Ch – 3 Transport Layer (Cont.)

Class 6

rdt2.1: receiver, handling garbled ACK/NAKs
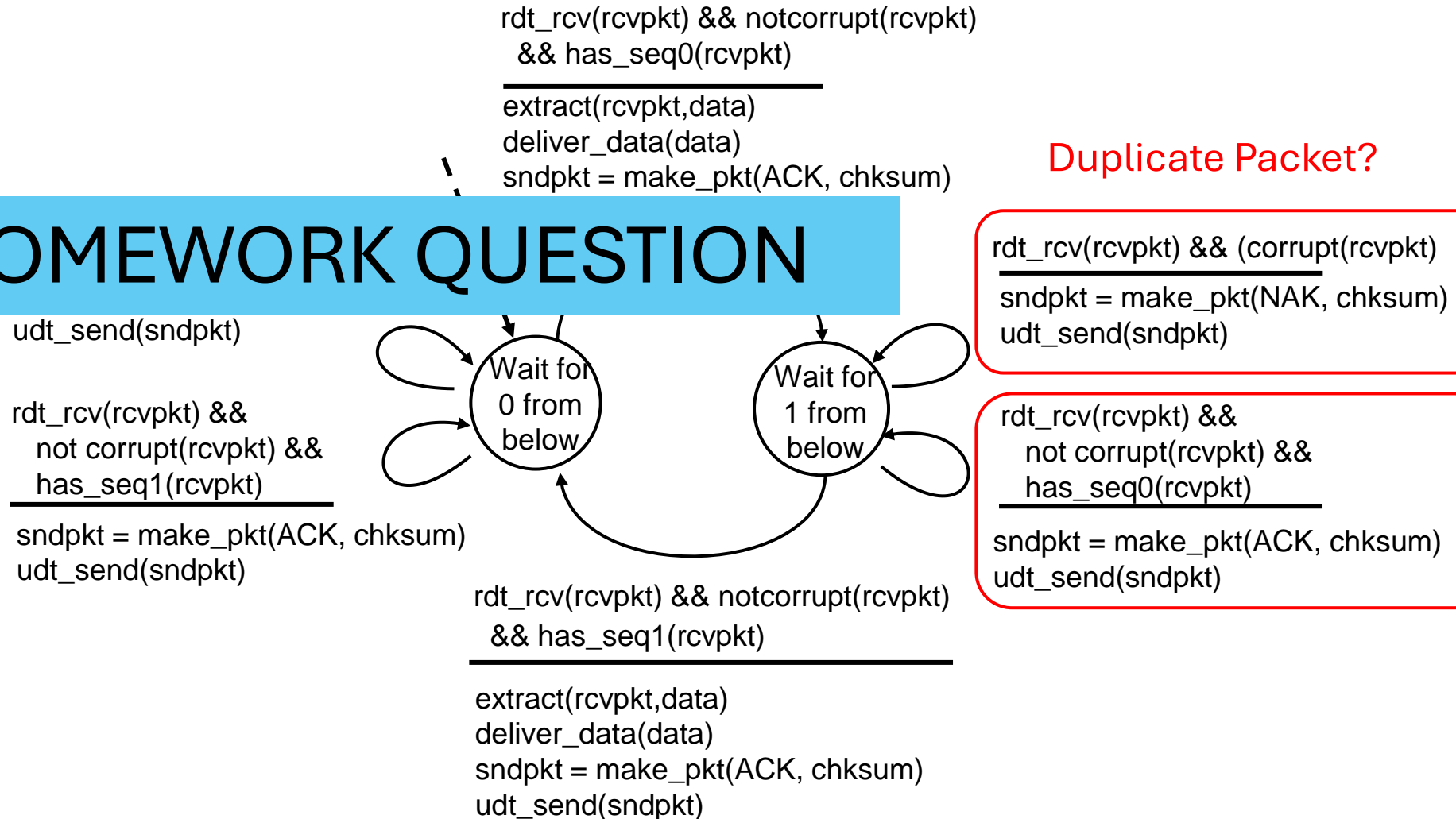
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)

Duplicate Packet?

HOMEWORK QUESTION

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Wait for
0 from
below

Wait for
1 from
below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

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:
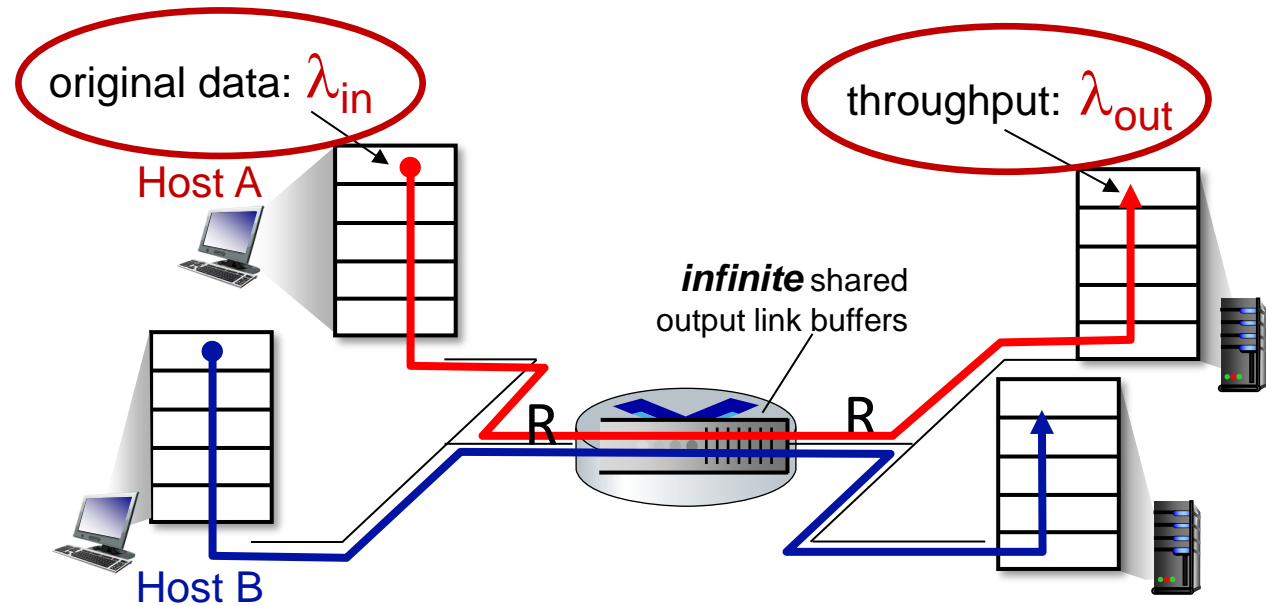too many senders,
sending too fast

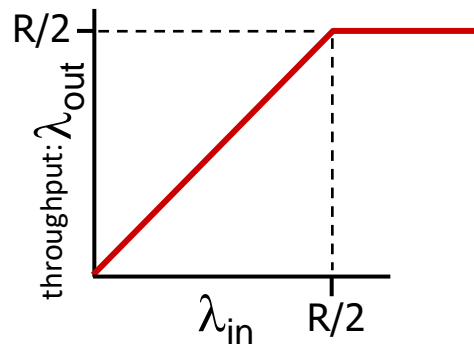**flow control:** one sender
too fast for one receiver

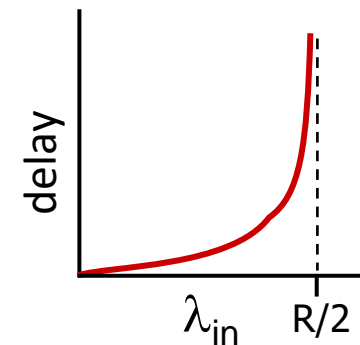# Causes/costs of congestion: scenario 1

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?

maximum per-connection throughput: R/2

large delays as arrival rate λ_{in} approaches capacity
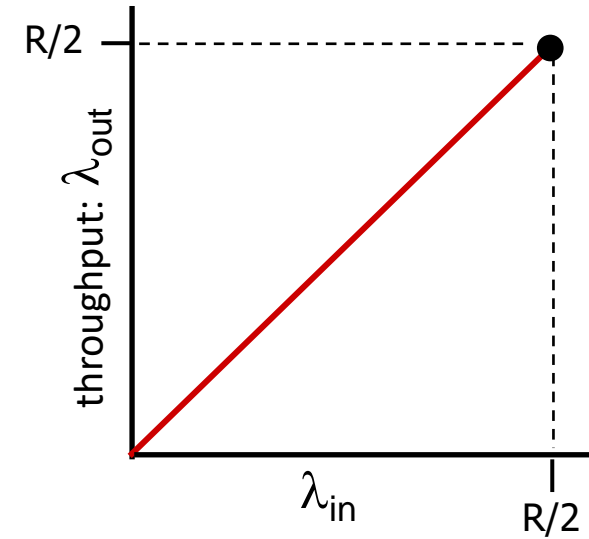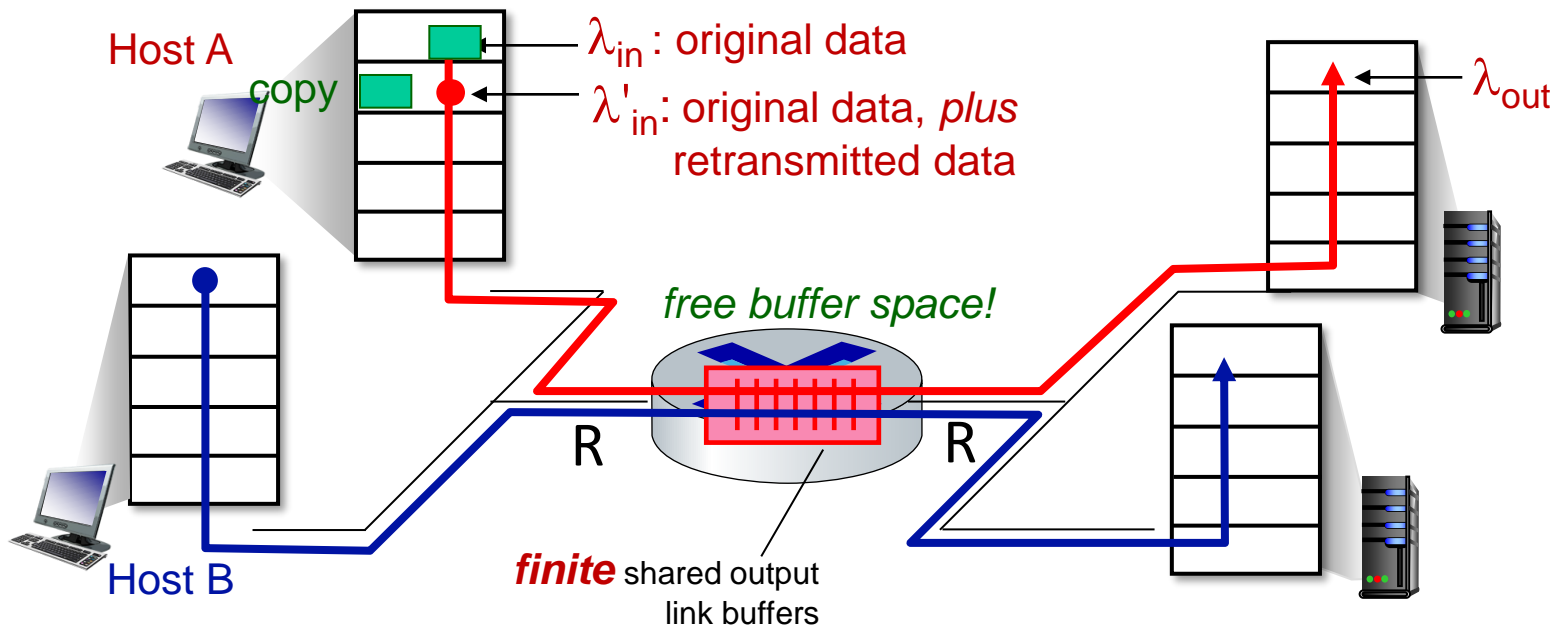
Causes/costs of congestion: scenario 2

