Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- **Principles of congestion control**
- TCP congestion control
- Evolution of transport-layer functionality

Principles of congestion control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"

- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)

- different from flow control!

- a top-10 problem!
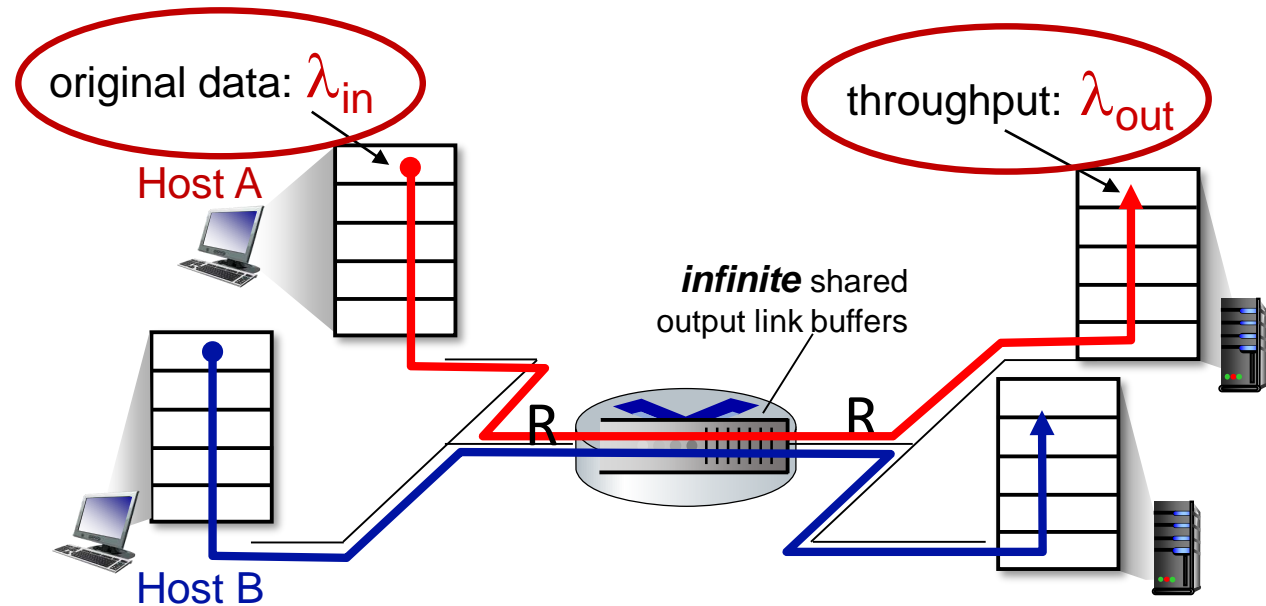

congestion control: too many senders, sending too fast

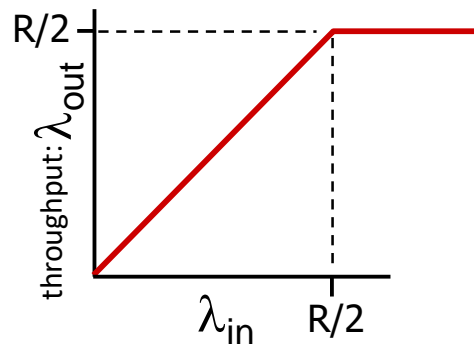flow control: one sender too fast for one receiver

Causes/costs of congestion: scenario 1
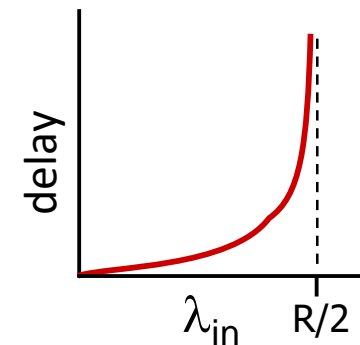
Simplest scenario:
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed

original data: λ_{in}

Host A

throughput: λ_{out}

infinite shared output link buffers

R   R

Host B

Q: What happens as arrival rate λ_{in} approaches R/2?

throughput: λ_{out}

R/2

λ_{in} R/2

maximum per-connection throughput: R/2

delay

λ_{in} R/2

large delays as arrival rate $\lambda\iota\nu\varepsilon$ approaches capacity
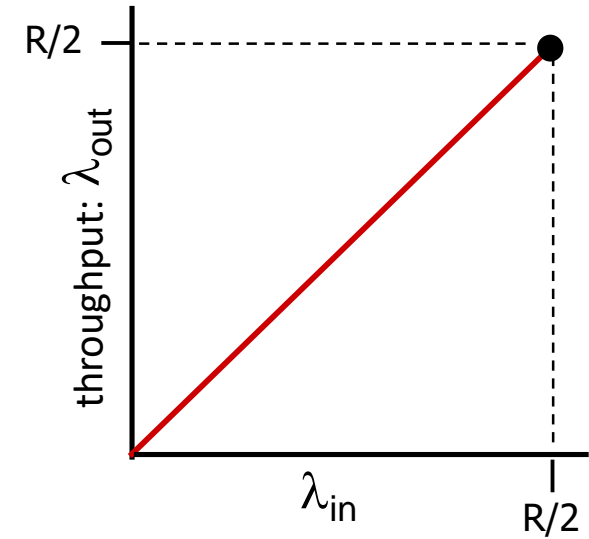
Causes/costs of congestion: scenario 2

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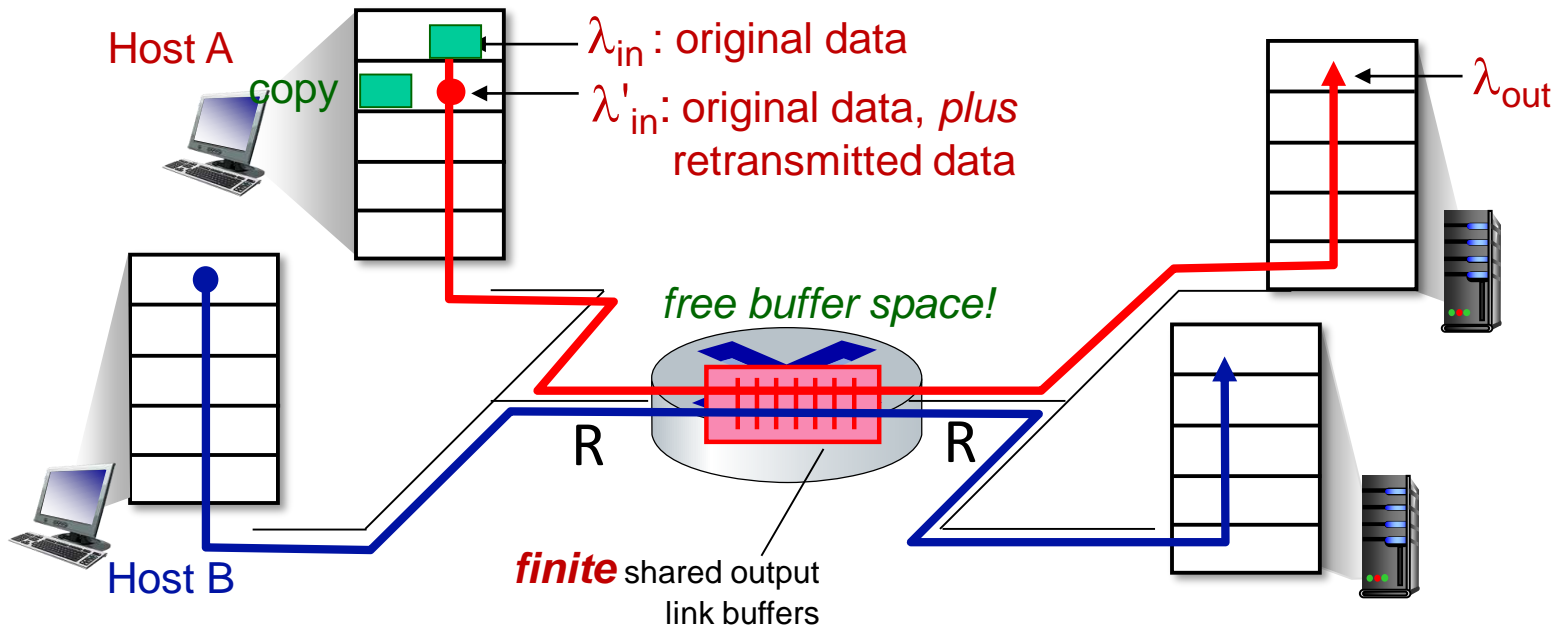
- sender retransmits lost, timed-out packet
    - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
    - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

Host A

λ_{in} : original data

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

λ_{out}

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

λ'_in: original data, *plus* retransmitted data

Host A

copy

free buffer space!

