UNIVERSITY OF WATERLOO

CS 456/656
Computer Networks

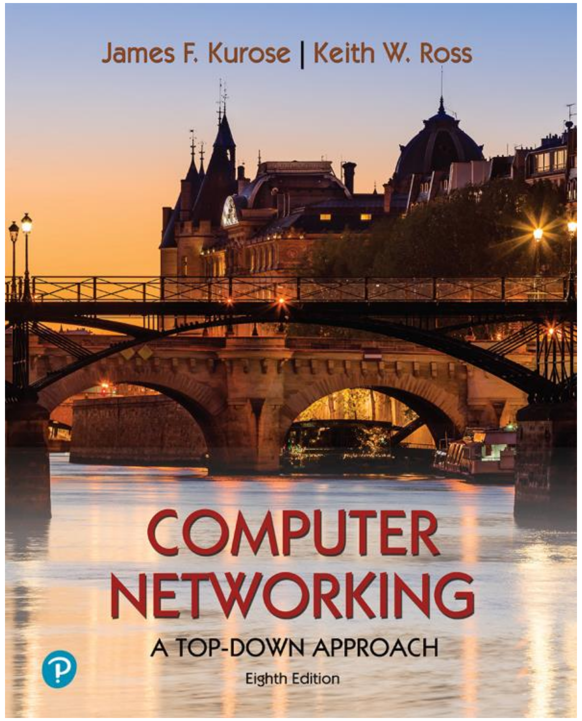
Lecture 8: Transport Layer – Part 4

Mina Tahmasbi Arashloo and Bo Sun

Fall 2024

A note on the slides

Adapted from the slides that accompany this book.



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport layer: roadmap

- Transport-layer overview
- 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:

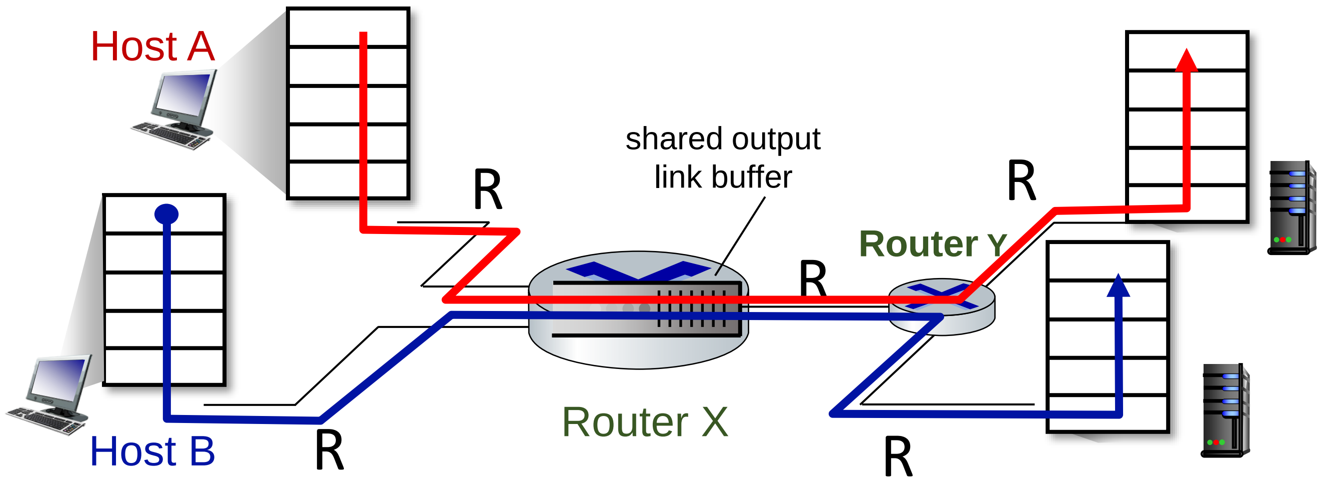
- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

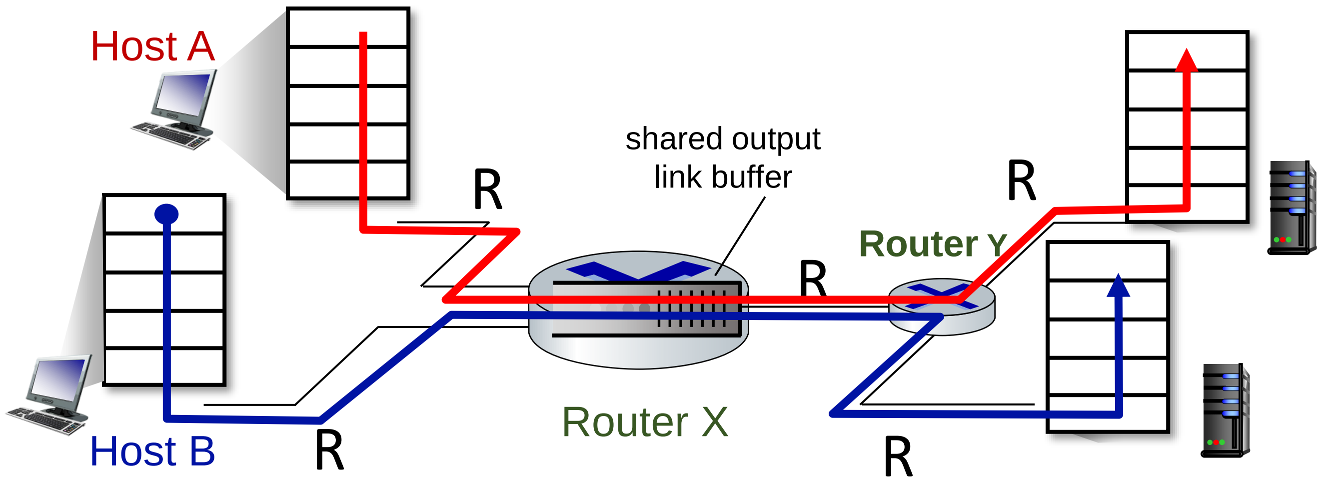
Throughput cannot exceed available capacity

- The transmission rate for all links is R bps
- So, if host A wants to send out data at R bps, the link can carry it to the router
- But, A has to share the link between Router X and Router Y with the traffic from Host B



Throughput cannot exceed available capacity

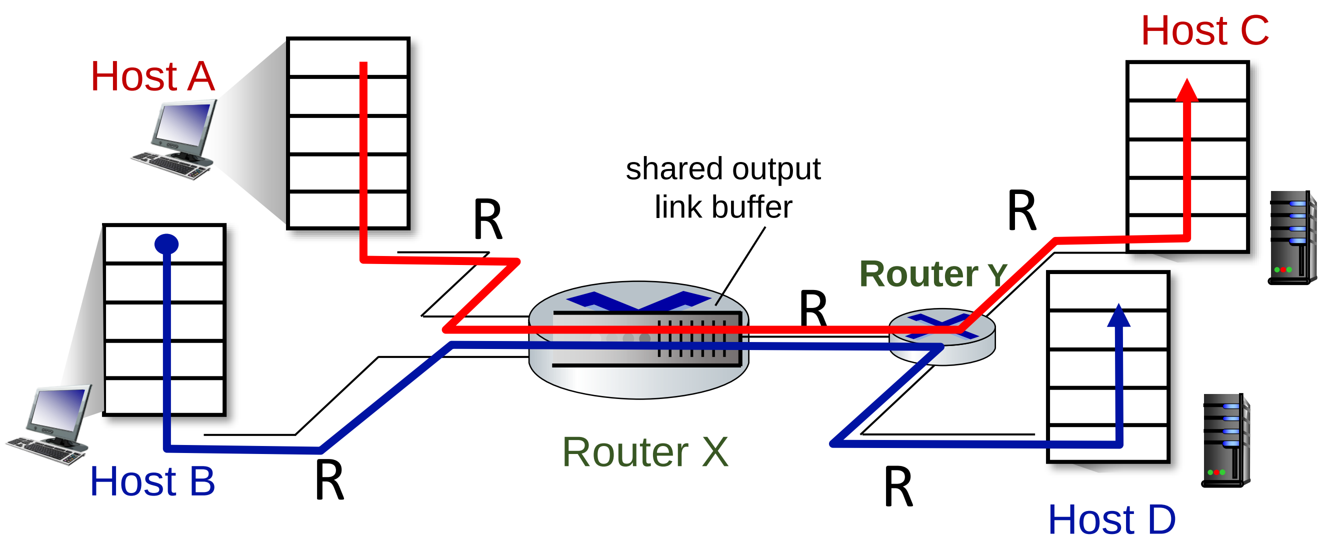
- Q. What happens if both Host A and Host B send data to their destinations at R bps?
  - Suppose the available bandwidth from Router X Router Y is shared fairly between traffic from A and B.



Throughput cannot exceed available capacity

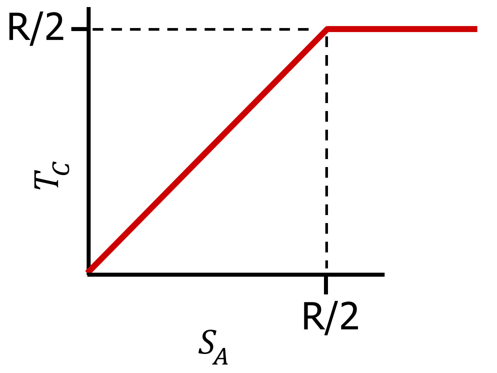
- No matter how fast A and B send data to the router, the router's bandwidth to Y is limited to R.
- So, host C can receive at most R/2 bps from A, and so does Host D from B
- In the best case, all the R/2 bits every second are sent exactly once
  - whatever is sent, it is delivered the first time

- So, in the best case, the throughput at which data is received by the application running in Host C is R/2 bps.

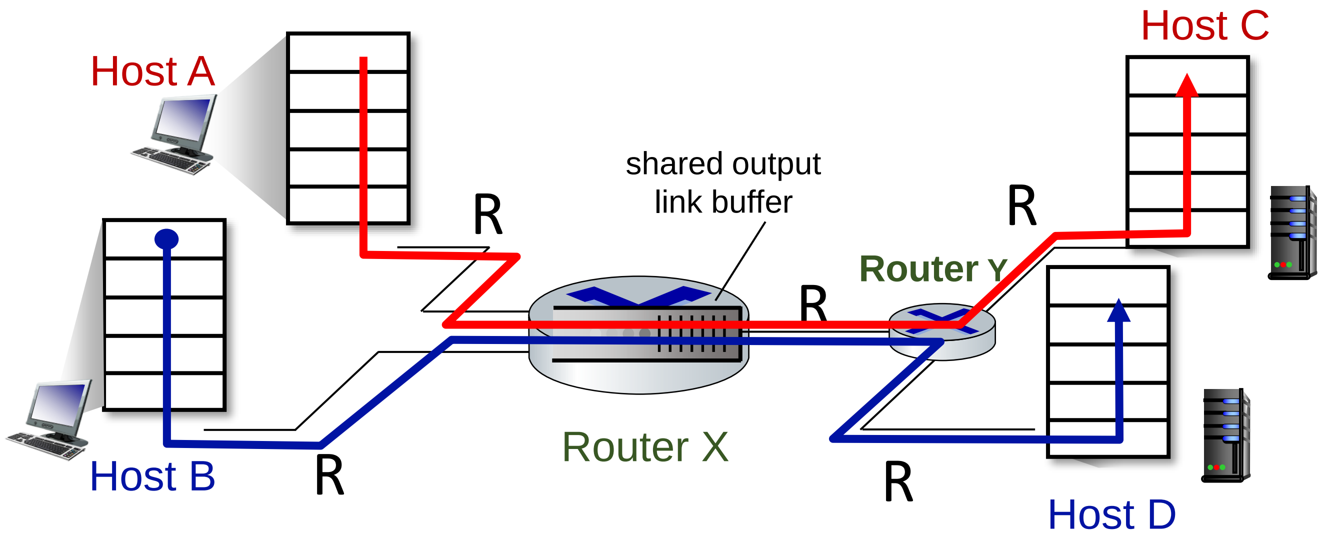


Throughput cannot exceed available capacity

- S_A: the rate at which host A sends data out.
- T_C: the rate at which new data is received by the application.
- Best case scenario: As S_A increases, T_C increases up to R/2.
  - $T_C = \min(S_A, \frac{R}{2})$
- Throughput can never exceed available capacity.

