Lecture 5

The Transport Layer

TCP - continued.

Subjects of today

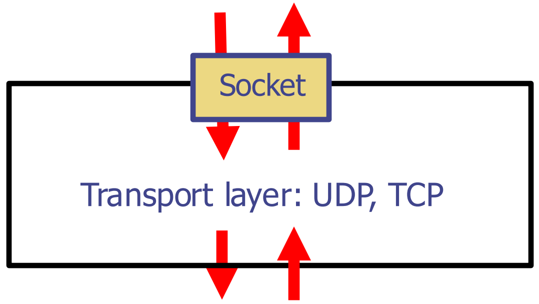
- Recap of TCP
- Timeout
- Flow Control
- Congestion control – Why?
- Congestion control – How?

4.1 Recap of TCP

The Transport Layer

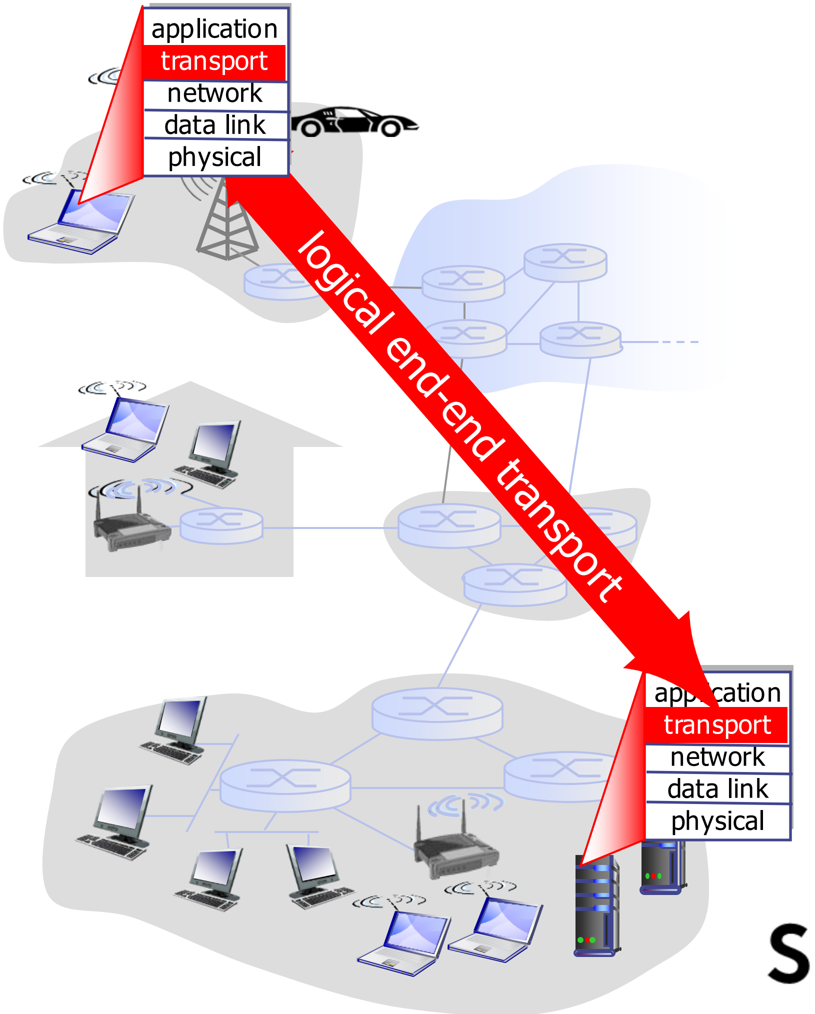
- Upper: Retrieves and delivers message through socket.
- Lower: Sends and receives segments to and from remote host.

- Packets at this level is called **Segments**

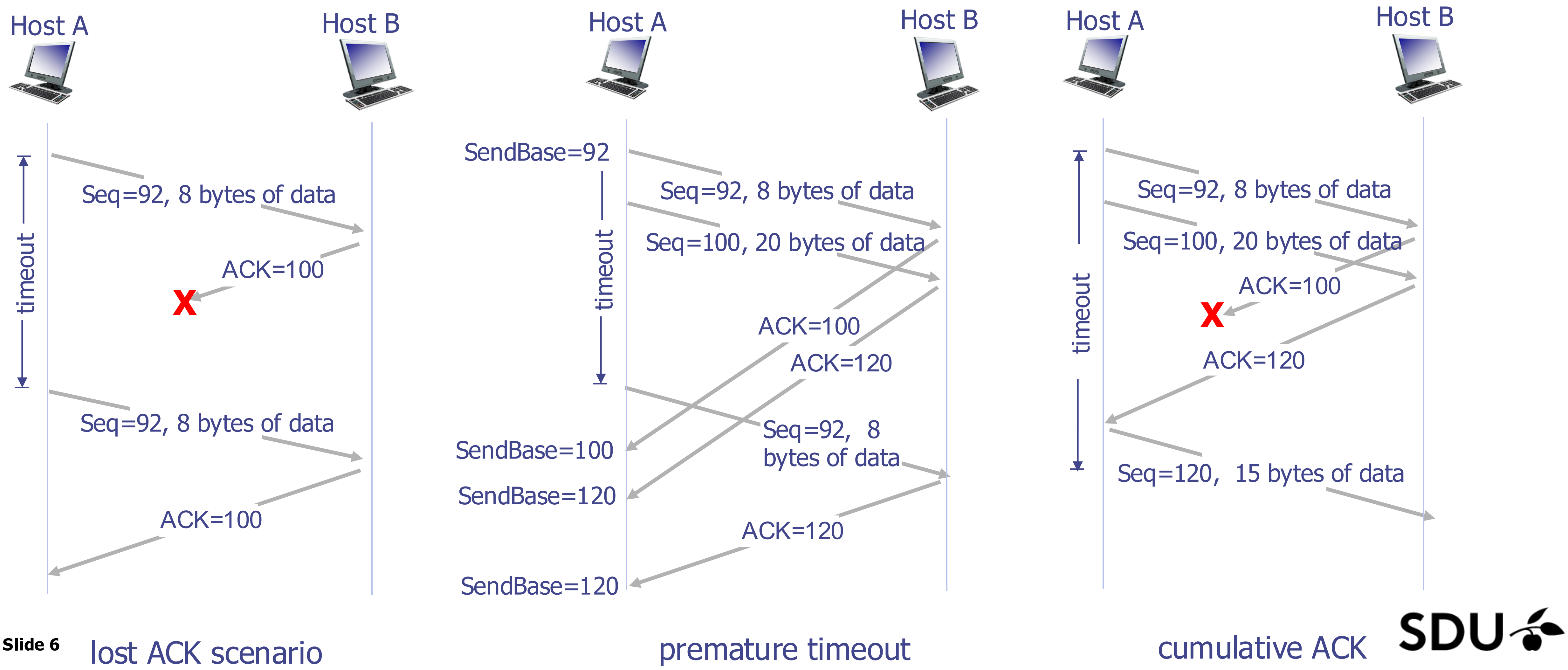


Summary of The Transport Layer's Role

- Provide *logical communication* between app processes running on different hosts by using:
  - IP addresses
  - Port numbers
- Transport protocols run in end systems
  - Send side: breaks app messages into *segments*, passes to network layer
  - Receiver side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP (most common)



TCP: Retransmission Scenarios



Transmission Control Protocol

- Must have:
  - Breaking messages into segments: Yes
  - Multiplexing/demultiplexing: Yes
- Connection management: Connection-oriented communication
- Reliable data transfer: Yes
  - Error detection: Checksum
  - Error recovery: Yes
  - In order delivery: Yes
- Timing: None
- Throughput: No
  - Flow control: Yes
  - Congestion Control: Yes
- Security: None

4.2 Timeout

TCP Round Trip Time, Timeout

How to set TCP timeout value?

- Longer than RTT
  - But RTT varies!
- *Too short:* Premature timeout, unnecessary retransmissions
- *Too long:* slow reaction to segment loss

How to estimate RTT?

SampleRTT: measured time from segment transmission until ACK receipt

- Ignore retransmissions

SampleRTT will vary, want estimated RTT "smoother"

