CS348: Computer Networks

Congestion Control in TCP

Dr. Manas Khatua
Assistant Professor
Dept. of CSE, IIT Guwahati
E-mail: manaskhatua@iitg.ac.in

Principles of Congestion Control

- We have discussed: reliable data transfer service in the face of packet loss
  - such loss typically results from the overflowing of router buffers as the network becomes congested

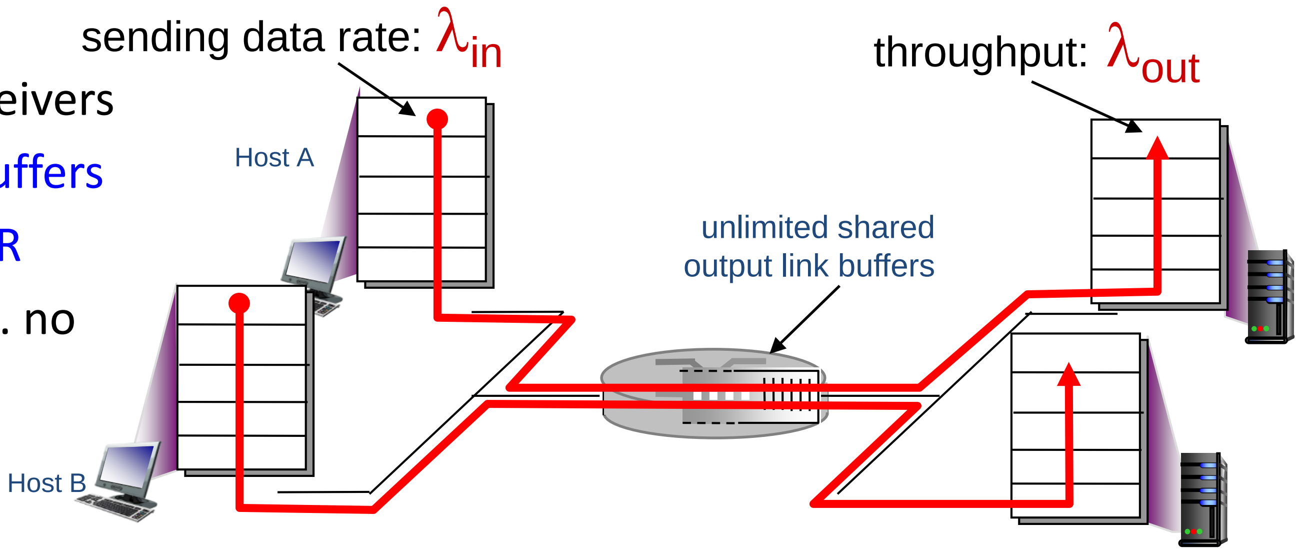
- Packet retransmission treats a symptom of network congestion but not the cause of network congestion

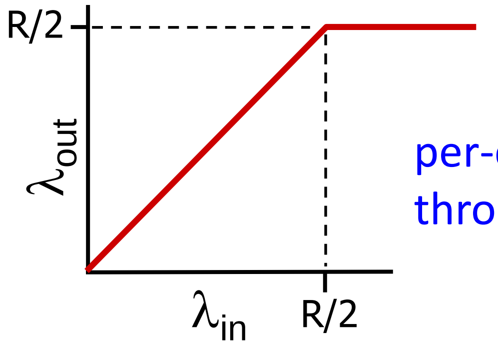
congestion:

- "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)
- a top-10 problem!

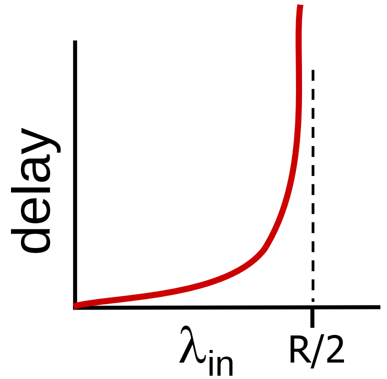
Causes/Cost of Congestion : scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no error recovery (i.e. no retransmission)





per-connection throughput



- max. per-connection throughput: R/2
- no matter how high their sending rates!
- as the router buffer is shared among them.

- large delays as arrival rate λ_{in} approaches capacity

Cont…

Conclusion:

- while operating at an aggregate throughput of near *R*
  - may be ideal from a throughput standpoint,
  - but, it is far from ideal from a delay standpoint!

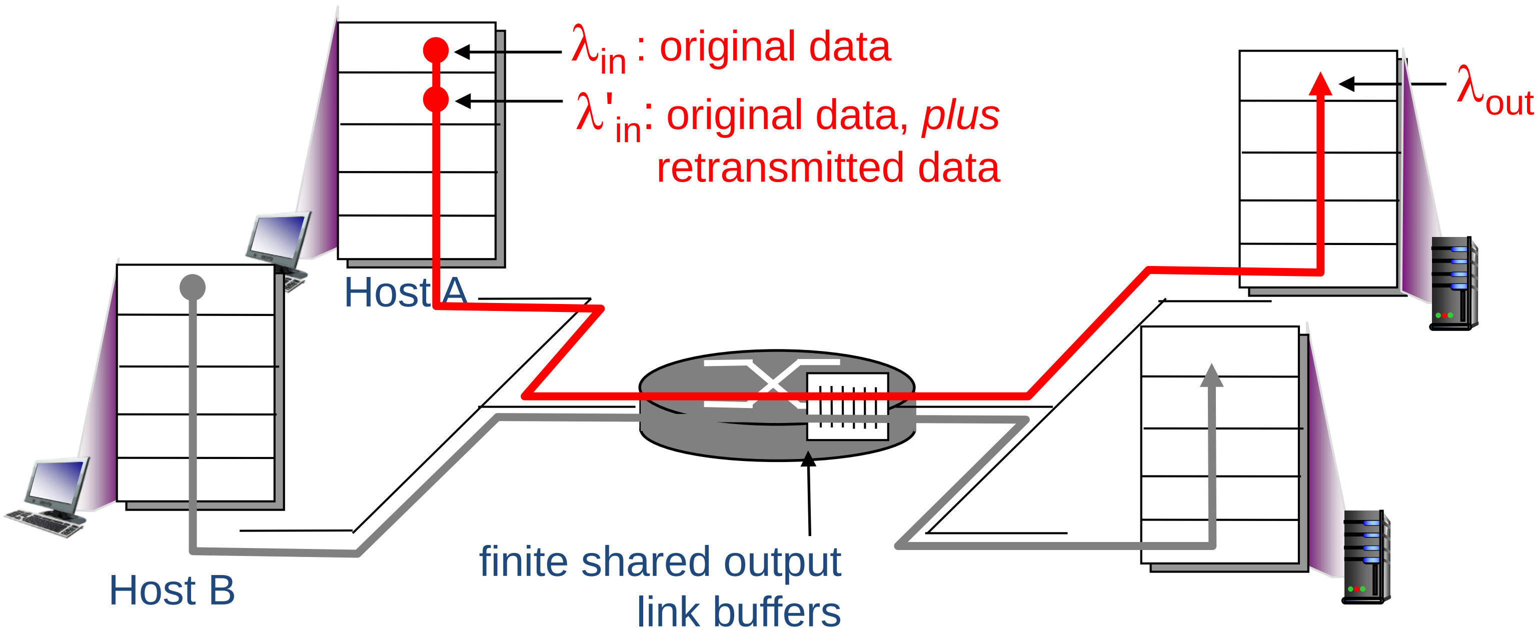
- Even in this (extremely) idealized scenario
  - “cost” of a congested network
    - large queuing delays are experienced as the packet arrival rate nears the link capacity.
    - assuming that: (i) the connections operate at these sending rates for an infinite period of time, (ii) there is an infinite amount of buffering available