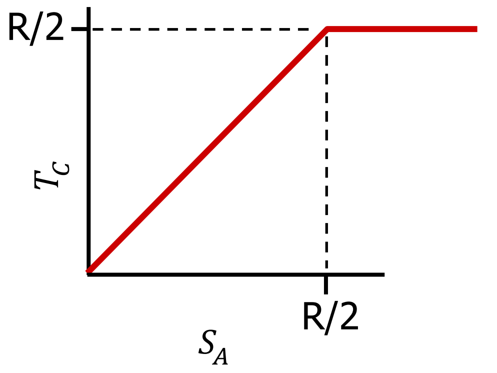


maximum per-connection throughput: R/2

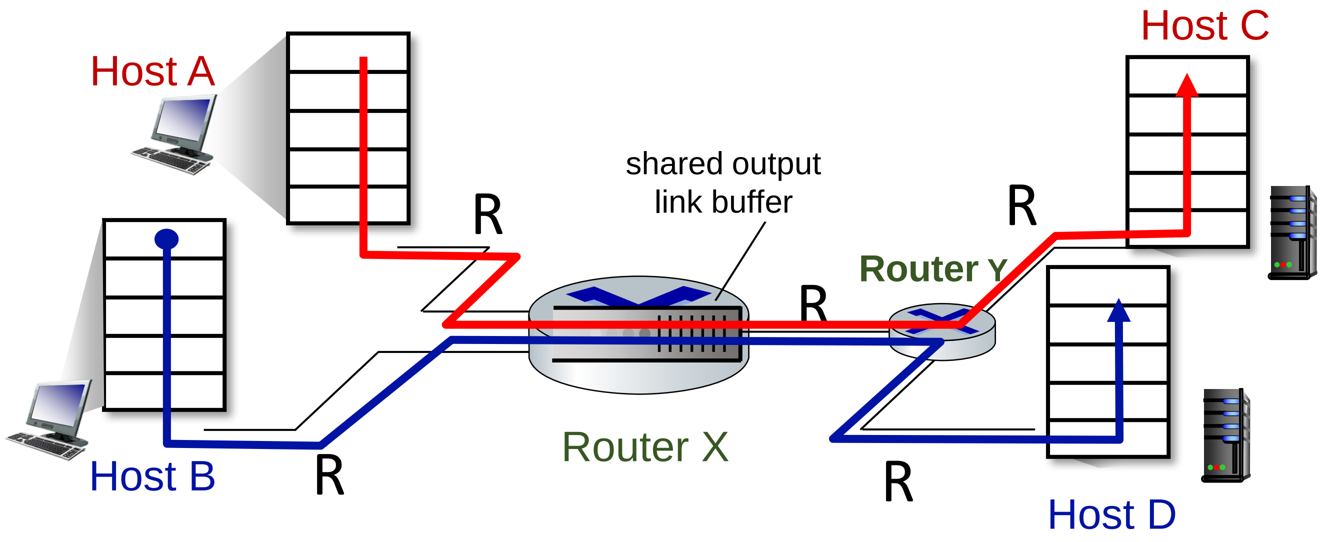


Ideal case 1: Infinite buffers

- When would this best case happen?
  - The buffer at Router X has infinite capacity.
  - So, no packets are dropped, they may just take longer and longer to get to Host C. (Why?)
  - No packet drops ⇨ all the R/2 bits per second getting to Host C have been sent exactly once.

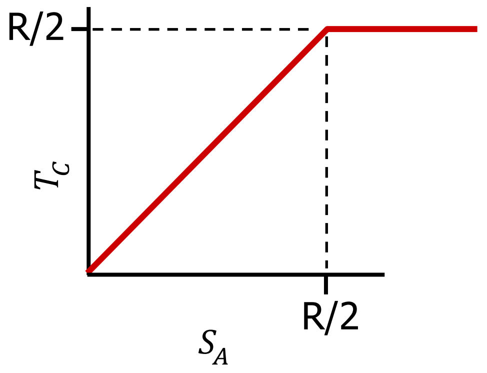


maximum per-connection throughput: R/2

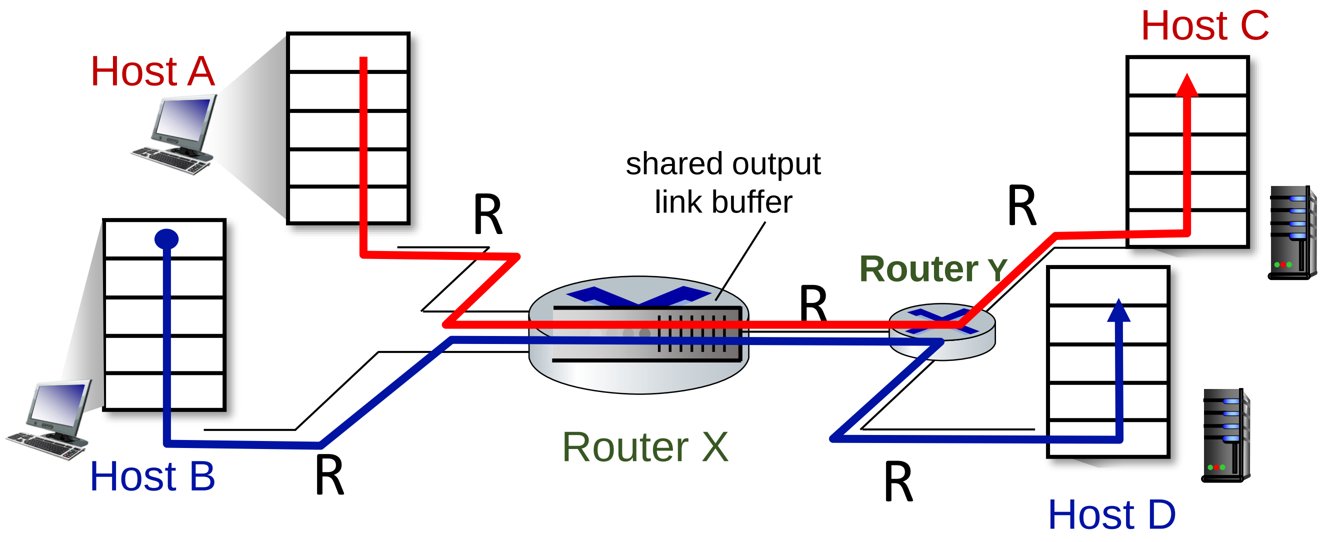


Ideal case 2: Finite buffers but perfect knowledge of capacity

- Could there be no packet loss if the buffer is finite?
  - Yes, if Host A has perfect knowledge of the available buffer capacity.
  - That is, if Host A only sends when router buffers are available.



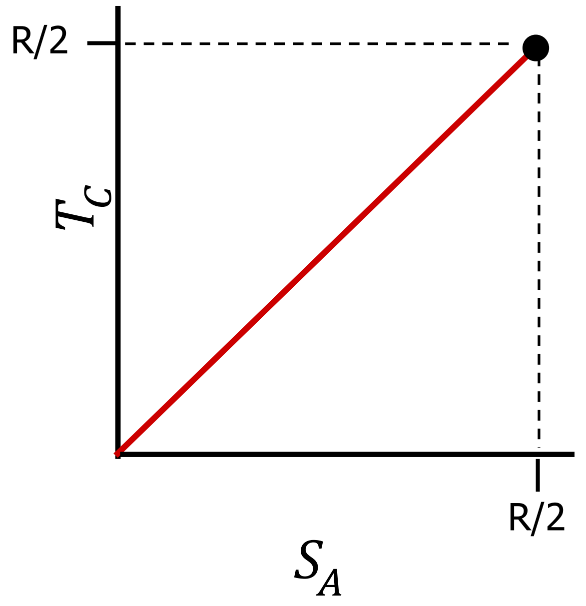
maximum per-connection throughput: R/2

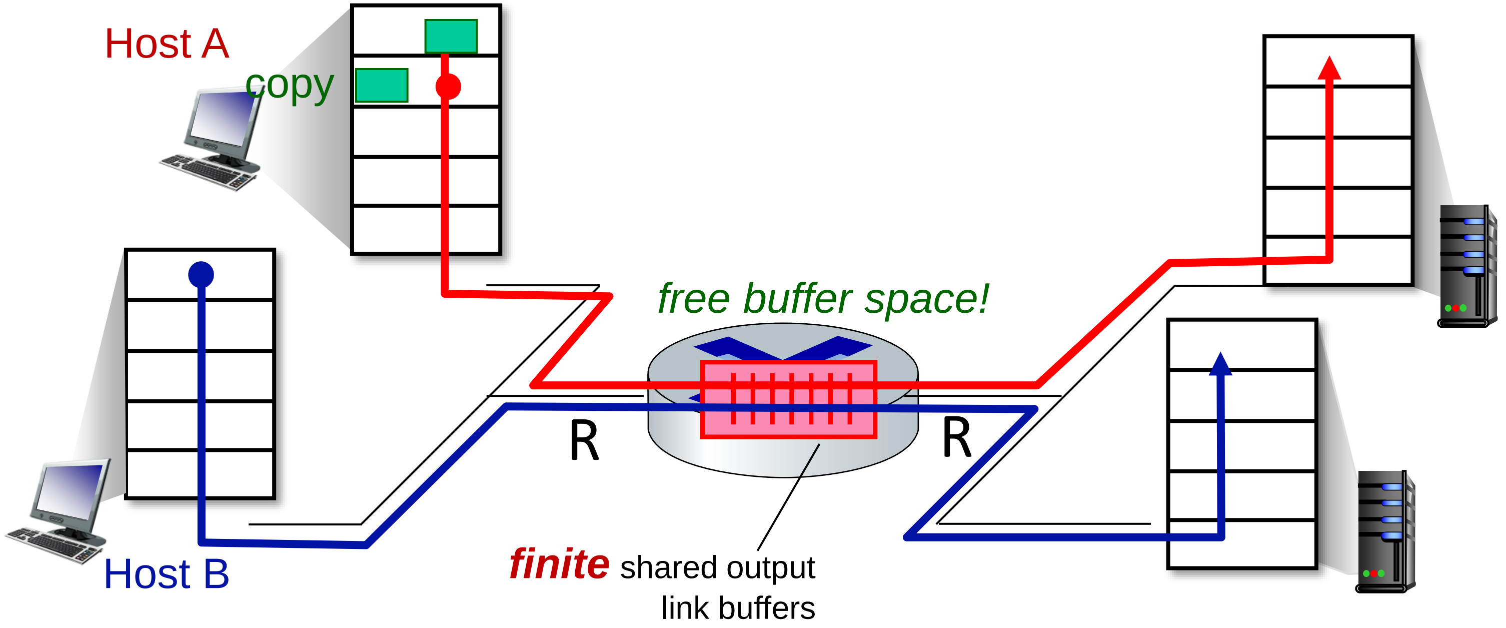


Ideal case 2: Finite buffers but perfect knowledge of capacity

Idealization: perfect knowledge

- sender sends only when router buffers available

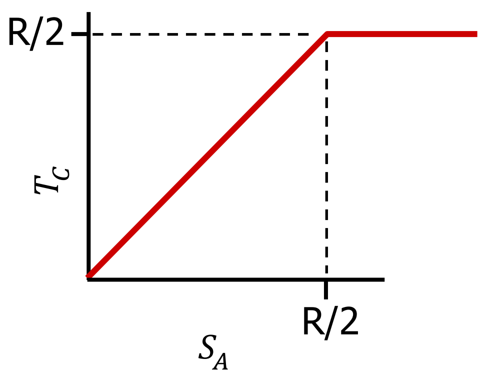




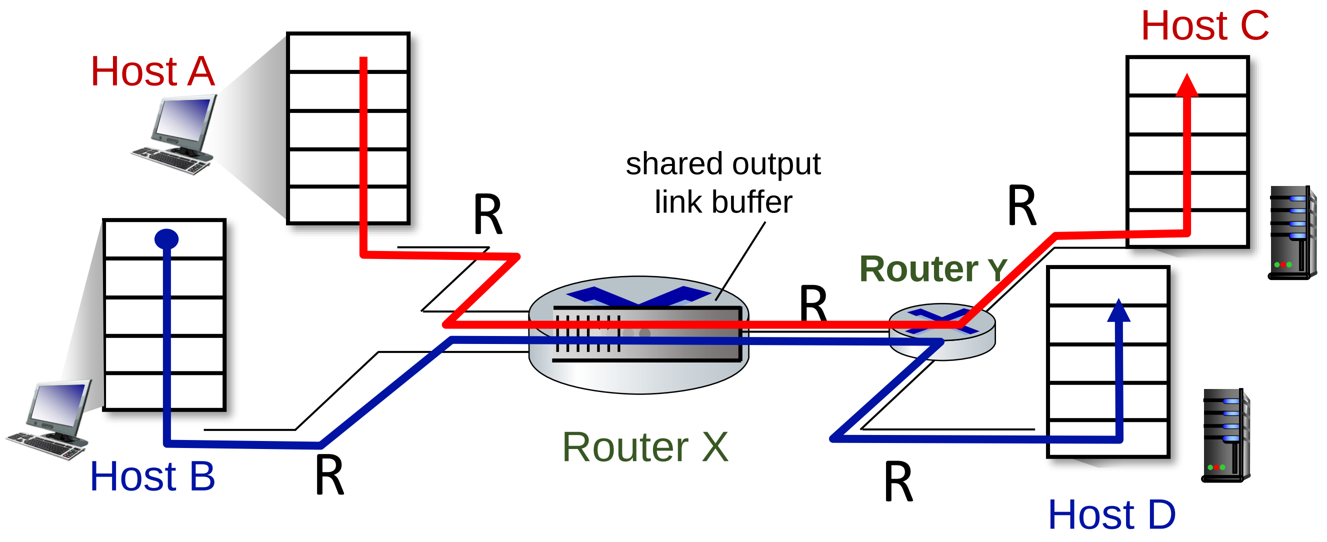
Ideal case 2: Finite buffers but perfect knowledge of capacity

- Could there be no loss if the buffer is finite?
  - Yes, if Host A has perfect knowledge of the available capacity.
  - That is, if Host A only sends when router buffers are available.
  - No packet drops all the R/2 bits per second getting to Host C have been sent exactly once.