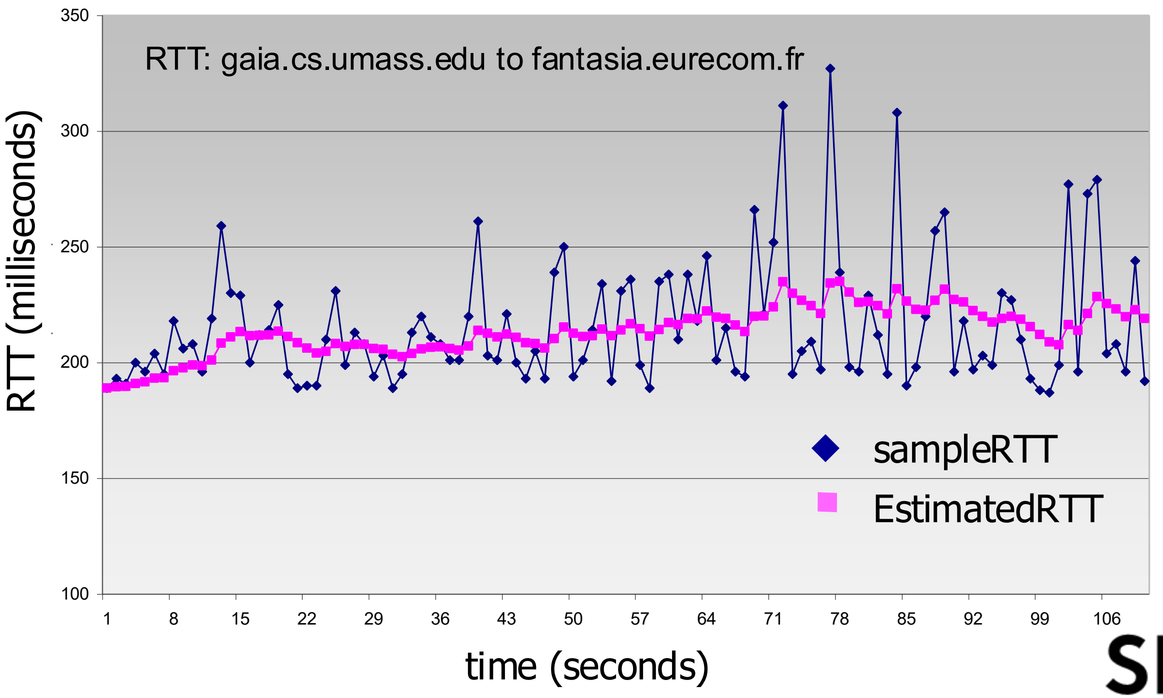
- Average several *recent* measurements, not just current **SampleRTT**

TCP Round Trip Time, Timeout

```
EstimatedRTT = (1-α)*EstimatedRTT + α*SampleRTT
```

- Exponential Weighted Moving Average (EWMA)
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$

How can you achieve
a safety margin?



TCP Round Trip Time, Timeout

- Timeout interval: **`EstimatedRTT`** plus “safety margin”
  - Large variation in **`EstimatedRTT:`** want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

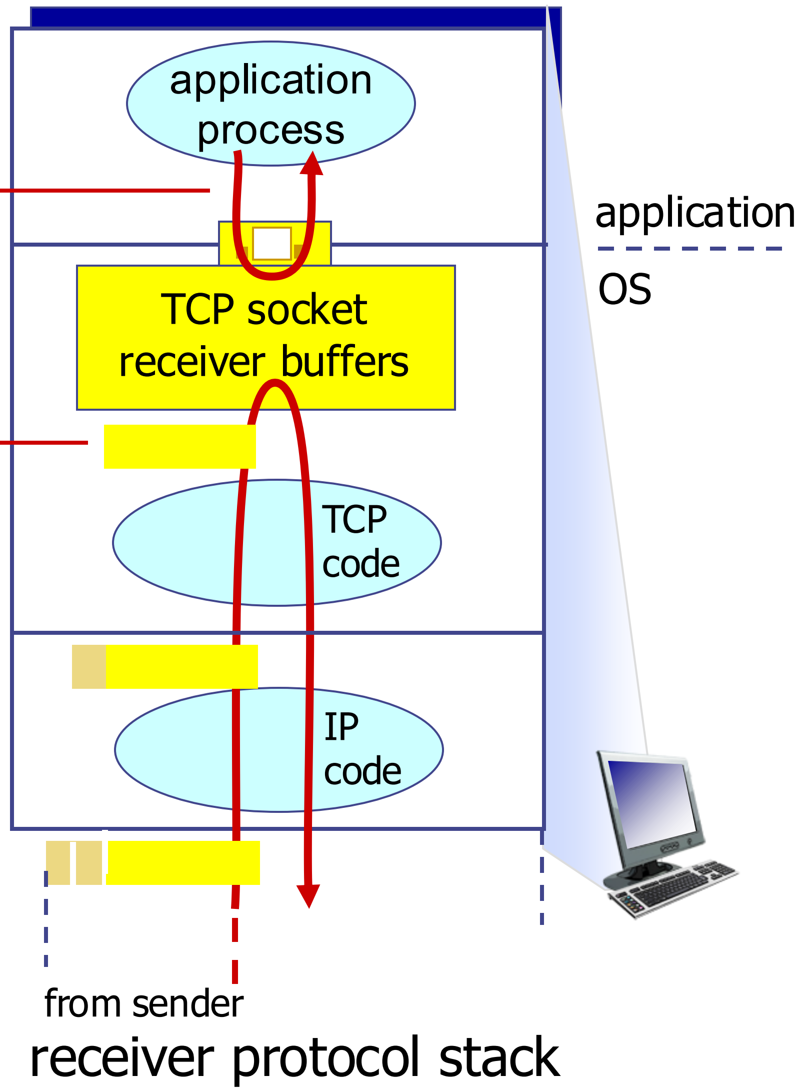
(typically, β = 0.25)

4.3 Flow Control

TCP Flow Control

application may remove data from TCP socket buffers ….

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

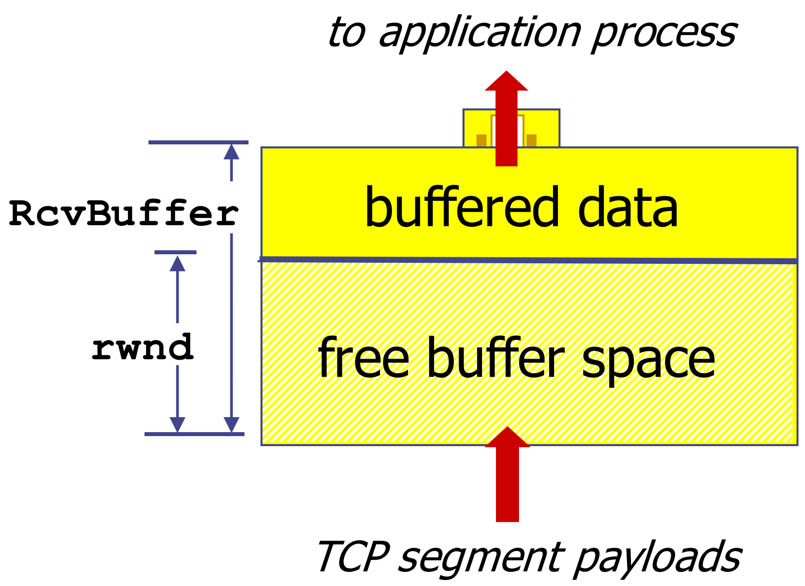


Flow control

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

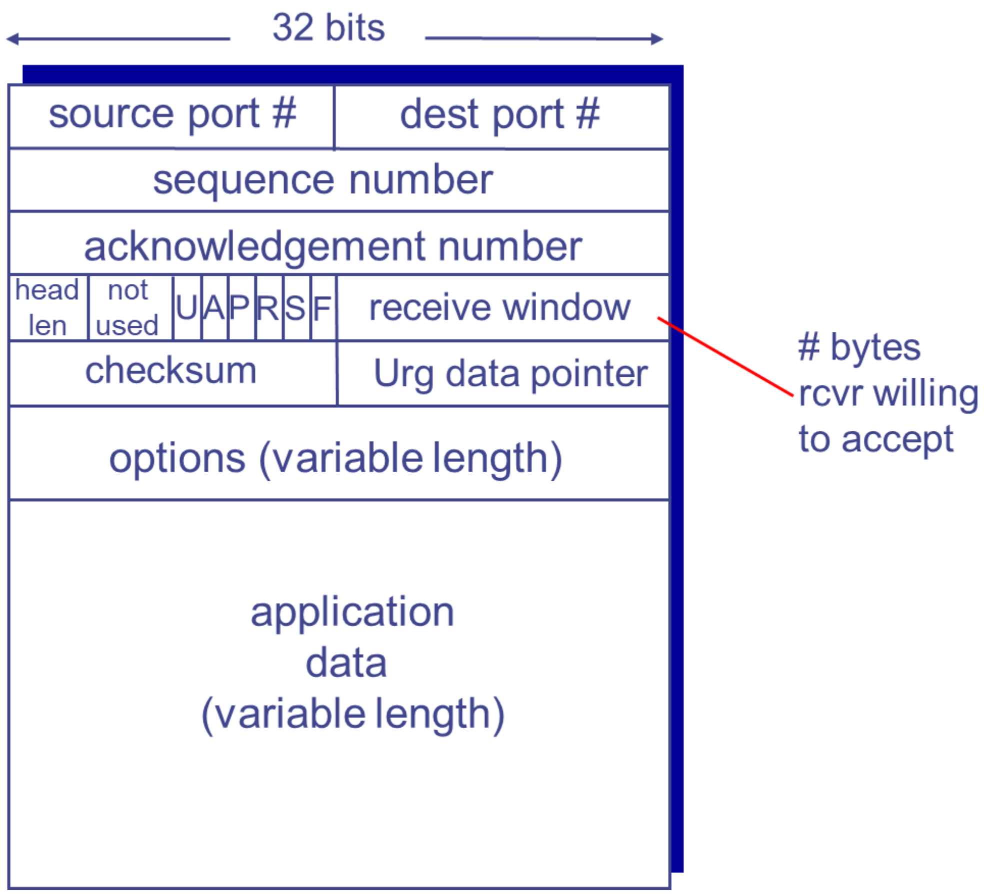
TCP Flow Control

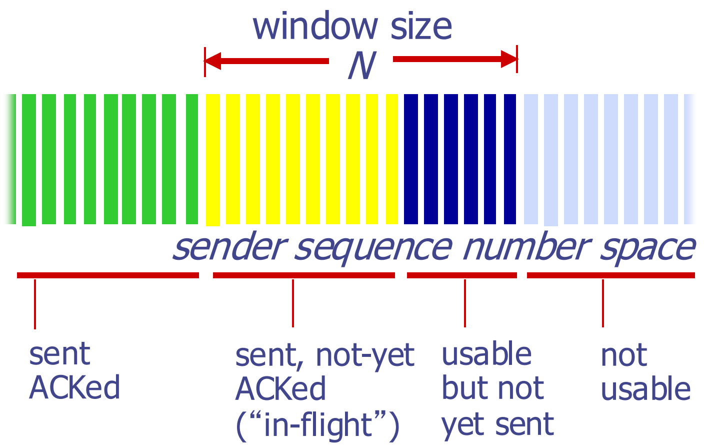
- Receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options
  - Many operating systems auto-adjust **RcvBuffer**
- Sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
  → Guarantees receive buffer will not overflow



