Lecture 12: Transport Layer Congestion Control

COMP 332, Spring 2024

Victoria Manfredi

WESLEYAN UNIVERSITY

Acknowledgements: materials adapted from Computer Networking: A Top Down Approach 7th edition: ©1996-2016, J.F Kurose and K.W. Ross, All Rights Reserved as well as from slides by Abraham Matta at Boston University, and some material from Computer Networks by Tannenbaum and Wetherall.

Today

1. Announcements
   - homework 5 due Thursday at 11:59p
   - Midterm is Wed after break

2. Midterm overview

3. Flow control

4. Congestion causes and costs

5. TCP congestion control

Midterm overview

In class on Wednesday, March 27

- closed book, closed notes
- covers through whatever we get through today
  - But limited questions that I will be able to ask you about congestion control (since have not done on homework yet)
- will post practice exam

5 or 6 questions

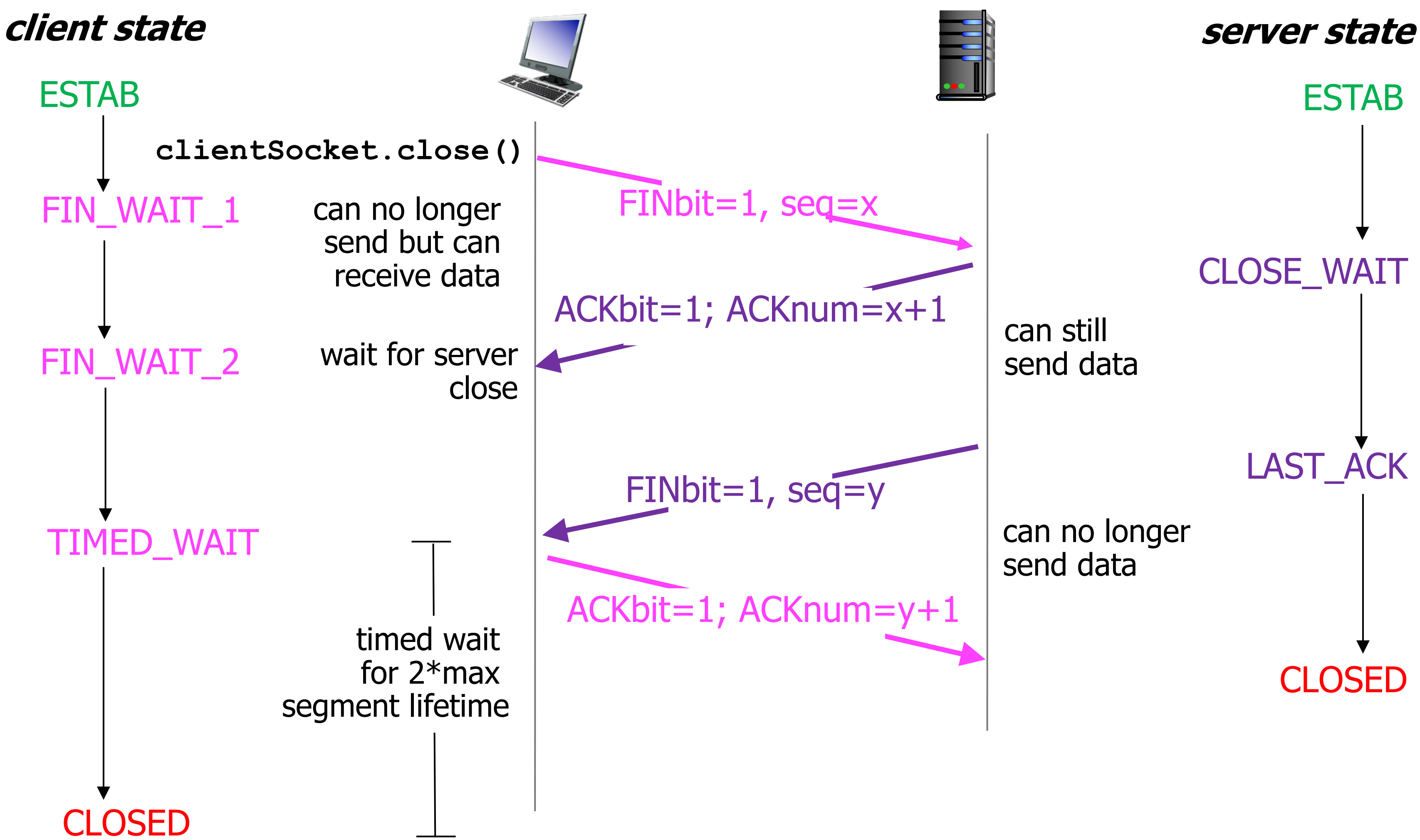
- Application layer questions: 6-8 in total, few sentences to answer
- Transport layer questions : 6-8 in total, few sentences to answer
- Deeper question on application layer protocol: likely HTTP
- Deeper question transport layer
- Small coding check?
- Something fun?

TCP
CONNECTION MANAGEMENT

TCP: politely closing a connection

Client, server each sends TCP segment with FIN bit = 1

– respond to received FIN with ACK (ACK can be combined with own FIN)



FIN segment in Wireshark

```
  241 4.063493     vmanfredismbp2.wireless.we…  40.97.120.226                                   54  55017 → 443 [FIN
▶ Frame 241: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
▶ Ethernet II, Src: 78:4f:43:73:43:26 (78:4f:43:73:43:26), Dst: 129.133.176.1 (3c:8a:b0:1e:18:01)
▶ Internet Protocol Version 4, Src: vmanfredismbp2.wireless.wesleyan.edu (129.133.187.174), Dst: 40.97.120.226 (40.97.12
▼ Transmission Control Protocol, Src Port: 55017 (55017), Dst Port: 443 (443), Seq: 3771, Ack: 6504, Len: 0
    Source Port: 55017
    Destination Port: 443
    [Stream index: 5]
    [TCP Segment Len: 0]
    Sequence number: 3771    (relative sequence number)
    Acknowledgment number: 6504    (relative ack number)
    Header Length: 20 bytes
  ▼ Flags: 0x011 (FIN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
    ▶ .... .... ...1 = Fin: Set
      [TCP Flags: *******A***F]
    Window size value: 8192
    [Calculated window size: 262144]
    [Window size scaling factor: 32]
  ▶ Checksum: 0xe59d [validation disabled]

0000  3c 8a b0 1e 18 01 78 4f  43 73 43 26 08 00 45 00   <.....xO CsC&..E.
0010  00 28 76 59 40 00 40 06  e5 ff 81 85 bb ae 28 61   .(vY@.@. ......(a
0020  78 e2 d6 e9 01 bb dd 11  e8 4a b0 93 7d 29 50 11   x....... .J..})P.
0030  20 00 e5 9d 00 00                                   .....
```

