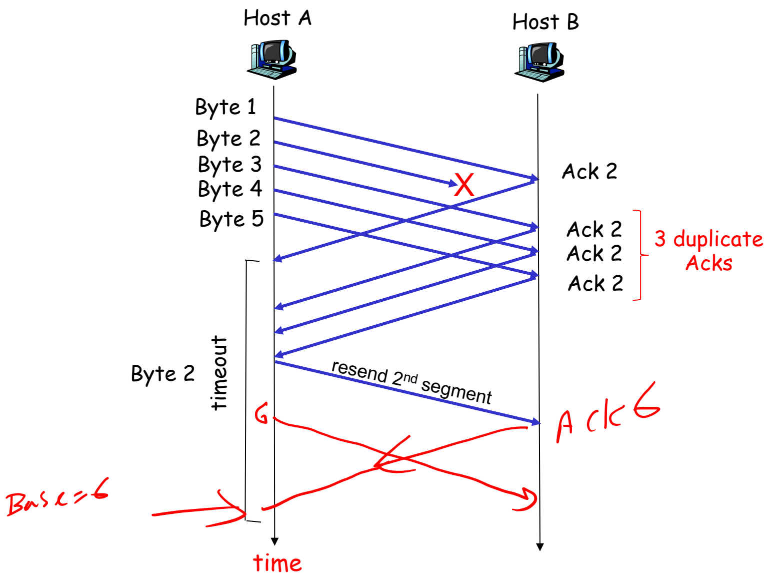
Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately 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 for last received byte in sequence, provided that segment starts at lower end of gap



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.

- if sender receives 3 duplicate ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires



Fast Retransmit: Resending a segment after triple duplicate ACK

Fast retransmit algorithm @sender:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) { ← Novel Ack
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
          }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
               }
```

a duplicate ACK for already ACKed segment

fast retransmit

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

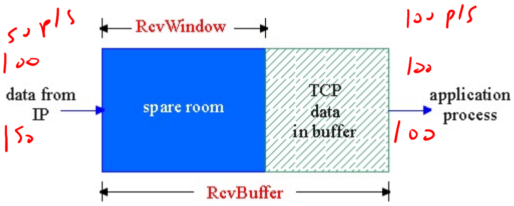
3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- TCP timeout and round trip time
- reliable data transfer
- flow control

TCP Flow Control

- receive side of TCP connection has a receive buffer:



- app process may be slow at reading from buffer

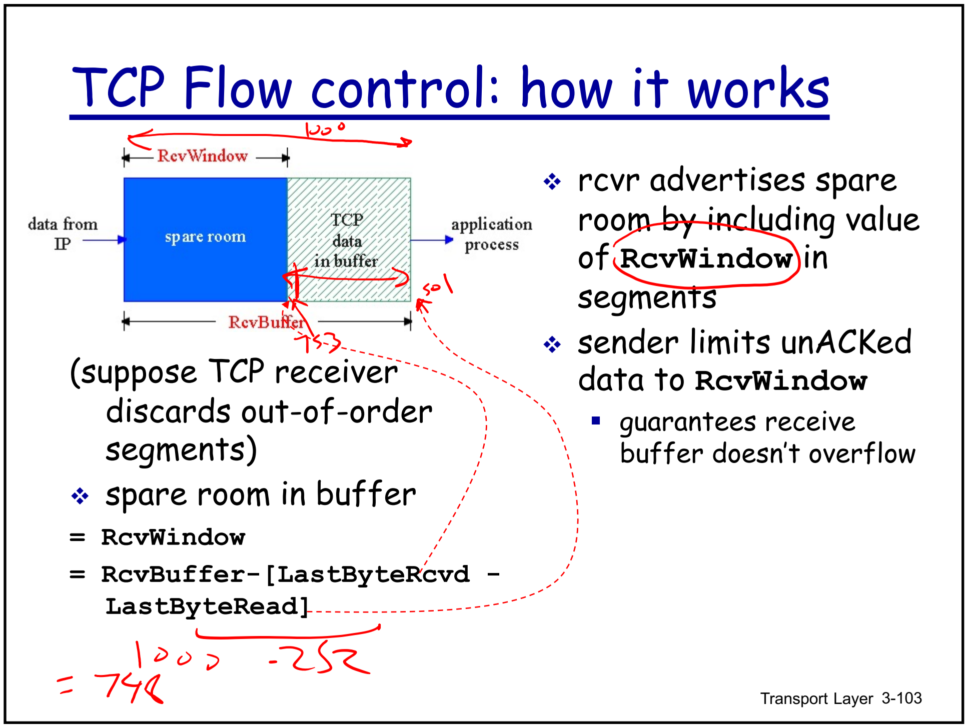
flow problem

receiver is not able to handle received packets fast enough -> buffer overflow