      → the above delay between source and destination becomes infinite

Causes/costs of Congestion: scenario 2

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

- Sending rate: λ_{in} ; Offered load: λ'_{in}

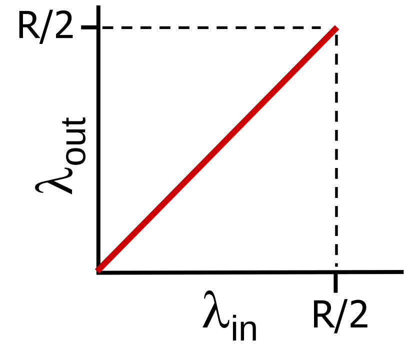


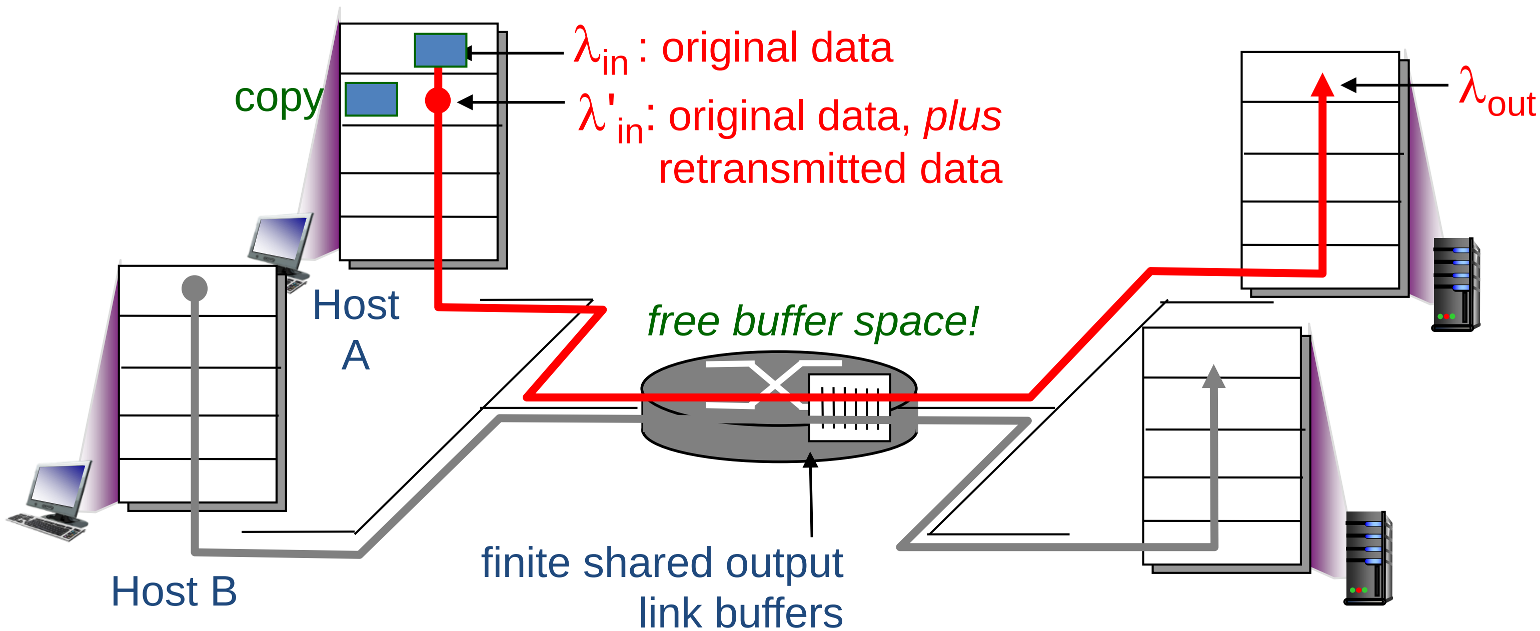
Cont…

Assume idealization: perfect knowledge with sender

- sender sends only when router buffers available

- so, no loss → $\lambda_{in} = \lambda_{out}$

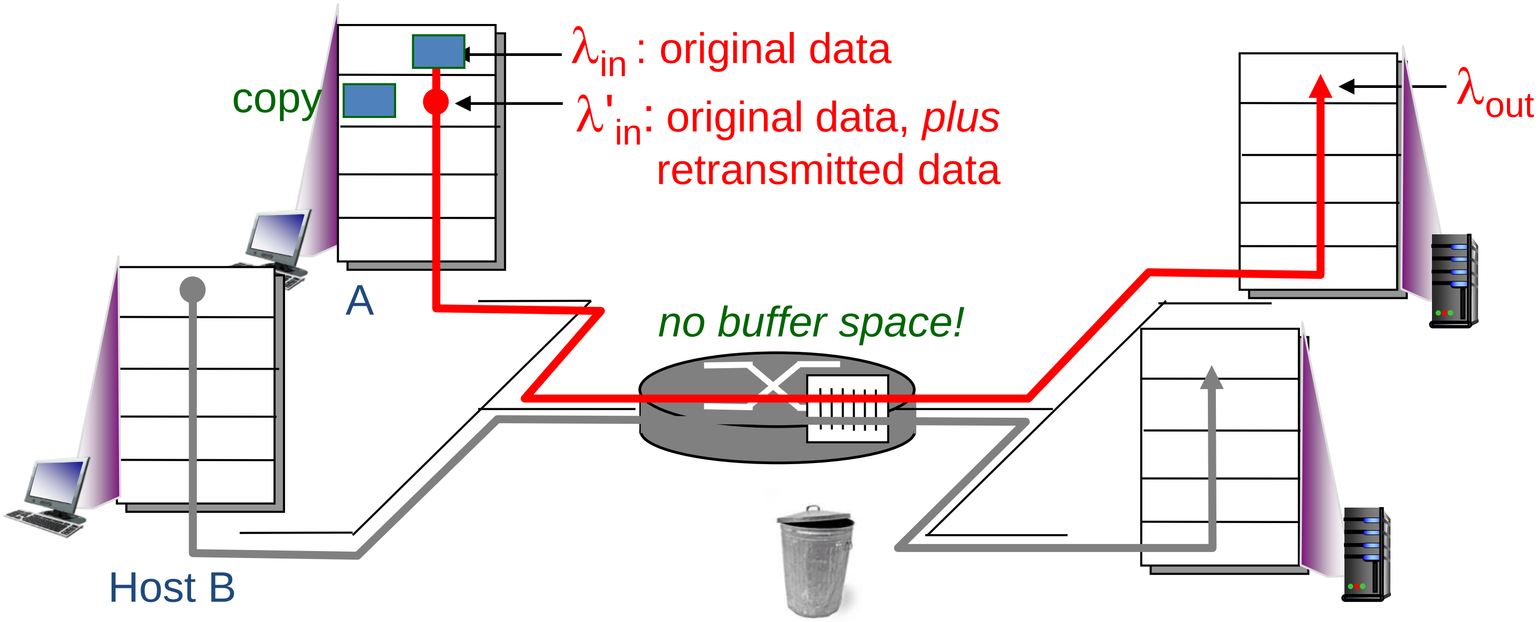




Cont…

Assume Idealization:
know when loss occur

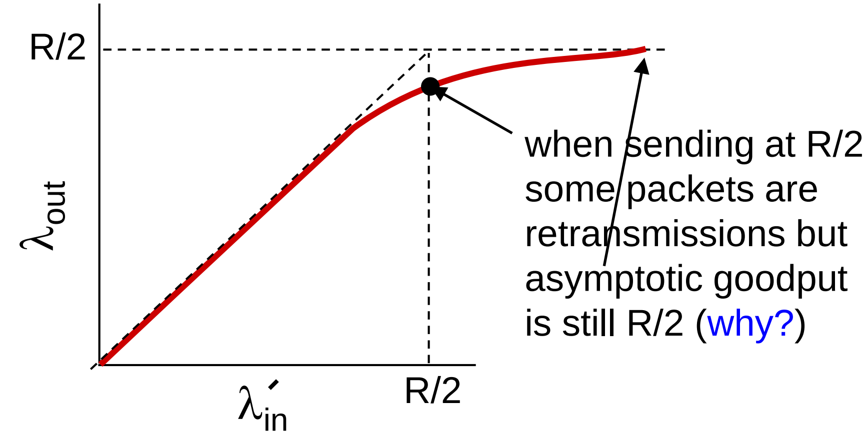
- sender only re-sends if packet *known* to be lost