receiver-side buffering

TCP Flow Control – In practise





4.4 Congestion Control – Why?

Principles of Congestion Control

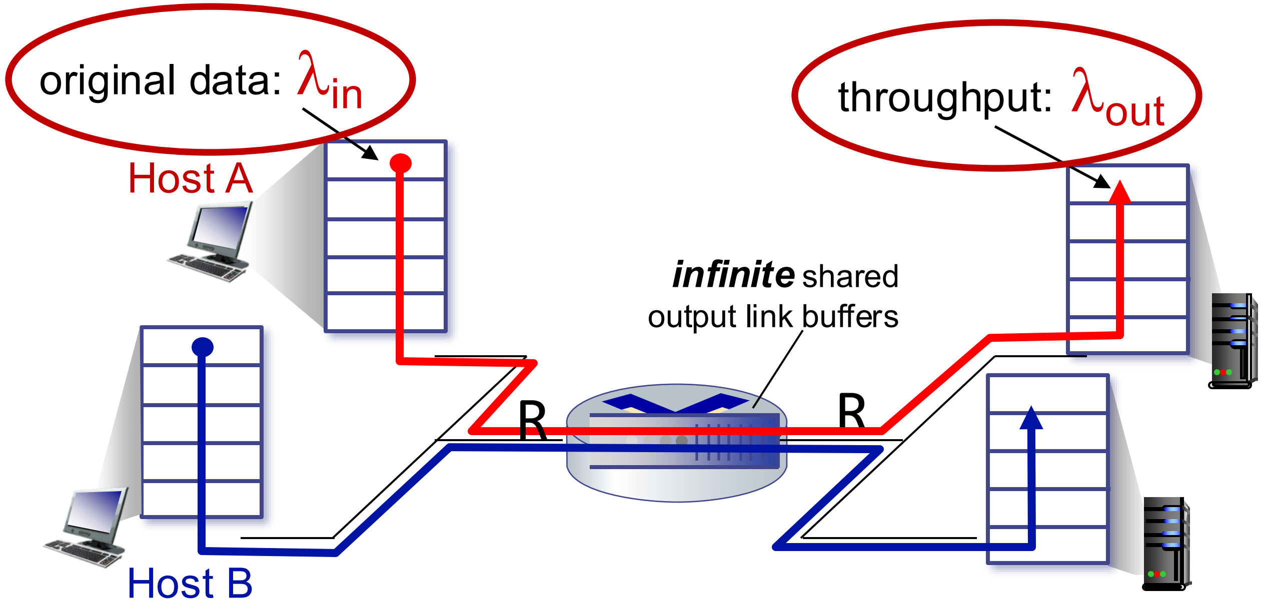
Congestion

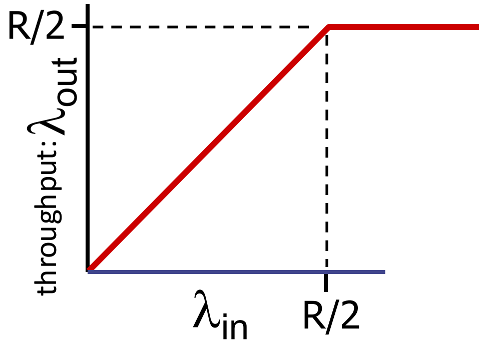
- "Too many sources sending too much data too fast for *network* to handle"
- Different from flow control!
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)

Causes/costs of congestion: scenario 1

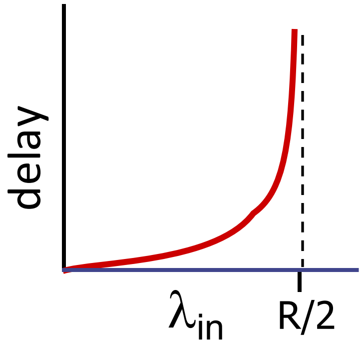
Simplest scenario:

- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed





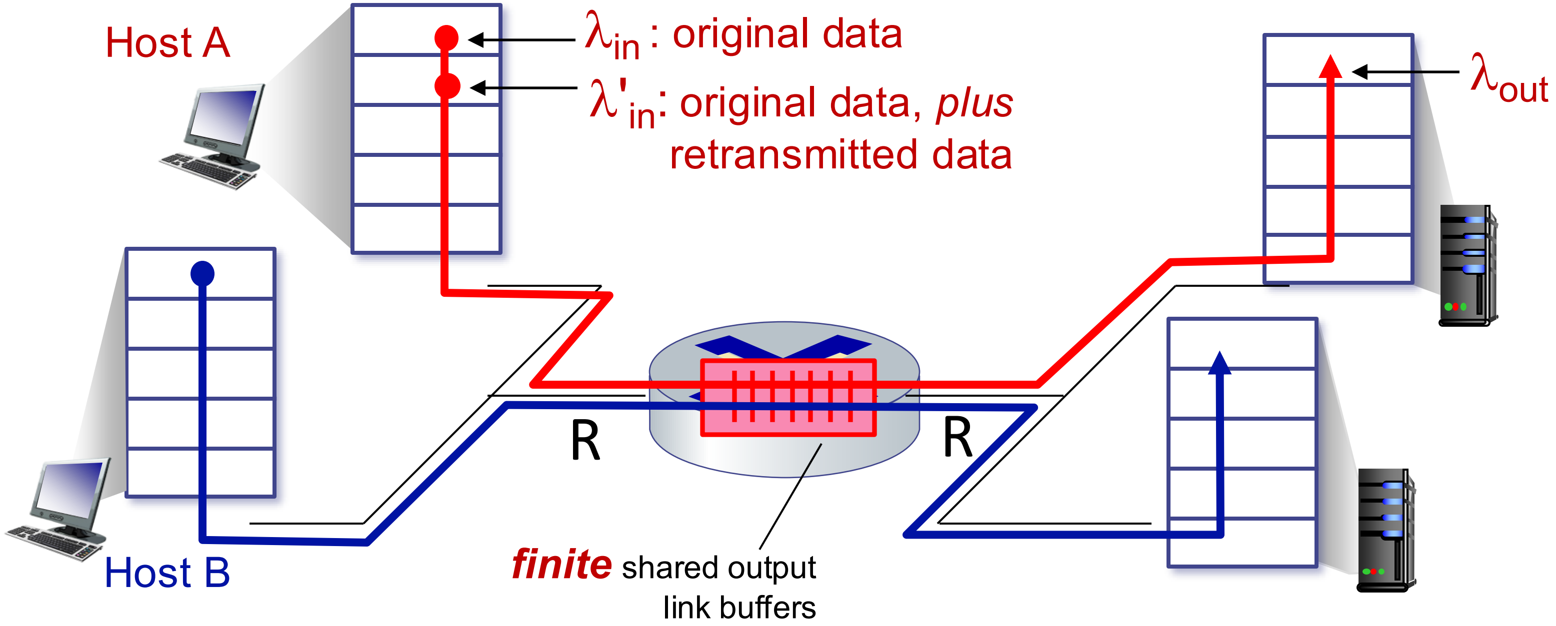
maximum per-connection throughput: R/2



large delays as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

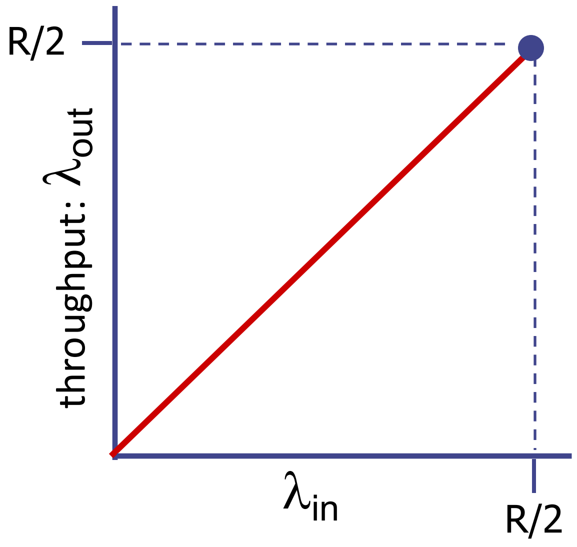
- one router, *finite* buffers
- sender retransmits lost, timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