R   R

finite shared output link buffers

Host B

λ_out

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers

- sender knows when packet has been dropped: only resends if packet *known* to be lost

λ_{in} : original data

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

Host A

copy

Host B

no buffer space!

R

R

finite shared output link buffers

Causes/costs of congestion: scenario 2

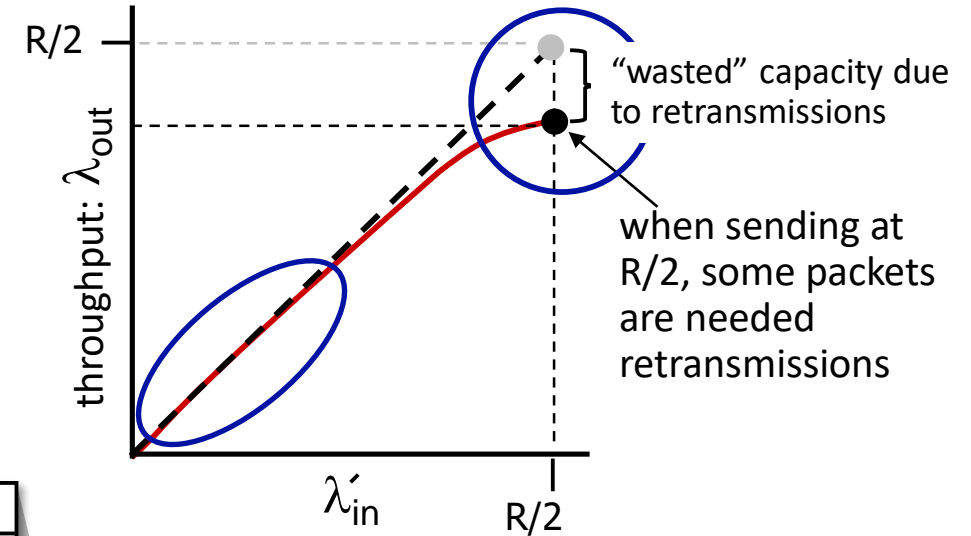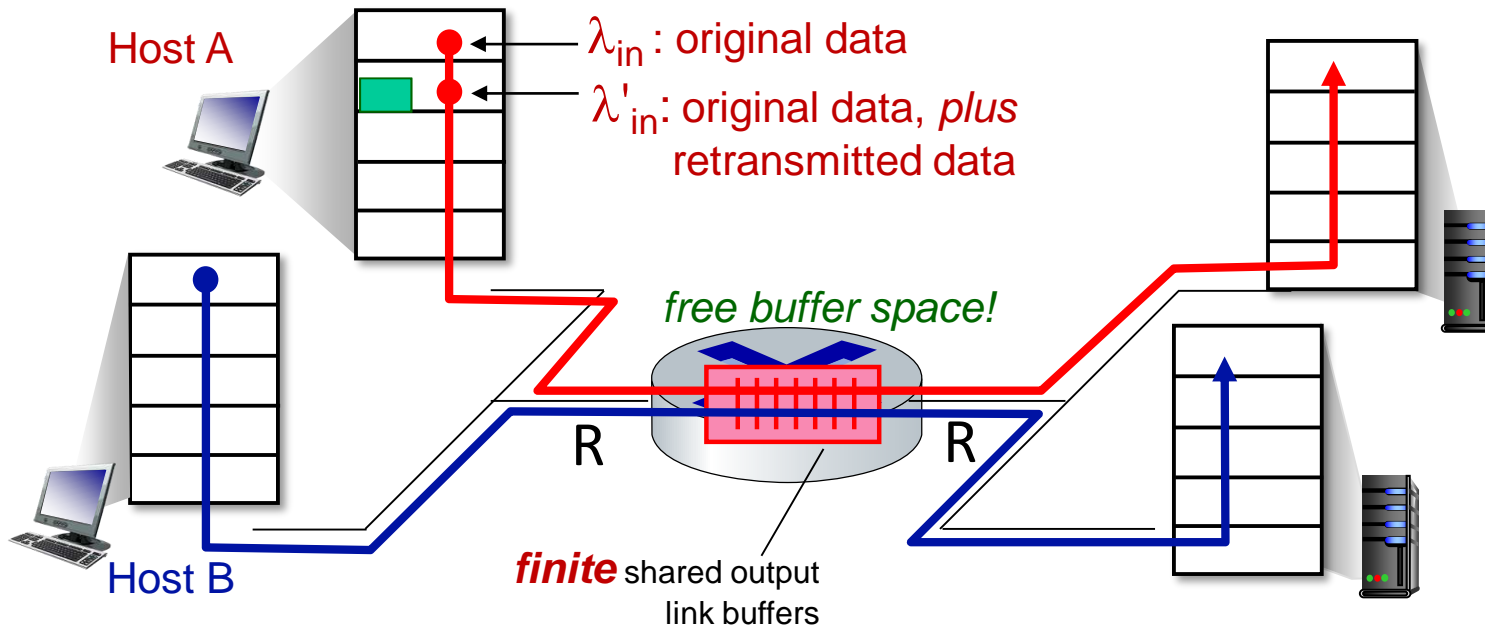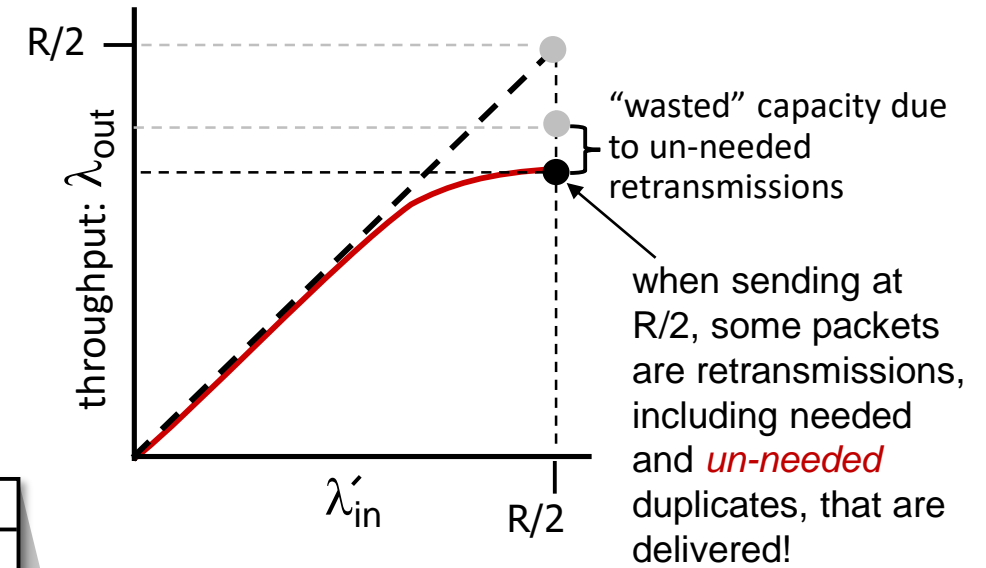## Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers

- sender knows when packet has been dropped: only resends if packet *known* to be lost

Host A

λ_{in} : original data

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

Host B

free buffer space!