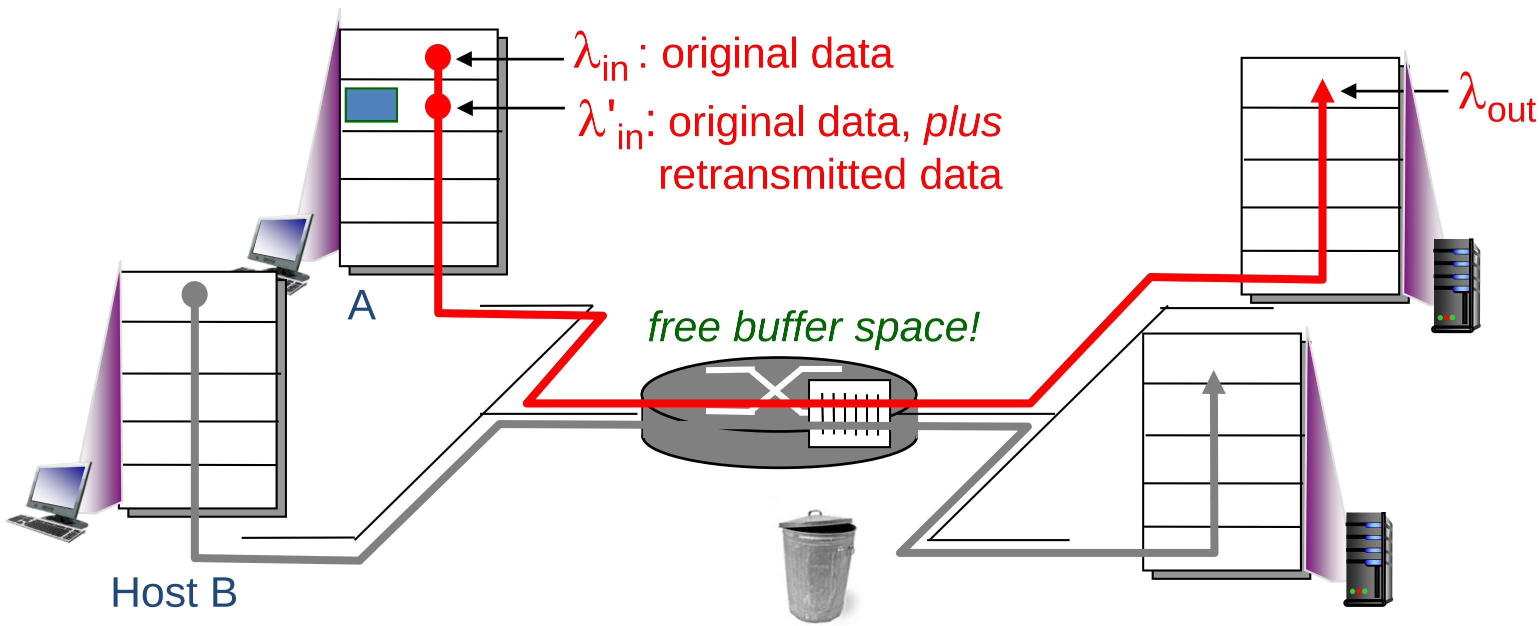


Cont…

Assume Idealization:
know when loss occur

- sender only re-sends if packet *known* to be lost

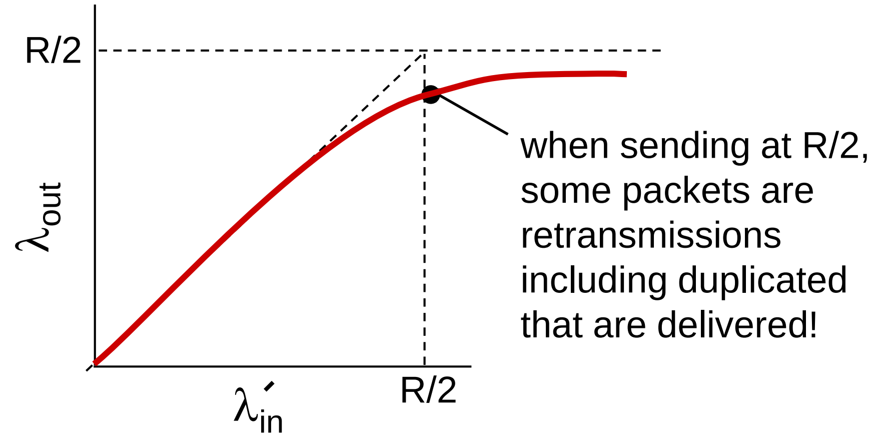


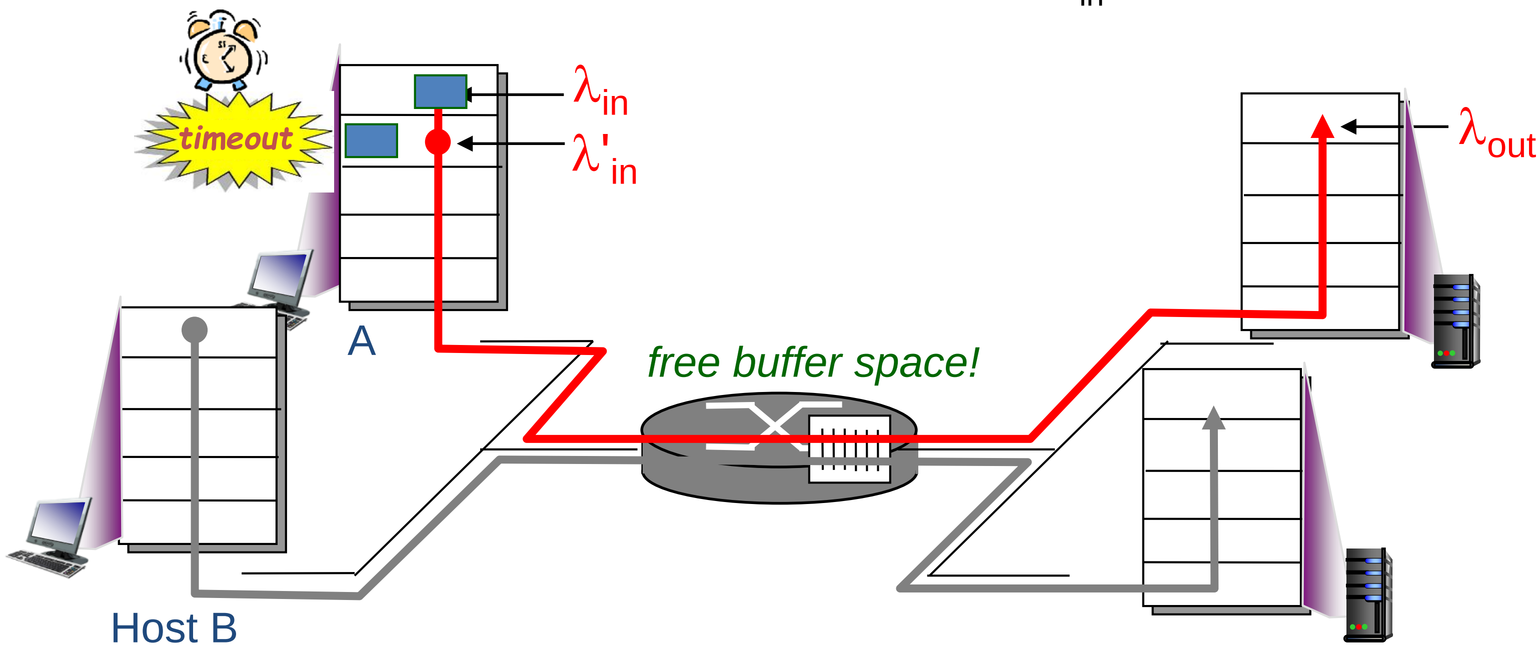


Cont...