Q. Can this ideal case happen in the Internet?
(hint: packet switching vs circuit switching)

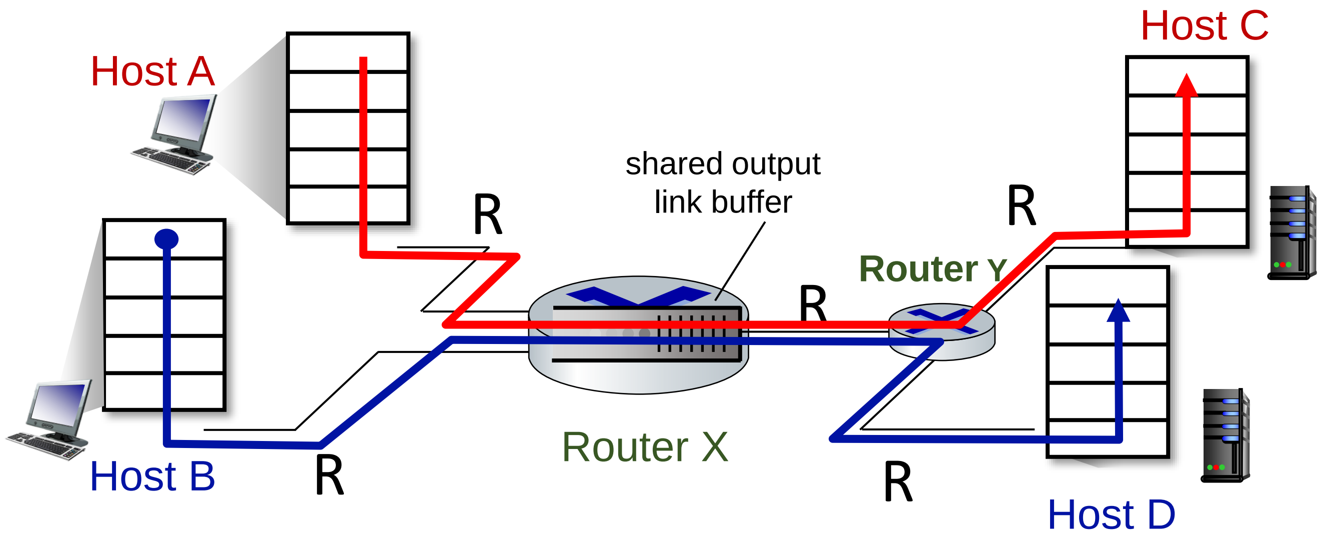


maximum per-connection throughput: R/2



What happens if packets are lost?

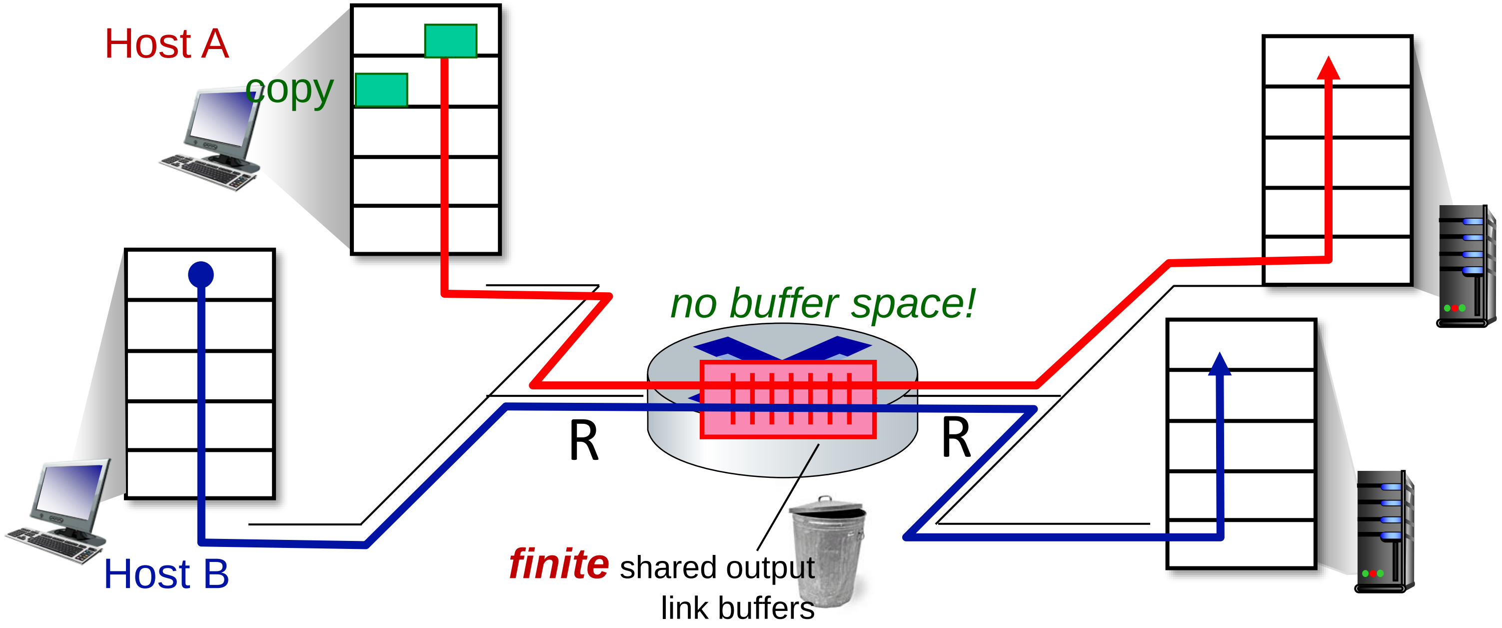
- In reality, host A may not have real-time information of the available buffer capacity.
- With reliable data transfer, if a packet is lost, the transport layer will retransmit the corresponding data segments.
- <u>Retransmission = Wasted capacity</u>
- Why?



What happens if packets are lost?

Idealization: *some* perfect knowledge

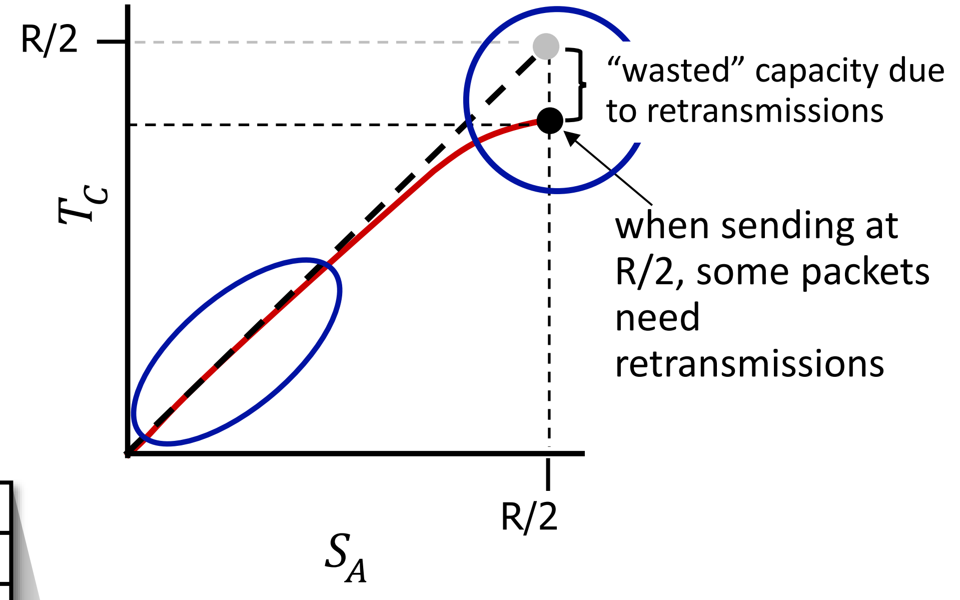
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

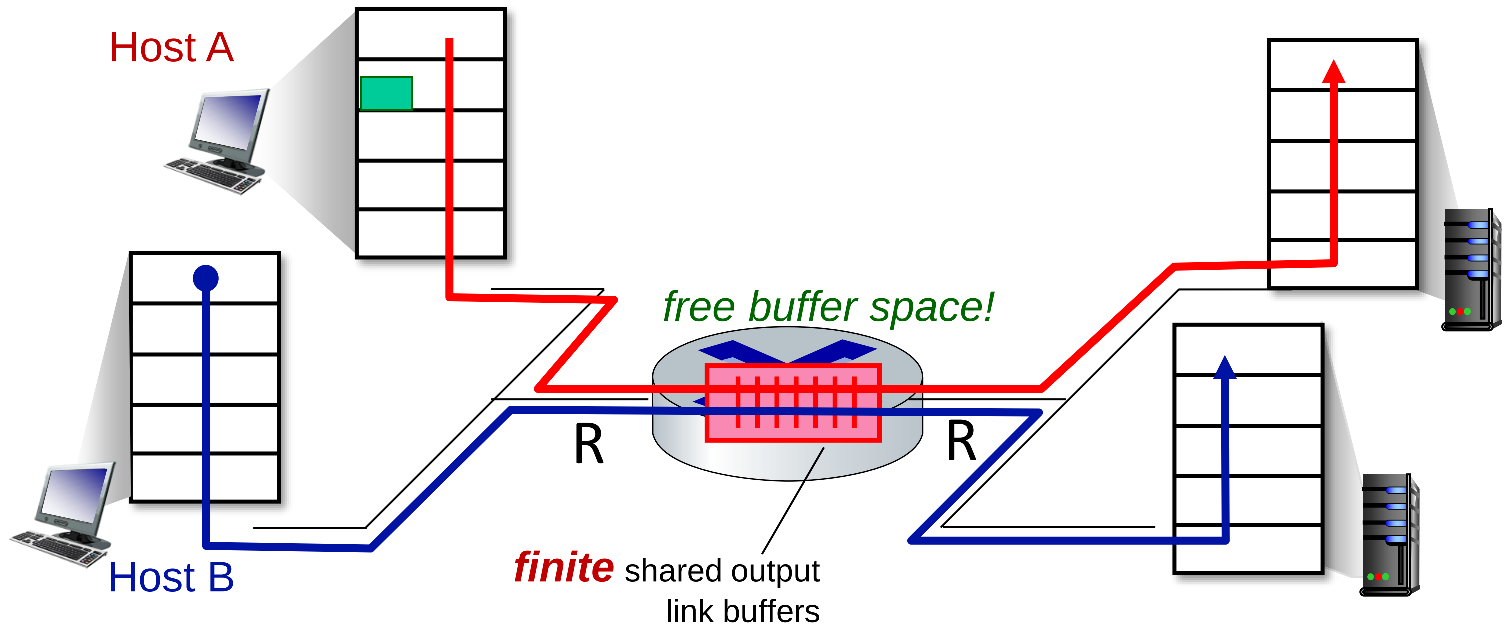


What happens if packets are lost?