- one router, *finite* buffers

- sender 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

R   R
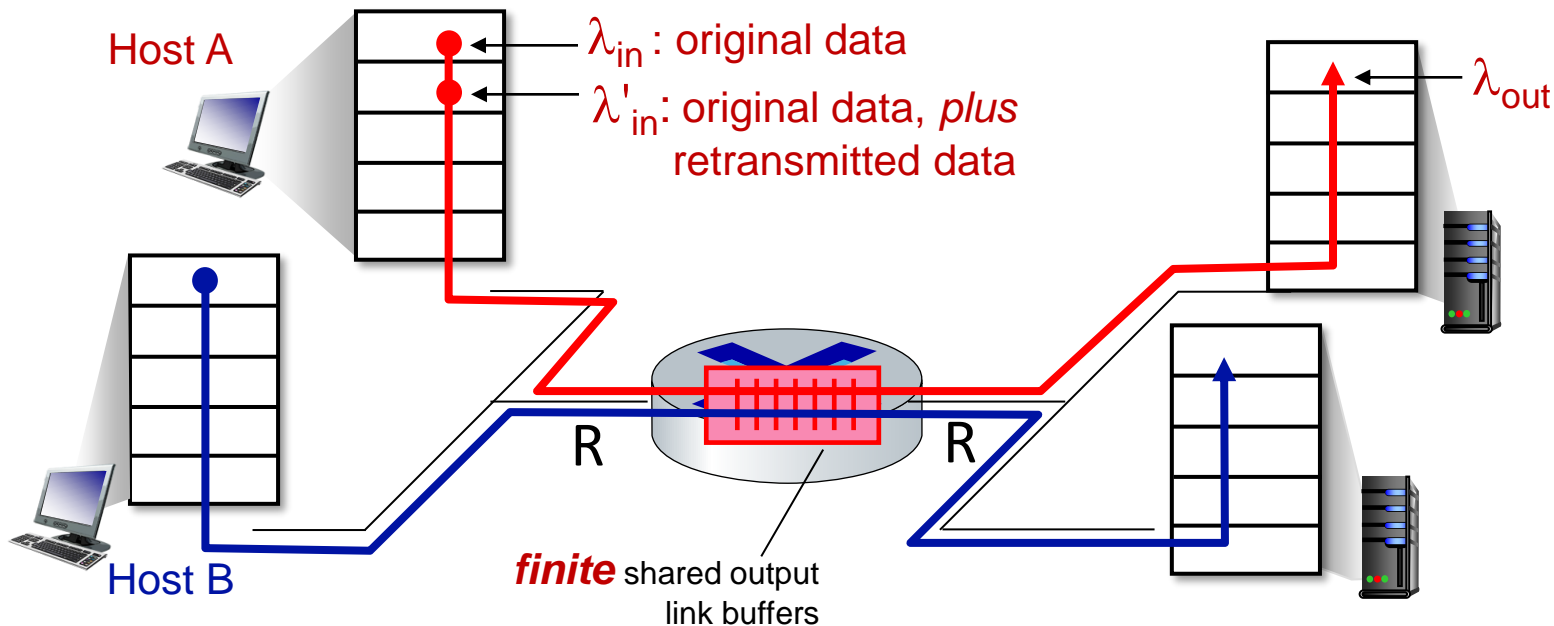
Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

Host A

copy

λ_{in} : original data

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

λ_{out}

free buffer space!

R   R

finite shared output link buffers

Host B

throughput: λ_{out}

R/2

λ_{in} R/2

Causes/costs of congestion: scenario 2

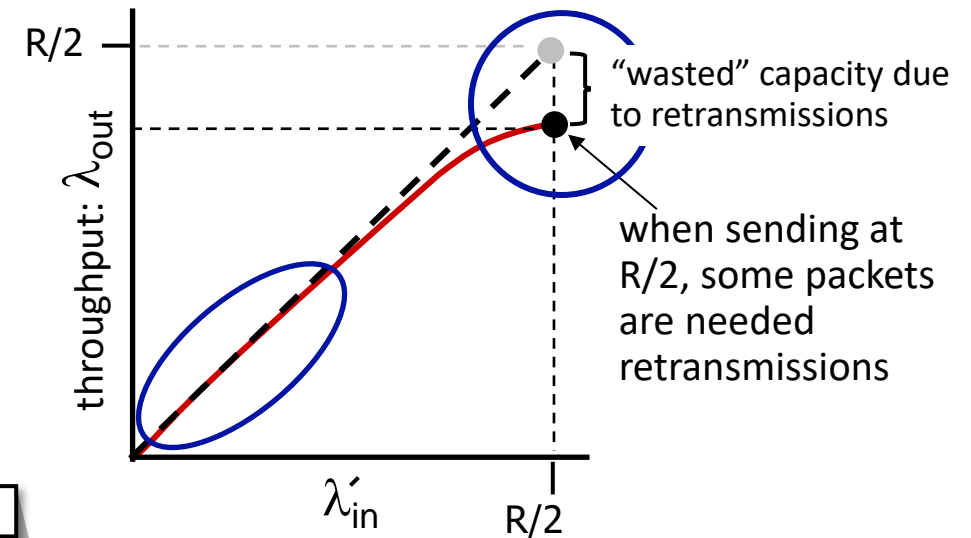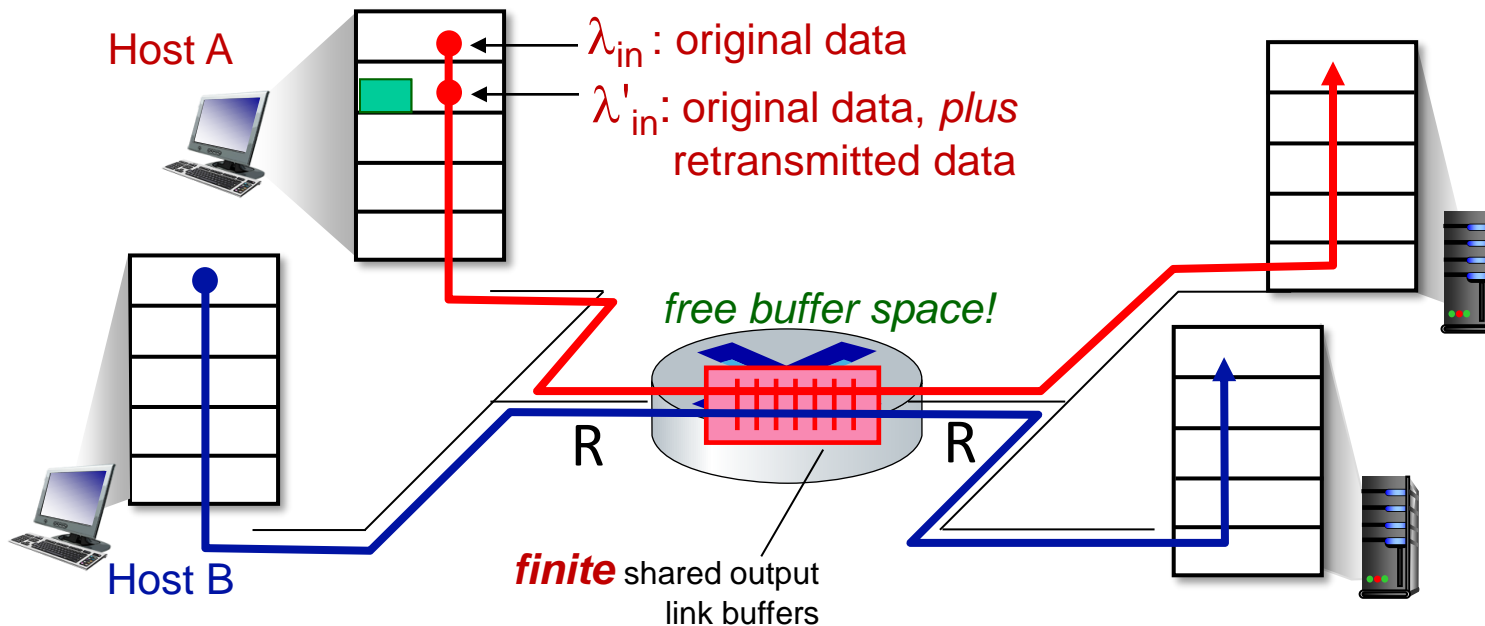## Idealization: *some* perfect knowledge

- 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

copy

λ_{in} : original data

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

Host B

no buffer space!