flow control

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

- speed-matching service: matching the send rate to the receiving app's drain rate



TCP Flow control: how it works

(suppose TCP receiver discards out-of-order segments)

- spare room in buffer

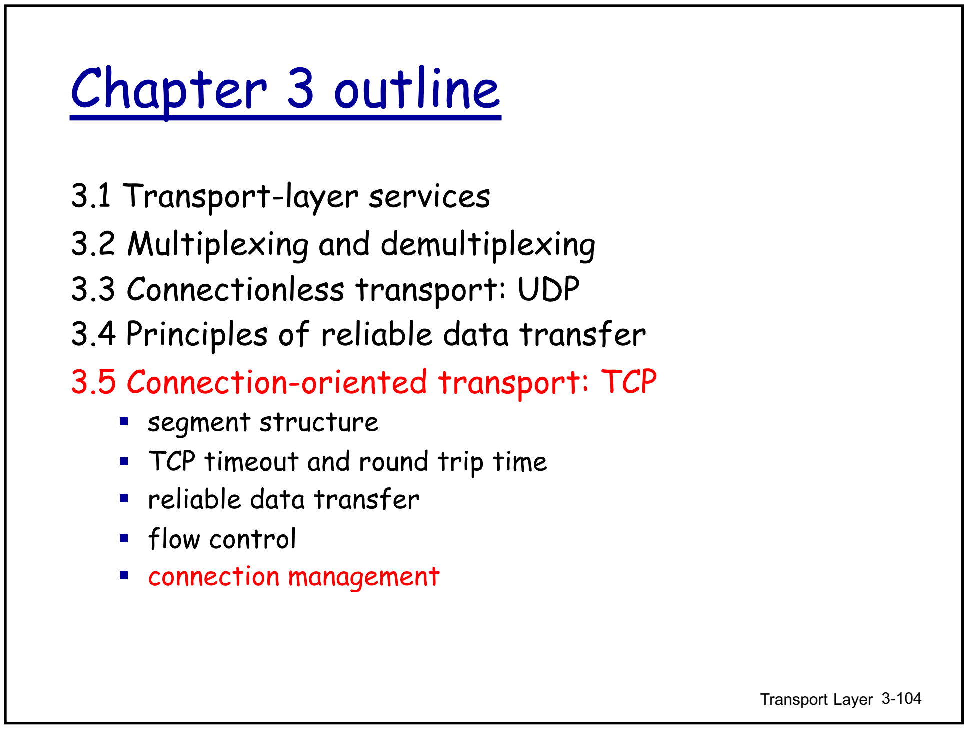
= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

1000 -252

= 748

- rcvr advertises spare room by including value of `RcvWindow` in segments
- sender limits unACKed data to `RcvWindow`
  - guarantees receive buffer doesn't overflow



Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- TCP timeout and round trip time
- reliable data transfer
- flow control
- connection management

TCP Connection Management

- TCP sender, receiver must establish "connection" before exchanging data segments
- Each side must initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *server: opens on a socket, and listens on it;*
- *client: opens a socket, and uses it to connect() to server;*
- *server:* contacted by client & accepts connection

May include rcvr window negotiations

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK (ACK & SYN bits set) segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (cont.)