Realistic: duplicates

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

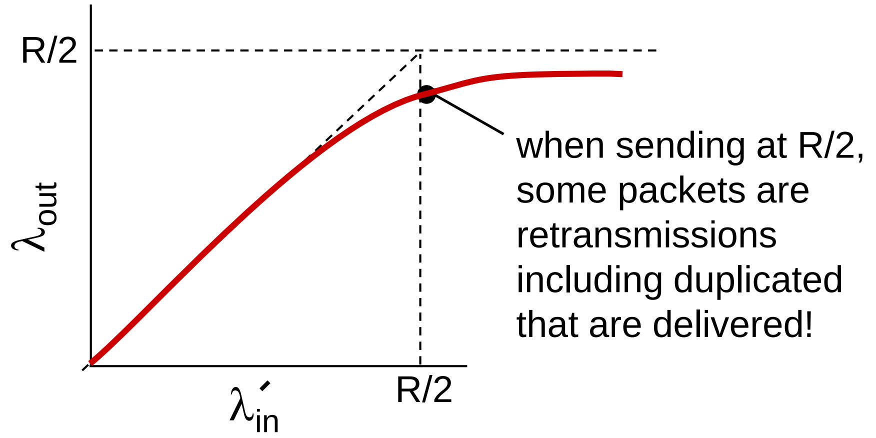




Cont…

Realistic: duplicates

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



"costs" of congestion:

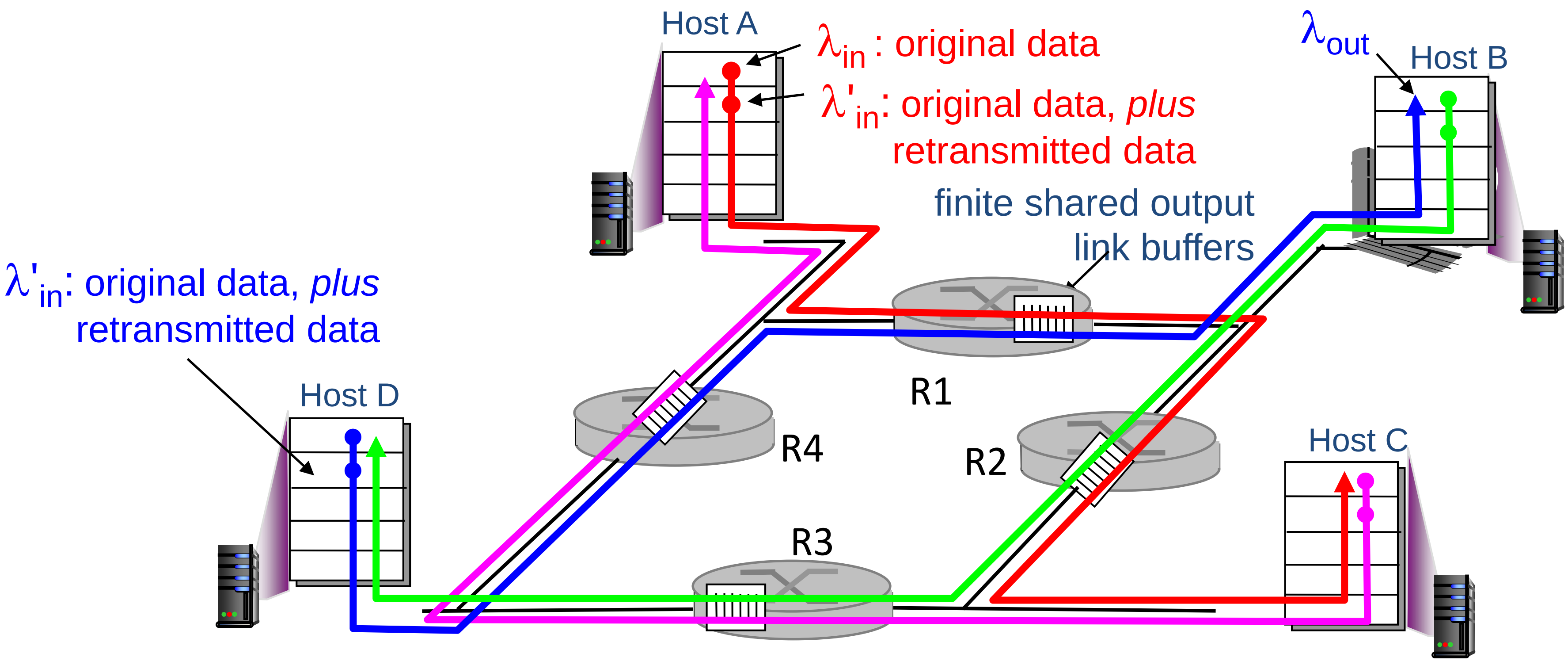
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit
- Overlapping paths
- all have same value of λ_{in}

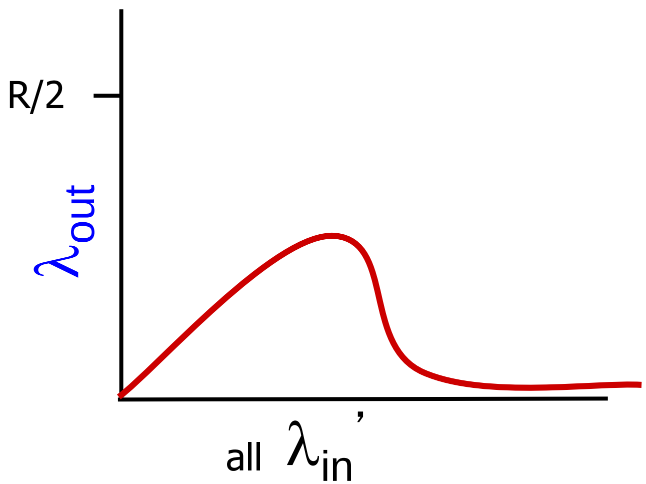
Q: what happens if all λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue (R1) are dropped; blue throughput → 0



Cont…

- For extremely small values of λ_{in}, buffer overflows are rare
  - the throughput approximately equals the offered load
- For slightly larger values of λ_{in}, overflows are still rare
  - the corresponding throughput is also larger,
- Thus, for small values of λ_{in}, an increase in λ_{in} results in an increase in λ_{out}



As red λ_{in}' increases, all arriving blue pkts at upper queue (in R1) are dropped, as R1 will give priority to red pkts;
So, blue throughput → 0

another "cost" of congestion:

- when packet is dropped, any "upstream transmission capacity" used for that packet was wasted! (e.g. work by R4 in above figure)

Congestion v/s Flow Control

- TCP cannot ignore the congestion in network (at the intermediate points) as it wants to provide end-to-end reliability
- The use of *flow control* in TCP cannot avoid congestion in intermediate routers because
  - a router may receive data from more than one sender
  - Flow control is for individual TCP sender
  - There is no congestion at the either end
  - there may be congestion in the middle.

Approaches to Congestion Control

two broad approaches towards congestion control:

end-to-end congestion control

- no explicit feedback from network
- congestion inferred from end-system who observed loss, delay
- approach taken by TCP
- suitable in datagram approach

network-assisted congestion control

- routers/switches provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM ABR)
  - explicit rate for sender to send at
  - Direct feedback: sent from a network router to the sender
  - Indirect feedback: router marks a field in a packet flowing from sender to receiver
- suitable for virtual-circuit approach

TCP Congestion control

- Basic approach:
  - each sender limit the rate at which it sends traffic into its connection
  - set the rate as a function of perceived network congestion.

- perceives less congestion along the path → increases its send rate
- perceives huge congestion along the path → reduces its send rate

- It should not aggressively send segments to the network
- It can not be very conservative, either, sending a small number of segments in each time interval

Cont…

- **Questions** need to answer:
  - How does a TCP sender limit the rate at which it sends traffic into its connection?
  - How does a TCP sender perceive that there is congestion on the path between itself and the destination?
  - What congestion control algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Answer of 1st Question:

- To control the number of segments to transmit, TCP uses another variable called Congestion Window (*cwnd*)
- Actually, the *cwnd* variable and the *rwnd* variable (used for flow control) together define the size of the send window in TCP
  - Actual send window size = min *(rwnd, cwnd)*
- The constraint above limits the amount of unacknowledged data at the sender and therefore indirectly limits the sender's send rate.