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

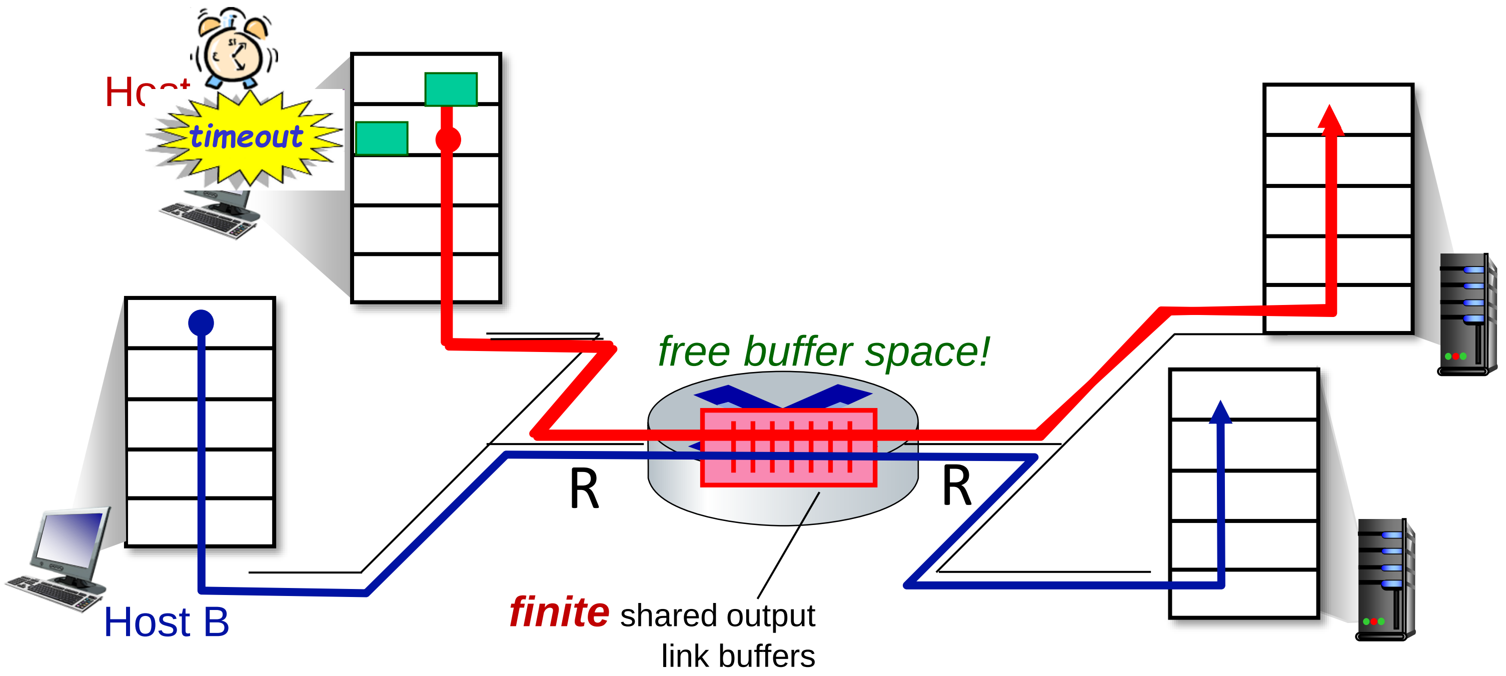


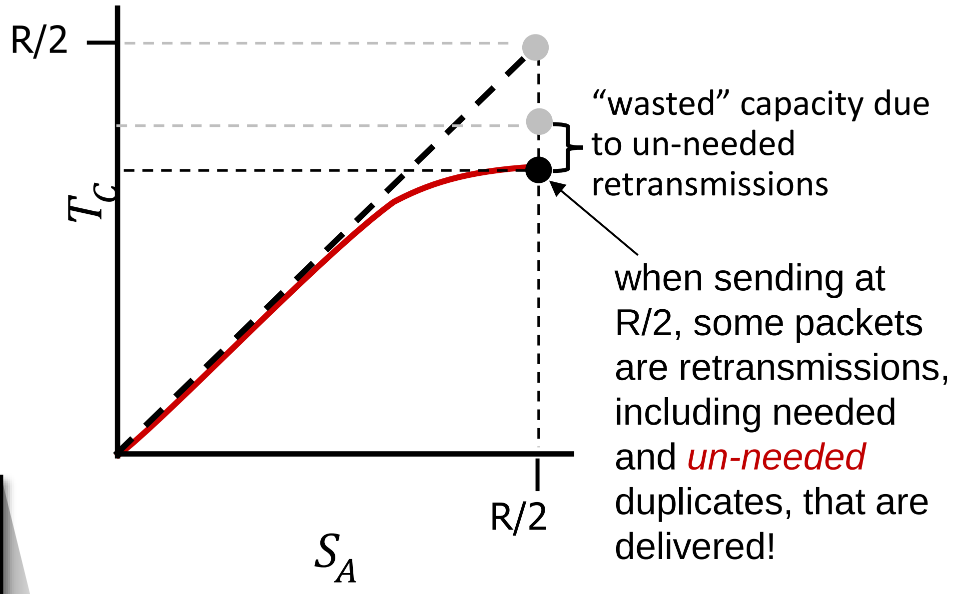


What happens if packets are lost?

Realistic scenario: *un-needed duplicates*

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

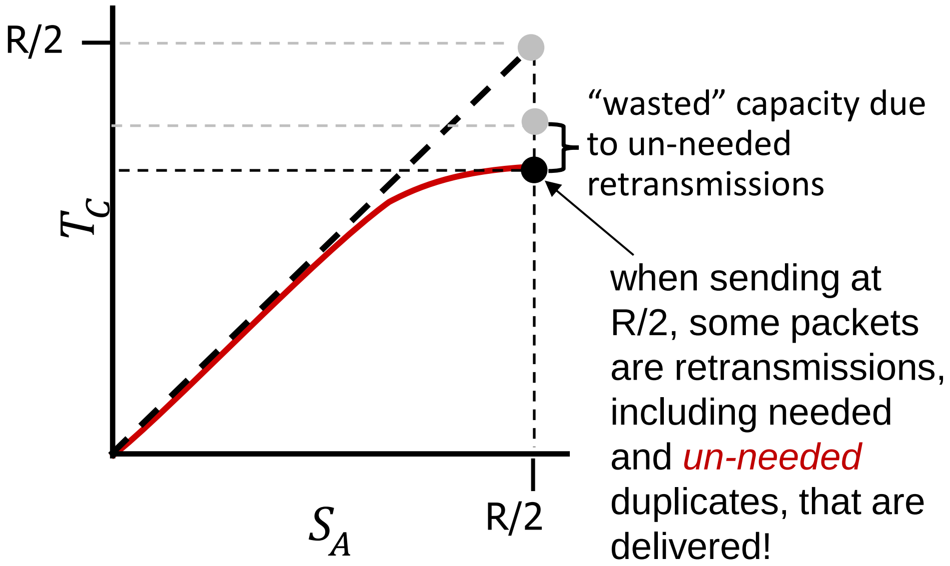




What happens if packets are lost?

Realistic scenario: *un-needed duplicates*

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



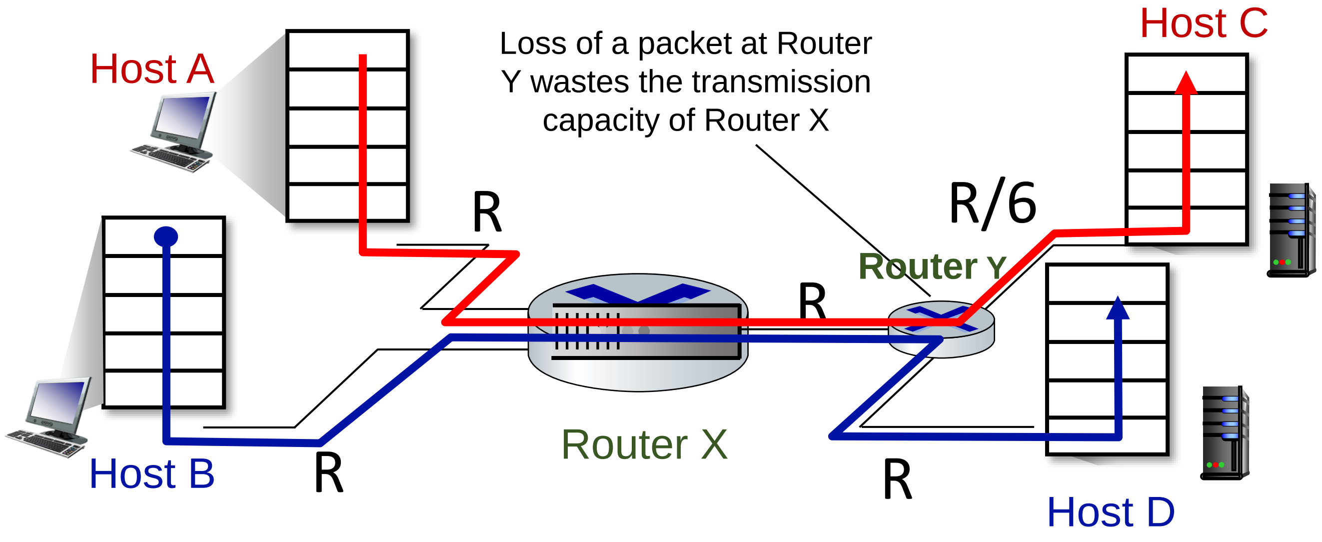
"costs" of congestion:

- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

What happens if packets are lost along a path?

Realistic scenario: *retransmissions triggered by loss throughout the network*

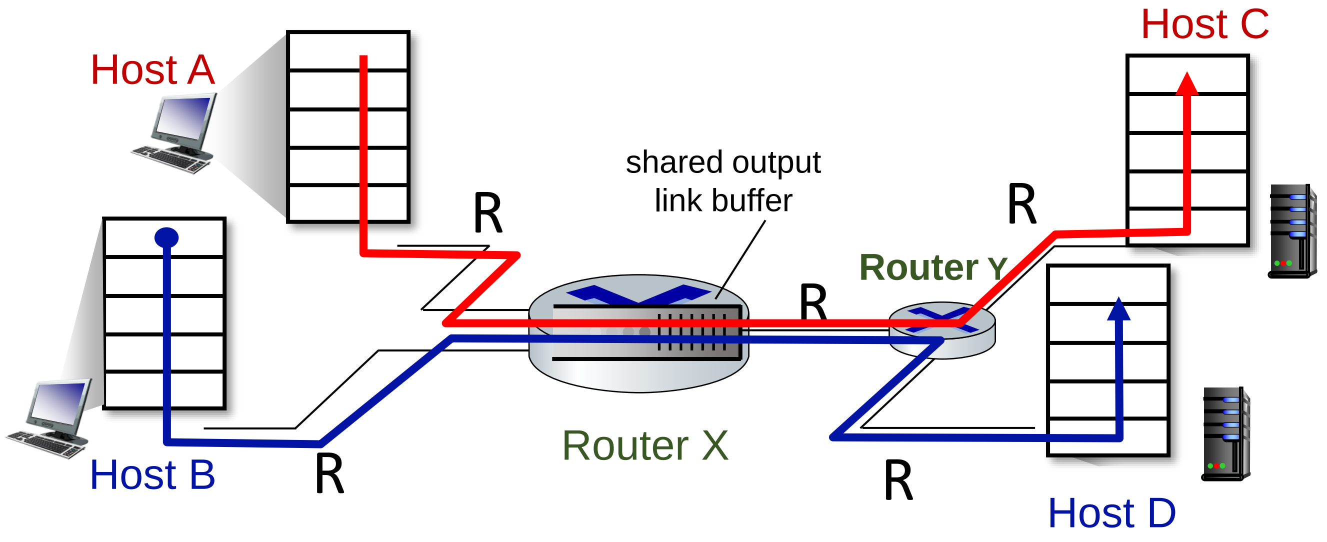
- whenever a packet is dropped at Router Y, the work done by Router X (buffering and forwarding) is wasted
- upstream transmission capacity / buffering wasted for packets lost downstream



- In extreme cases, this can lead to a situation called congestion collapse, where the network keeps carrying retransmitted packets, only for them to be dropped later in the path.
- No data gets delivered.
- This happened in the early days of the Internet!

How can we avoid congestion?