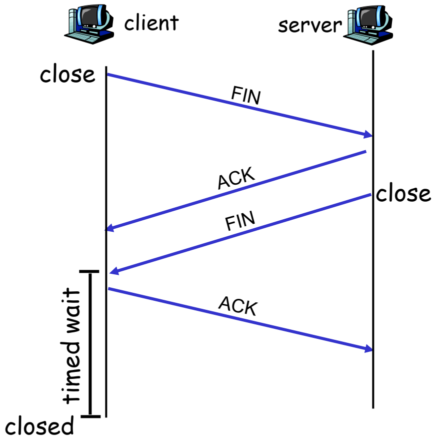
Closing a connection:

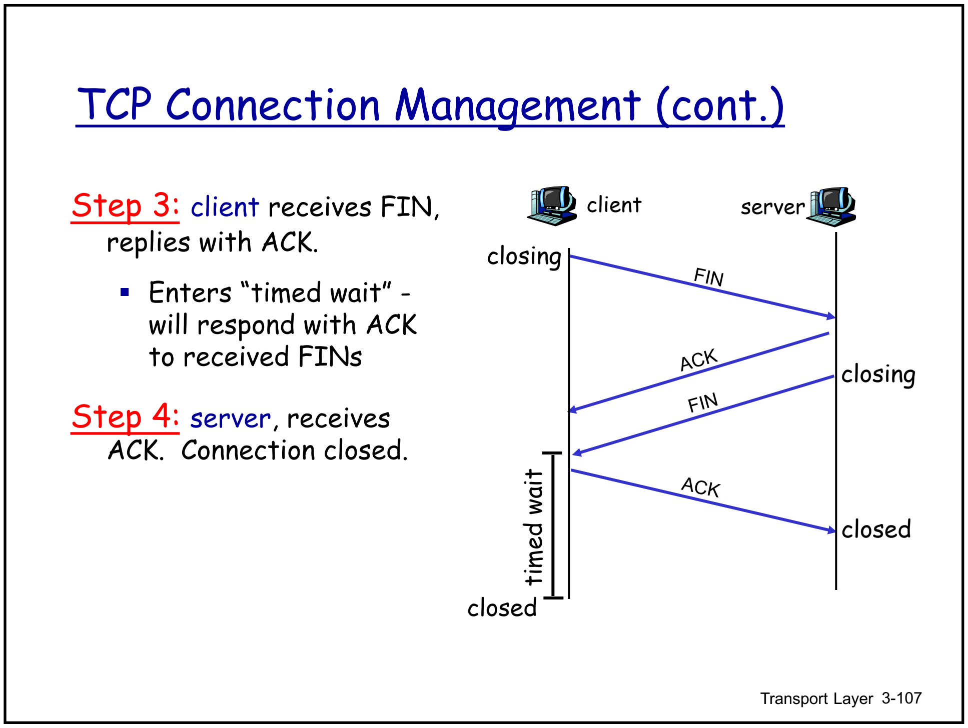
client closes socket:
`close();`

Four way handshake:

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



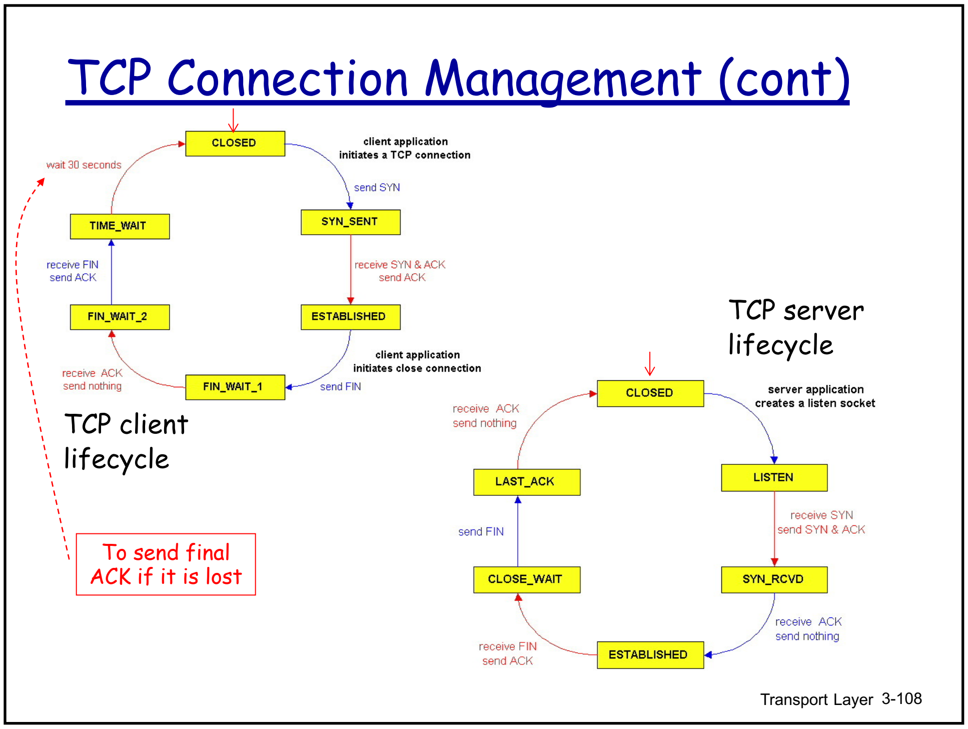


TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.



TCP Connection Management (cont)

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- TCP timeout and round trip time