R

R

finite shared output link buffers

Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge
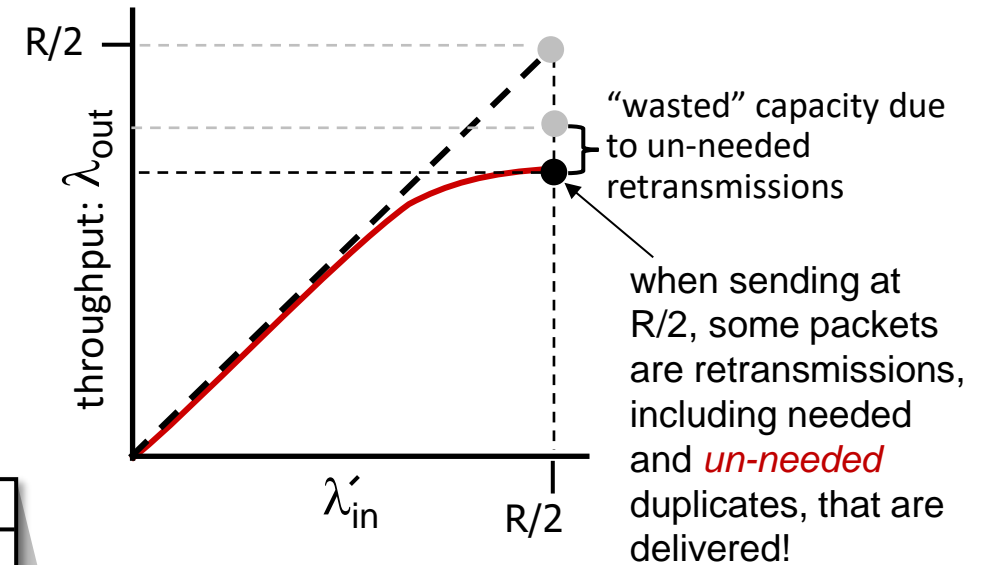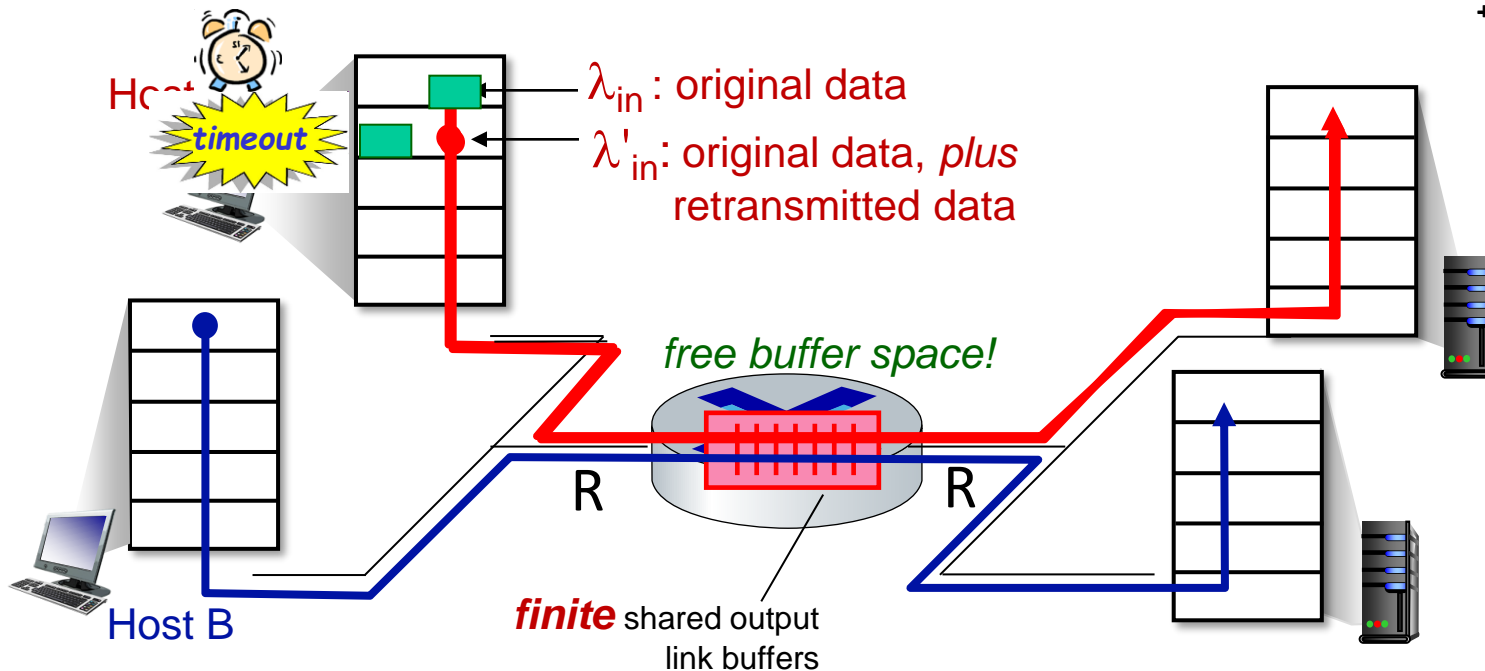
- packets can be lost (dropped at router) due to full buffers

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

"wasted" capacity due to retransmissions

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

Host A

λ_{in} : original data

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

free buffer space!

R   R

finite shared output link buffers

Host B

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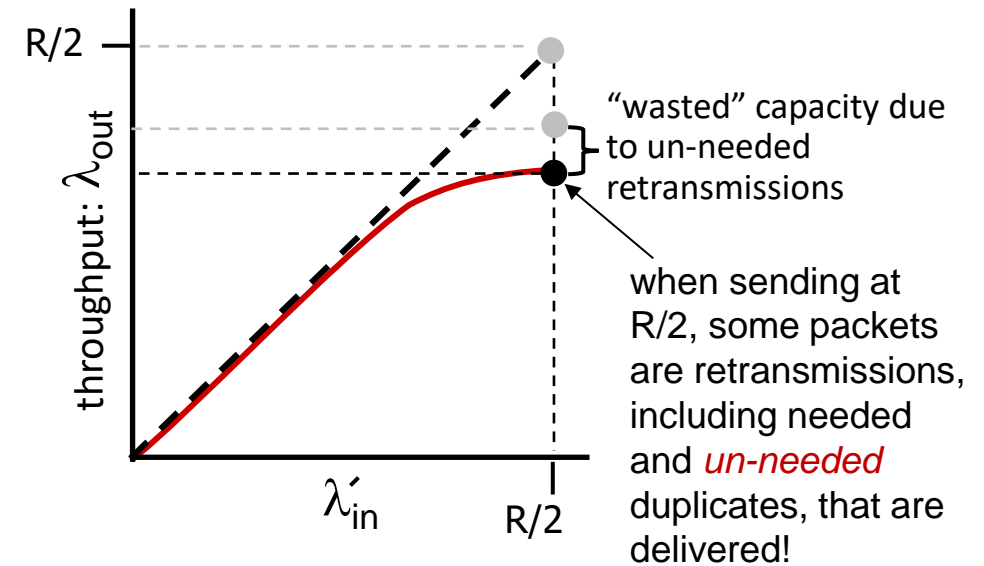
Realistic scenario: *un-needed duplicates*

- 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

free buffer space!

finite shared output link buffers

Host B

throughput: λ_{out}

R/2

R/2

λ'_{in}

"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!

timeout

Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

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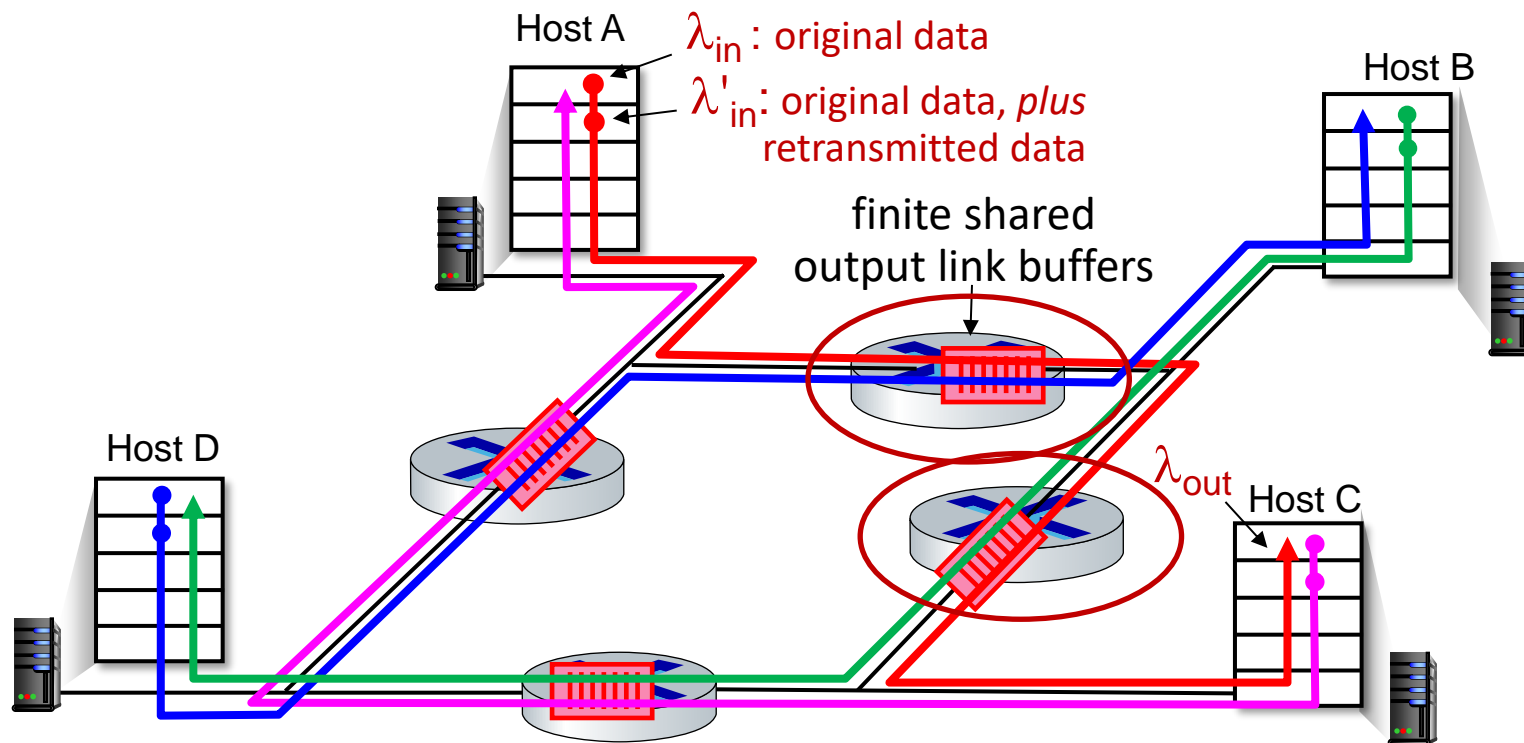- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered