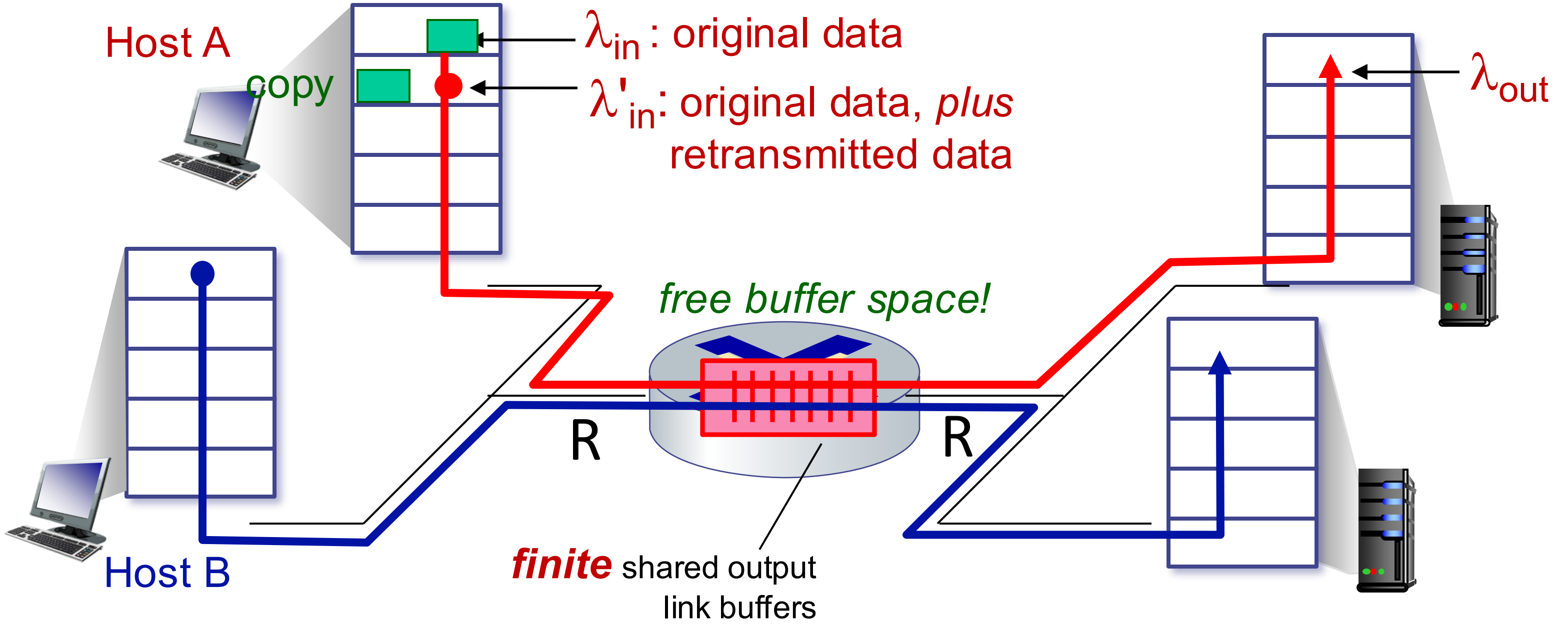


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

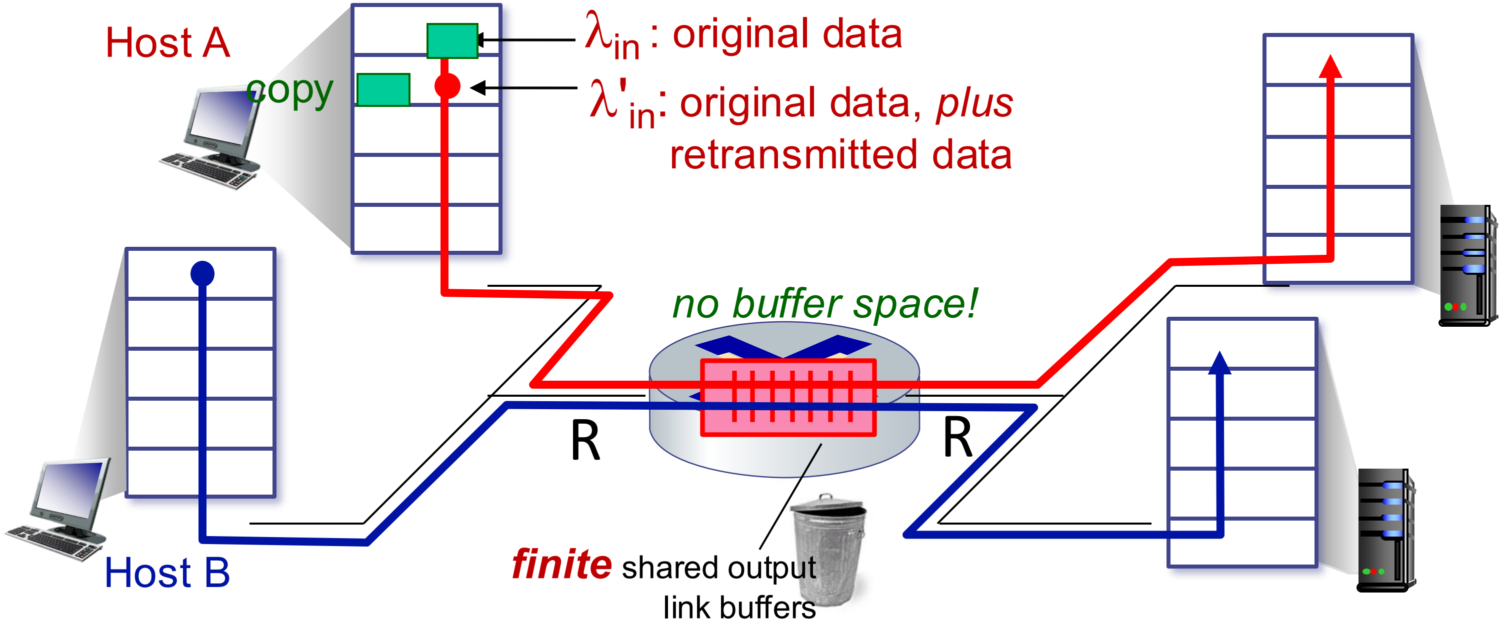




Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

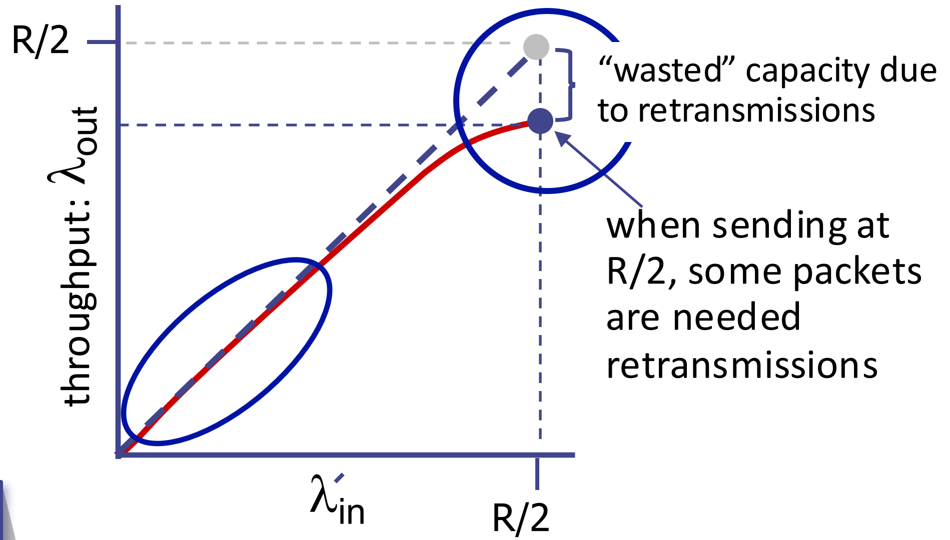
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