TCP
FLOW CONTROL

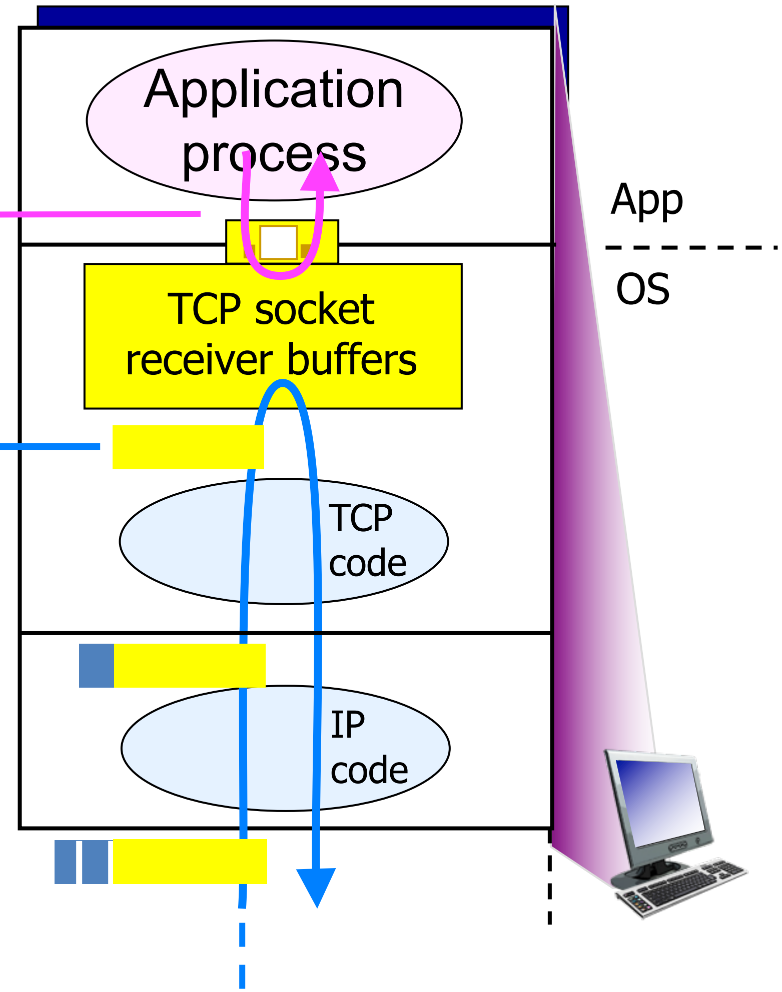
What if sender overwhelms receiver?

Problem

Application may remove data from TCP socket buffers ….

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

Receiver protocol stack



from sender

TCP flow control

Receiver provides feedback to sender

- so sender doesn't overflow receiver's buffer
- sender and receiver each maintain window

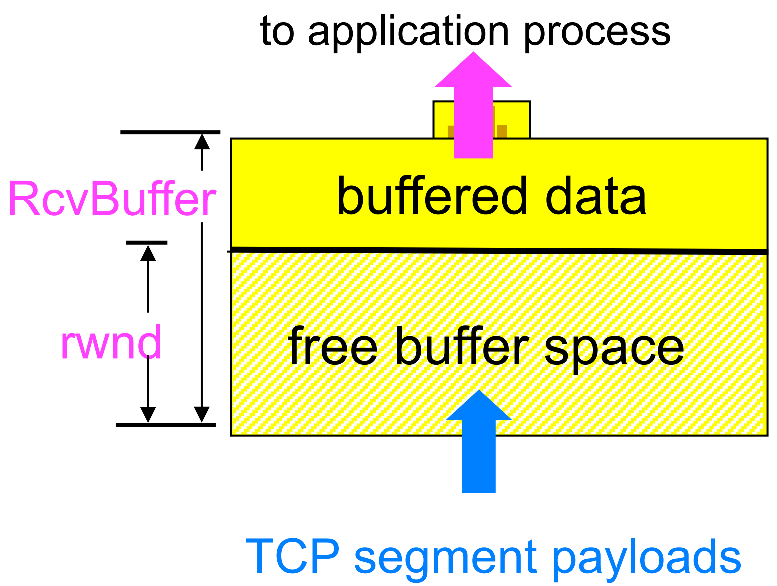
Receiver

- rwnd: free space in RcvBuffer
- puts rwnd in TCP header of receiver-to-sender segments