R/2

throughput: λ_{out}

"wasted" capacity due to un-needed retransmissions

λ'_{in} R/2

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
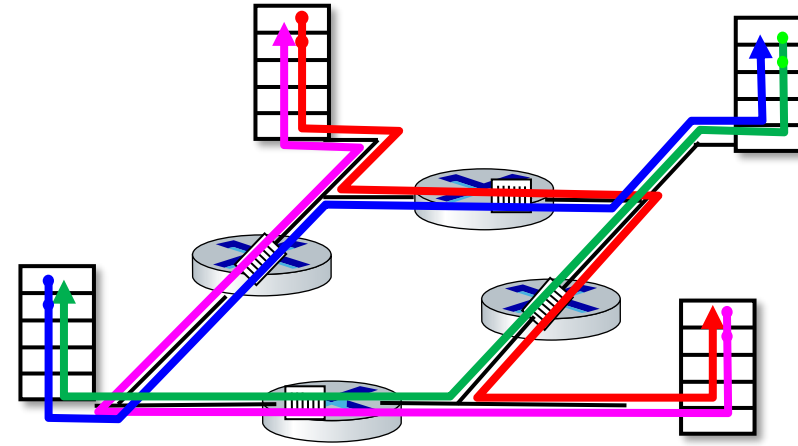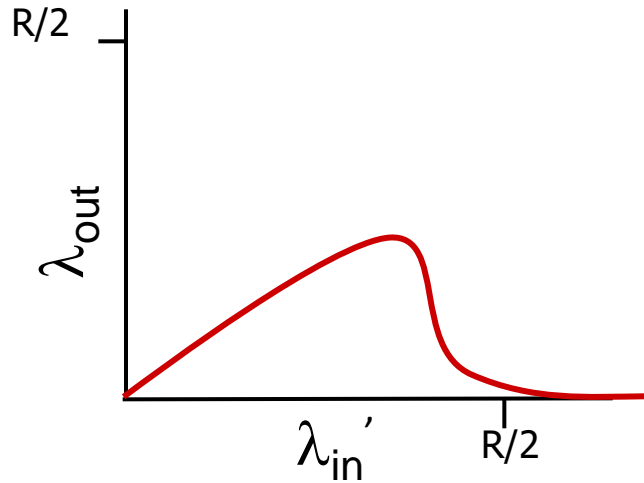
Causes/costs of congestion: scenario 3

- *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

Host B

finite shared output link buffers

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
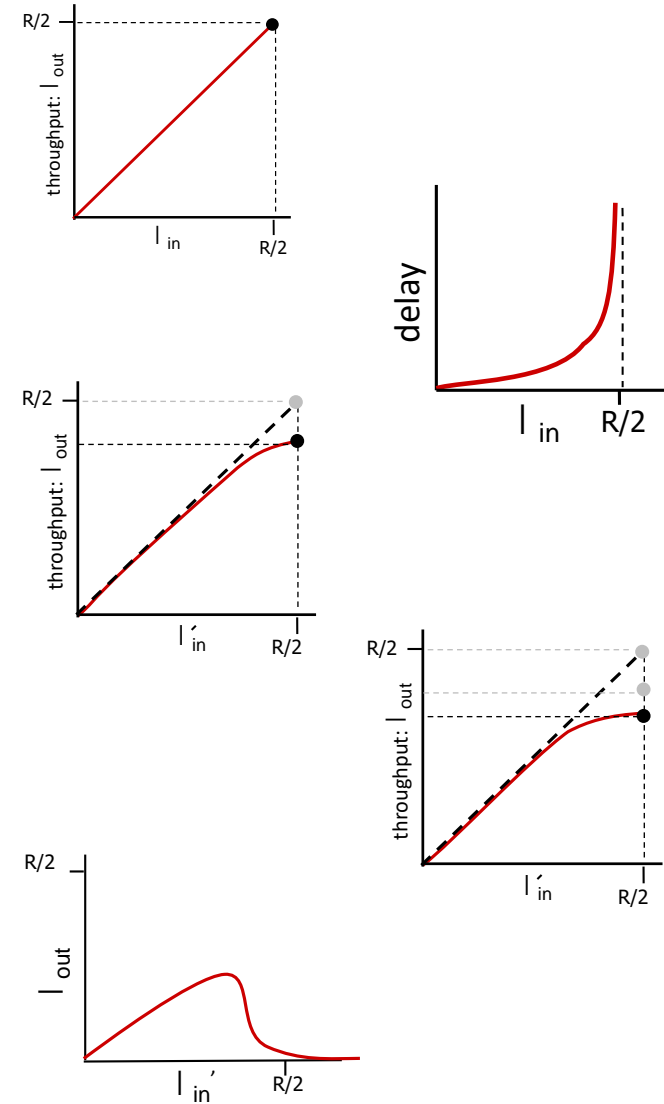
Approaches towards congestion control
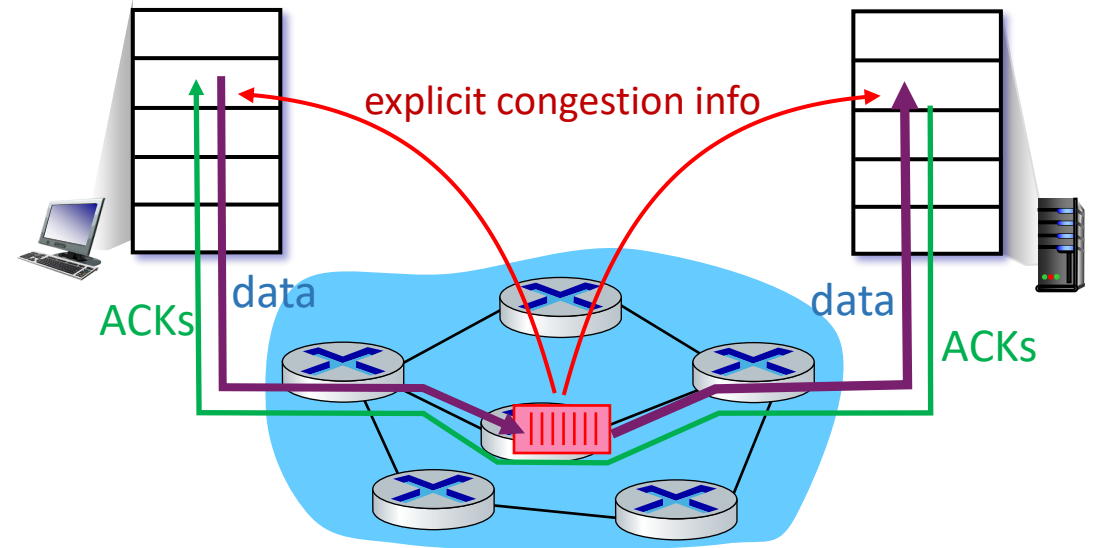
End-end congestion control:

- no explicit feedback from network

- congestion *inferred* from observed loss, delay

  ▪ approach taken by TCP
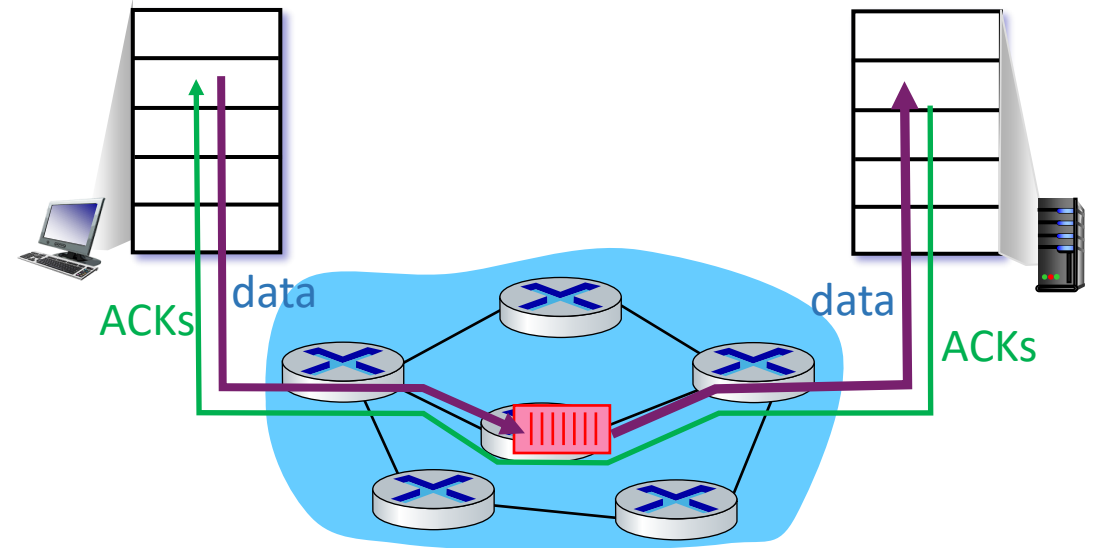
Approaches towards congestion control

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

- 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
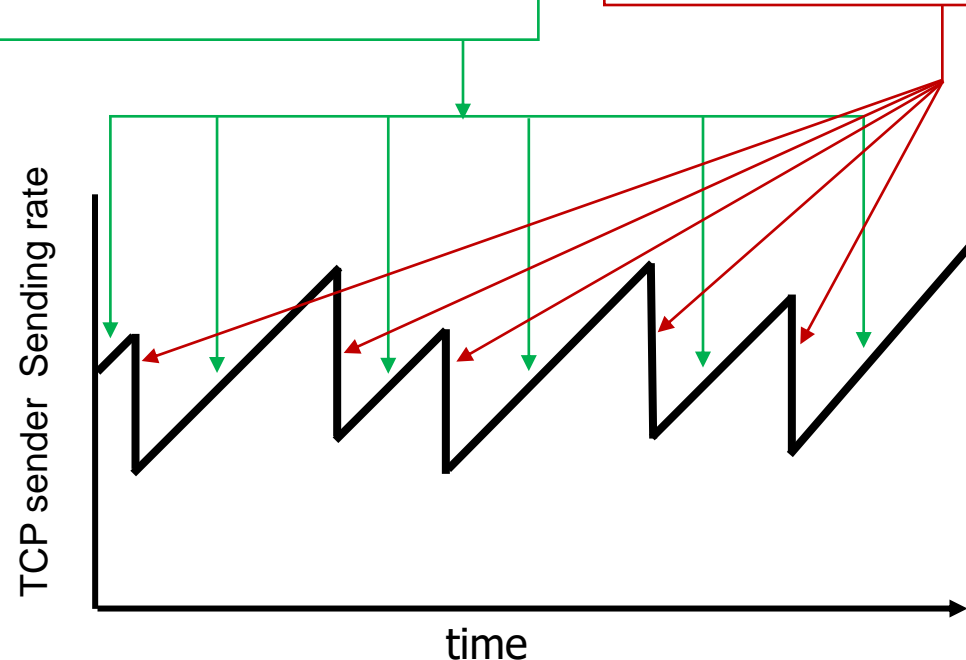
TCP congestion control: AIMD