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

Principles of Congestion Control

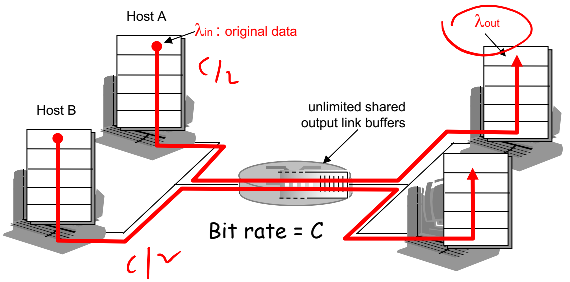
Hot spot
congestion

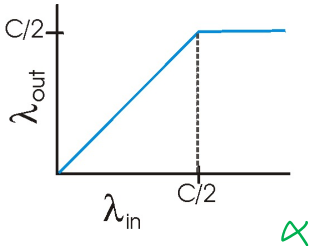
Congestion:

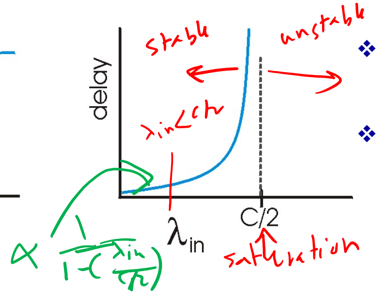
- formally: requesting more resources (bit rate, buffers, etc) than what the network has:
  - demand > available resources
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow problem!