Sender

- limits unacked data to rwnd
- ensures RcvBuffer will not overflow

Receiver-side buffering

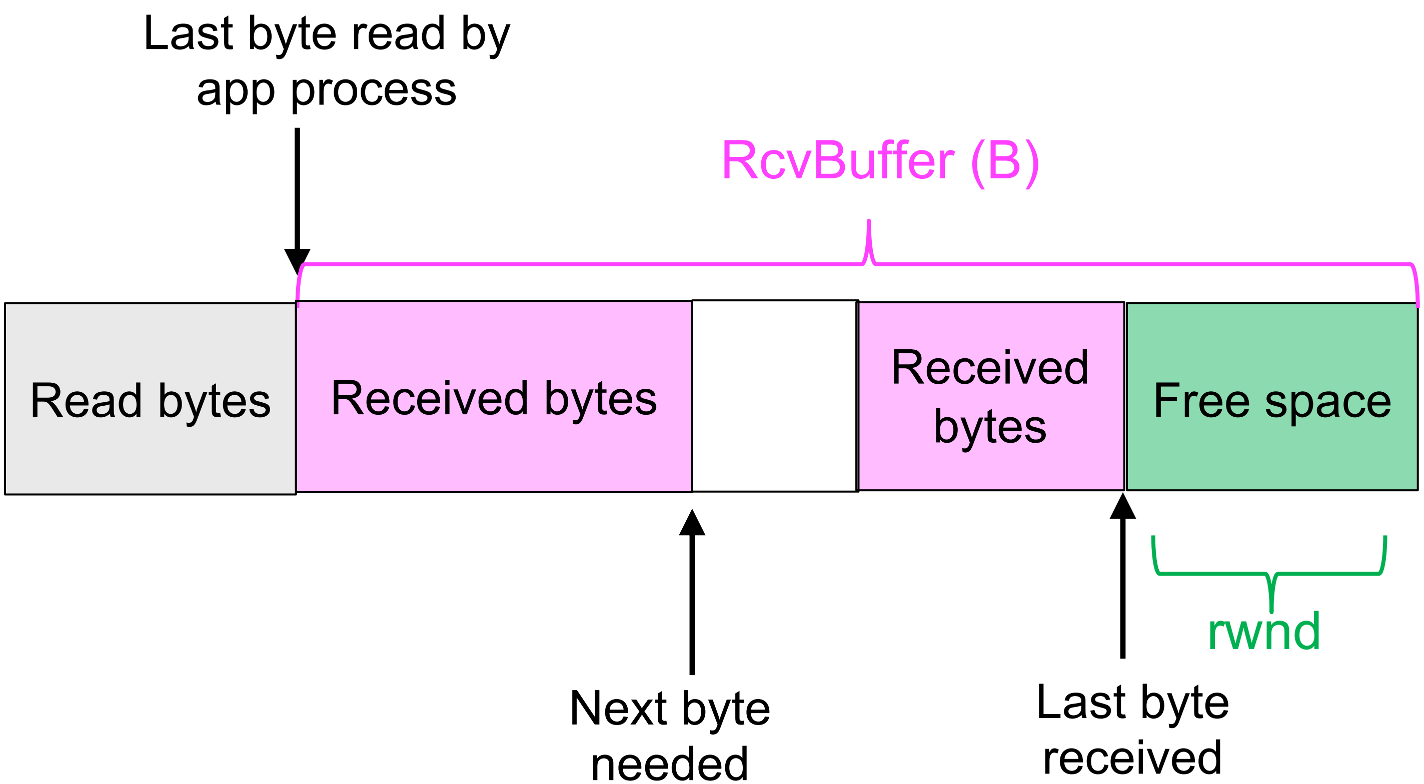


Receive window (rwnd)

```
▼ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 52232 (52232), Seq: 0, Ack: 1,
    Source Port: 443
    Destination Port: 52232
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 1    (relative ack number)
    Header Length: 32 bytes
  ▼ Flags: 0x012 (SYN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ▶ .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: *******A**S*]
    Window size value: 8190
    [Calculated window size: 8190]
```

Receiver use of receive window (rwnd)

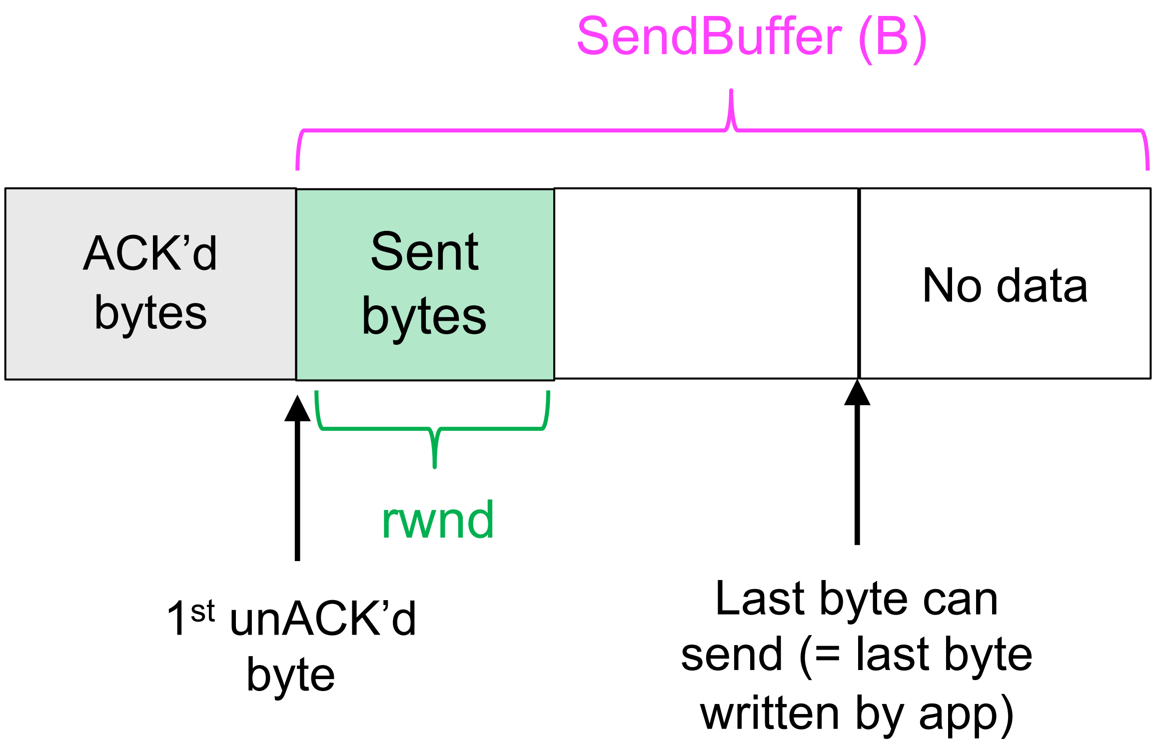
Keeps track of available space in its RcvBuffer



rwnd = B – (last byte received – last byte read)

Sender use of receive window (rwnd)

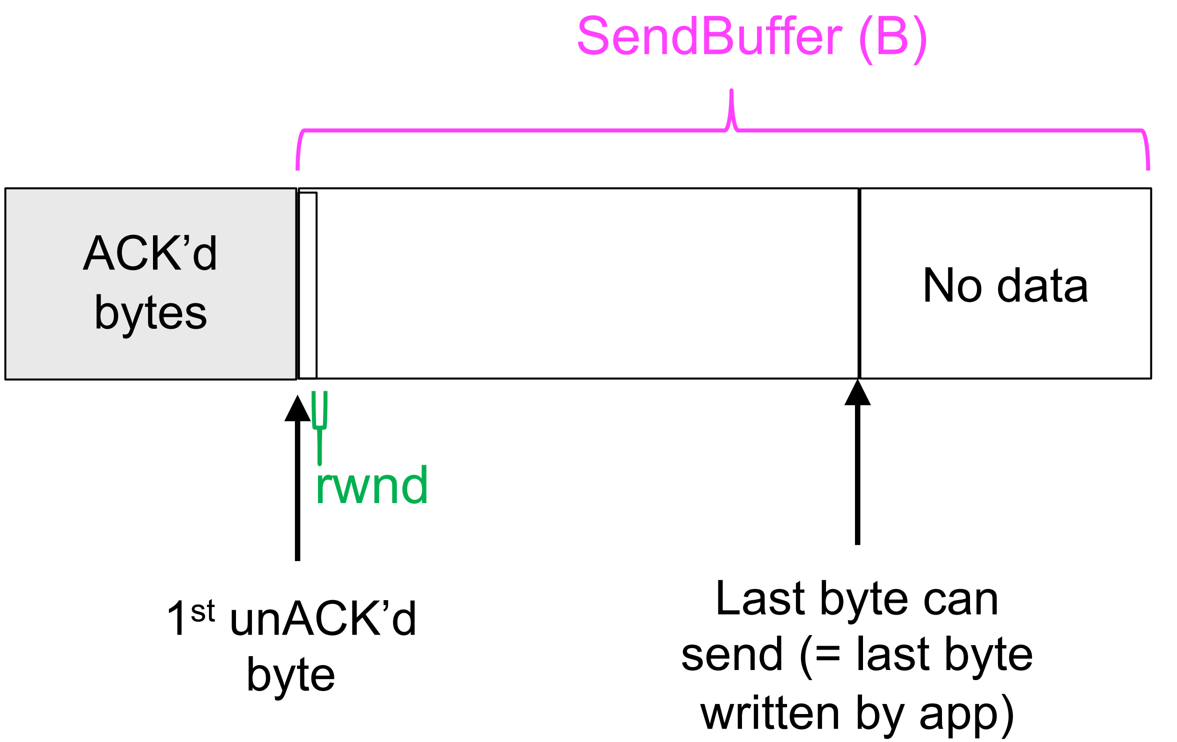
Limits # of in-flight segments of sender