R   R

finite shared output link buffers

R/2

throughput: λ_{out}

"wasted" capacity due to retransmissions

when sending at R/2, some packets are needed retransmissions

λ'_{in} R/2

Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

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

λ_{in} : original data

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

timeout

Host

free buffer space!

R   R

finite shared output link buffers

Host B

throughput: λ_{out}

R/2

λ'_{in} R/2

"wasted" capacity due to un-needed retransmissions

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that are delivered!

Causes/costs of congestion: scenario 2
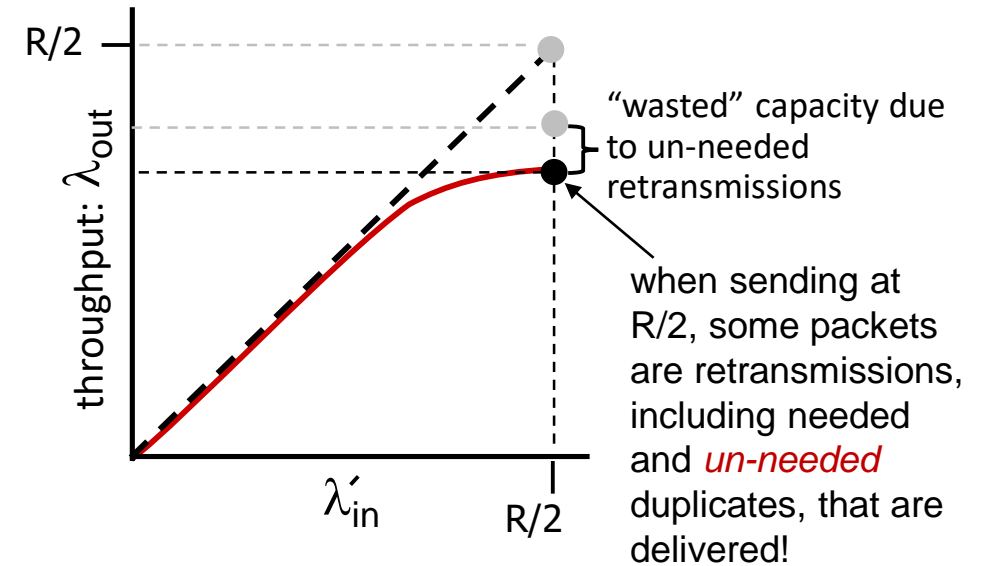
Realistic scenario: *un-needed duplicates*

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

"wasted" capacity due to un-needed retransmissions

when sending at R/2, some packets are retransmissions, including needed and *un-needed* duplicates, that are delivered!
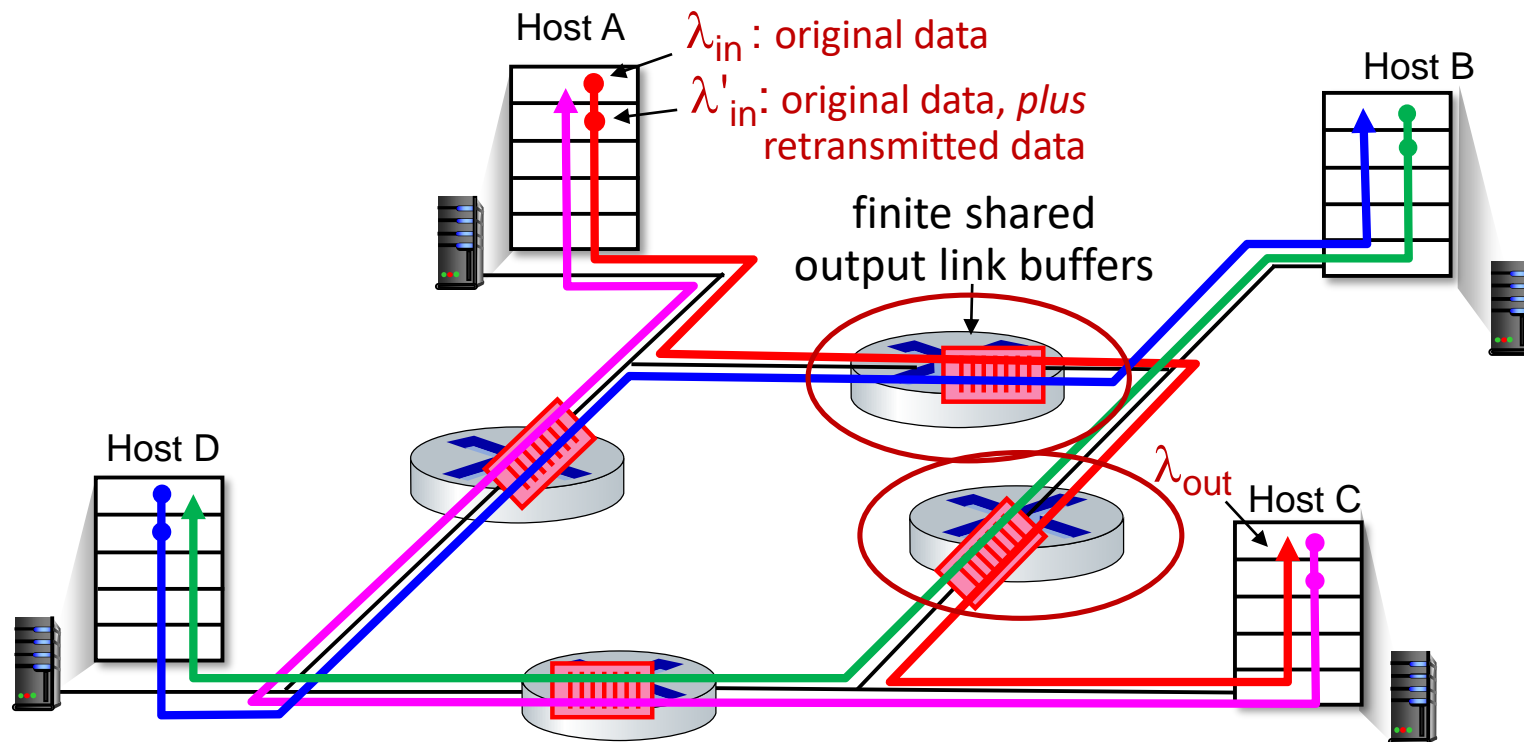
"costs" of congestion:

- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput
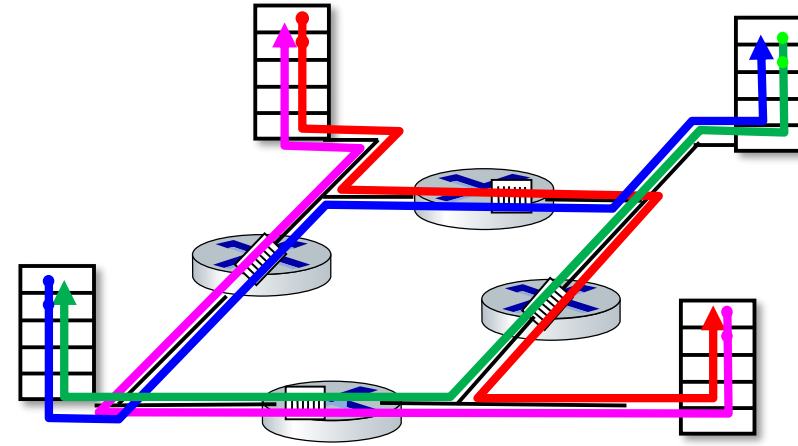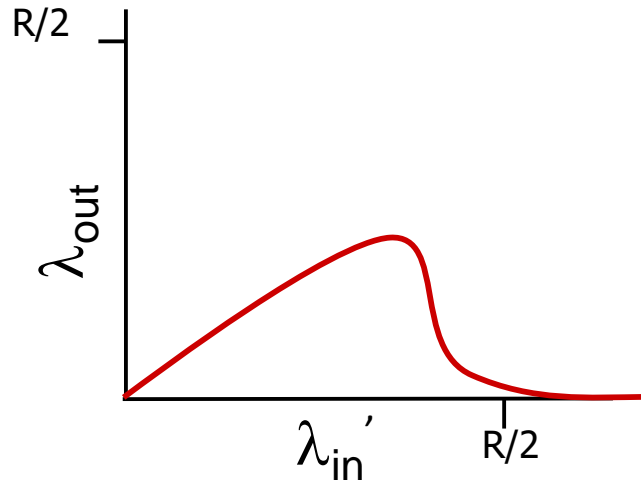
Causes/costs of congestion: scenario 3

- *four* senders
- *multi-hop* 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

Host A

λ_{in} : original data

λ_{in}': original data, *plus* retransmitted data

finite shared output link buffers

Host B

Host D

λ_{out}
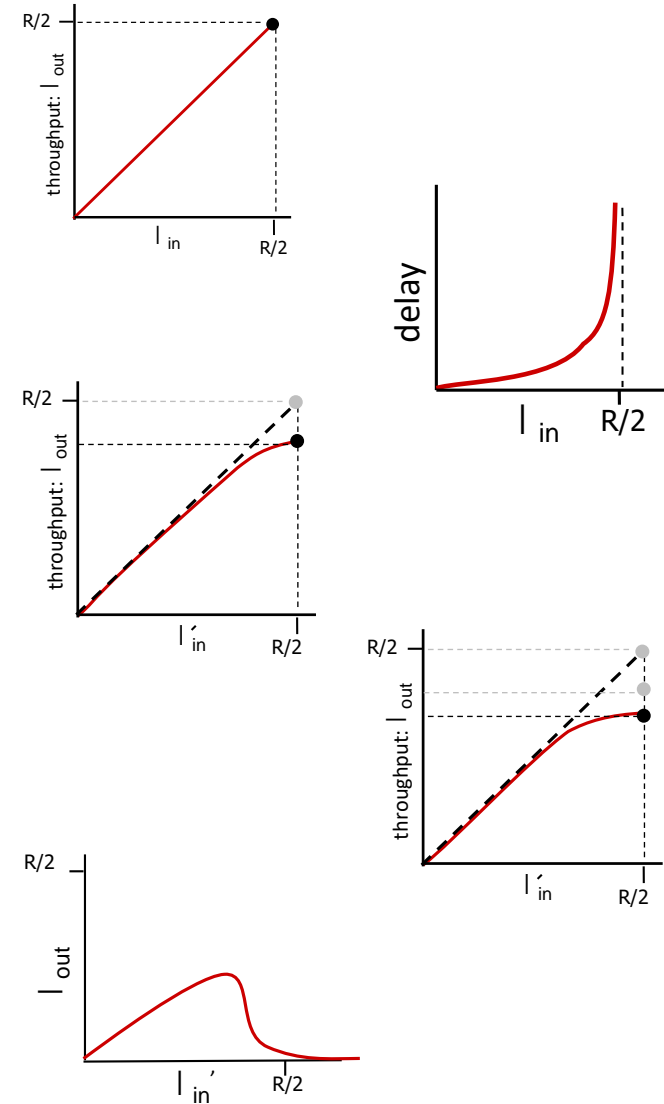
Host C

Causes/costs of congestion: scenario 3

another "cost" of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!
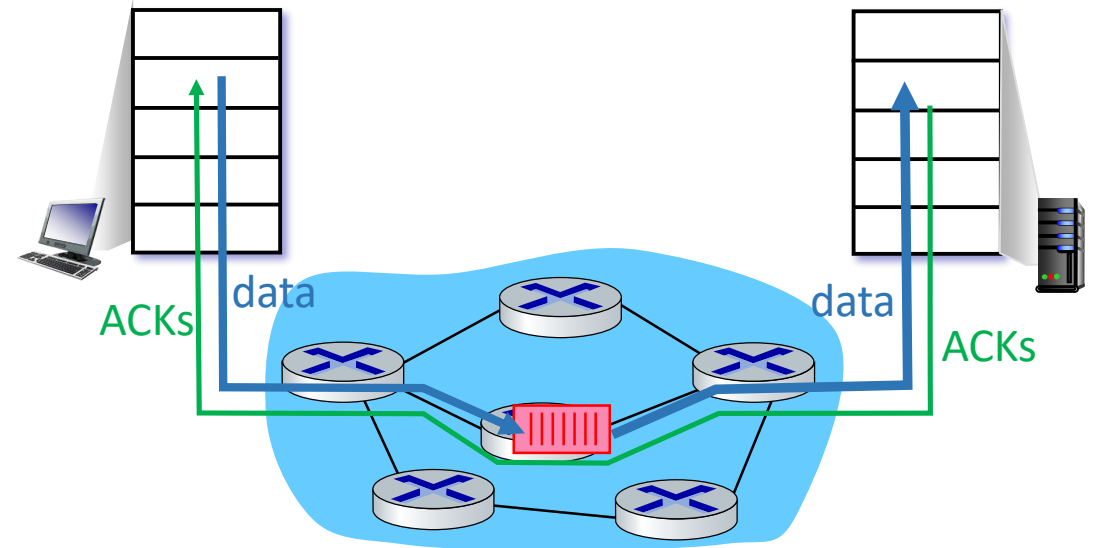
Causes/costs of congestion: insights

- throughput can never exceed capacity

- delay increases as capacity approached

- loss/retransmission decreases effective throughput

- un-needed duplicates further decreases effective throughput

- upstream transmission capacity / buffering wasted for packets lost downstream

Approaches towards congestion control
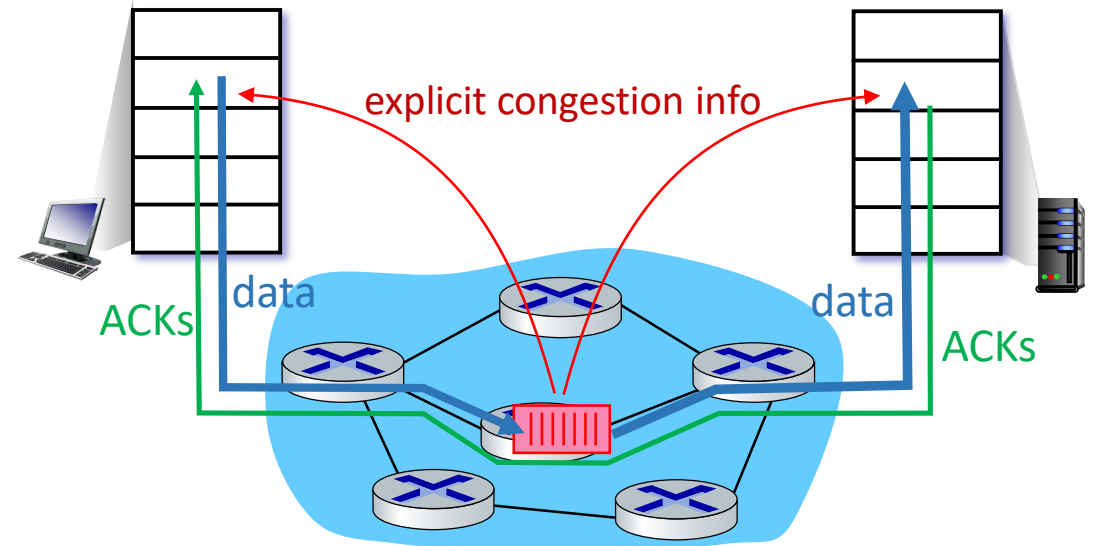
End-end congestion control:

- no explicit feedback from network

- congestion *inferred* from observed loss, delay

- approach taken by TCP

Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router

- may indicate congestion level or explicitly set sending rate

- TCP ECN, ATM, DECbit protocols

Chapter 3: roadmap

TCP congestion control: AIMD
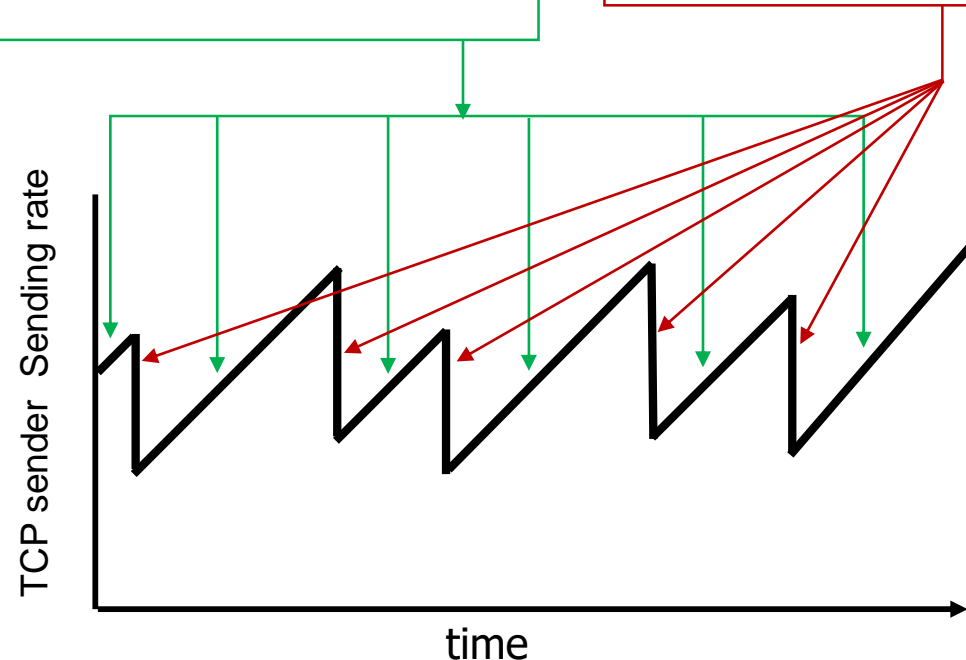
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event

AIMD sawtooth behavior: *probing* for bandwidth

TCP sender Sending rate
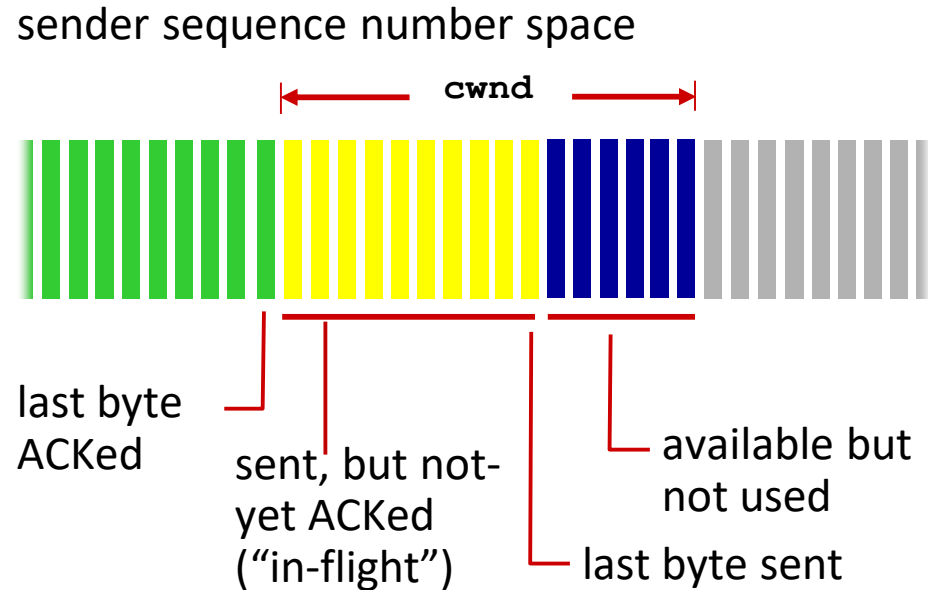
time

TCP AIMD: more

Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP congestion control: details

sender sequence number space

TCP sending behavior:

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

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

- TCP sender limits transmission: `LastByteSent- LastByteAcked` $\leq$ `cwnd`
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing 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:* initial rate is slow, but ramps up exponentially fast