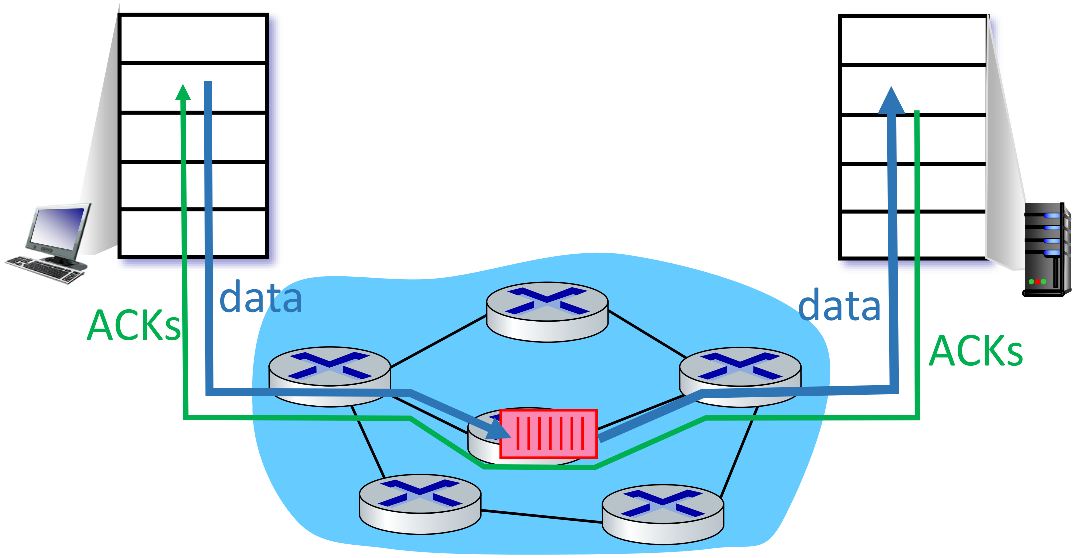
- Throughput can't exceed available capacity
- Sending over capacity ⇨ packet loss or long delays
- Packet loss or long delay ⇨ retransmission
- Retransmission ⇨ Wasted capacity
- Constant retransmission throughout the network ⇨ congestion collapse
- Congestion control: Have each sender estimate the available capacity in the network before sending, and only send out what the network can handle.



Approaches towards congestion control

End-end congestion control:

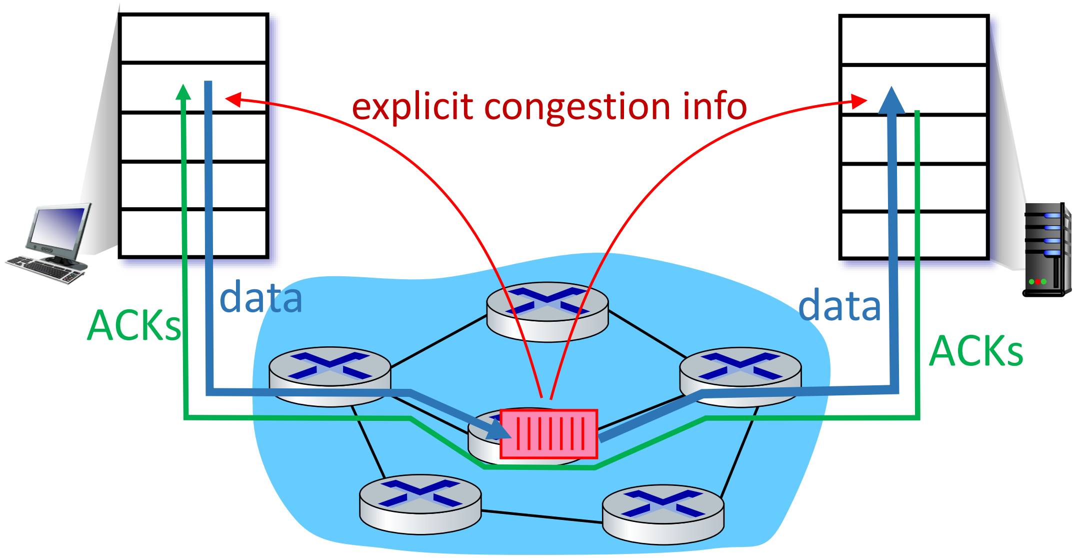
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



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



Discussion

- What if some senders decide to send more data than the available network capacity anyway?

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

TCP congestion control: AIMD

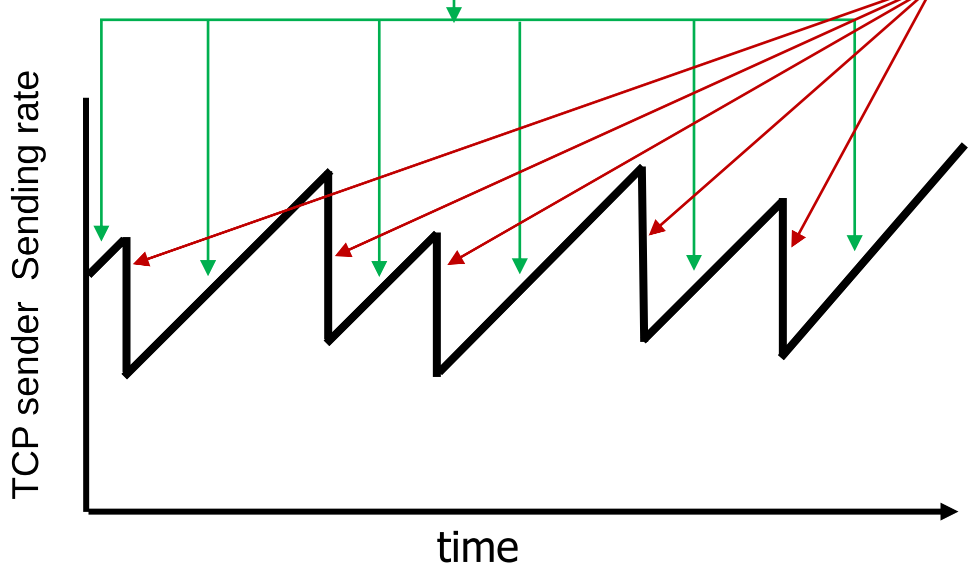
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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 AIMD: more

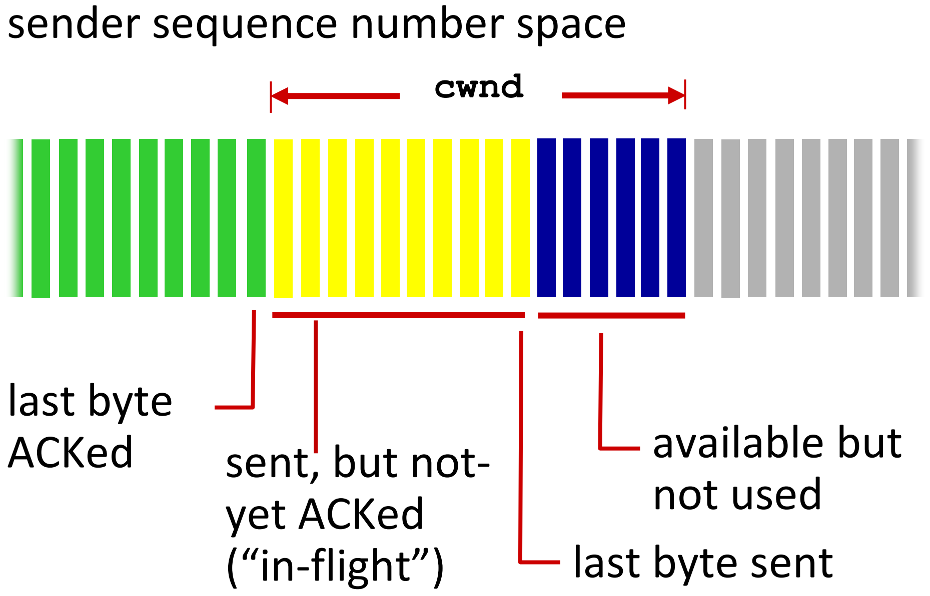
Multiplicative decrease detail: sending rate is

- 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



TCP sending behavior:

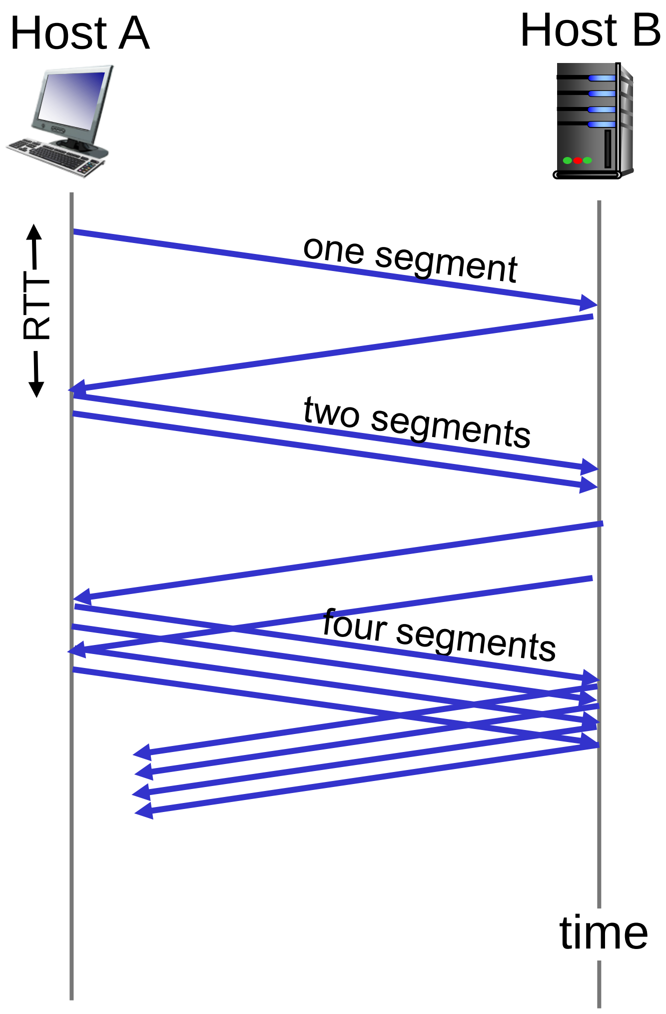
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: **`LastByteSent- LastByteAcked ≤ cwnd`**
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

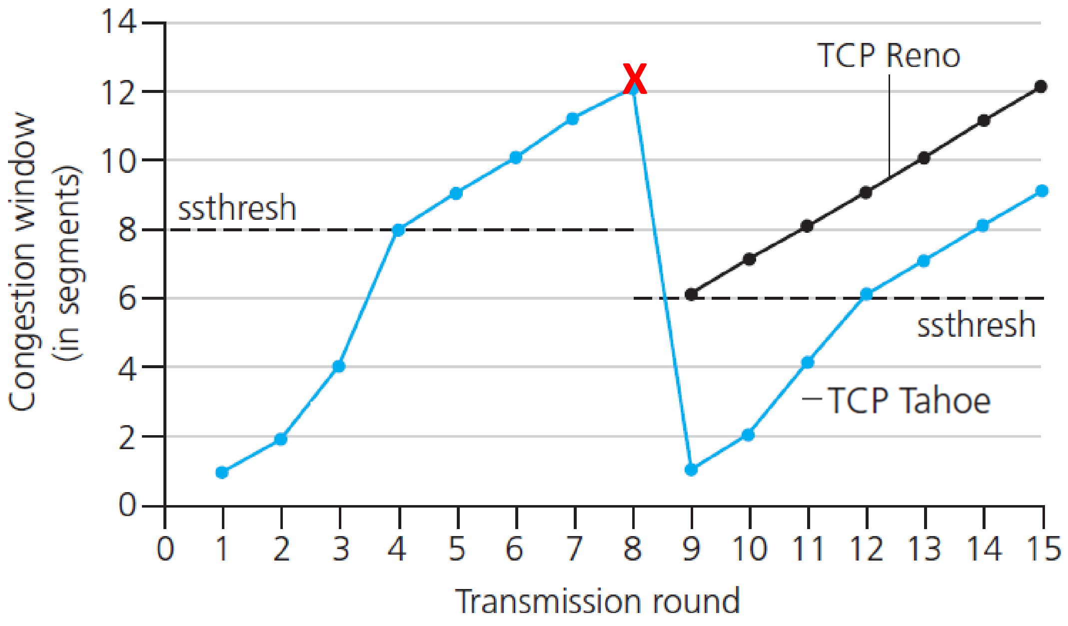
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