Sending rate limited to: rwnd bytes/RTT seconds

Sender use of receive window (rwnd)

Problem: if rwnd = 0, what happens?



No ACKs sent: receiver has no way to let sender know rwnd increased

Solution: send segments with 1 byte of data, which receiver ACKs

Congestion

CAUSES AND COSTS

What if sender overwhelms network?

Receive buffer is not only resource limitation

- every packet travels through path of routers
- routers may be congested, have long queues …

Causes of network congestion

- many senders compete for network resources
- senders lack knowledge
  - amount of resources available (bandwidth)
  - # of other senders competing

Costs of network congestion

As queues in bottleneck link fill up: large packet delays, dropped packets

Bad feedback loop!

As timeouts expire at sender due to delays/drops: packets retransmitted

Problem

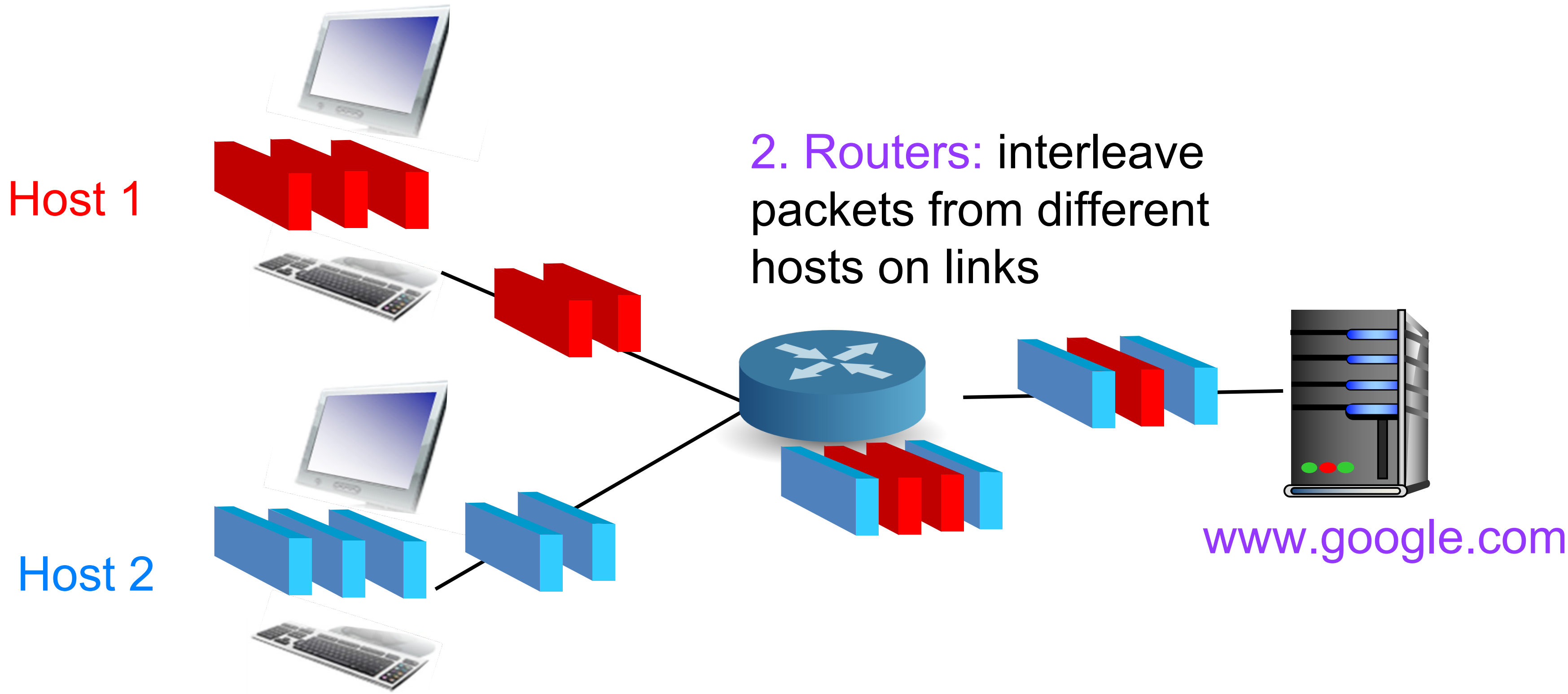
– retransmission treats symptoms but not underlying problem

Q: how to solve underlying problem of congestion?

– reduce sending rate … but what should sending rate be?
  - depends on available bandwidth
  - sender increases/decreases sending rate based on congestion level