- *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
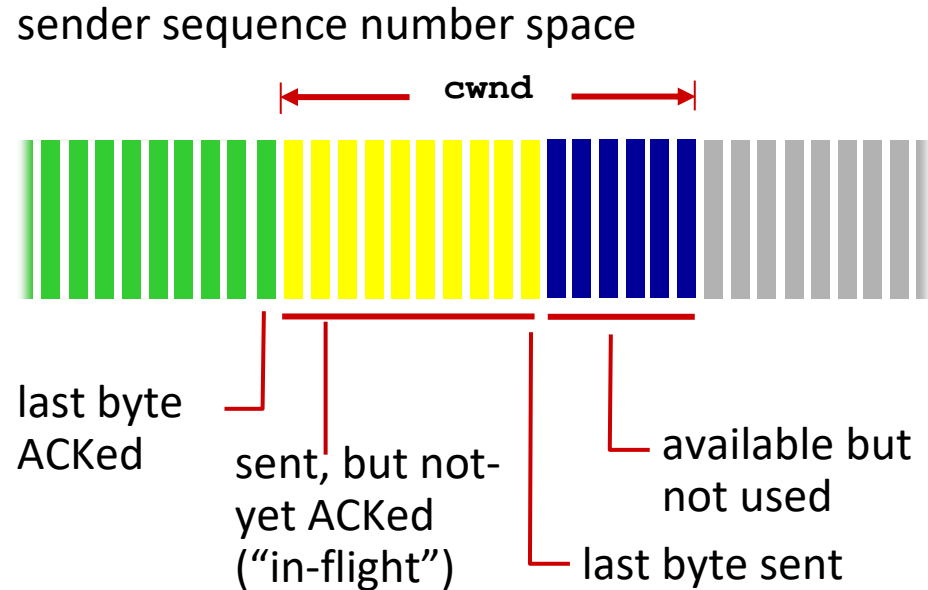
TCP sender Sending rate

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
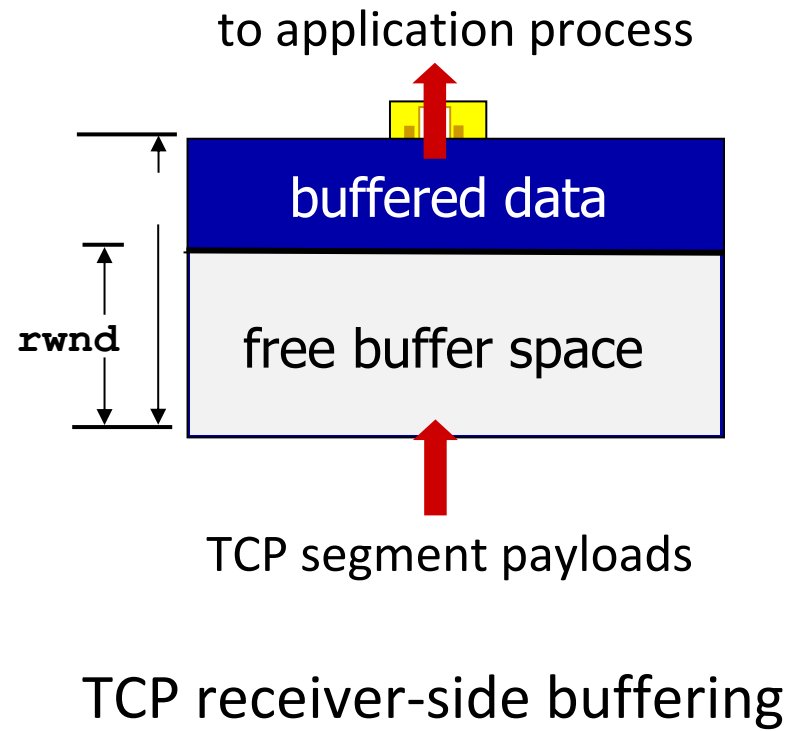
TCP congestion control: details

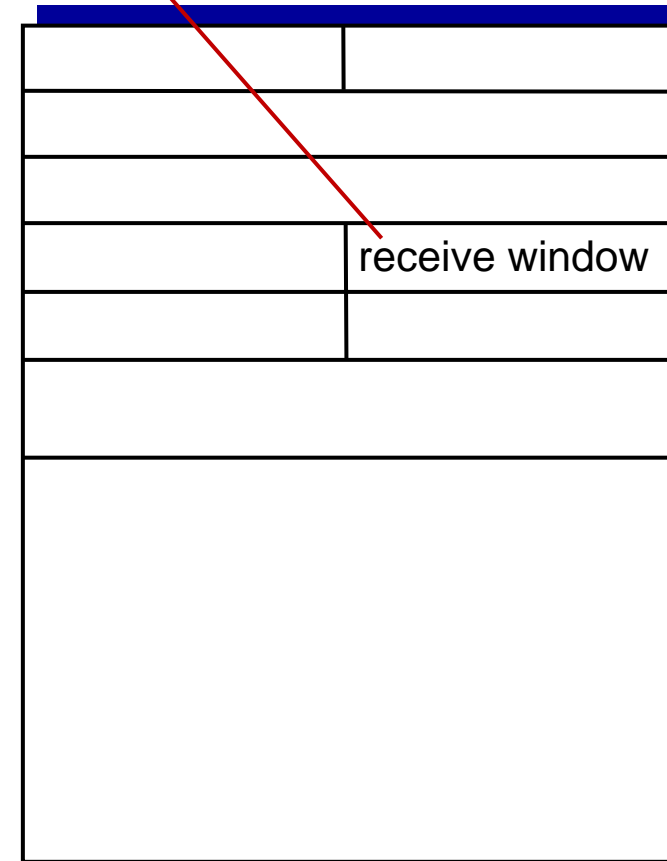sender sequence number space

cwnd

last byte ACKed

sent, but not-yet ACKed ("in-flight")

last byte sent

available but not used

TCP sending behavior:

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

TCP rate $\approx \dfrac{\mathtt{cwnd}}{\mathrm{RTT}}$ bytes/sec

- TCP sender limits transmission: `LastByteSent- LastByteAcked` $\leq$ `cwnd`

- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

to application process

buffered data

rwnd

free buffer space

TCP segment payloads

TCP receiver-side buffering

flow control: # bytes receiver willing to accept

receive window

TCP segment format

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

Host A   Host B

RTT

one segment

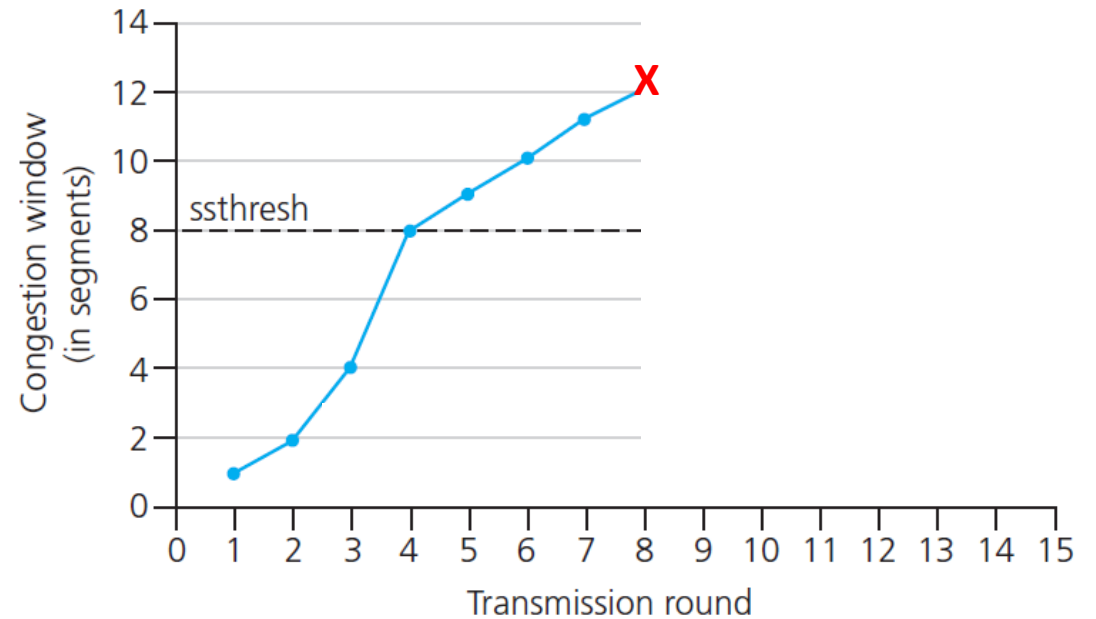two segments

four segments
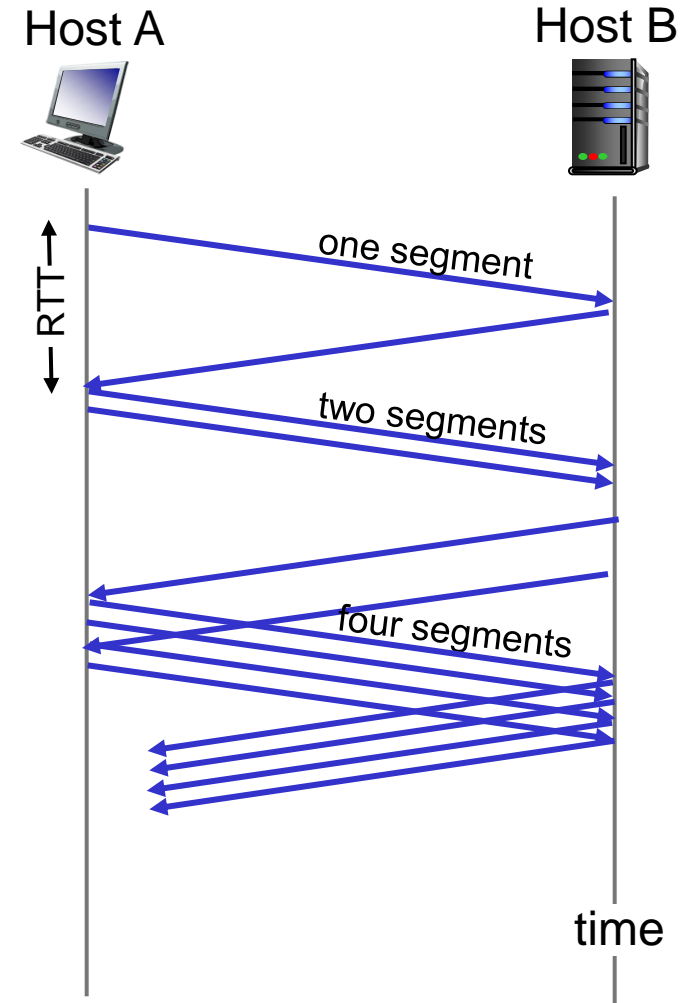
time

TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?
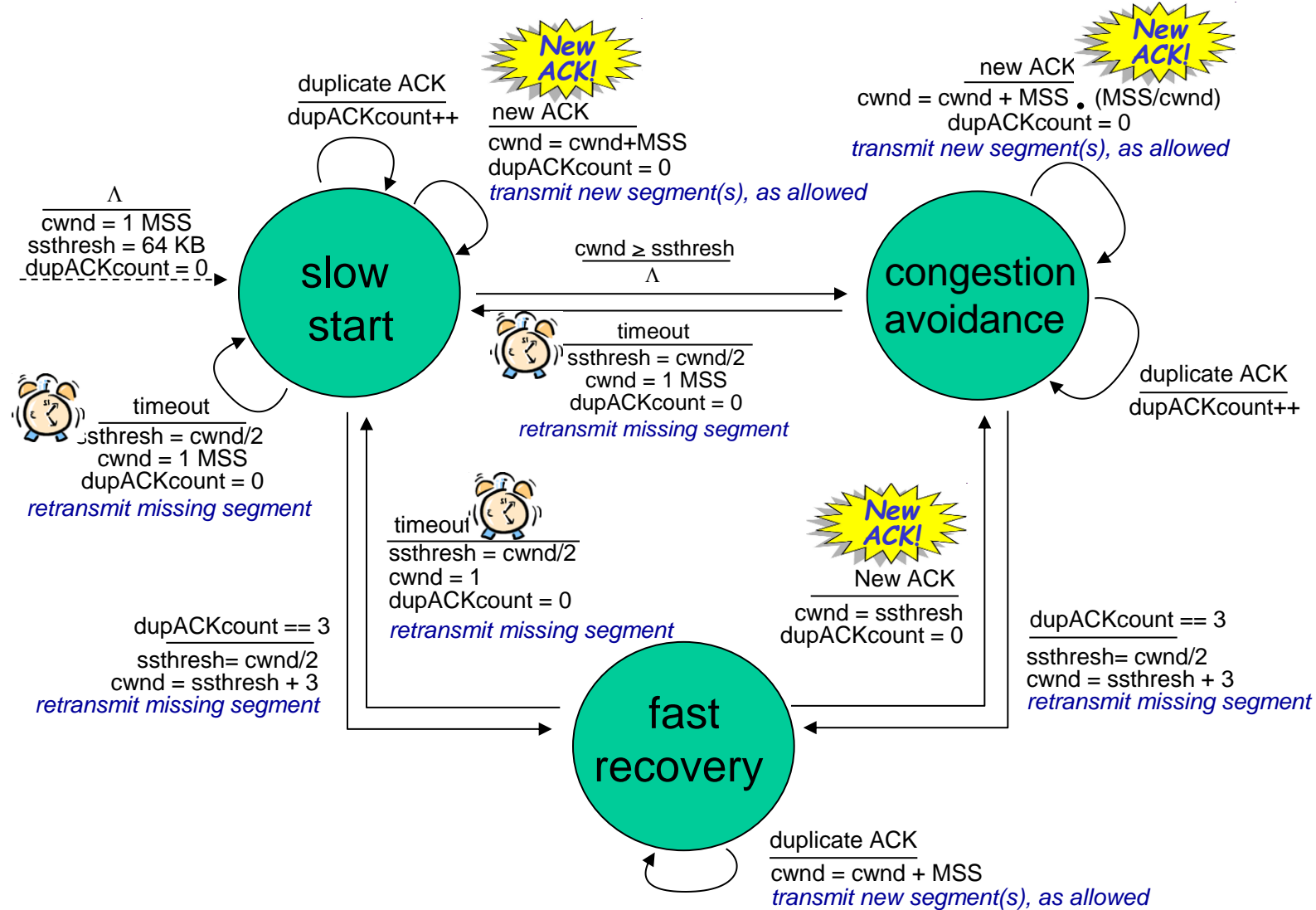
A: when **cwnd** gets to 1/2 of its value before timeout.

Implementation:

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

Summary: TCP congestion control

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

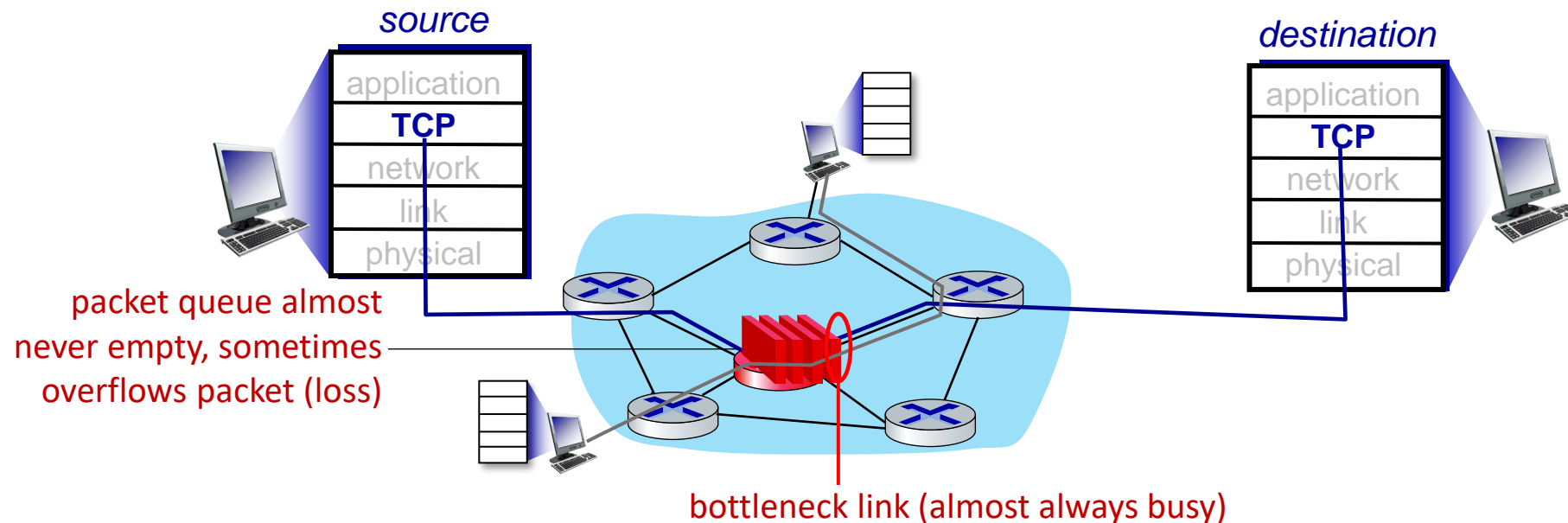TCP CUBIC - higher throughput in this example
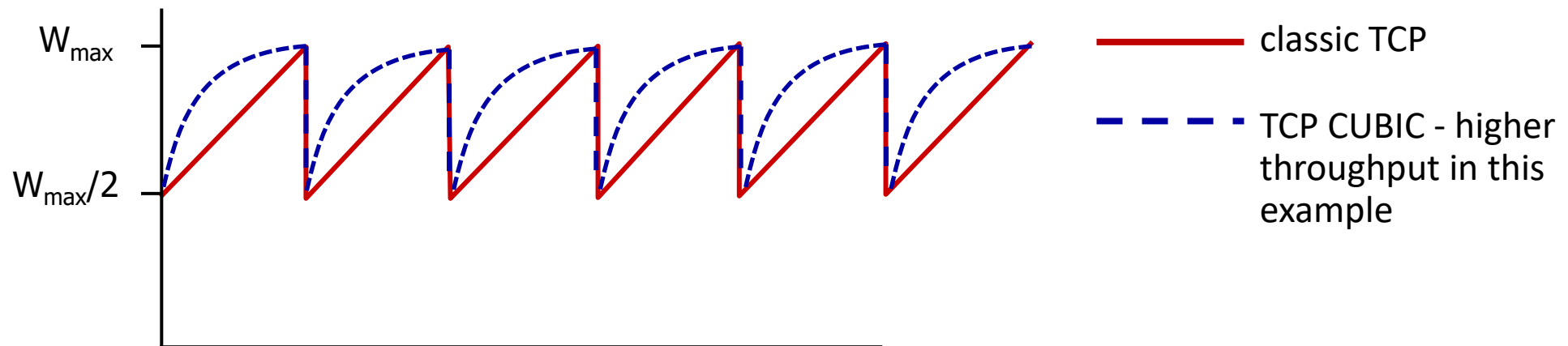
TCP CUBIC