Cont…

Answer of 2nd Question:

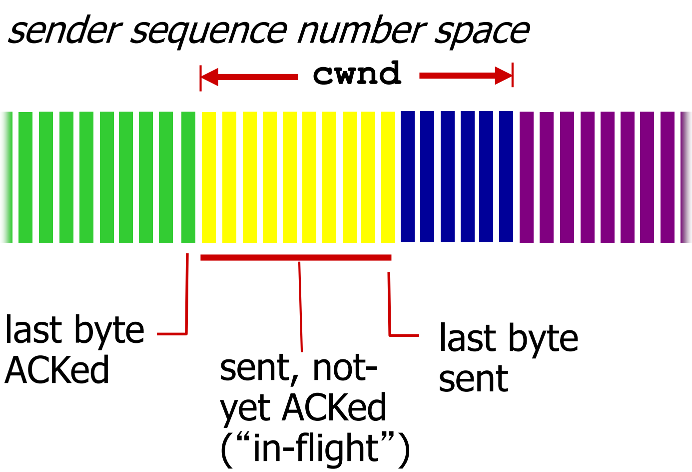
- TCP sender uses the occurrence of two events as signs of congestion:
  - time-out
  - 3 duplicate ACKs

Answer of 3rd Question:

- There exist many congestion control algorithm for adjusting the value of *cwnd* based upon end-to-end congestion
  - Default/basic approach

- Modified TCP with congestion control algorithms
  - Tahoe TCP: both signs of occurrence are treated equally
  - Reno TCP: both signs of occurrence are treated differently
  - New Reno TCP: TCP checks to see if more than one segment is lost in the current window when 3 duplicate ACKs arrive

TCP Congestion Control: details



- sender limits transmission:

$$\boxed{\texttt{LastByteSent-LastByteACKed} \le \texttt{cwnd}}$$

- **cwnd** is dynamic, function of perceived network congestion

TCP sending rate:

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

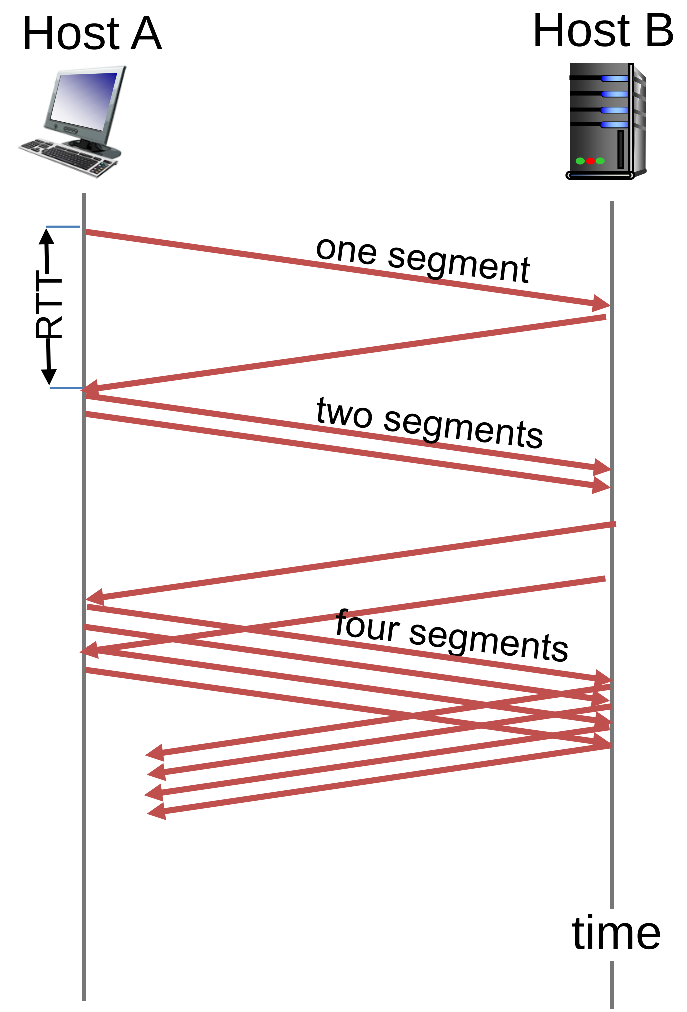
$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP congestion control algo has three components:

- *slow start*
- *congestion avoidance*
- *fast recovery*

Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS (maximum-sized segments)
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast
- This process results in a doubling of the sending rate every RTT.



When growth ends?

- 1st case, a loss event indicated by a timeout 
  - Indicates congestion
  - *cwnd* sets to 1 MSS
  - begins the **slow start** process anew.
  - *ssthresh* (slow start threshold) sets to *cwnd/2*.

- 2nd case, when the value of *cwnd* equals *ssthresh*,
  - TCP transitions into **congestion avoidance** state
  - *cwnd* grows linearly

- 3rd case, if 3 duplicate ACKs are detected,
  - dupACKs indicate network capable of delivering some segments
  - TCP performs a fast retransmit and enters **fast recovery** state
  - *ssthresh* sets to *cwnd/2*.
  - *cwnd* sets to *ssthresh + 3 MSS*.
  - *cwnd* grows linearly

Slow-start strategy is slower in the case of delayed ACK.

If two segments are ACKed cumulatively, the size of the *cwnd* increases by 1, not 2. With one ACK for every two segments, the growth is a power of 1.5, but still exponential