- manifestations:
  - long delays (queueing in router buffers) -> approaching congestion
  - lost packets (buffer overflow at routers) -> congestion has occurred

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission


Host A
λ_{in} : original data
C/2
Host B
c/2
unlimited shared output link buffers
λ_{out}
Bit rate = C

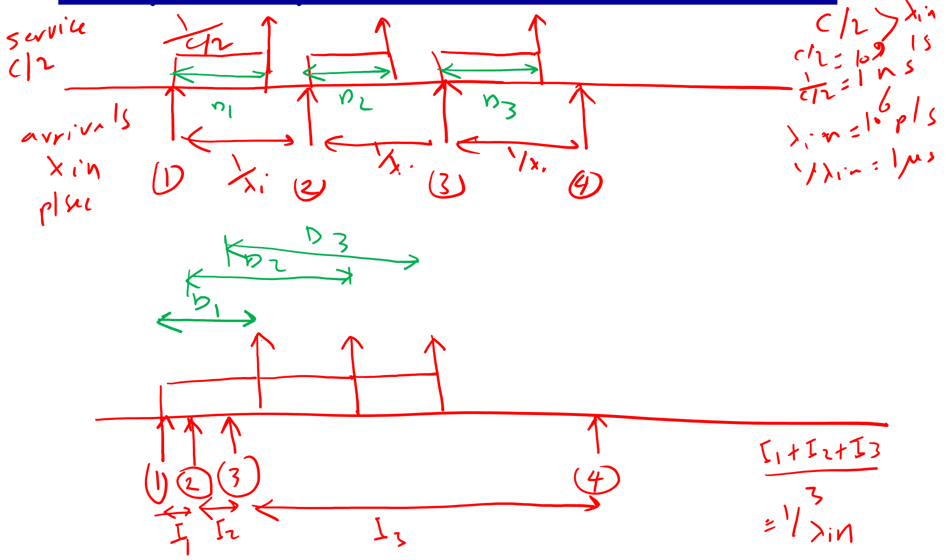




$\propto \frac{1}{1-\left(\frac{\lambda_{in}}{C/2}\right)}$

- large delays when congested
- maximum achievable throughput

Why delay increases with load?



$C/2 > \lambda_{in}$

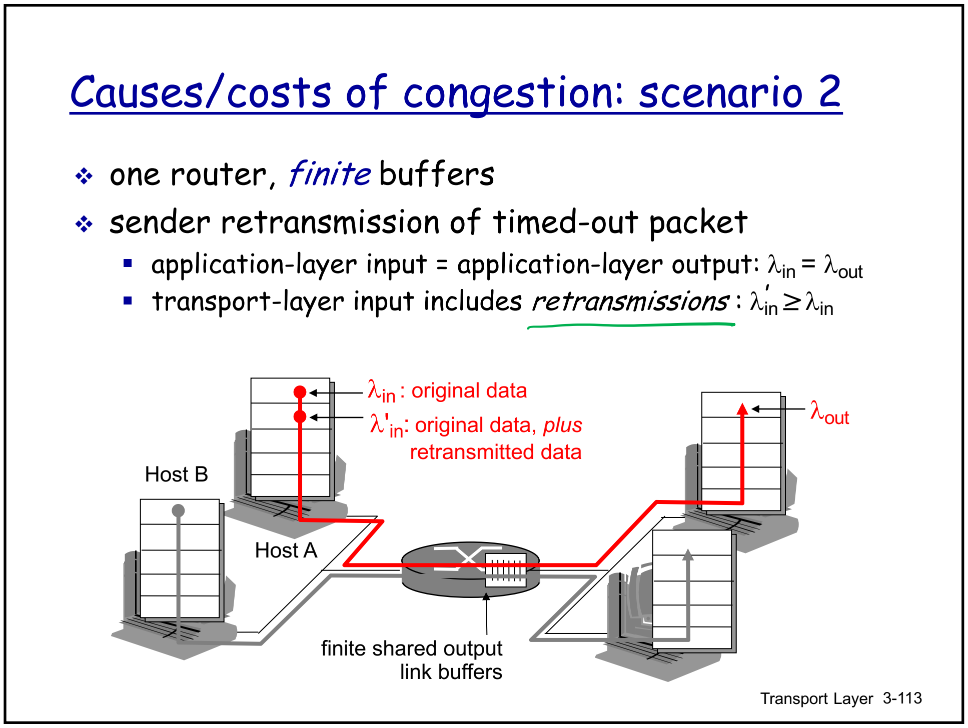