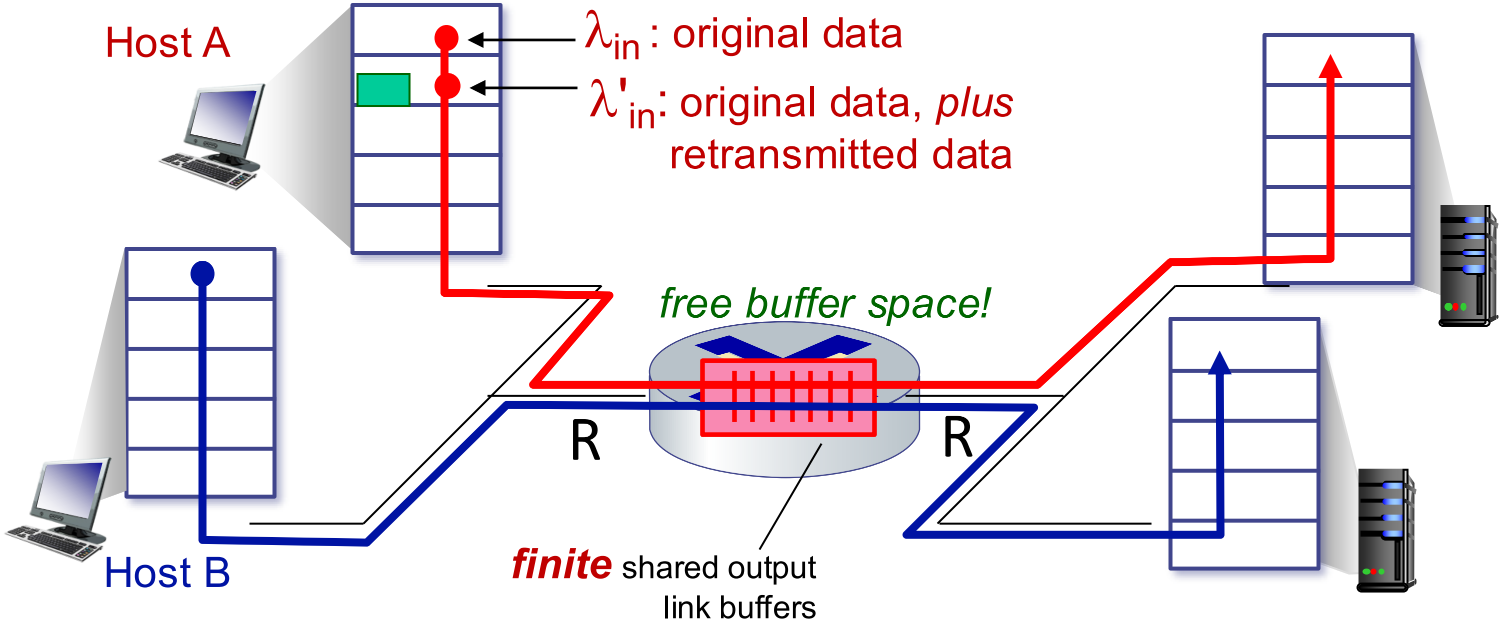


Causes/costs of congestion: scenario 2

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



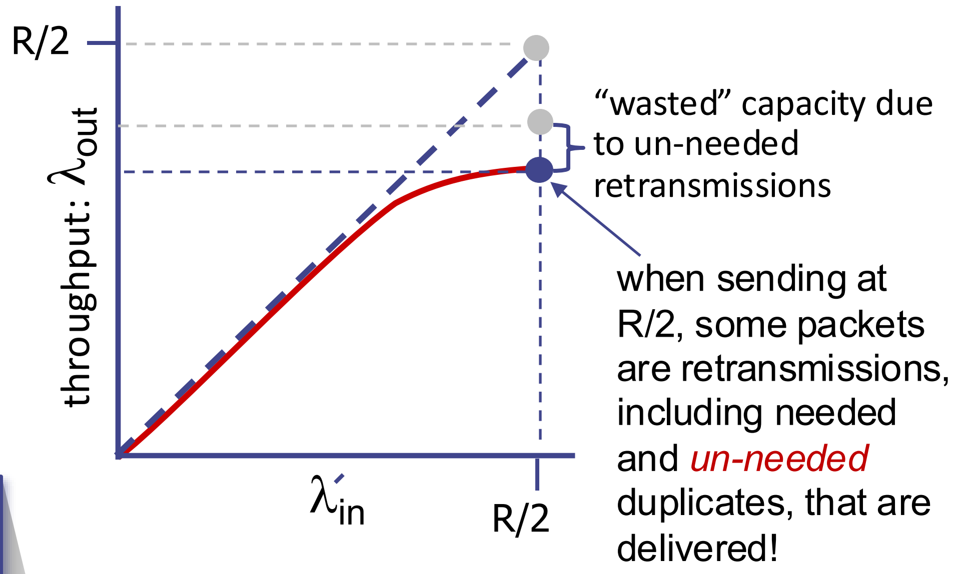


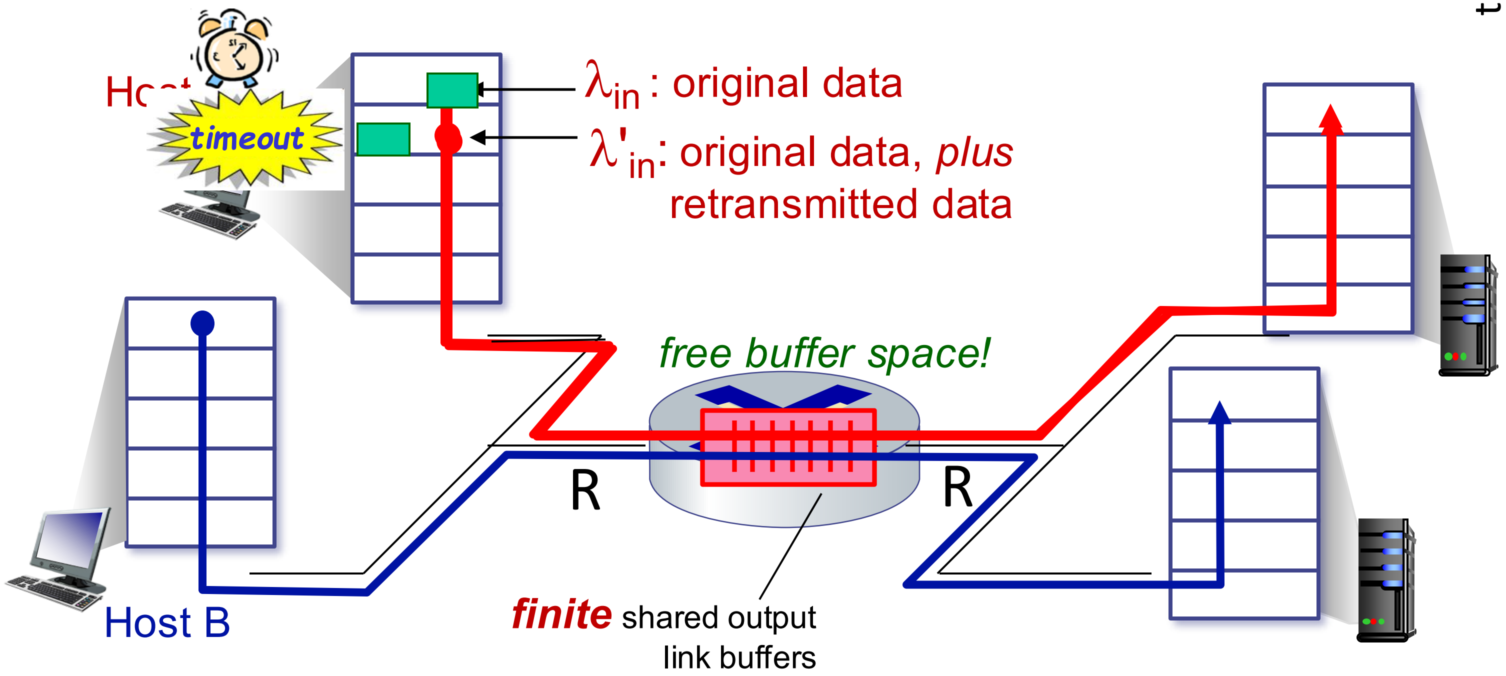
SDU

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



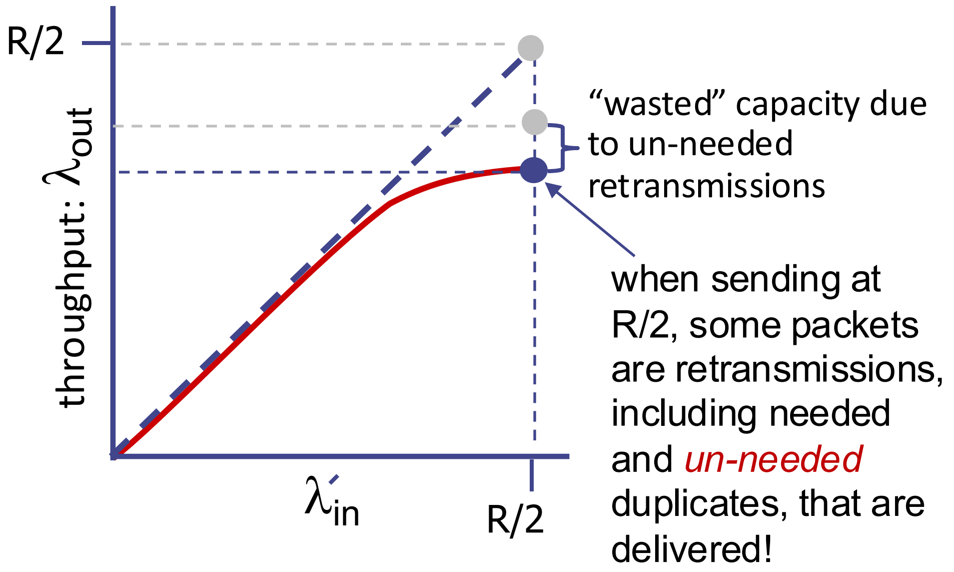


SDU

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



"costs" of congestion:

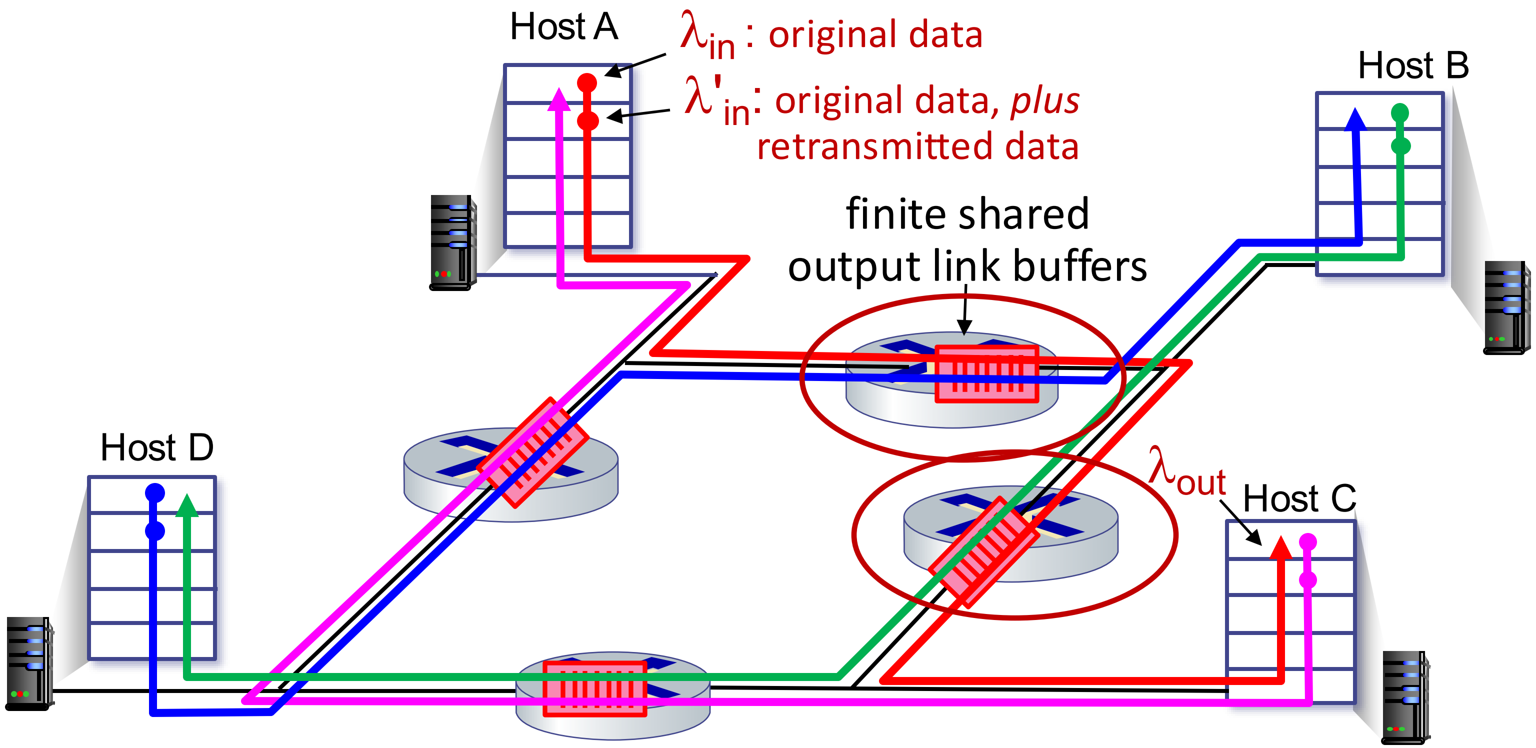
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3

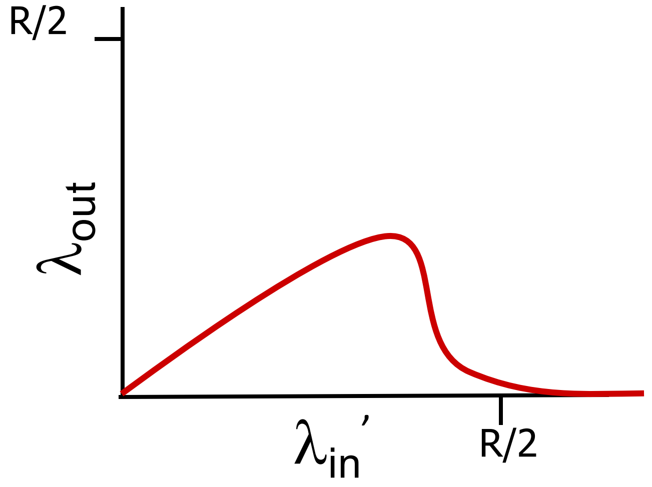
- *four* senders
- *multi-hop* paths
- timeout/retransmit

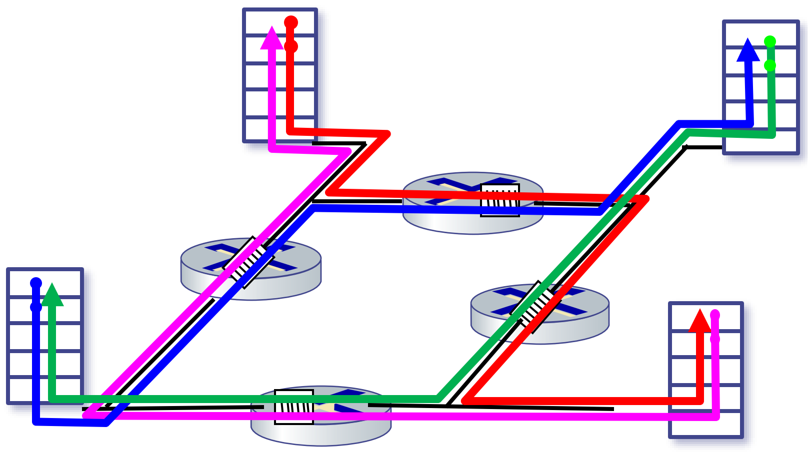
Q: what happens as λ_{in} and λ_{in}' increase ?

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



Causes/costs of congestion: scenario 3



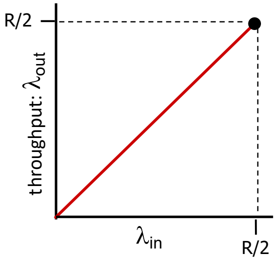


another "cost" of congestion:

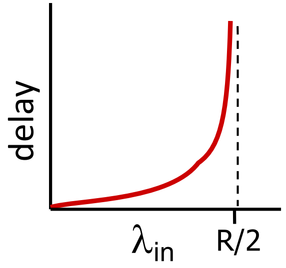
- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

Summary of Key Points

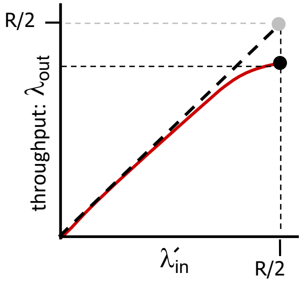
- throughput can never exceed capacity



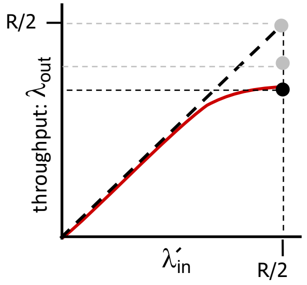
- delay increases as capacity approached



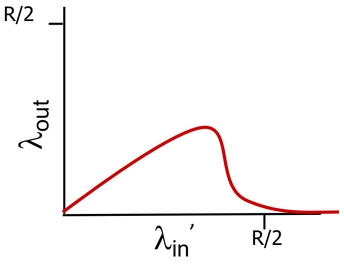
- loss/retransmission decreases effective throughput