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/

TCP congestion control

Slow start:

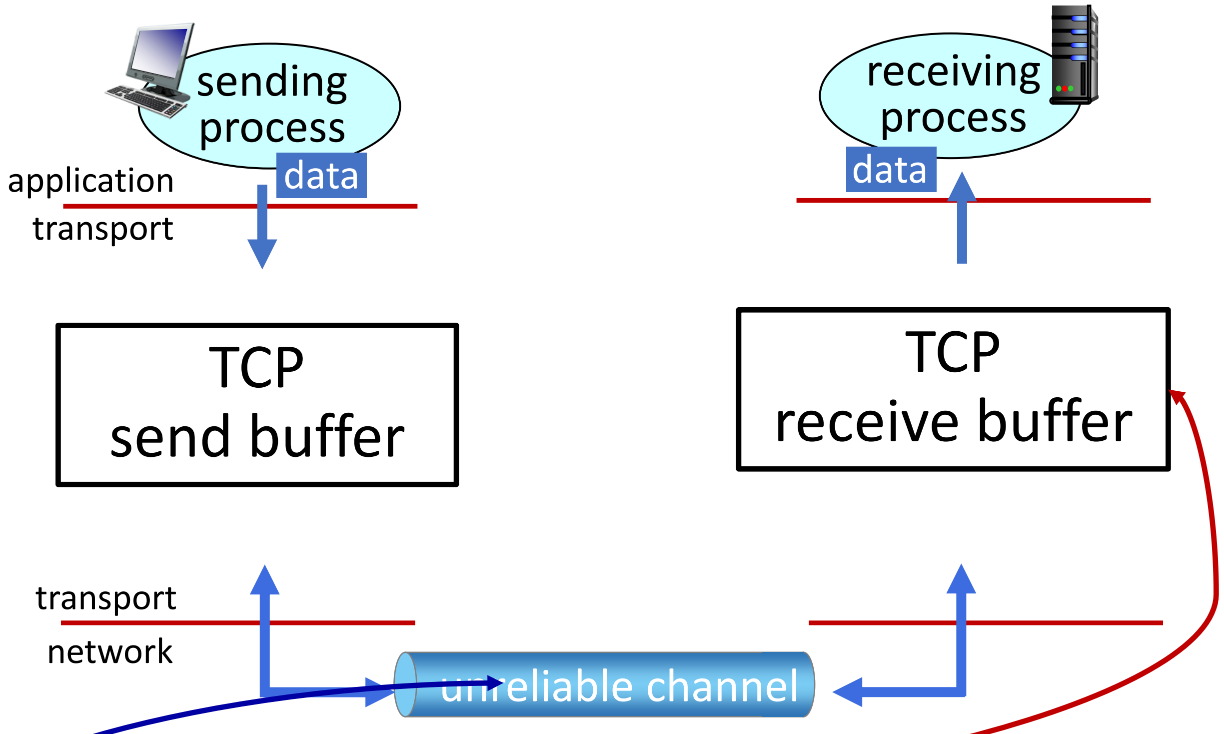
- New ACK
  - if **cwnd** < **ssthresh**, **cwnd** grows exponentially
  - if **cwnd** ≥ **ssthresh**, go to congestion avoidance
- Three duplicate ACKs
  - set **ssthresh** ← **cwnd**/2
  - set **cwnd** ← **ssthresh**
  - go to congestion avoidance
- Timeout
  - set **ssthresh** ← **cwnd**/2
  - set **cwnd** ← 1

Congestion avoidance:

AIMD

- New ACK
  - **cwnd** increases linearly
- Three duplicate ACKs
  - set **ssthresh** ← **cwnd**/2
  - set **cwnd** ← **ssthresh**
- Timeout
  - set **ssthresh** ← **cwnd**/2
  - set **cwnd** ← 1
  - go to slow start

Note: Congestion control ≠ Flow control

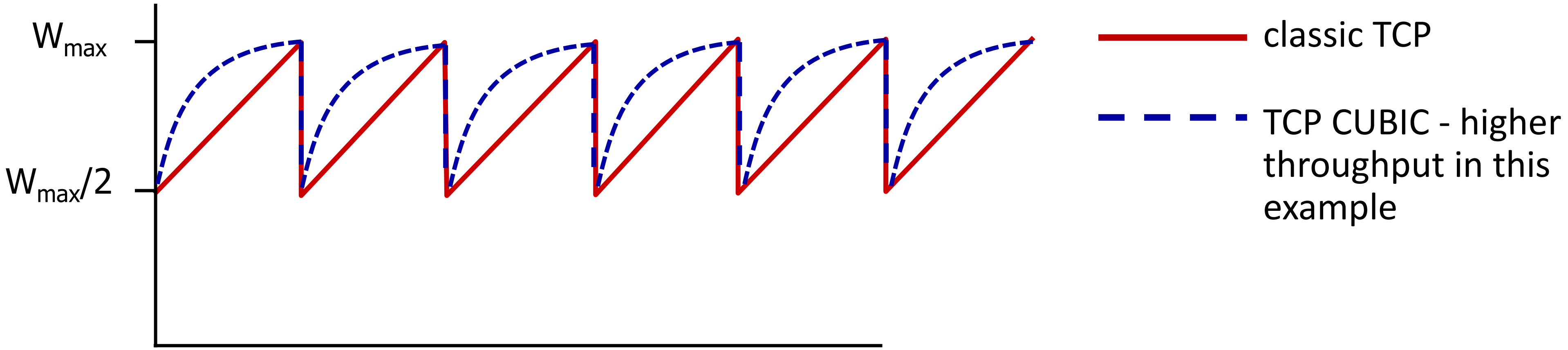


- Flow control
  - Sender will not overwhelm receiver
- Congestion control
  - Sender will not overwhelm the network

- In rdt tools, windows are used to manage pipelined transfer
- TCP has two windows
  - Flow control window
  - Congestion control window
- Sender is limited by the smallest window

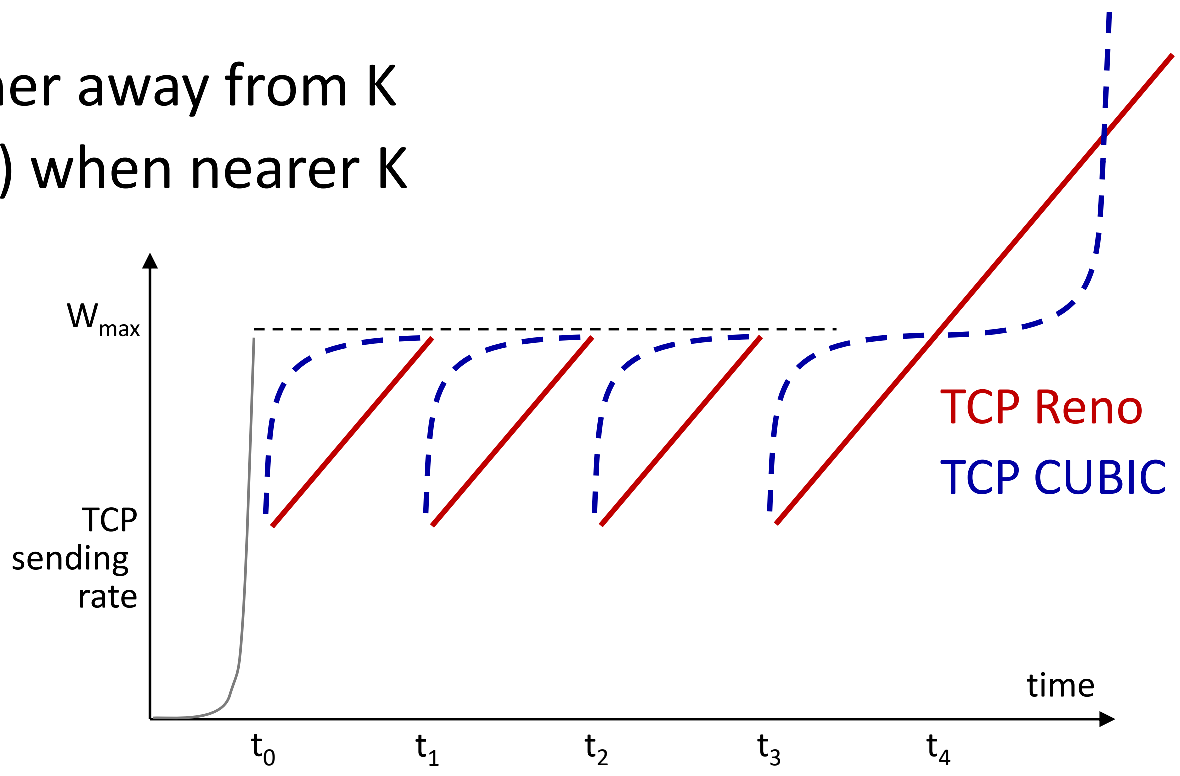
TCP CUBIC

- 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*



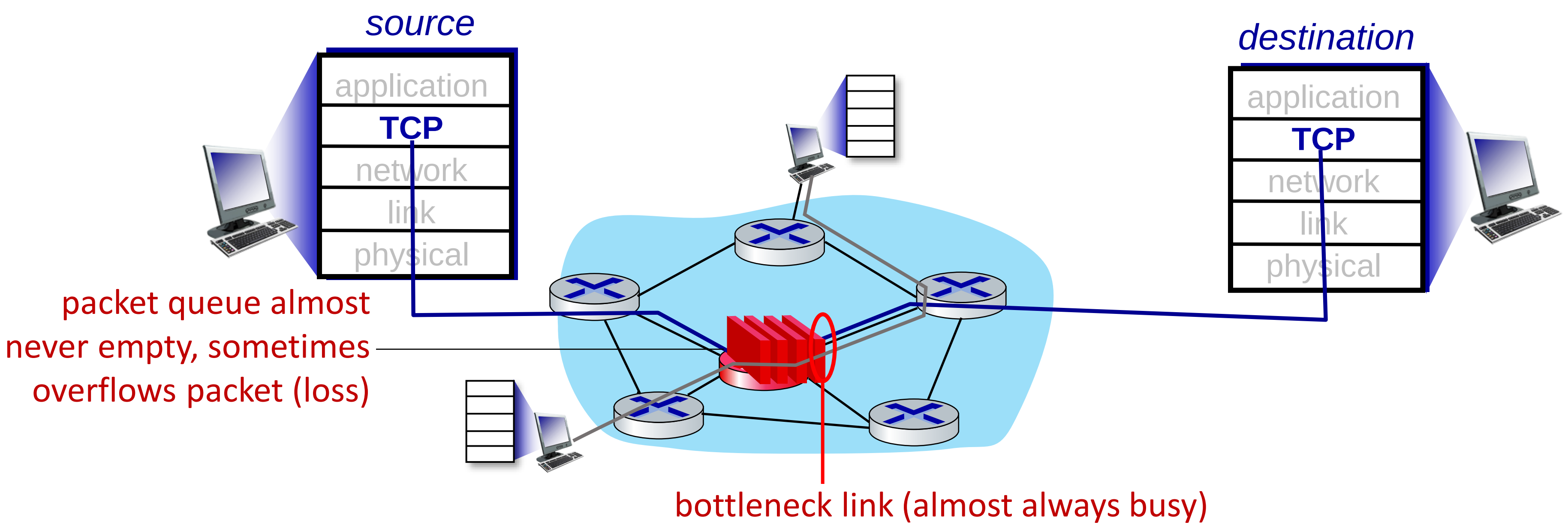
TCP CUBIC

- 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 and the congested "bottleneck link"