Recall link and network resources are shared

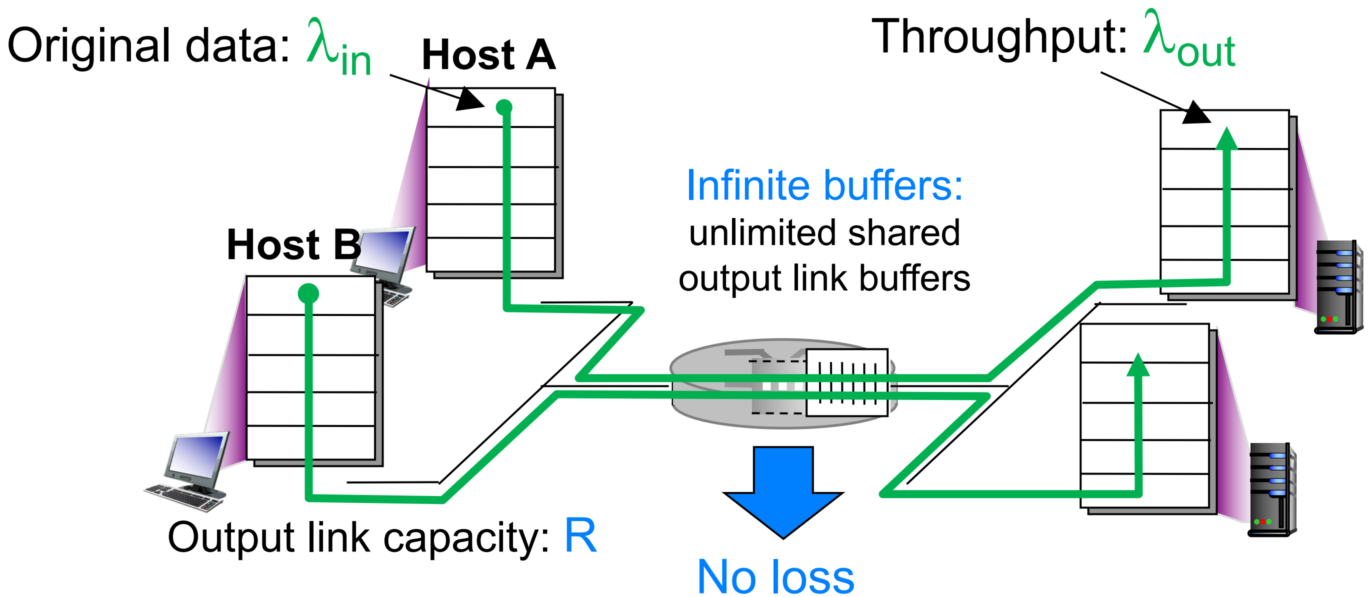
1. Hosts: divide data to send into fixed-length packets

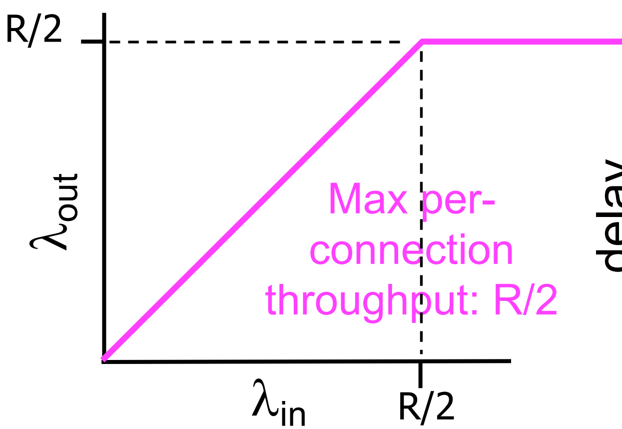


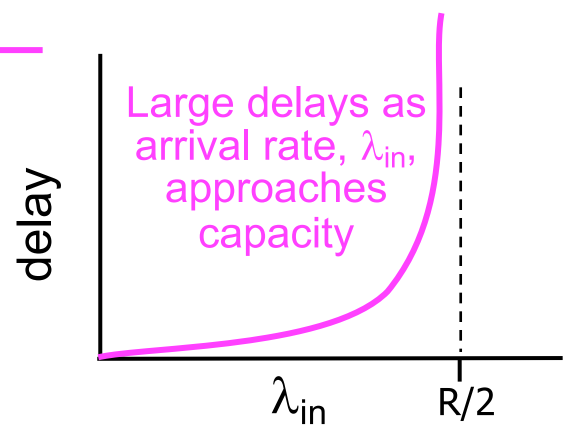
2. Routers: interleave packets from different hosts on links

Scenario 1

No retransmission, 2 senders, 2 receivers







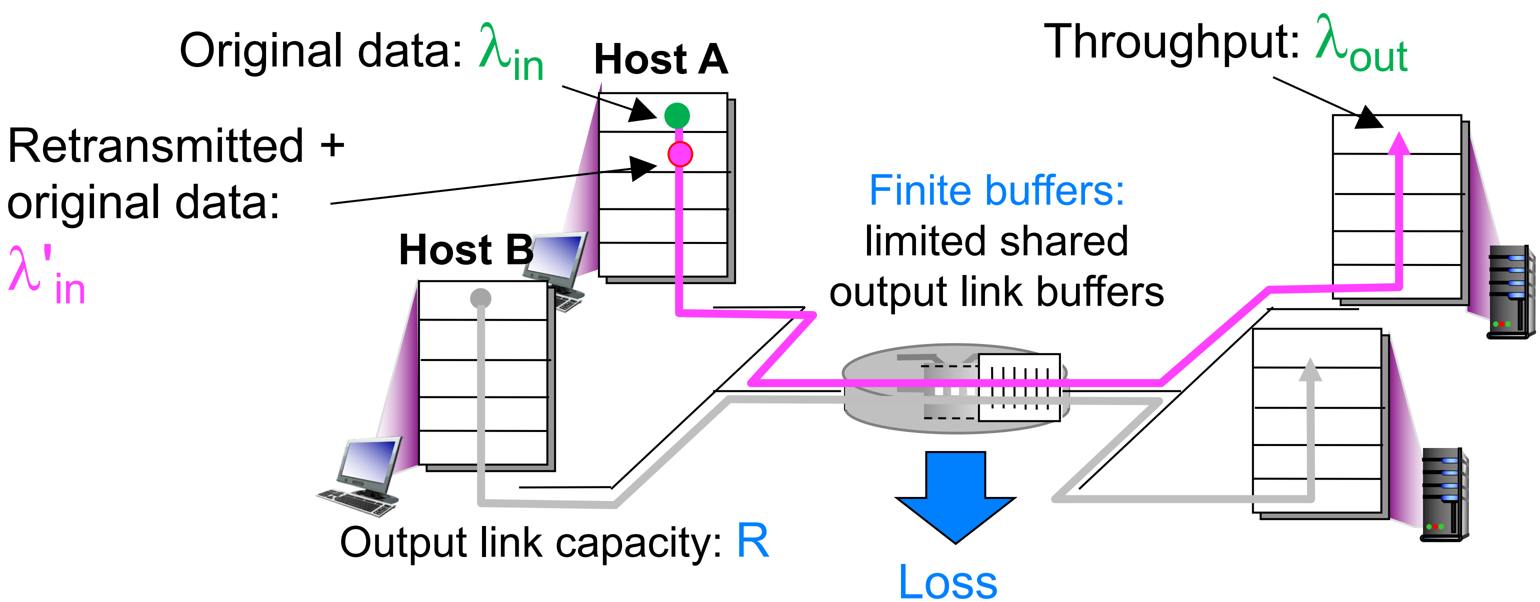
Even though high throughput when close to capacity, also high delay!

Q: Why R/2?