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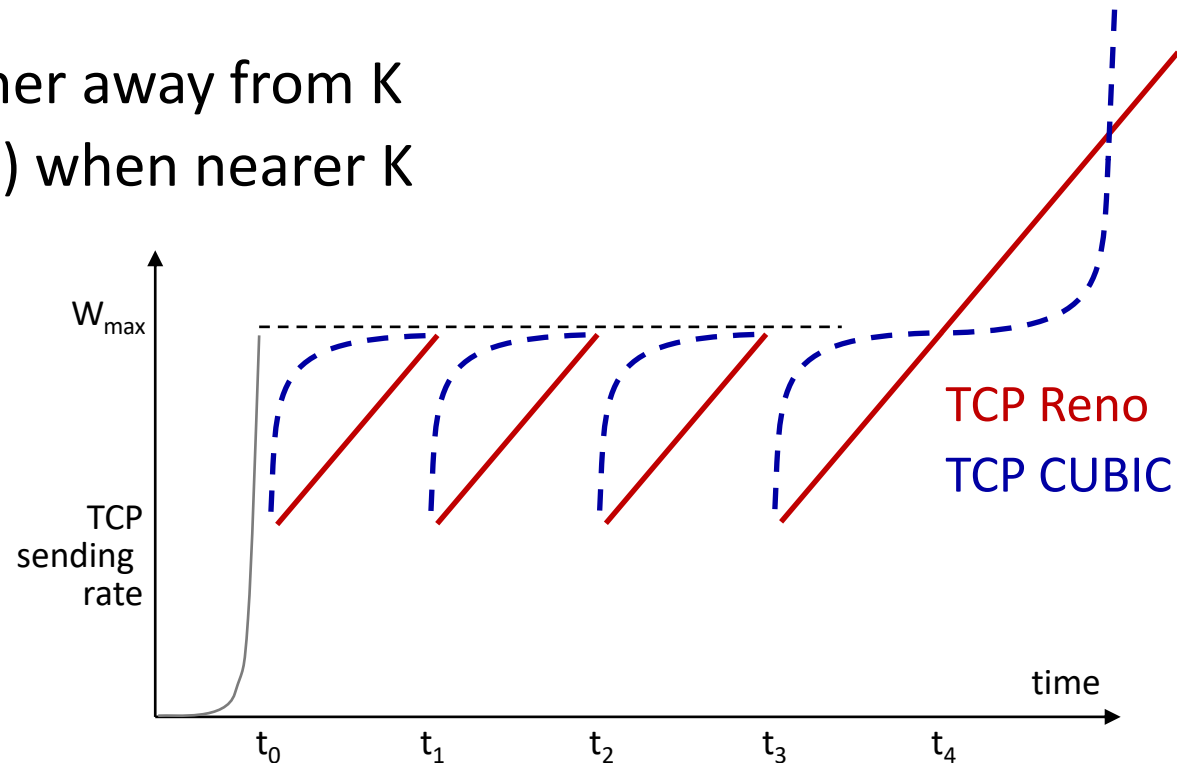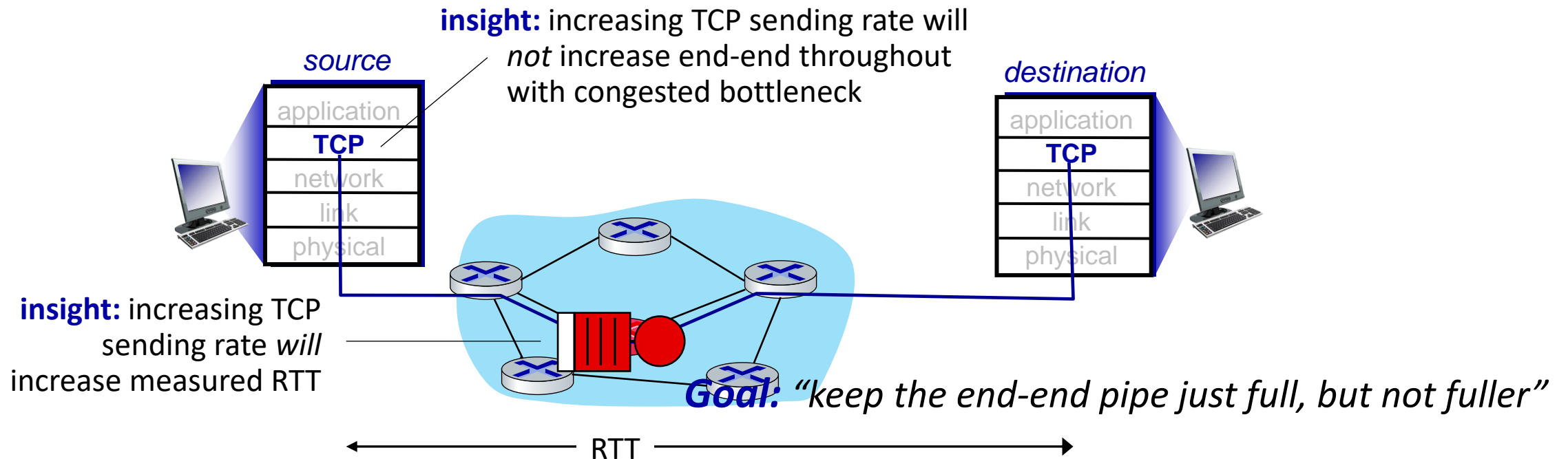

TCP Reno

TCP CUBIC

W_{max}

TCP sending rate

time

t_0 t_1 t_2 t_3 t_4

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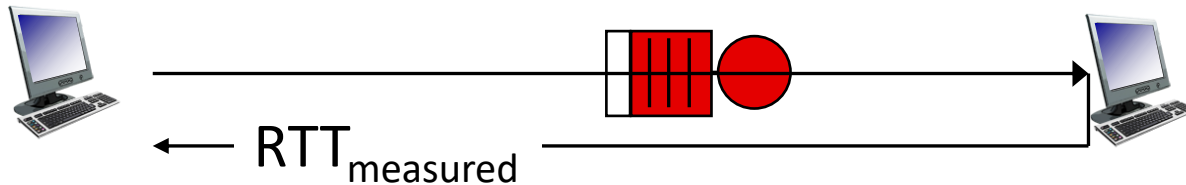
$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

$RTT_{measured}$

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)
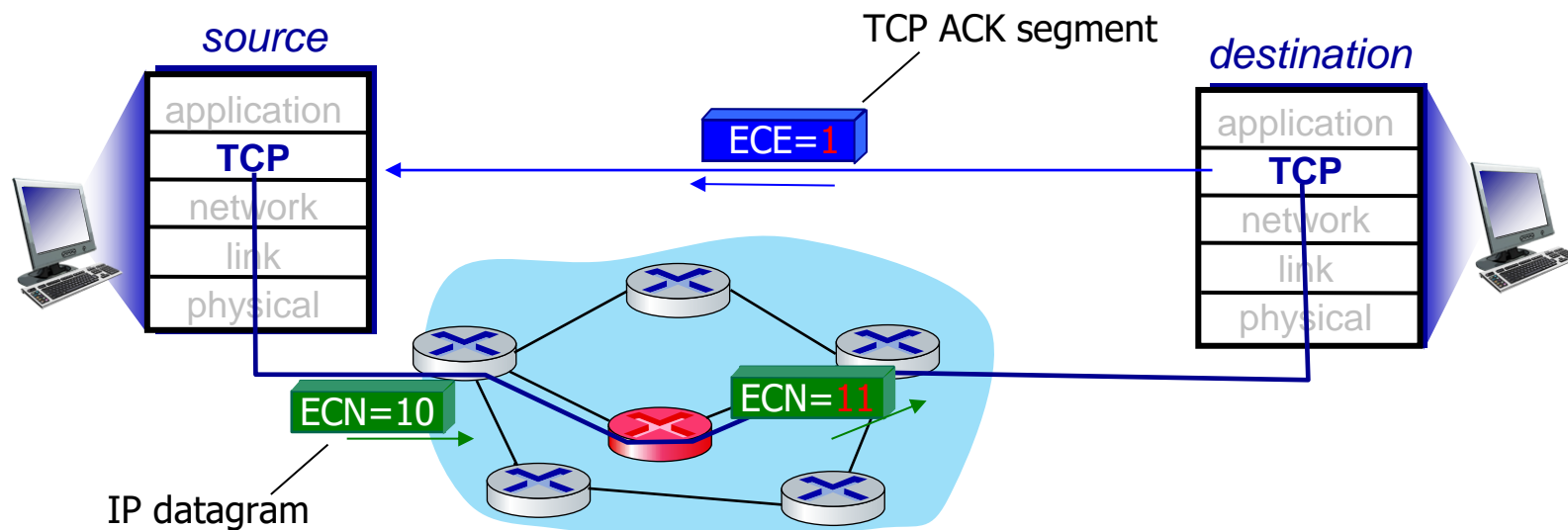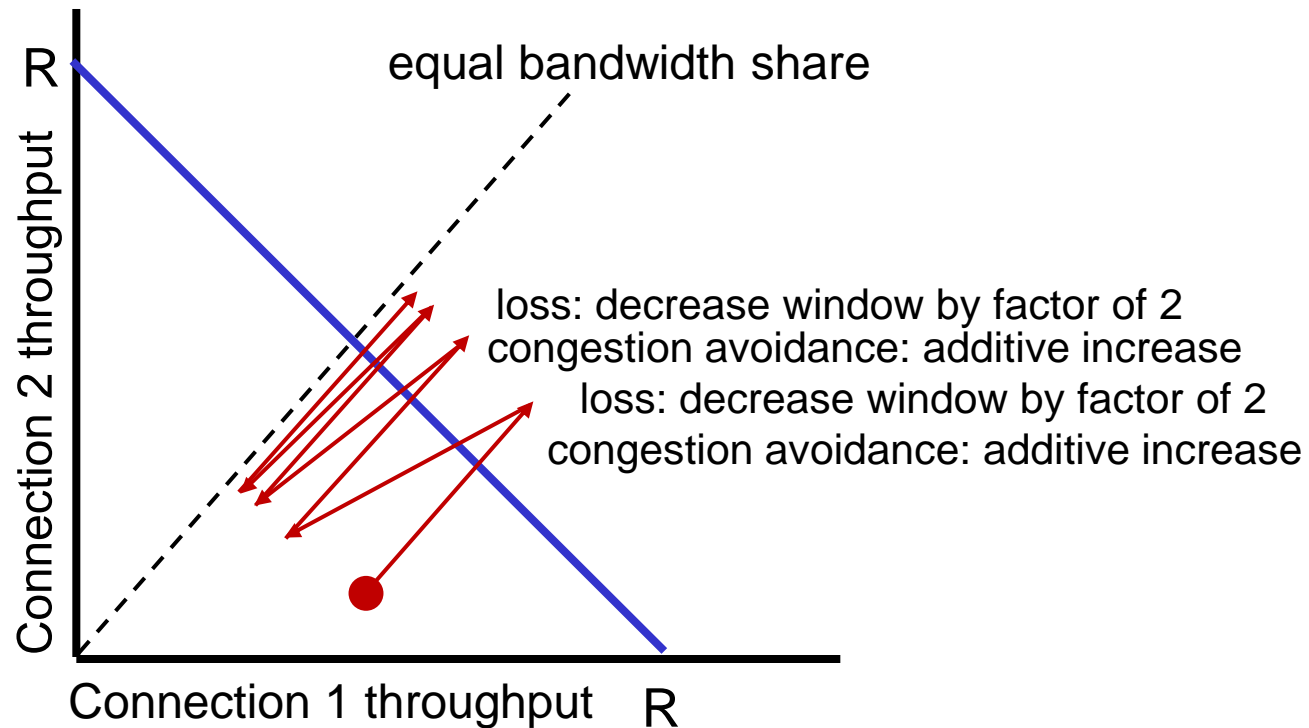
TCP fairness