$C/2 = 10^9$ /s

$\frac{1}{C/2} = 1$ ns

$\lambda_{in} = 10^6$ p/s

$1/\lambda_{in} = 1\mu s$

$\frac{I_1+I_2+I_3}{3} = 1/\lambda_{in}$



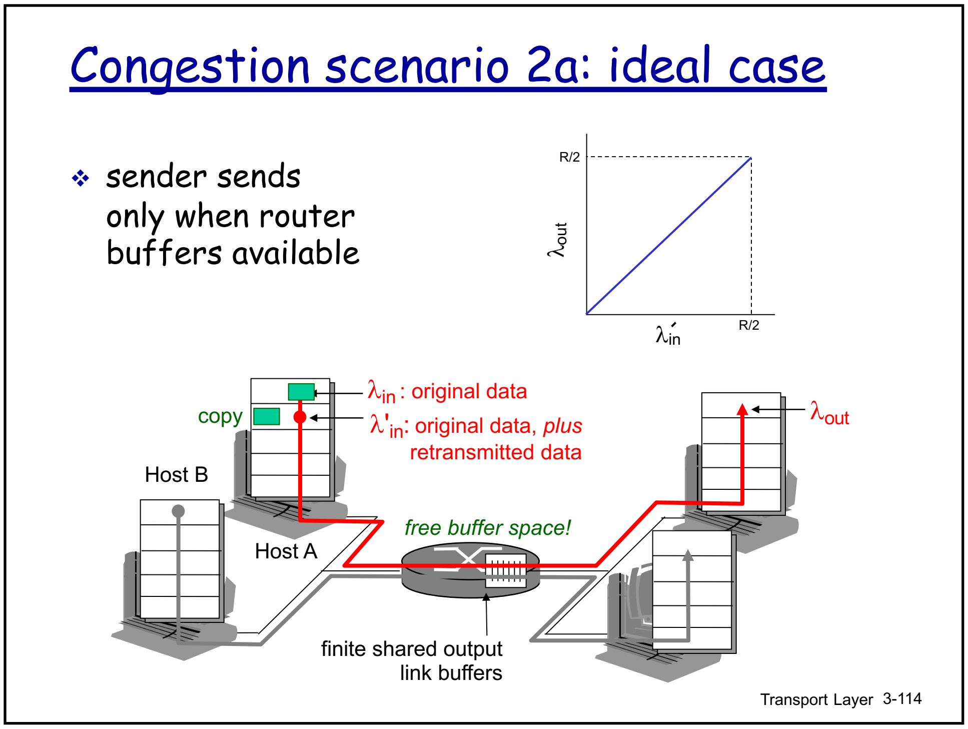
Causes/costs of congestion: scenario 2

- 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 B
Host A
finite shared output link buffers

Transport Layer 3-113

113



Congestion scenario 2a: ideal case

- sender sends only when router buffers available

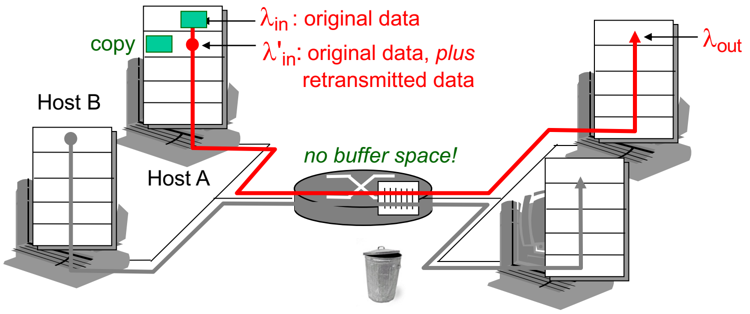
R/2
λ_{out}
λ'_{in}
R/2
copy
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
λ_{out}
Host B
Host A
free buffer space!