Congestion Avoidance

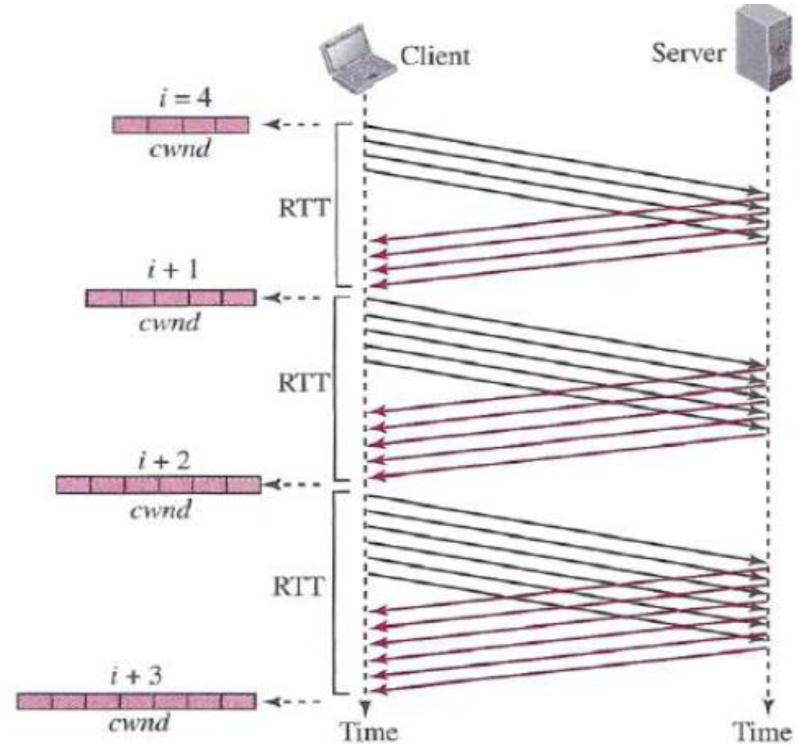
- On entry to this state, the value of *cwnd* is approx half its value when congestion was last encountered

- To avoid congestion before it happens, we must slow down the exponential growth of *cwnd*

- the additive phase begins.

- If 3 dupACKs are detected at this state,
  - TCP performs a fast retransmit and enters the fast recovery state
- If timeout occurs at this state
  - TCP enters into slow start

If a new ACK arrives, cwnd = cwnd + MSS. (MSS/cwnd)



Fast Recovery

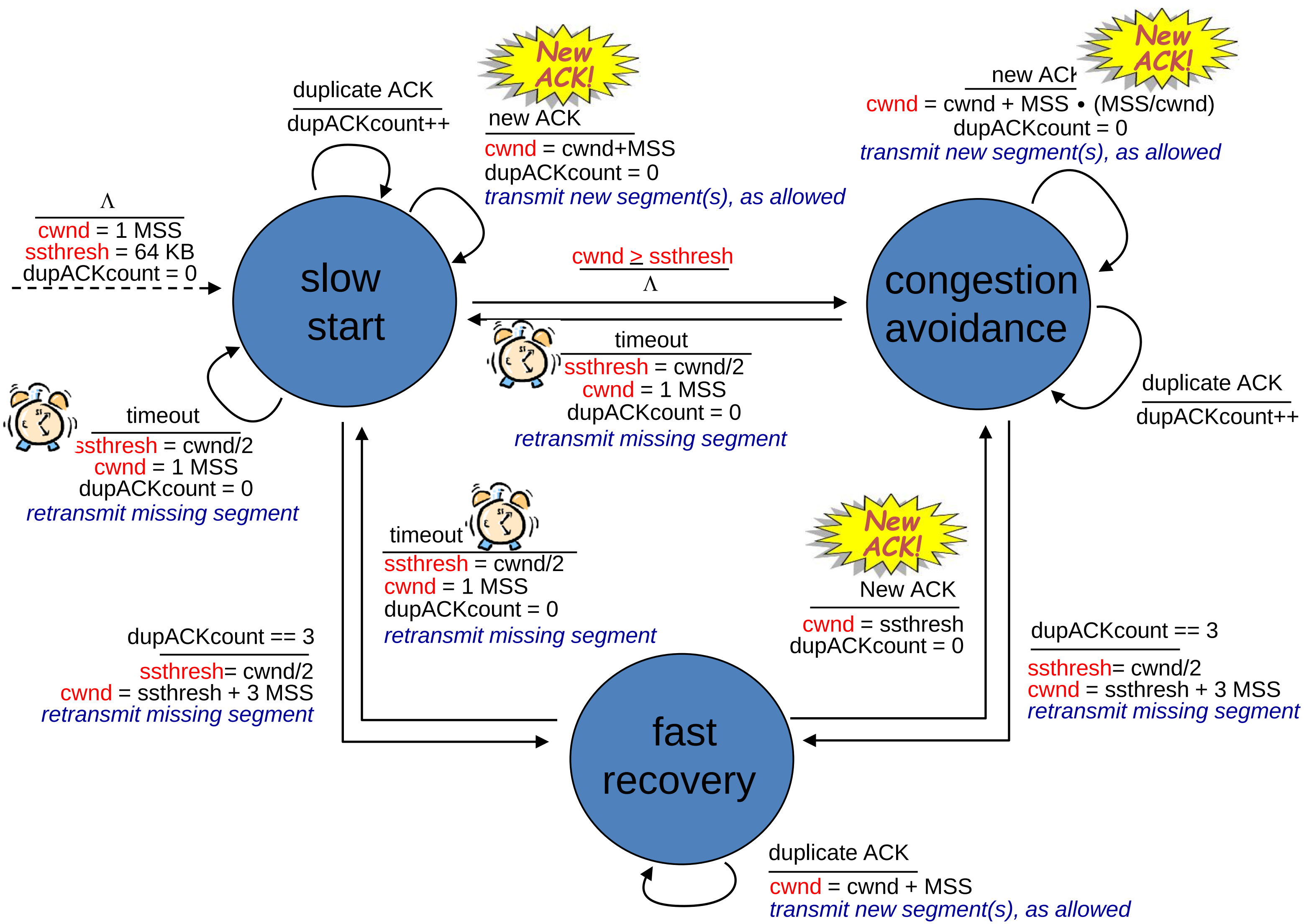
- this algorithm is also an additive increase, but it starts when 3 duplicate ACK arrives

- If a duplicate ACK arrives (after the 3 duplicate ACK which triggers the recovery)
  - *cwnd = cwnd + (1/ cwnd)*

- If timeout occurs, TCP moves back to **slow start** state
- If any new ACK arrives, TCP moves back to **congestion avoidance** state

- This state is recommended, but not mandatory in TCP

FSM of TCP Congestion Control



Different Versions

- TCP Tahoe
  - signs of congestion occurrence (time-out, 3 duplicate ACK) are treated equally
  - uses only *slow start* and *congestion avoidance* states

- TCP Reno
  - signs of congestion occurrence (time-out, 3 duplicate ACK) are treated differently
  - three states in FSM: *slow start*, *congestion avoidance, fast recovery*

- TCP New Reno
  - It differs from RENO in that it doesn't exit fast-recovery until all the data which was outstanding at the time it entered fast recovery is ACKed.
  - It is most common today