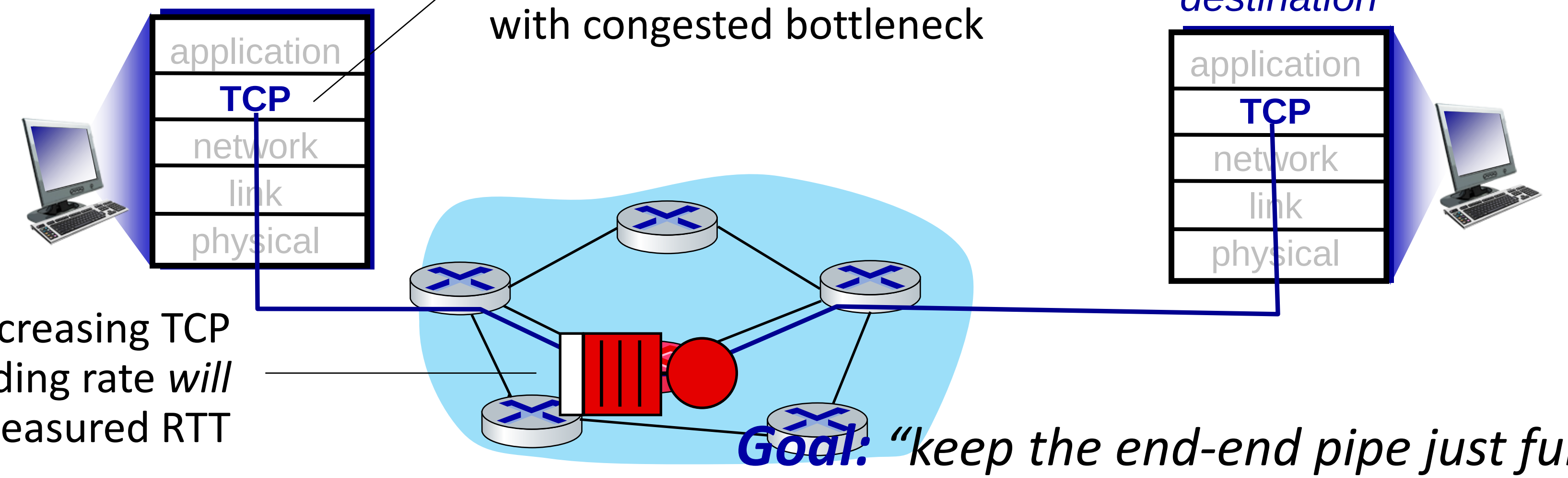
- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



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



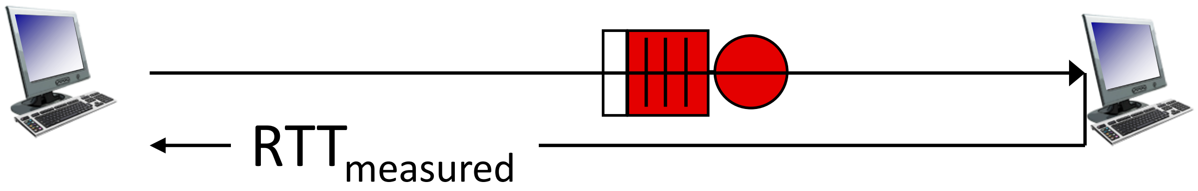
insight: increasing TCP sending rate *will* increase measured RTT