finite shared output link buffers

Transport Layer 3-114

114

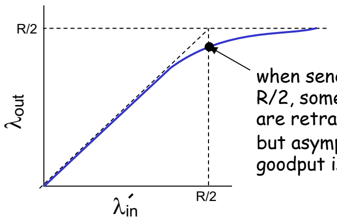
Congestion scenario 2b: *known* loss

- packets may get dropped at router due to full buffers
  - sometimes lost
- sender only resends if packet *known* to be lost (admittedly idealized)

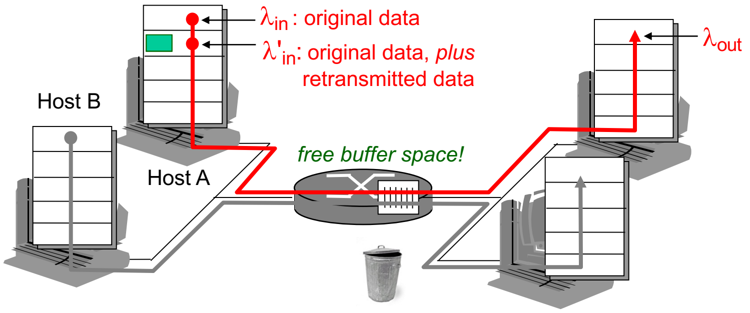

copy
λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
λ_{out}
Host B
Host A
no buffer space!

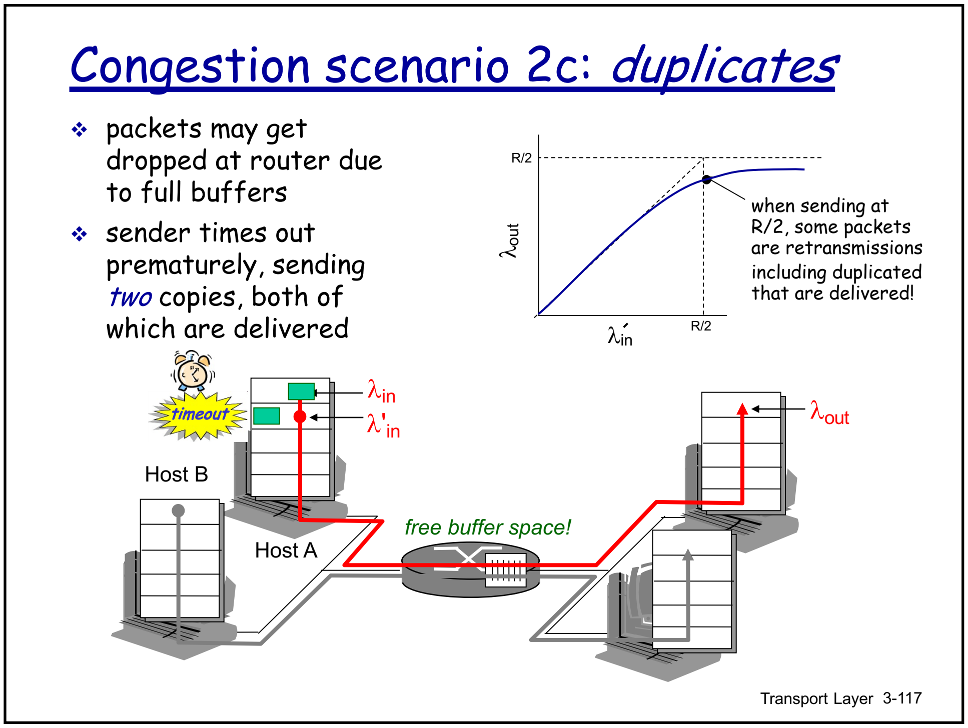
Congestion scenario 2b: *known* loss

- packets may get dropped at router due to full buffers
  - sometimes not lost
- sender only resends if packet *known* to be lost (admittedly idealized)



when sending at R/2, some packets are retransmissions but asymptotic goodput is still R/2


λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
λ_{out}
Host B
Host A
free buffer space!