TCP: from slow start to congestion avoidance

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
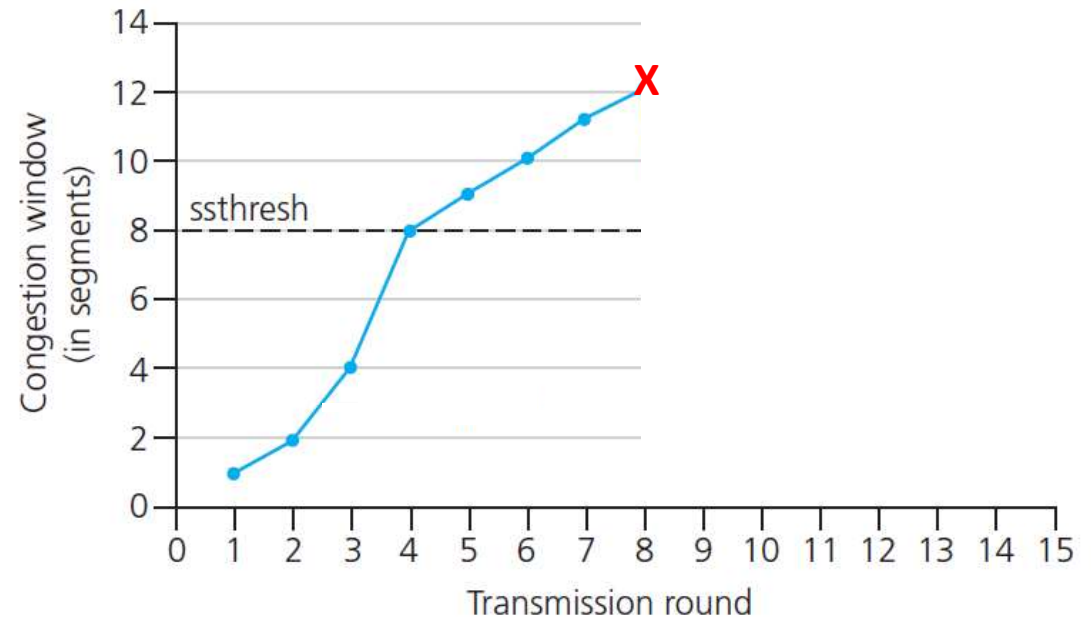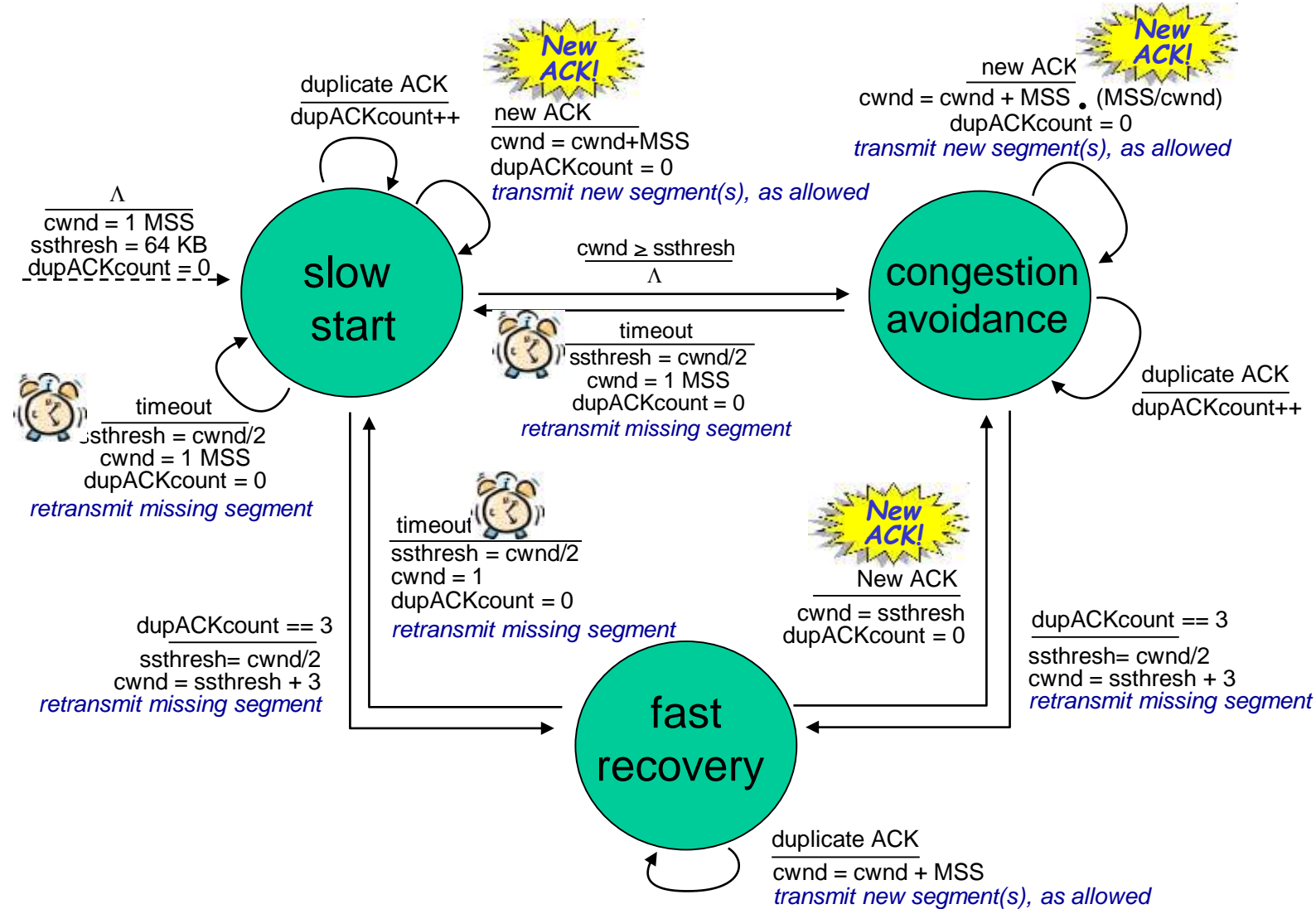

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Summary: TCP congestion control

New ACK!

duplicate ACK
——————————
dupACKcount++

new ACK
——————————
cwnd = cwnd+MSS
dupACKcount = 0
transmit new segment(s), as allowed

New ACK!

new ACK
——————————
cwnd = cwnd + MSS . (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed

Λ
——————————
cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0

slow start

cwnd ≥ ssthresh
——————————
Λ

congestion avoidance

timeout
——————————
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment

duplicate ACK
——————————
dupACKcount++

timeout
——————————
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment

timeout
——————————
ssthresh = cwnd/2
cwnd = 1
dupACKcount = 0
retransmit missing segment

New ACK!

New ACK
——————————
cwnd = ssthresh
dupACKcount = 0

dupACKcount == 3
——————————
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment

dupACKcount == 3
——————————
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment

fast recovery

duplicate ACK
——————————
cwnd = cwnd + MSS
transmit new segment(s), as allowed

TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
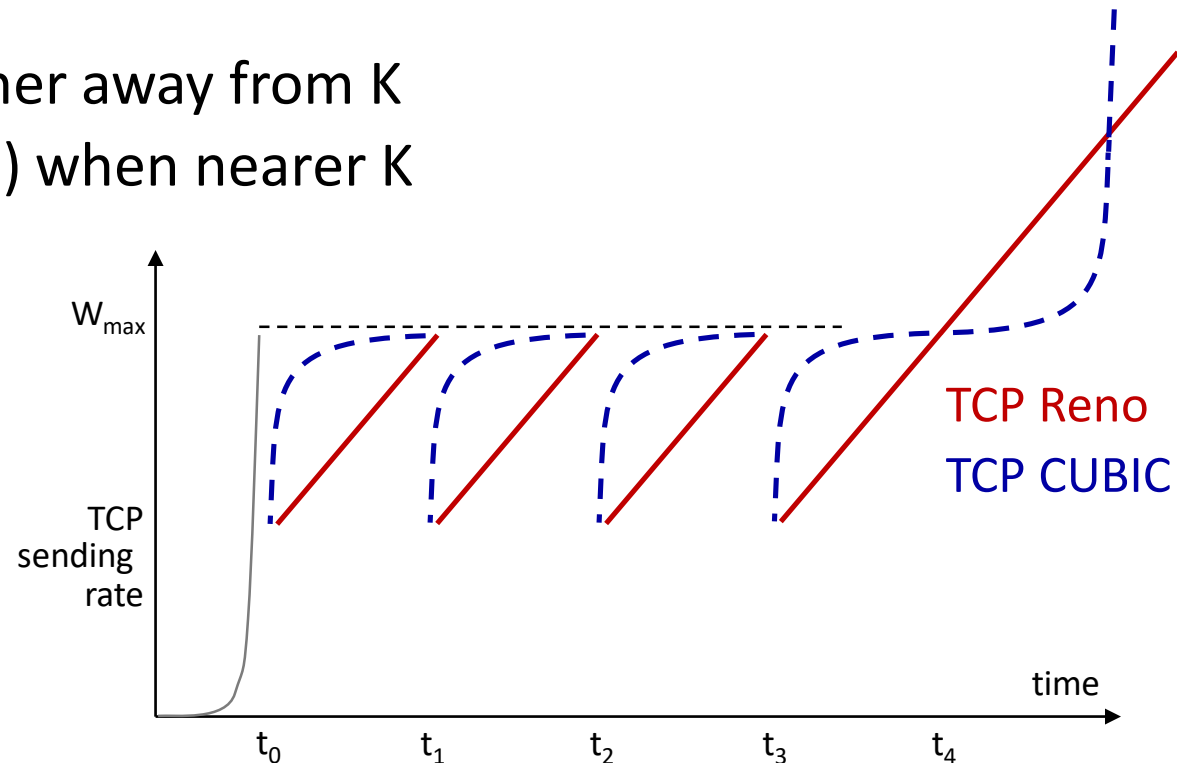- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*