- un-needed duplicates further decreases effective throughput

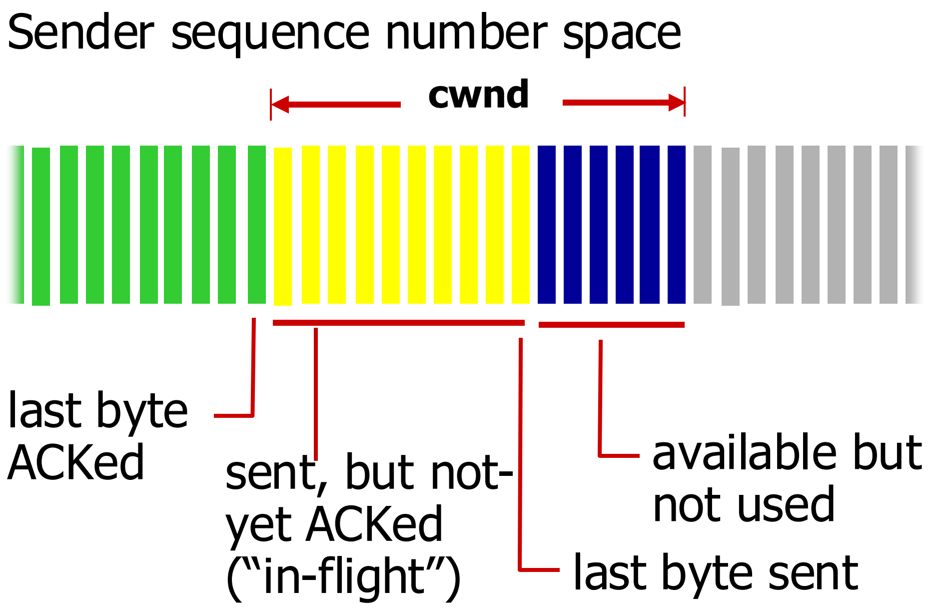


- upstream transmission capacity / buffering wasted for packets lost downstream



4.4 Congestion Control – How?

TCP Congestion Control: Congestion window



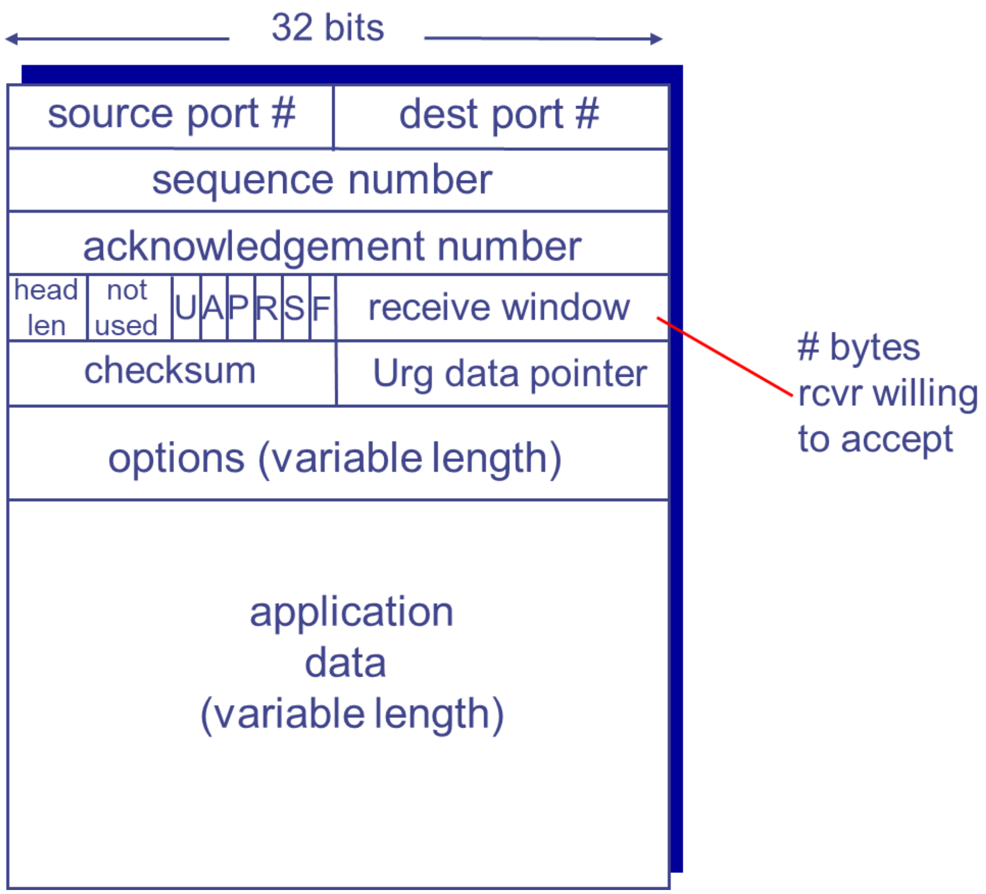
TCP sending behavior:

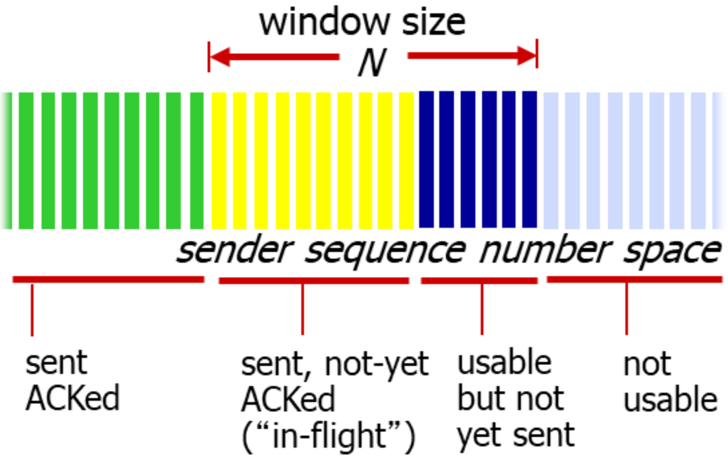
- *Roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

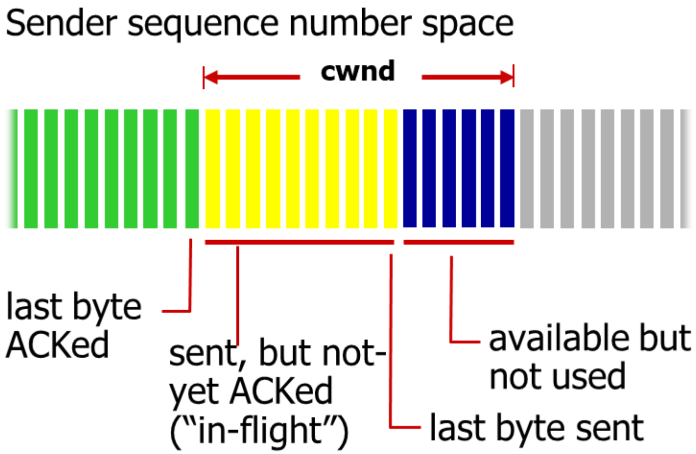
$$\textbf{TCP rate} \approx \frac{\textbf{cwnd}}{\textbf{RTT}} \textbf{ bytes/sec}$$

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

TCP Cogestion Control – Congestion window





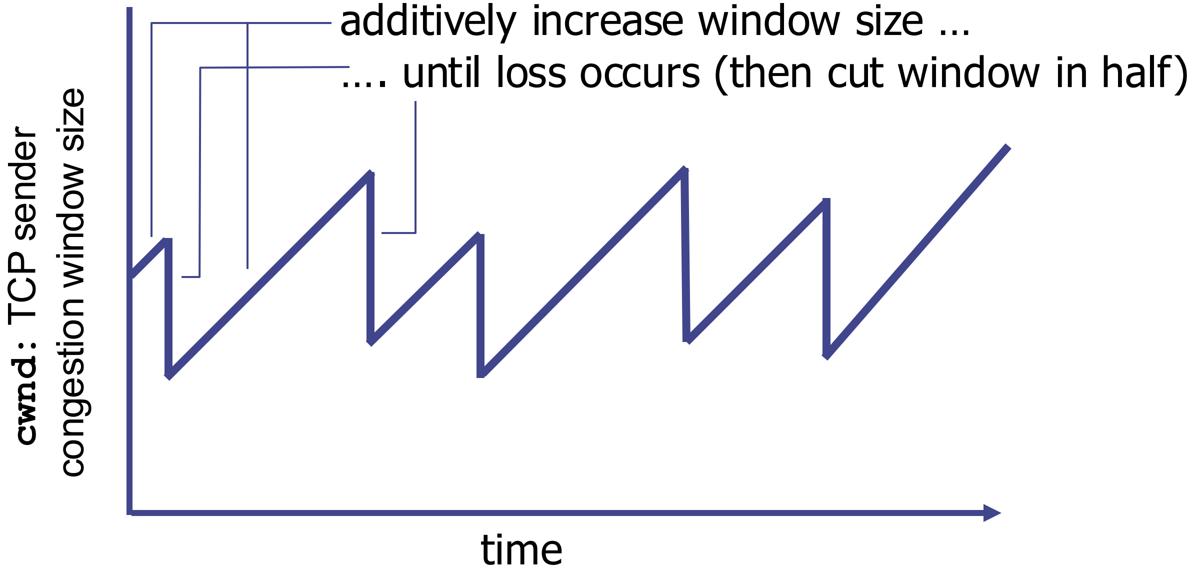


Why is there not congestion window field in the header?

TCP Congestion Control: AIMD

- Additive Increase Multiplicative Decrease
- **Approach**: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **Additive increase:** increase **cwnd** by 1 MSS (Max. Segment Size) every RTT until loss detected
  - **Multiplicative decrease:** cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth



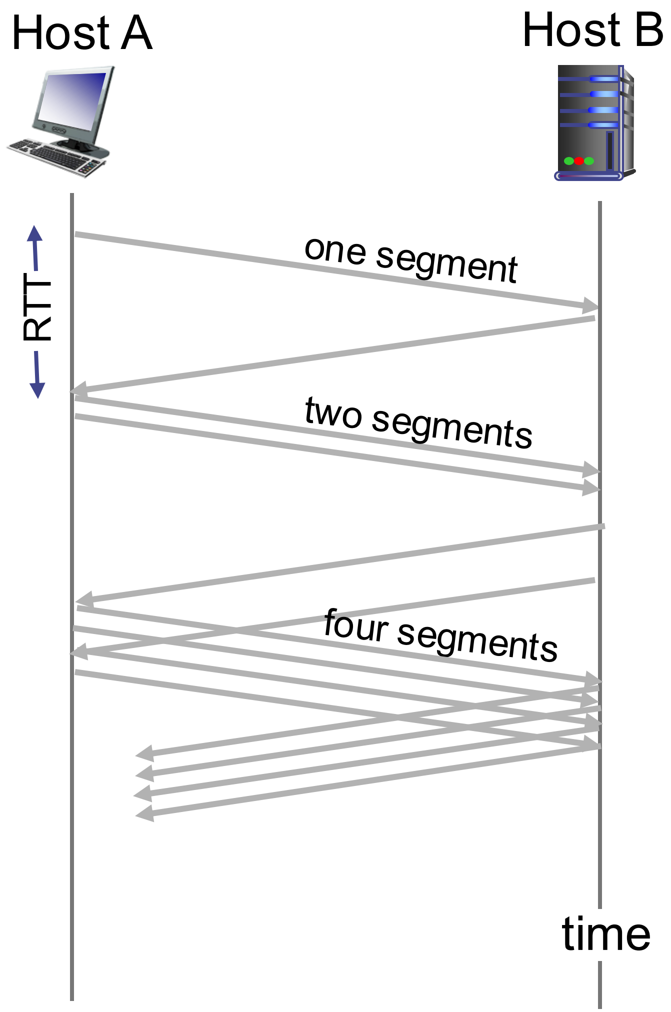
TCP Congestion Control: Mechanisms

- **Information**
  - A lost segment implies congestion
  - An acknowledged segment indicates no congestion
  - = Bandwidth probing