Scenario 2: retransmission

Sender retransmits timed-out packet

- $\lambda_{in} = \lambda_{out}$: app-layer input equals app-layer output
- $\lambda'_{in} \geq \lambda_{in}$: transport-layer input includes retransmissions

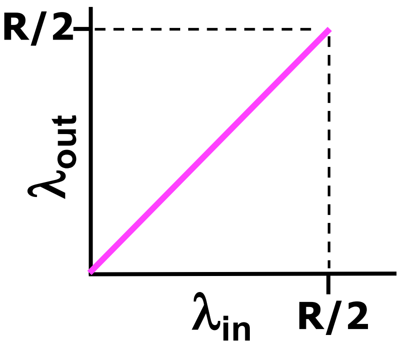


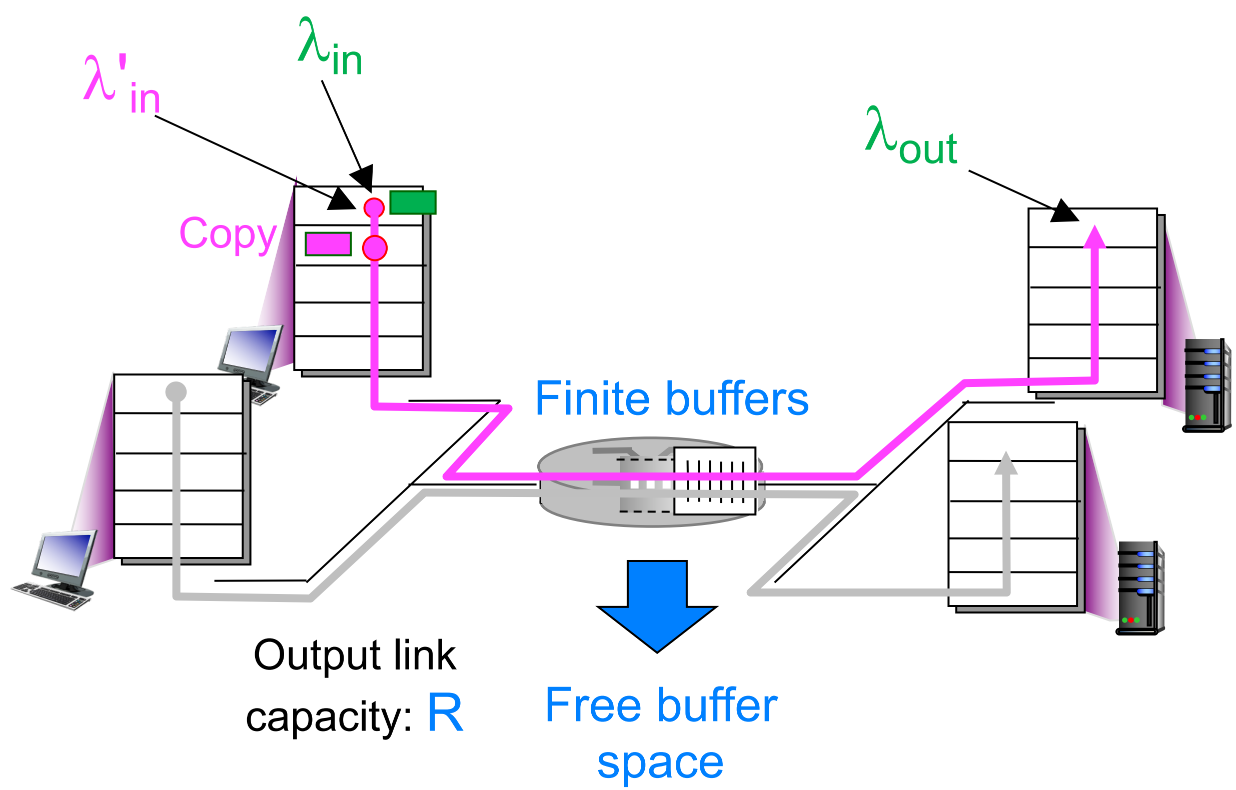
Performance depends on how retransmission performed…

Scenario 2: retransmission + perfect knowledge

Idealization: perfect knowledge

- sender sends only when router buffers available
- no loss occurs, so $\lambda'_{in} = \lambda_{in}$