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 */
```

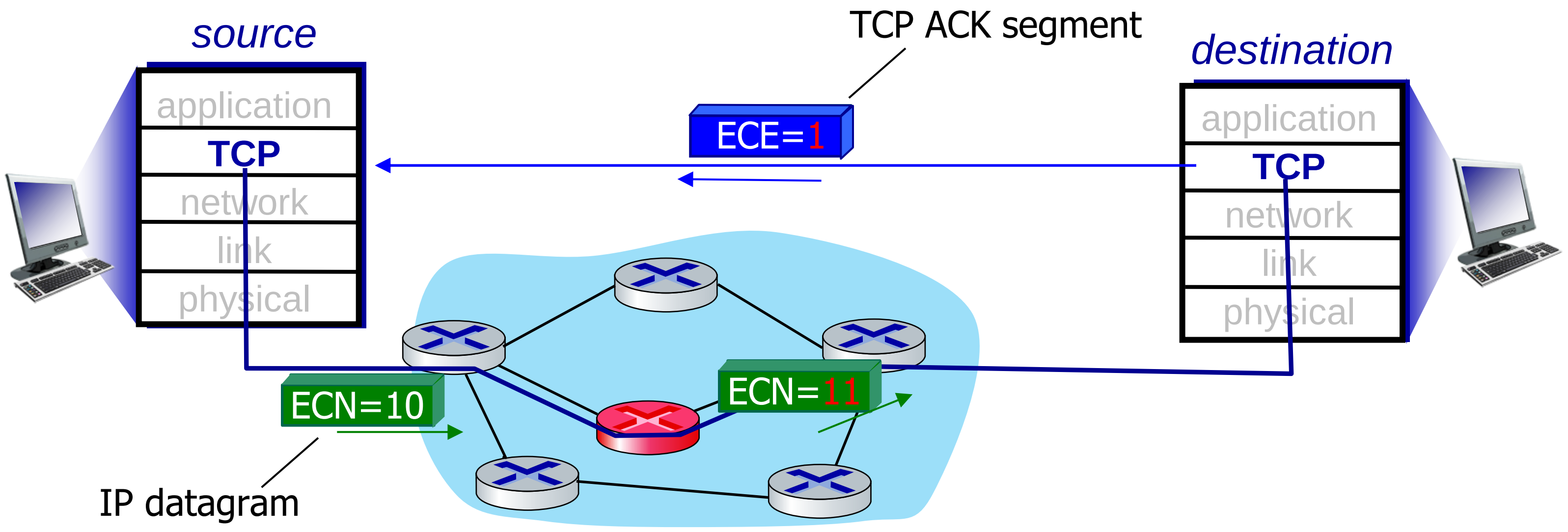
Delay-based TCP congestion control

- 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)

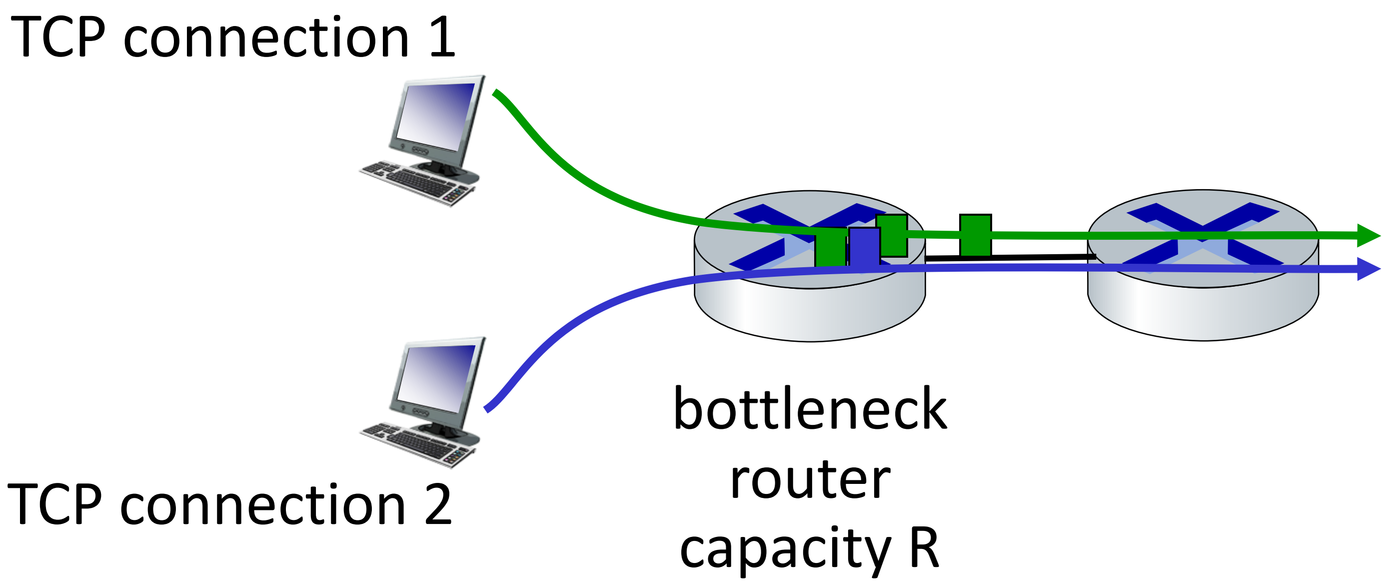
TCP deployments often implement *network-assisted* congestion control:

- 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

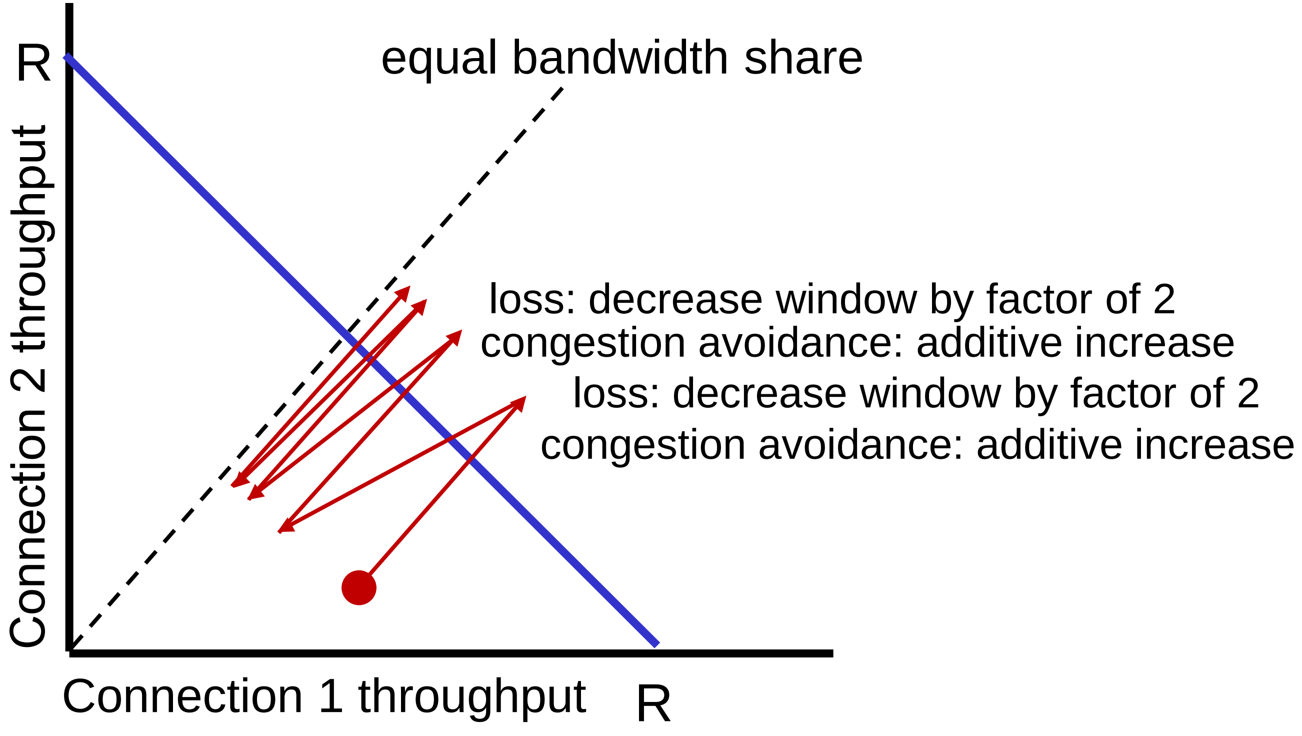
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

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



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

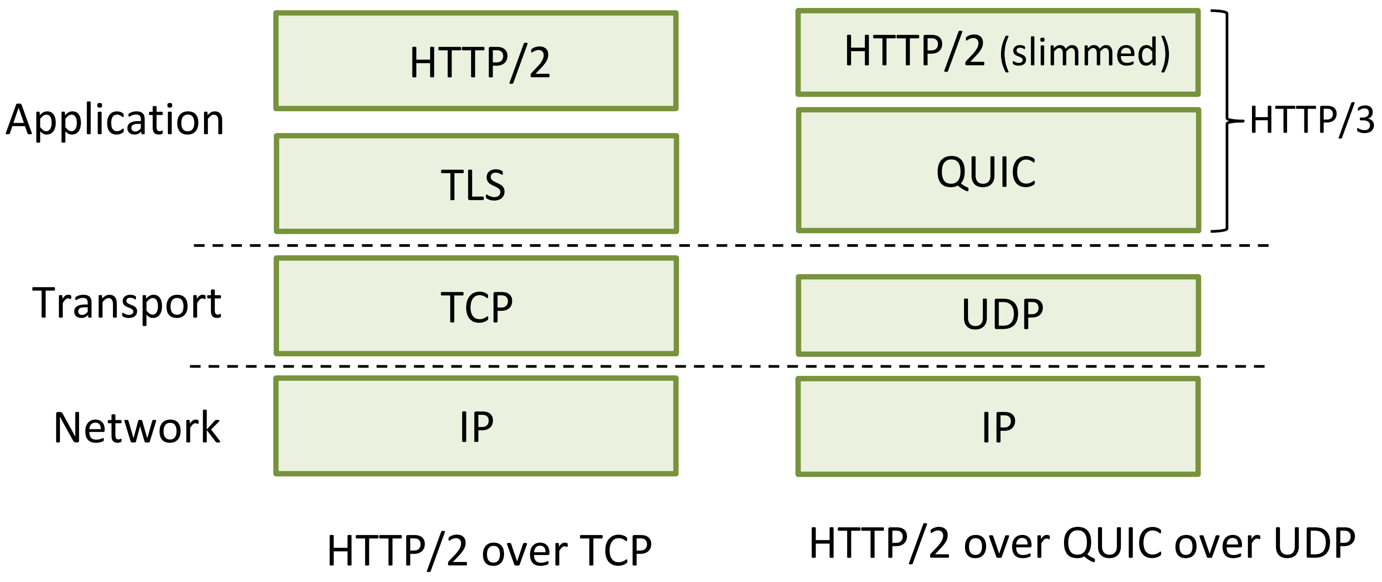
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

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)



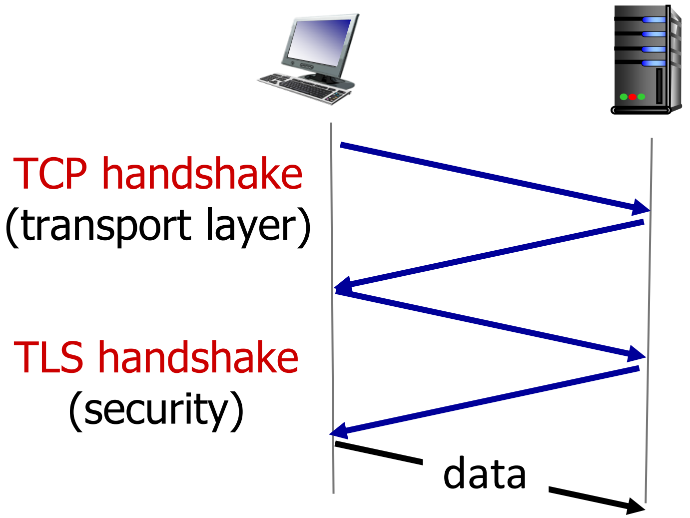
QUIC: Quick UDP Internet Connections

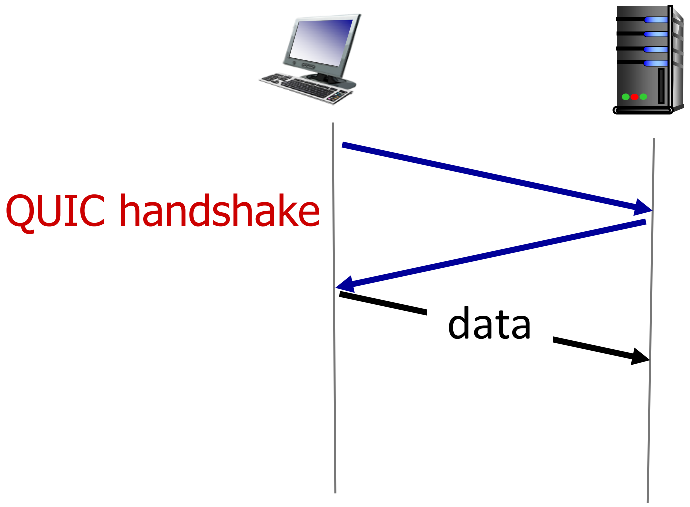
adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **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





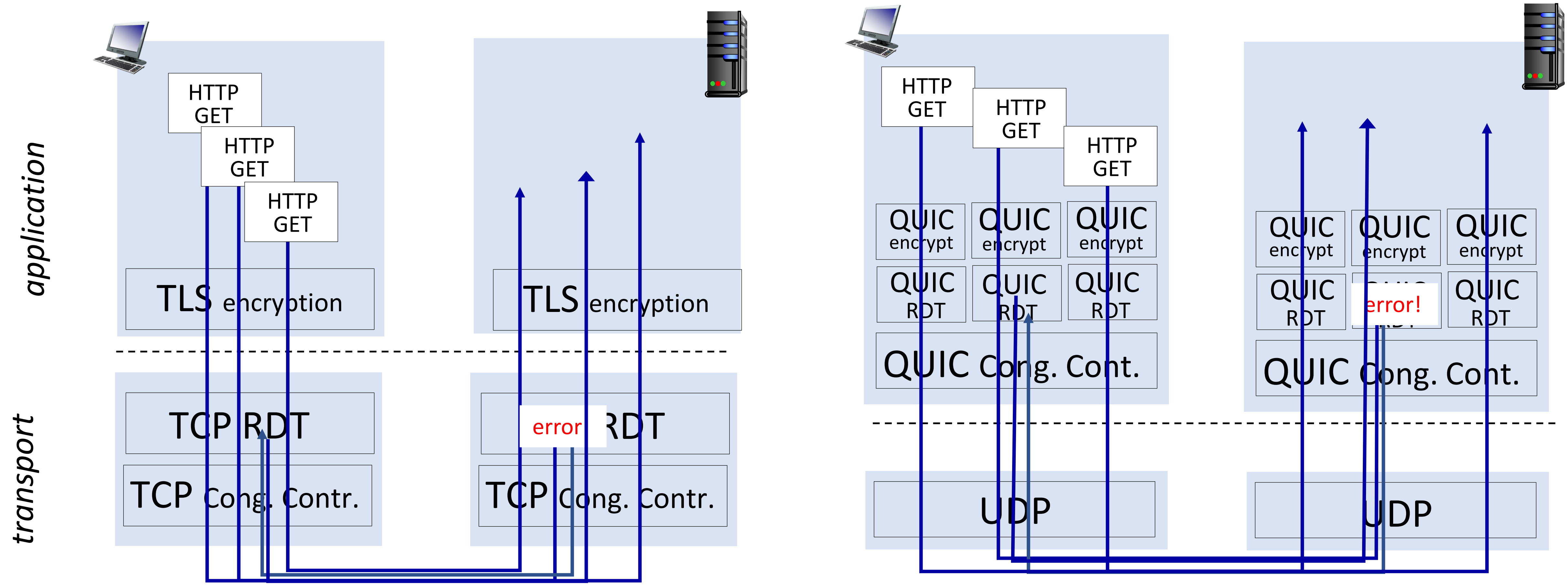
TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

(b) HTTP/2 with QUIC: no HOL blocking

Transport layer: 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"

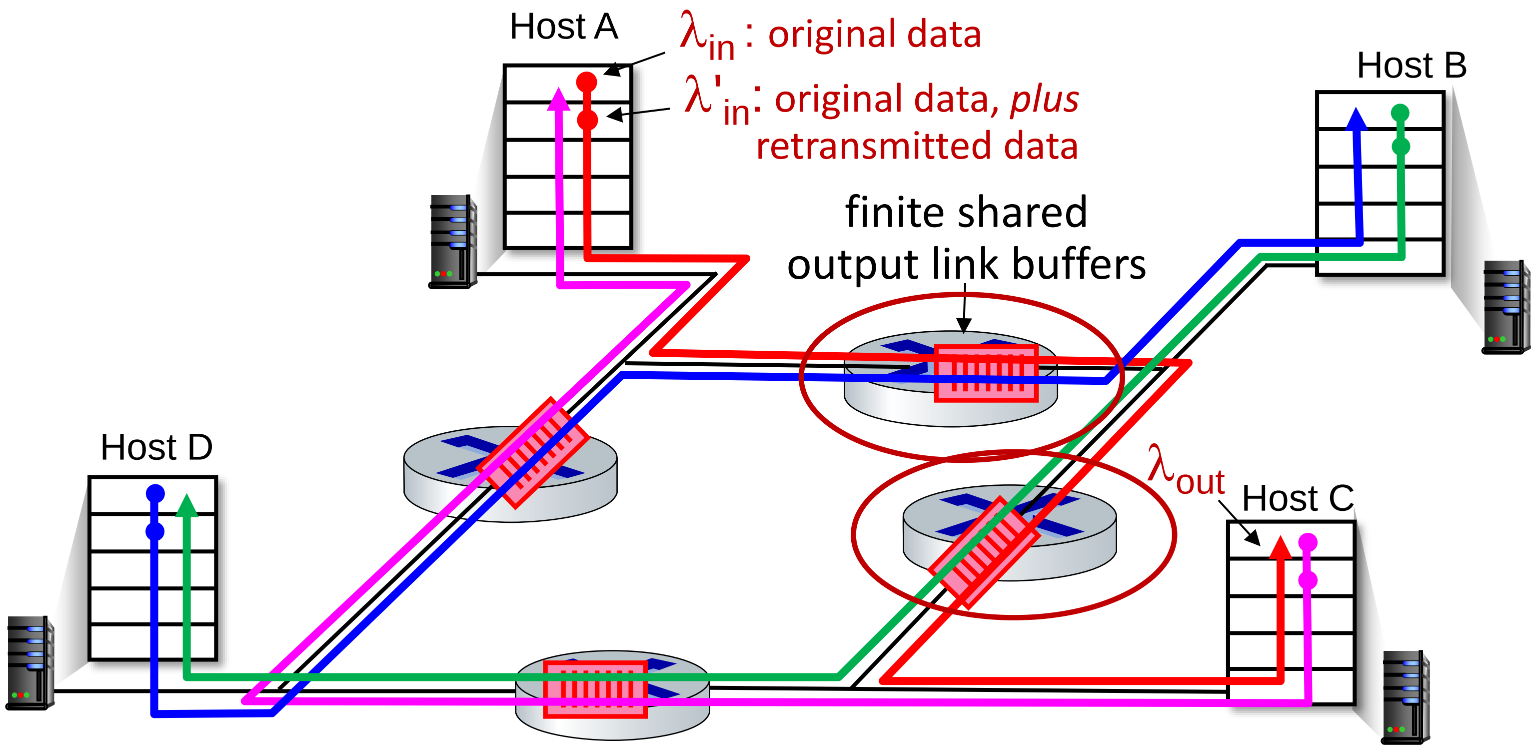
Additional Slides

Packet drops along the path

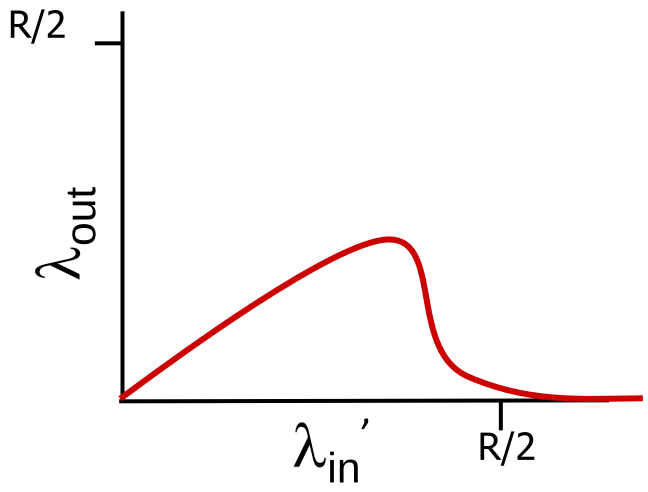
- *four* senders
- *multi-hop* paths
- timeout/retransmit

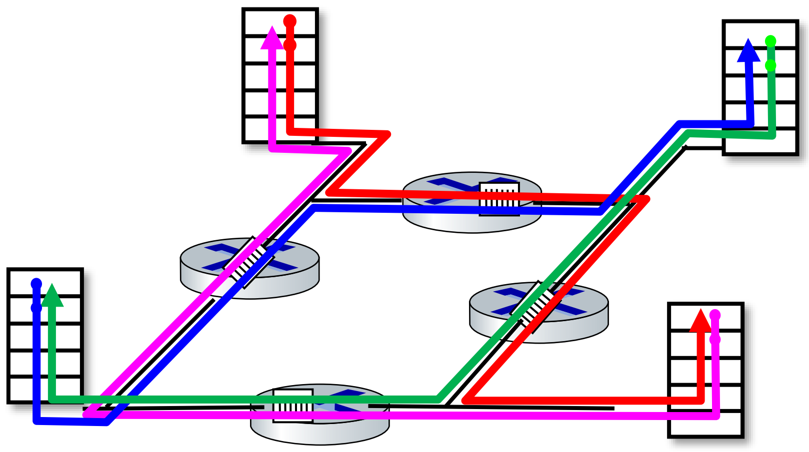
Q: what happens as λ_{in} and λ_{in}' increase ?

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



Packet drops along the path





another “cost” of congestion:

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