- **Control States**
  - Slow Start
  - Congestion Avoidance
  - Fast Recovery

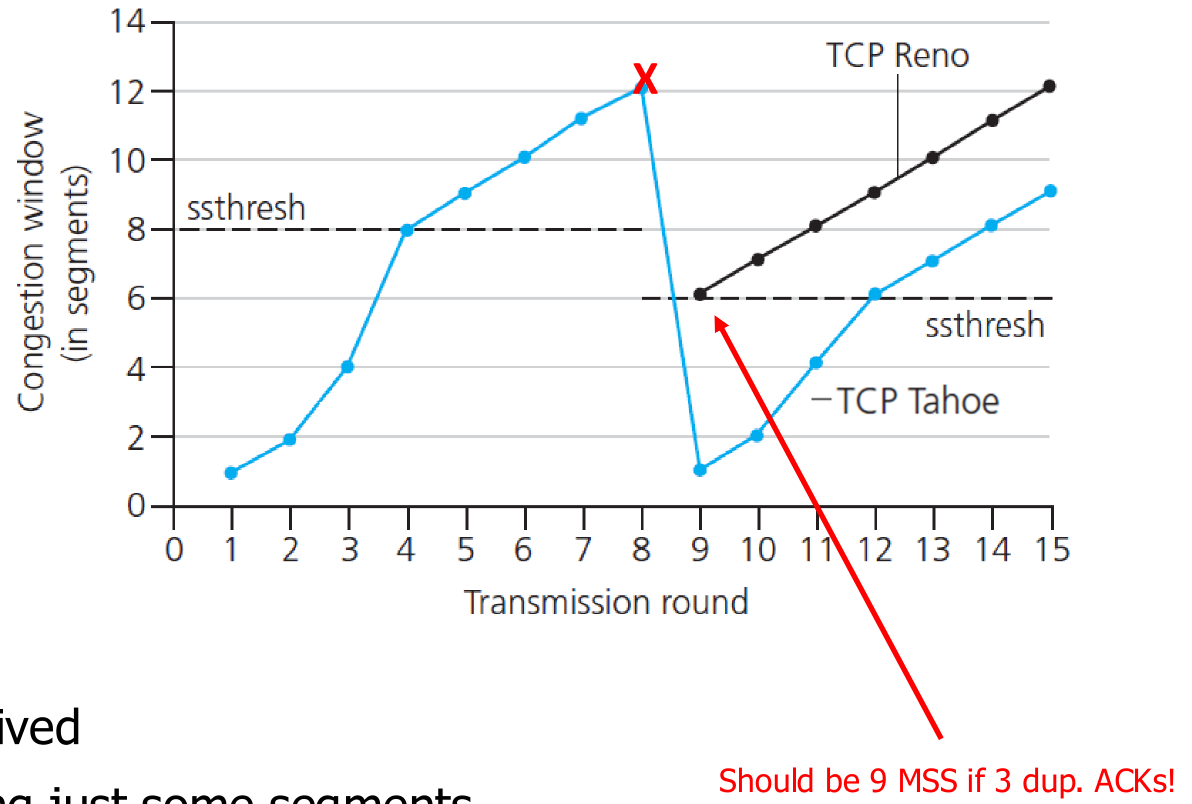
Slow Start

- 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
  - Resets to **cwnd** = 1MSS on timeout
- **Summary:** initial rate is slow but ramps up exponentially fast



From Slow Start to Congestion Avoidance

TCPs exponential increase of **cwnd** switches to **linear** (CA) when **cwnd** gets to 1/2 of its (peak) value before timeout.



Implementation:

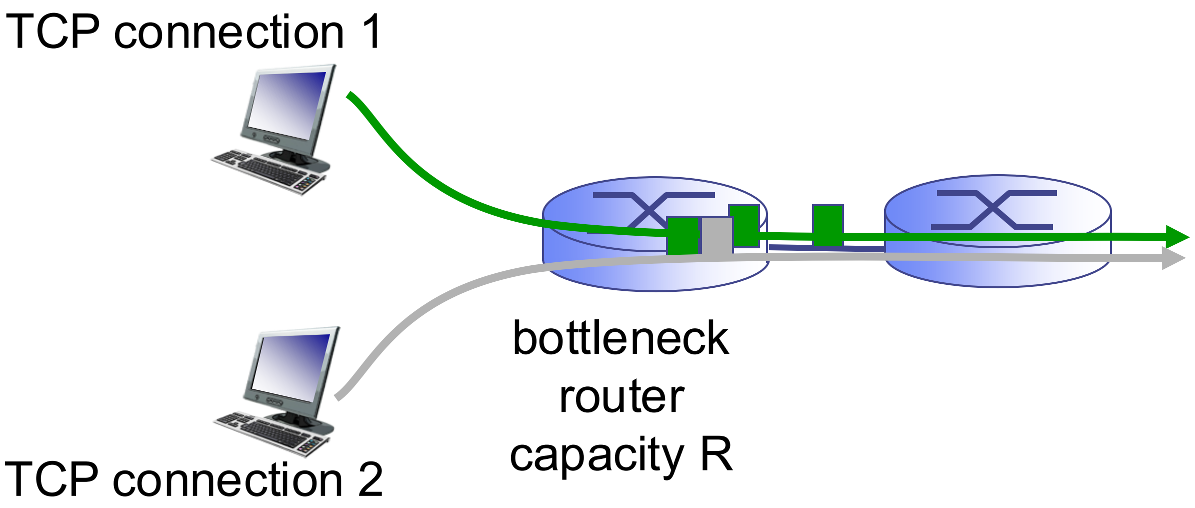
- Variable: **ssthresh**
- On loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Fast recovery:

- Enters this mode when duplicate ACKs are received
- Dup. ACKs indicate network capable of delivering just some segments
- Increase by 1 MSS per dup. ACK

TCP Fairness

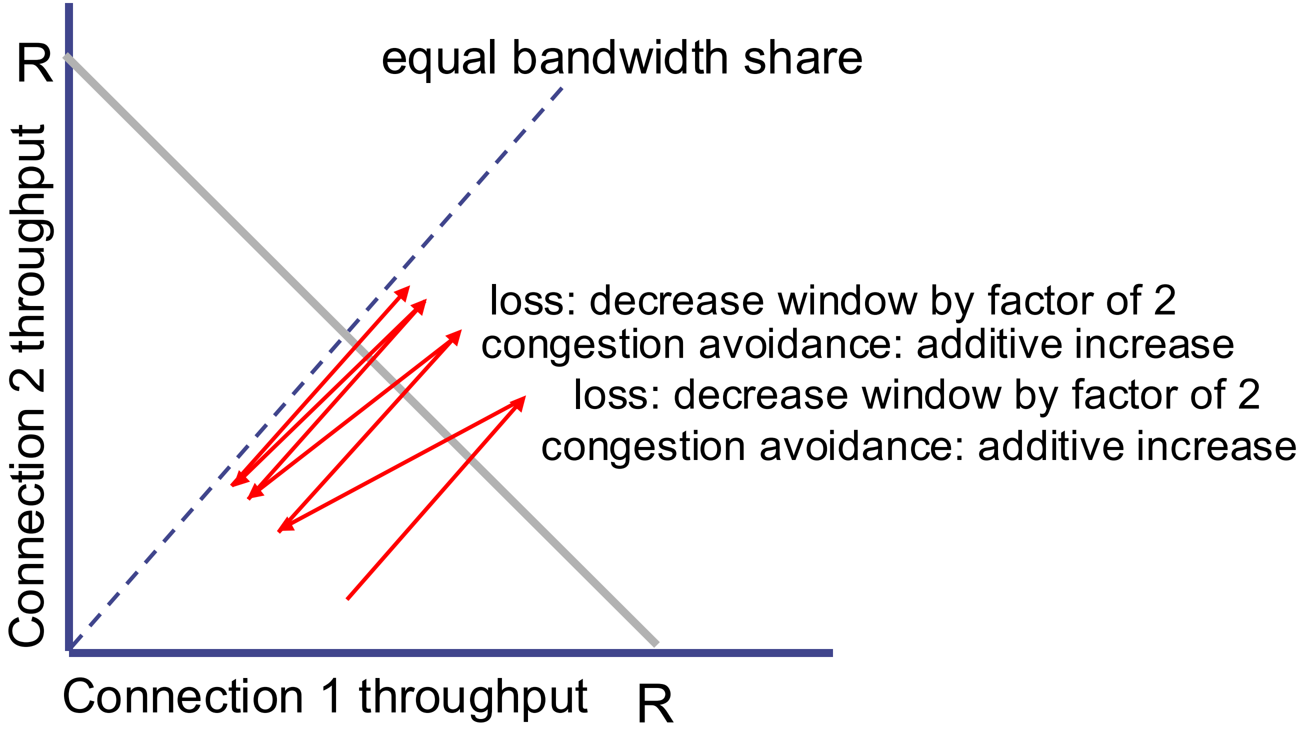
Fairness goal: If K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



TCP Fairness

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



TCP Fairness

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

Fairness, parallel TCP connections

- Application can open multiple parallel connections between two hosts
- 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

Next time(s)

The next section is reserved exclusive for exercises in socket programming. No additional preparation required. Expected knowledge:

- Sockets
  - Application
  - Transport layer
- Programming (e.g. Python Lab prep.)

In week 44 we will talk about the data plane of the network layer.
Read chapter 4 in the book (page 333-389).