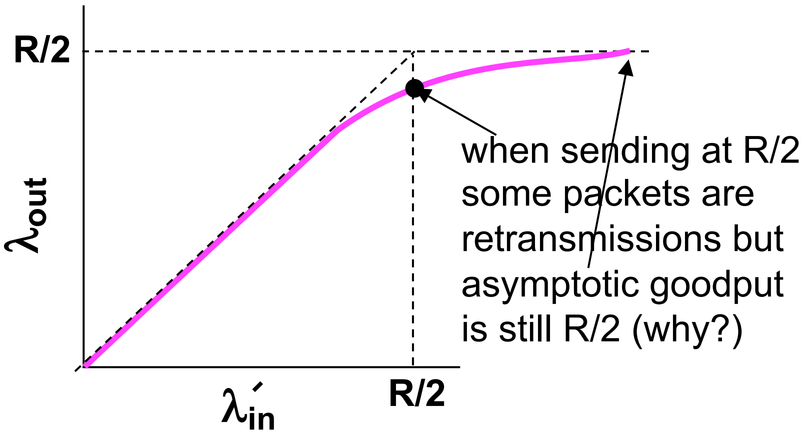


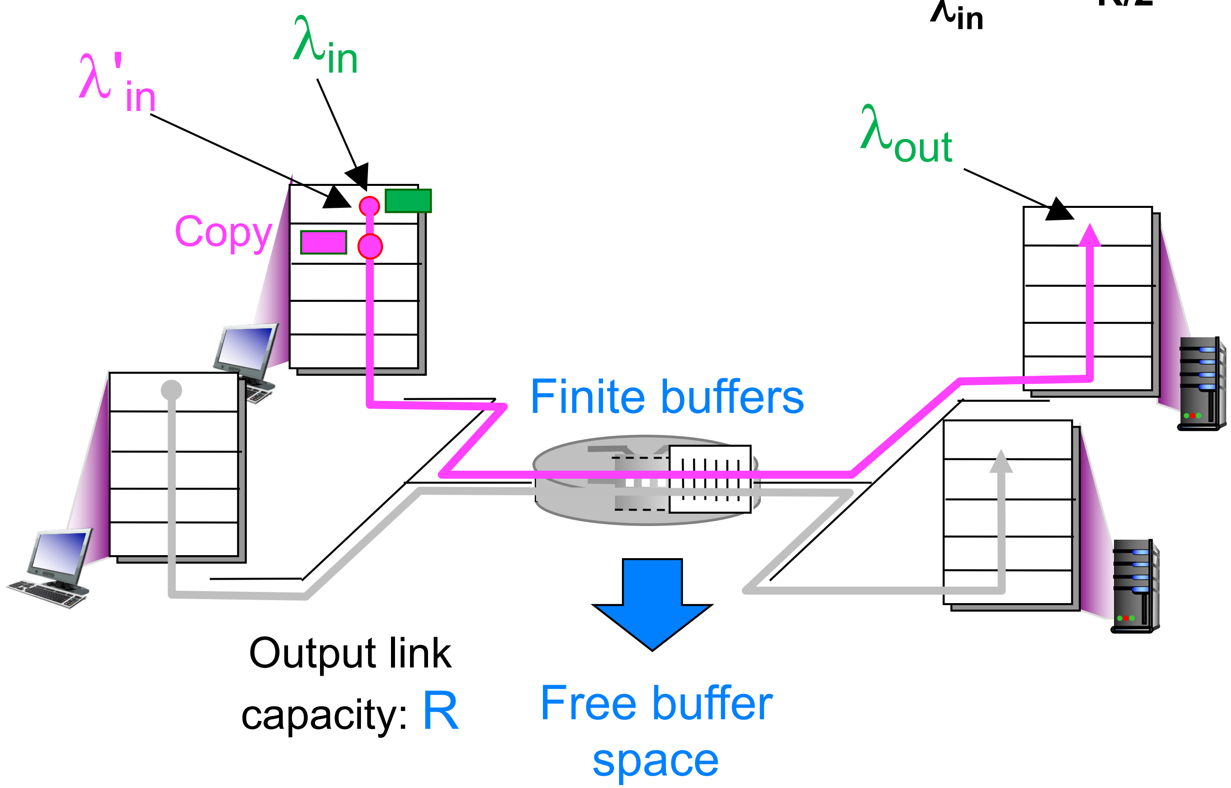


Scenario 2: retransmission only when lost

Idealization: known loss

- packets can be lost, dropped at router due to full buffers
- only resend packet known to be lost

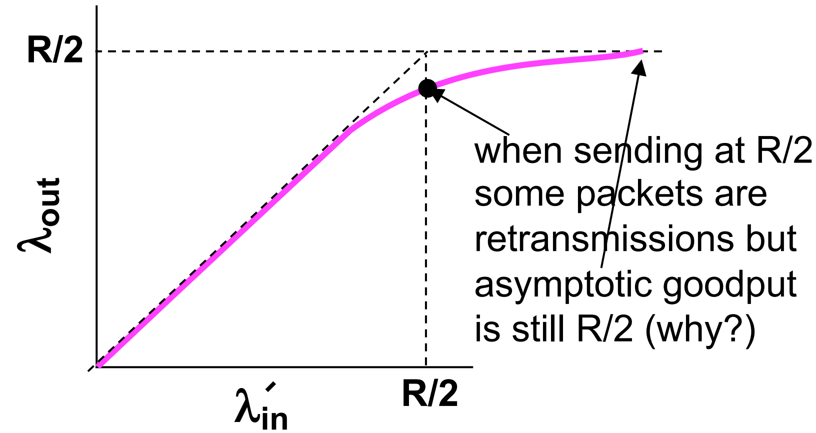


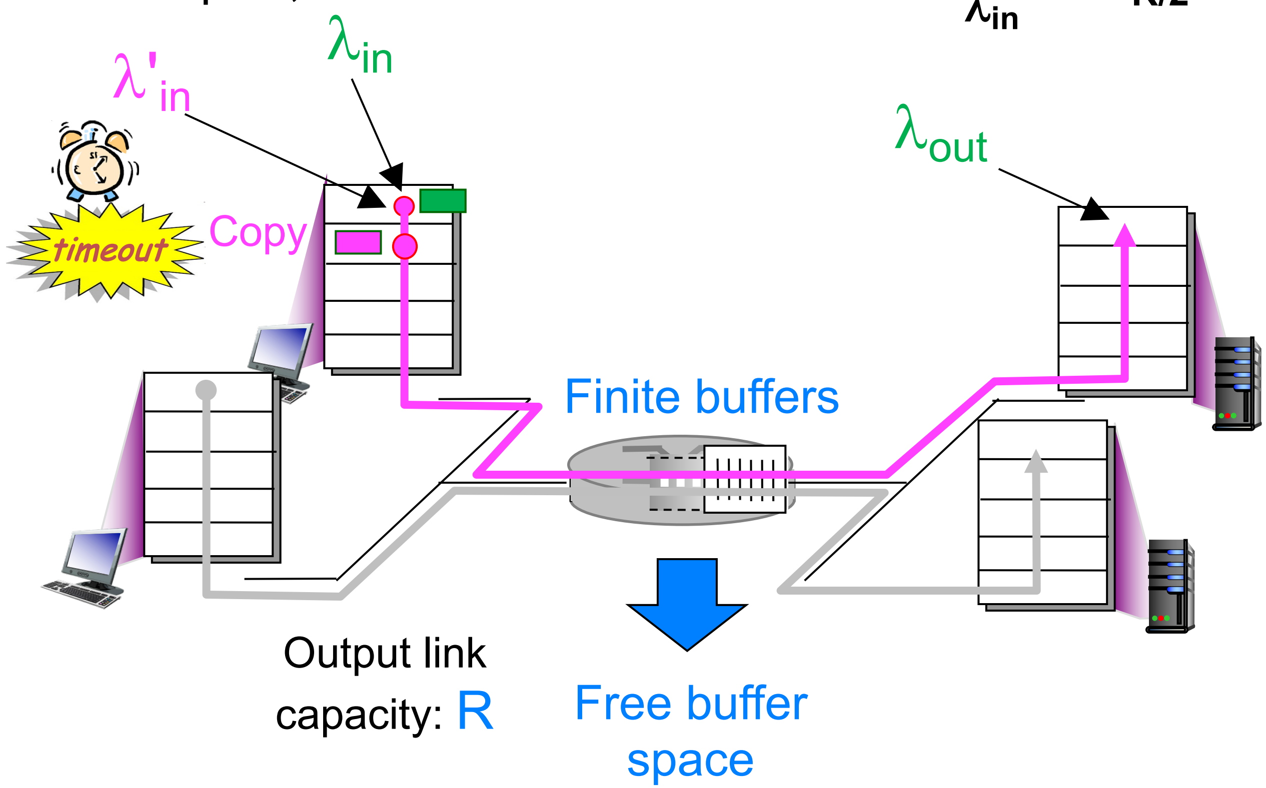


Scenario 2: retransmission causing duplicates

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely
  - sends 2 copies, both delivered





TCP
CONGESTION CONTROL

Goals of TCP congestion control

1. Discover available bandwidth
   - how much bandwidth can be used without causing congestion
     - will vary over time
   - estimate starting from no information