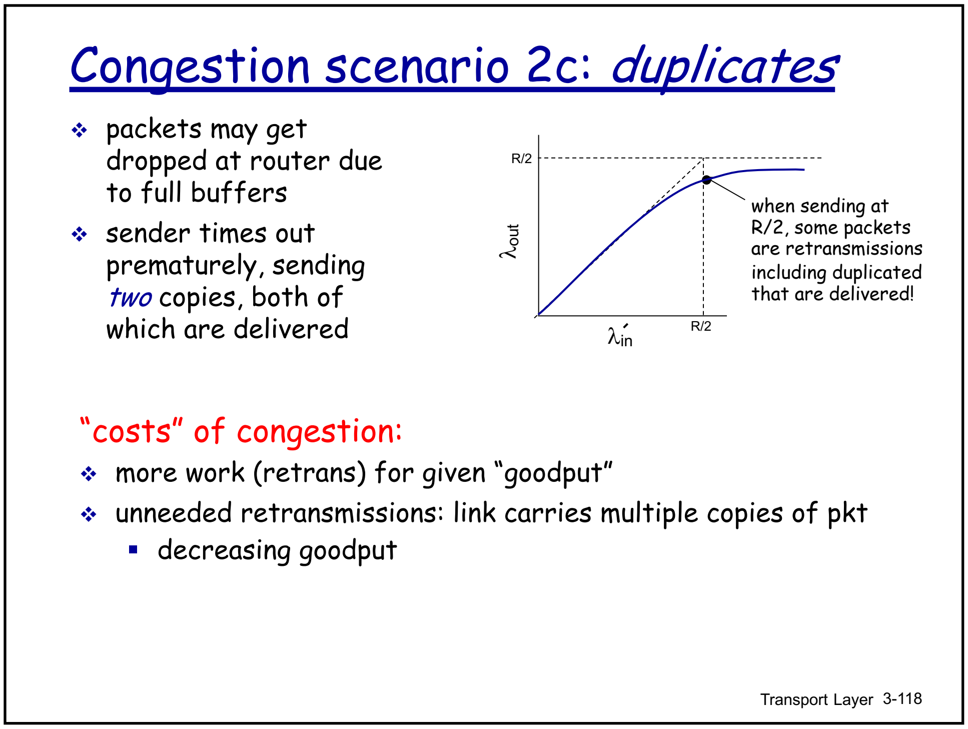


Congestion scenario 2c: *duplicates*

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

R/2
λout
λ'in
R/2
when sending at R/2, some packets are retransmissions including duplicated that are delivered!
timeout
λin
λ'in
Host B
Host A
free buffer space!
λout

117



Congestion scenario 2c: *duplicates*

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

"costs" of congestion:

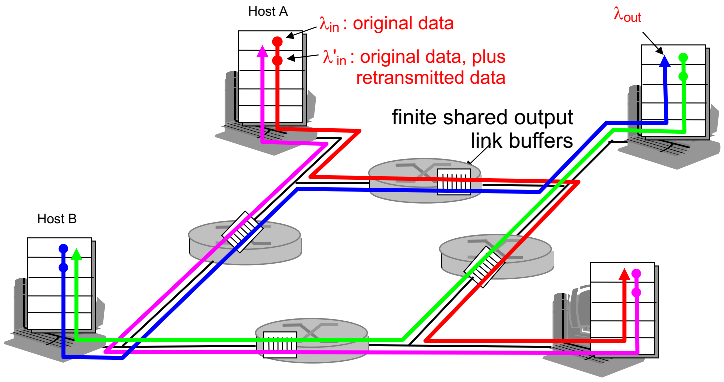
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

118

Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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



Host A

λ_{in} : original data

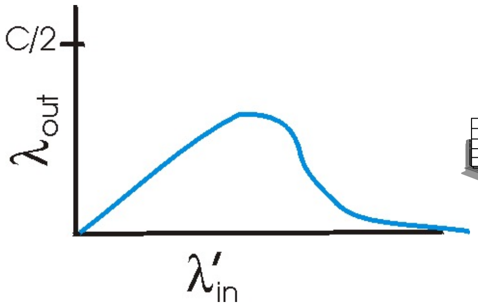
λ'_{in} : original data, plus retransmitted data

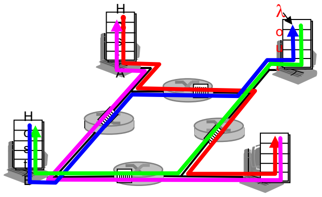
λ_{out}

finite shared output link buffers

Host B

Causes/costs of congestion: scenario 3





another "cost" of congestion:

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