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
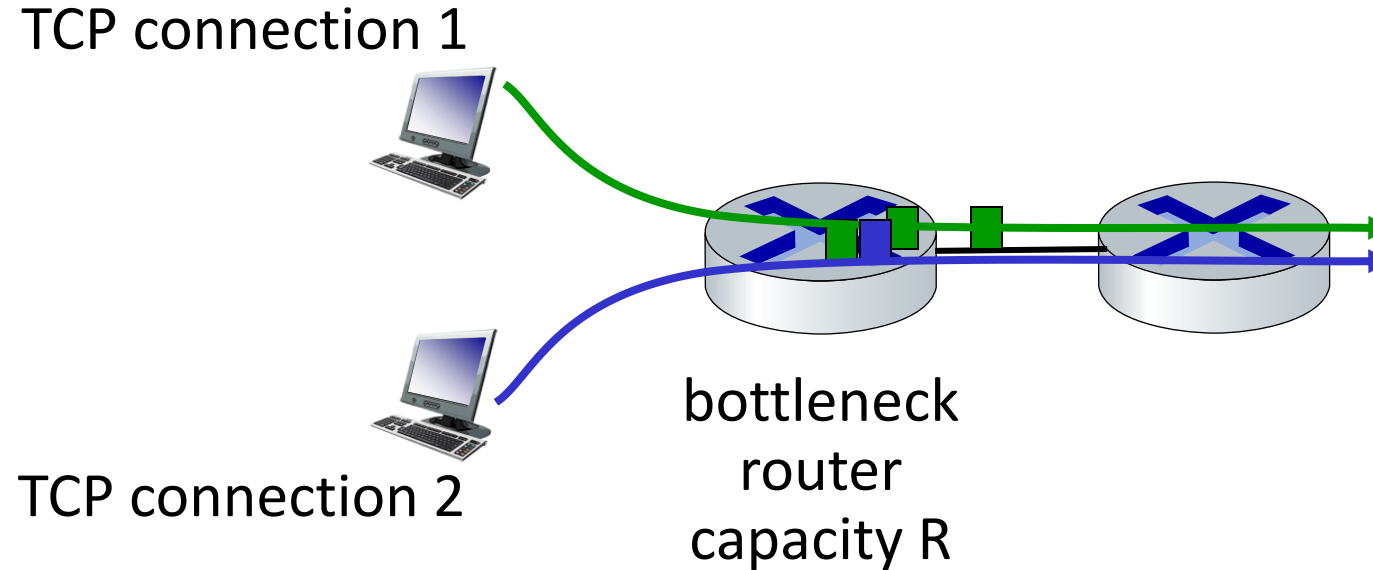
Q: is TCP Fair?

Example: two competing TCP sessions:

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

equal bandwidth share

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

Connection 2 throughput

Connection 1 throughput

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

Transport layer: 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**

Evolving transport-layer functionality

- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets "in flight"; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, "background" TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

■ application-layer protocol, on top of UDP
  • increase performance of HTTP
  • deployed on many Google servers, apps (Chrome, mobile YouTube app)

|	
	HTTP/2
Application	TLS
Transport	TCP
Network	IP

HTTP/2 over TCP
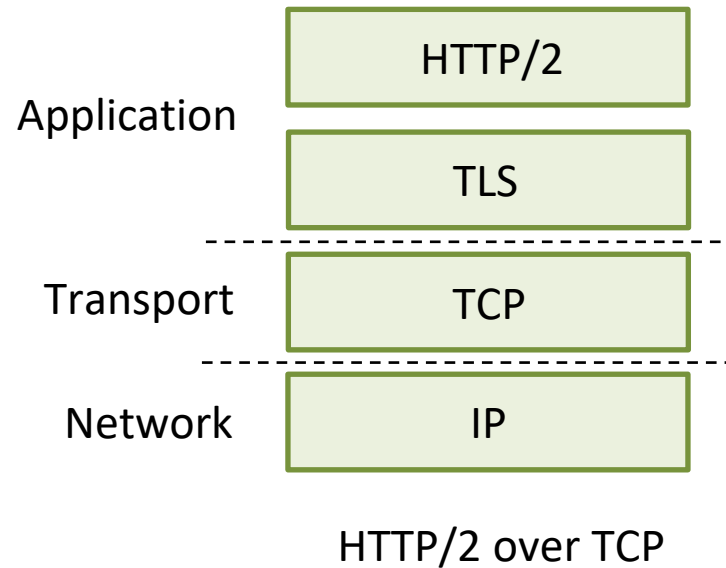
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment
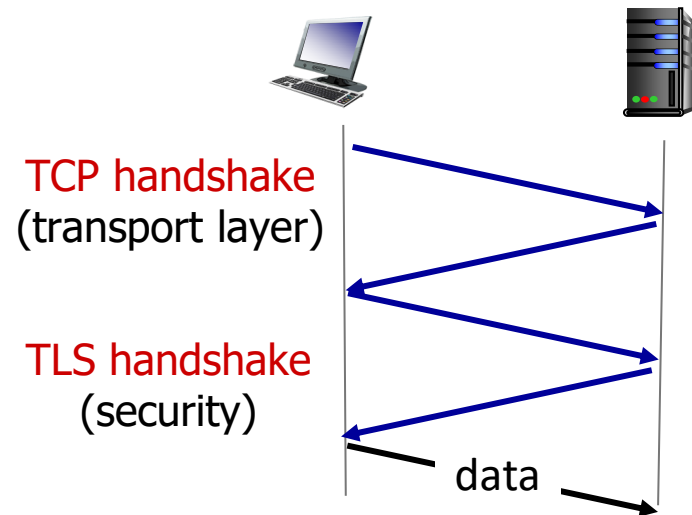
TCP (reliability, congestion control state) + TLS (authentication, crypto state)
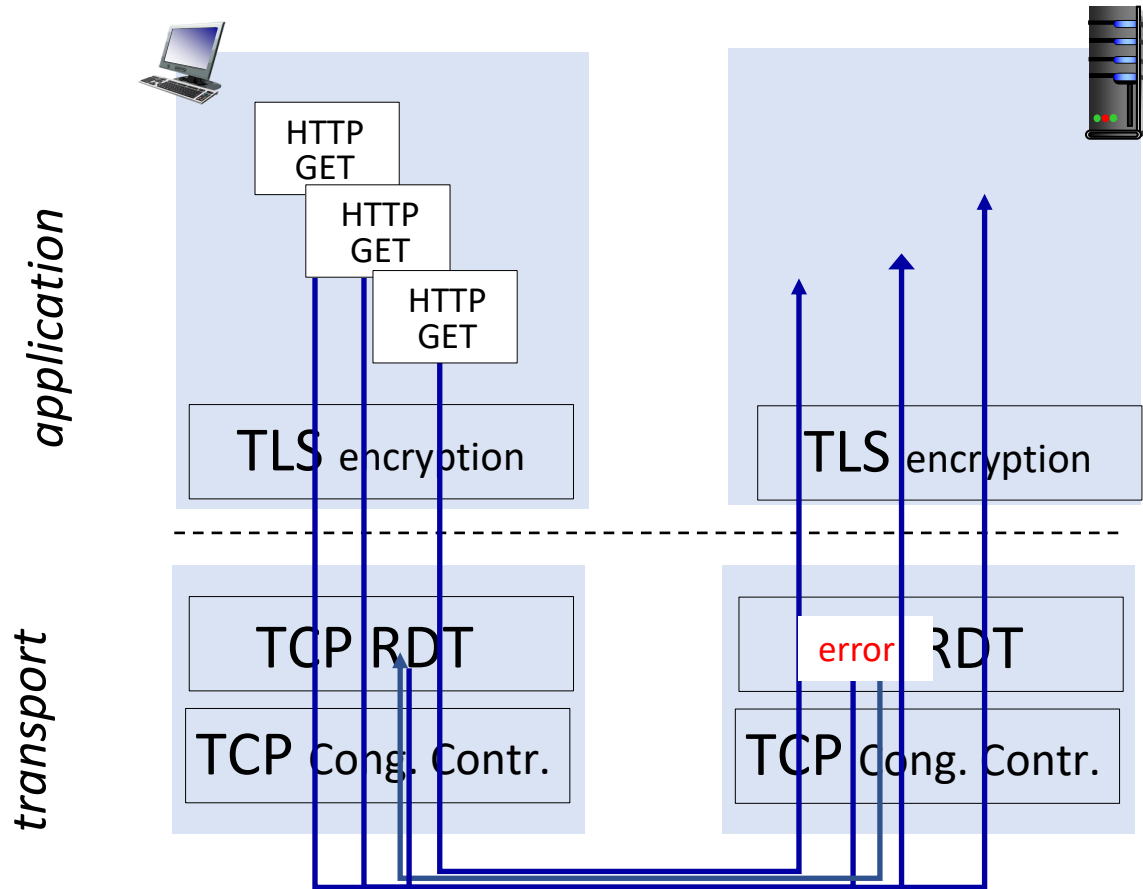
- 2 serial handshakes
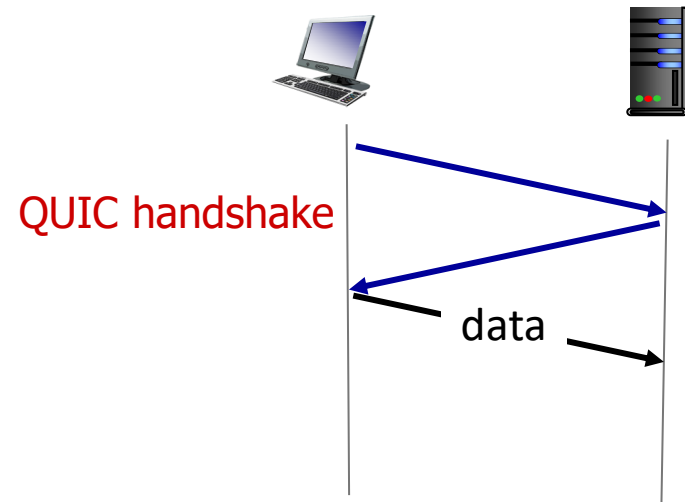
QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking

(a) HTTP 1.1

Chapter 3: summary

- **principles behind transport layer services:**
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- **instantiation, implementation in the Internet**
  - UDP
  - TCP

Up next:

- **leaving the network "edge"** (application, transport layers)
- **into the network "core"**
- **two network-layer chapters:**
  - data plane
  - control plane