2. Correctly set sending rate
   - should not exceed available bandwidth

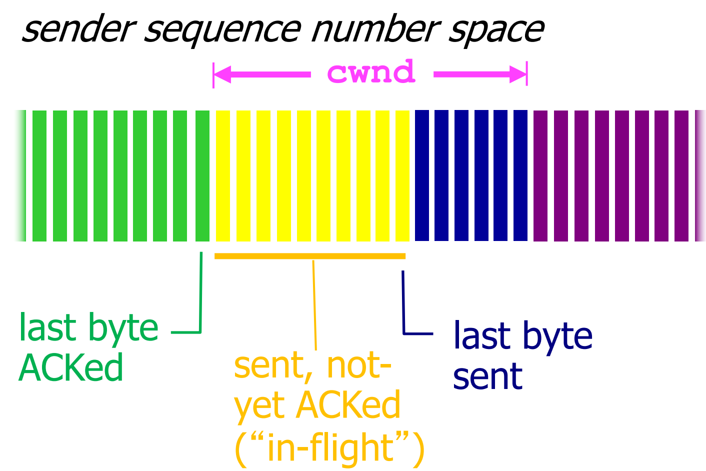
3. Fairness
   - no user gets all of the bandwidth

TCP Congestion Control

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
- send more bytes

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

Q: How does sender estimate cwnd?

To estimate cwnd

Detect congestion

- **delays**
  - large RTTs: too variable to be used in practice
- **duplicate ACKs**
  - isolated loss
- **timer expired**
  - multiple losses

Use to adjust cwnd,
affecting sending rate

How to intuitively adjust cwnd

- ACK received: increase cwnd
- loss detected: decrease cwnd

3 states in TCP finite state machine

Goal: send segments, adjust cwnd as needed

1. Slow start

– determine available bandwidth starting from no info

2. Congestion avoidance

– deal with fluctuations in bandwidth

3. Fast recovery

– quickly recover from isolated lost packets

We'll first look at different states, then full FSM

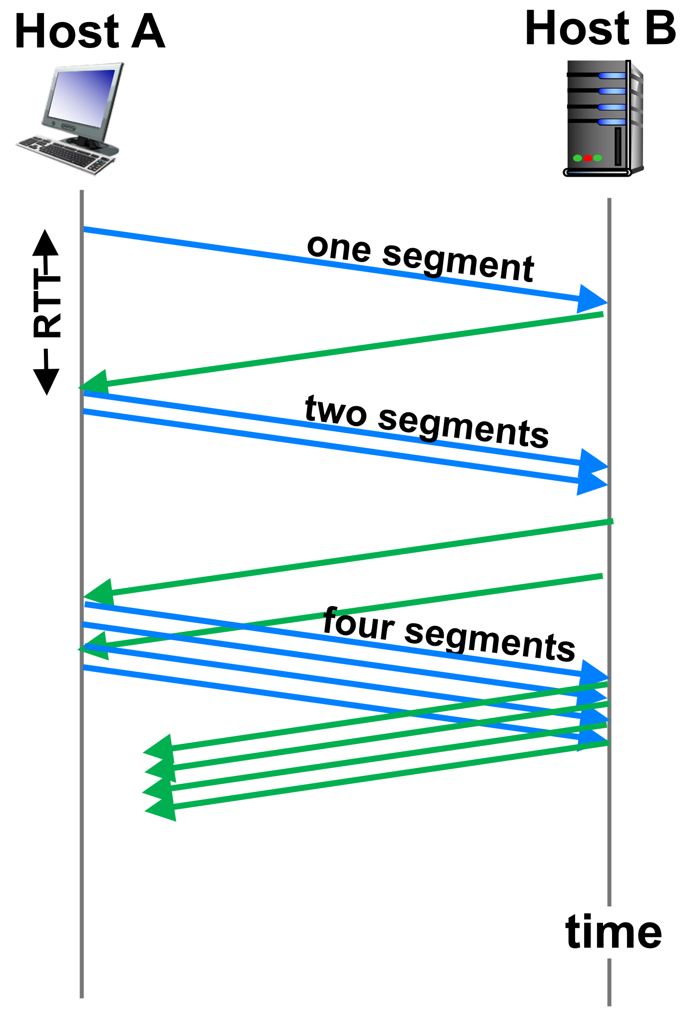
Slow start: initialization

Initial rate is "slow"

- relative to original TCP which had no congestion control
- initially cwnd = 1 MSS

Ramp up exponentially fast

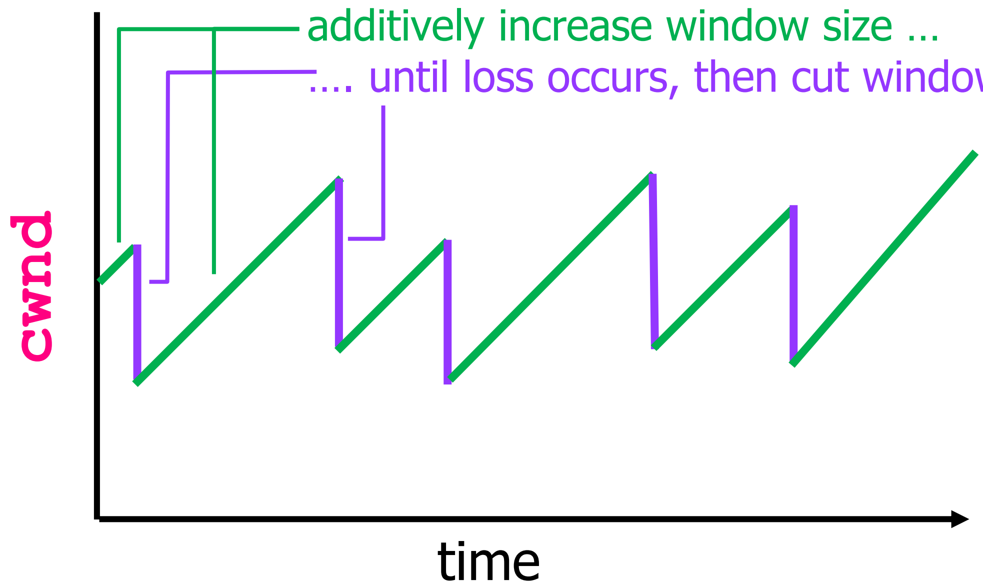
- every time ACK received
  - cwnd = cwnd + MSS
- essentially doubles cwnd every RTT



Congestion avoidance

Additive Increase Multiplicative Decrease (AIMD)

– probe cautiously for usable bandwidth
– additive increase
    - **cautious:** increase cwnd by 1 MSS every RTT until loss detected
– multiplicative decrease
    - **aggressive:** cut cwnd in half after loss



AIMD saw tooth behavior: probing for bandwidth

Finite state machine