classic TCP

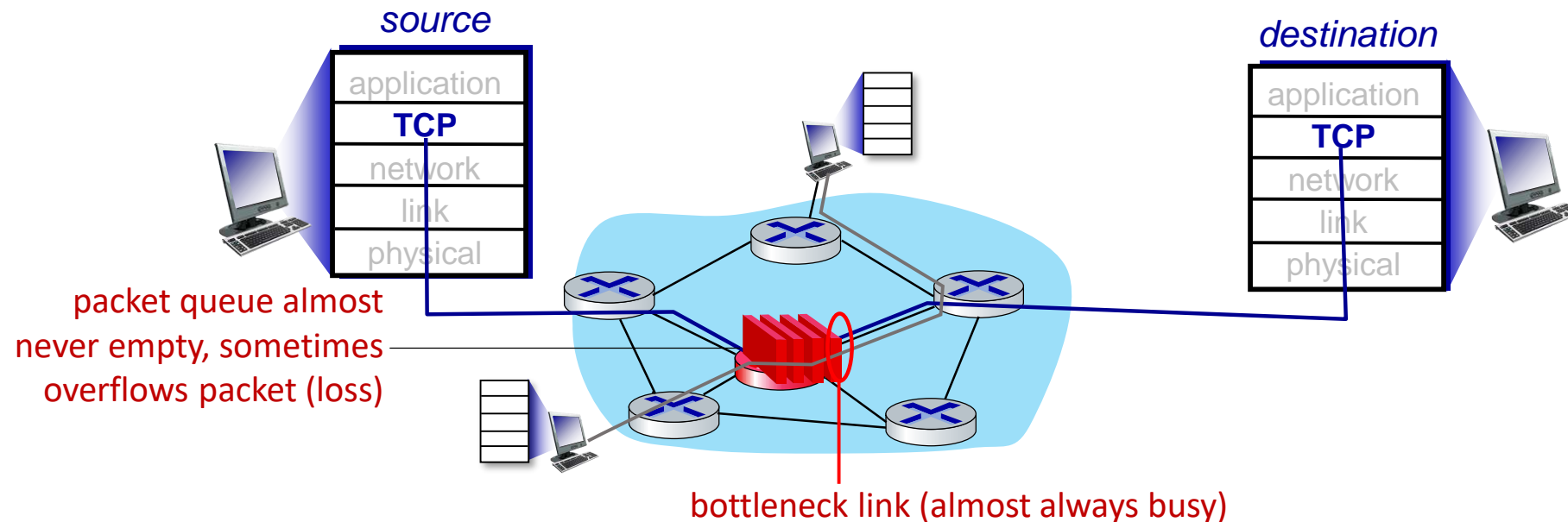TCP CUBIC - higher throughput in this example

TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tunable

- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K

- TCP CUBIC default in Linux, most popular TCP for popular Web servers

TCP Reno

TCP CUBIC

TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*

source

application
TCP
network
link
physical

destination

application
TCP
network
link
physical

packet queue almost never empty, sometimes overflows packet (loss)
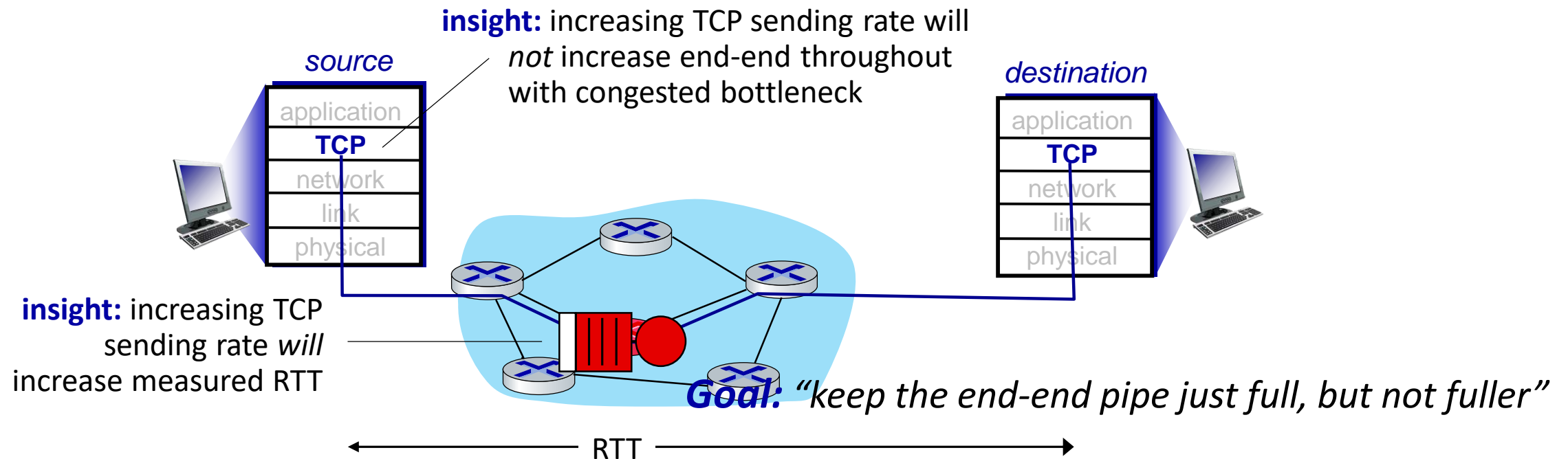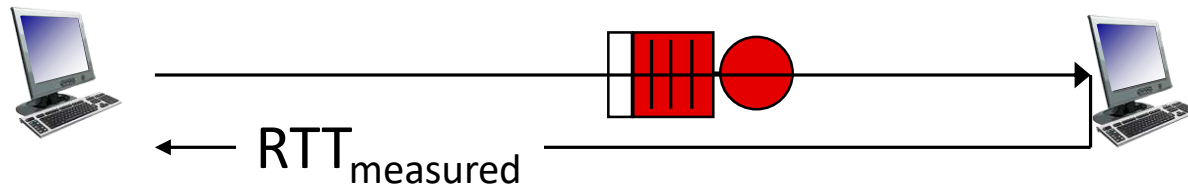
bottleneck link (almost always busy)

TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*

- understanding congestion: useful to focus on congested bottleneck link

insight: increasing TCP sending rate will *not* increase end-end throughout with congested bottleneck

source

| application |
| TCP |
| network |
| link |
| physical |

destination

| application |
| TCP |
| network |
| link |
| physical |

insight: increasing TCP sending rate *will* increase measured RTT

Goal: "keep the end-end pipe just full, but not fuller"

RTT

Delay-based TCP congestion control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering

$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}

if measured throughput "very close" to uncongested throughput
     increase `cwnd` linearly   /* since path not congested */
else if measured throughput "far below" uncongested throughout
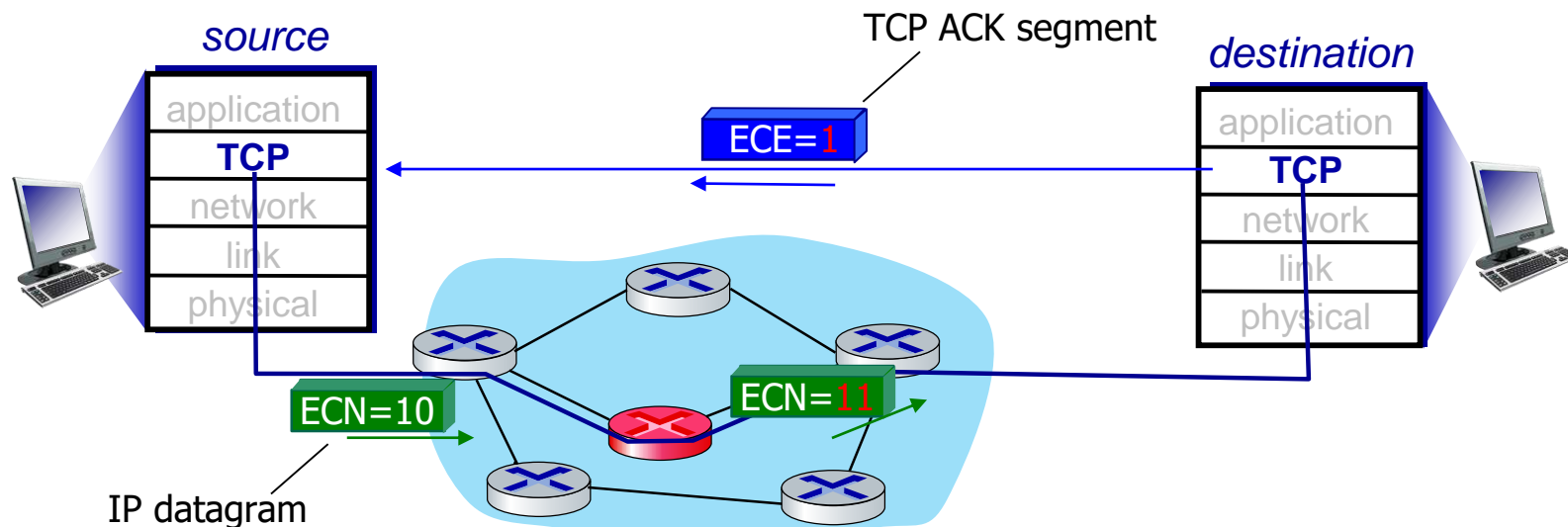   decrease `cwnd` linearly   /* since path is congested */

Delay-based TCP congestion control

- congestion control without inducing/forcing loss

- maximizing throughout ("keeping the just pipe full... ") while keeping delay low ("...but not fuller")

- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network
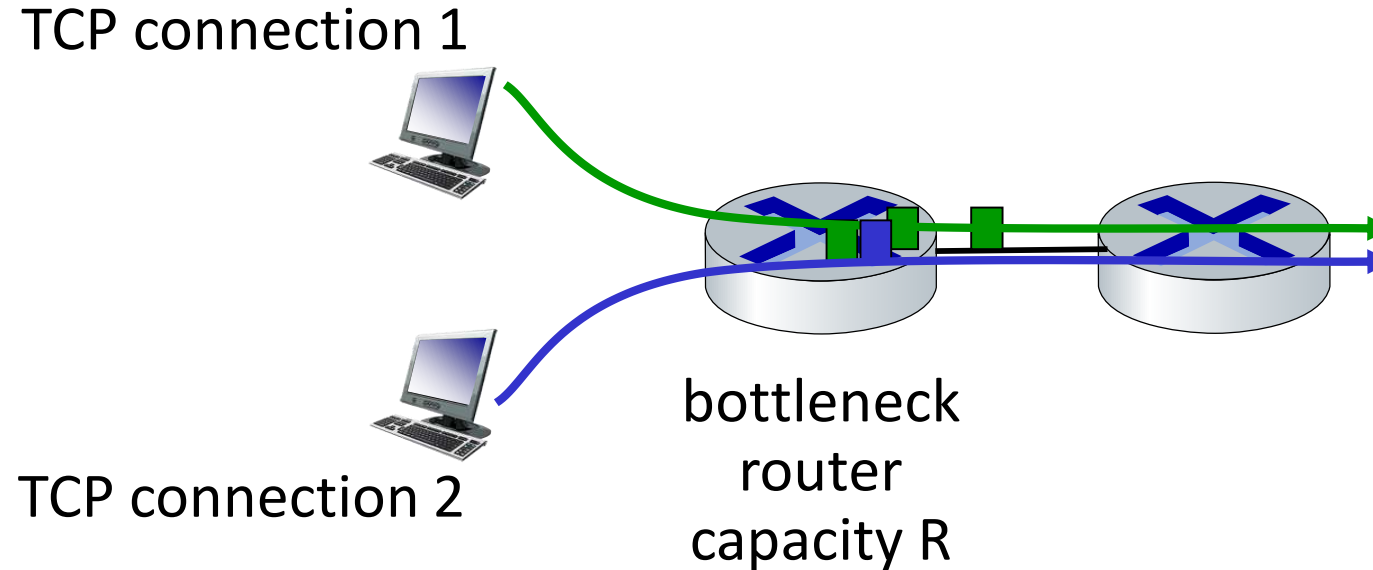
Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)
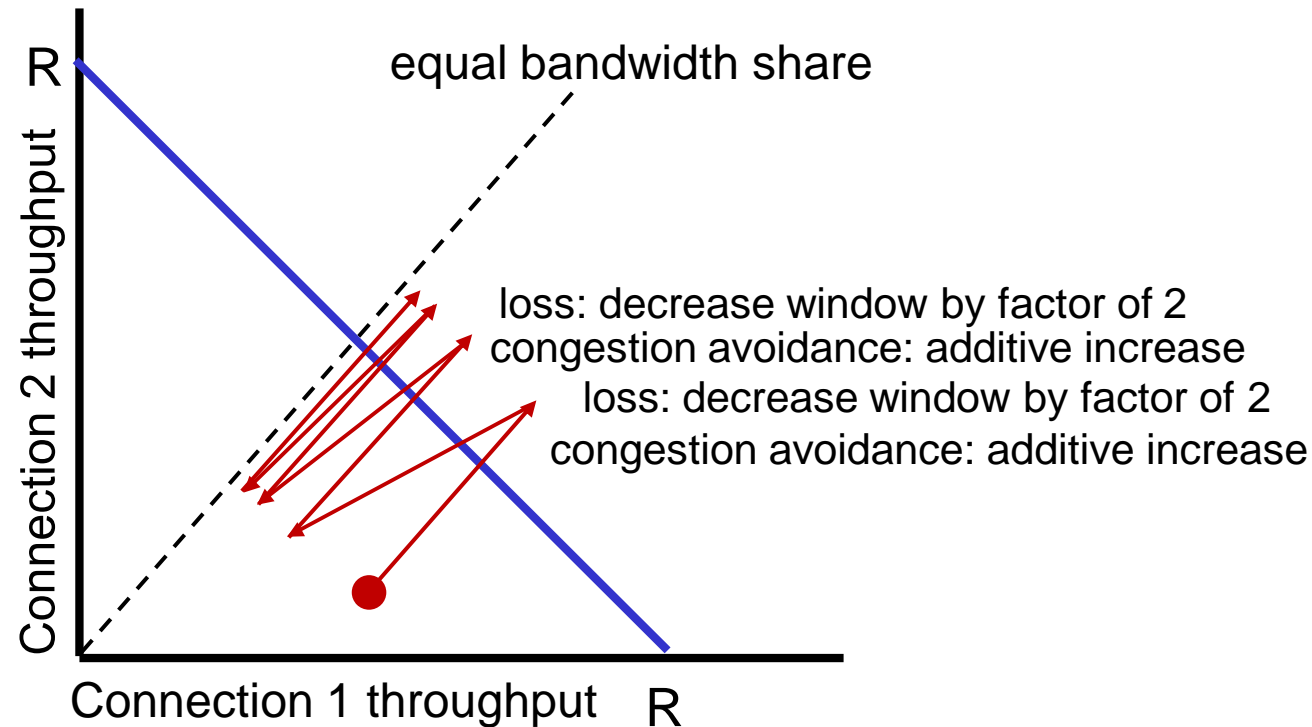
TCP fairness

Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*

TCP connection 1

TCP connection 2

bottleneck
router
capacity R

Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally

R   equal bandwidth share

Connection 2 throughput

loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase

Connection 1 throughput   R

Is TCP fair?

A: Yes, under idealized assumptions:
- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be "fair"?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no "Internet police" policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2