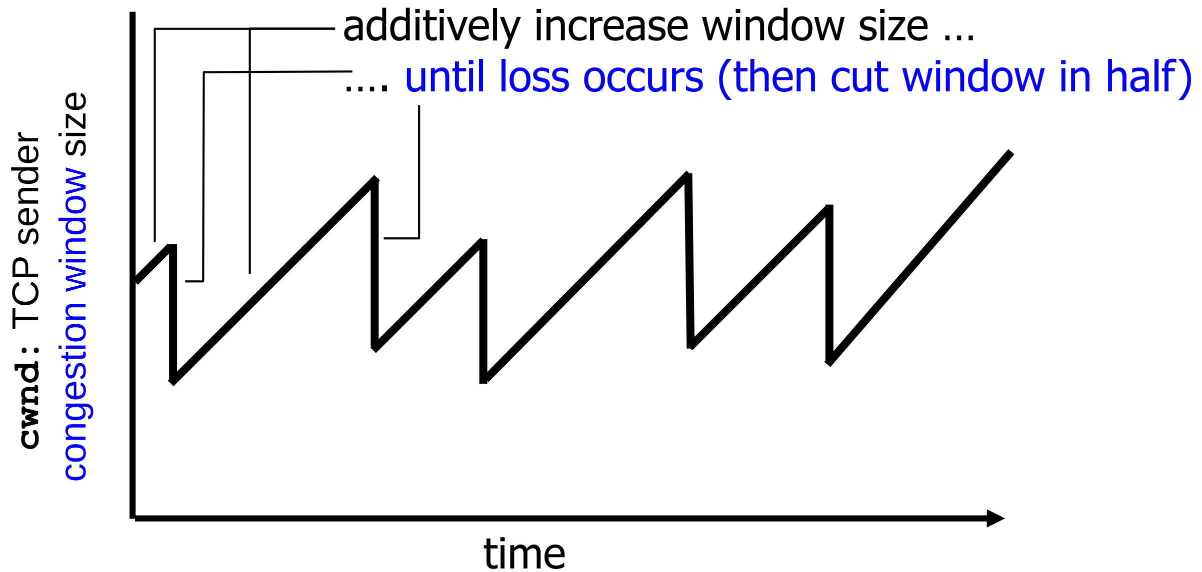
- TCP Vegas
  - variations of the Reno algorithm
  - attempts to avoid congestion while maintaining good throughput
  - The basic idea of Vegas is to
    - (1) detect congestion in the routers between source and destination *before* packet loss occurs, and
    - (2) lower the rate linearly when this imminent packet loss is detected.

Additive Increase Multiplicative Decrease

- TCP congestion control is often refereed to as AIMD form of congestion control.

❖ *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase window by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut window in half after loss

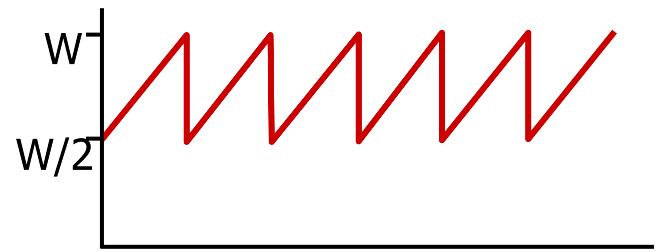
AIMD saw-toothed behavior: probing for bandwidth



TCP Throughput

- What the average throughput of a long-lived TCP connection would be?
- we'll ignore the slow-start phases that occur after timeout events as these phases are typically very short.
- the rate at which TCP sends data is a function of *cwnd* and current *RTT*
  - Rate = cwnd/RTT
- Let, *cwnd* = *W* when a loss event occurs.

If we ignore slow-start then



- Assume that *RTT* and *W* are approximately constant over the duration of the connection (i.e. in steady-state)
  - the TCP transmission rate ranges from (*W* /2 *RTT*) to (*W* /*RTT*)
- So, the average throughput of a connection = ½ ((*W* /2 *RTT*) + (*W* /*RTT*)) = 0.75*(W/RTT)

TCP over "High-Bandwidth" path

- Example of high speed TCP needed in present era:
  - 1500 byte segments, 100ms RTT,
  - We want 10 Gbps throughput

- So, using previous formula --> it requires W = 83,333 in-flight segments
- What would happen the case of loss?

- throughput in terms of segment loss probability, L [Mathis 1997]:

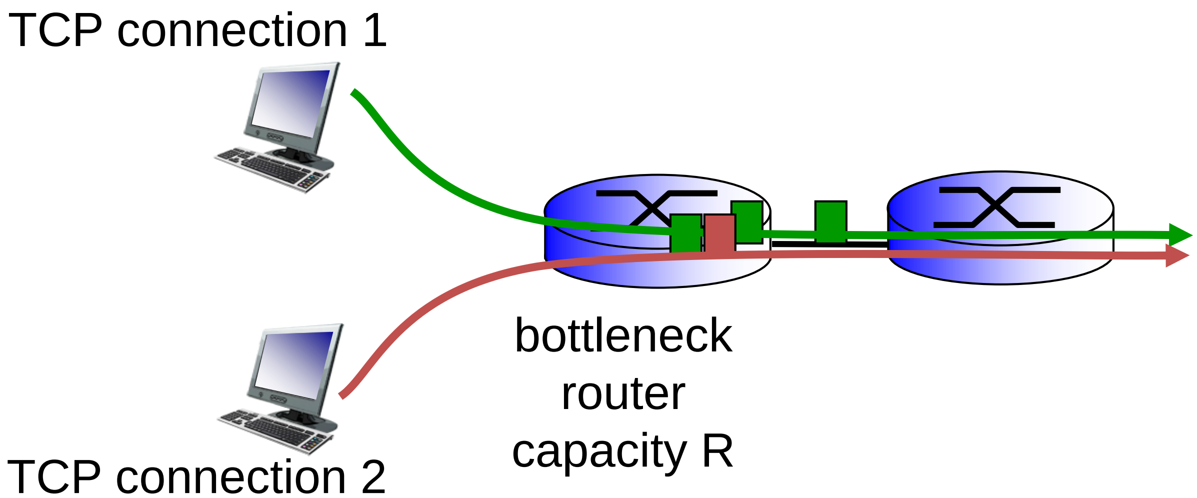
$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$
it means very small loss rate!

- new versions of TCP for high-speed

TCP Fairness

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



Thanks!

Content of this PPT are taken from:

1) **Computer Networks: A Top Down Approach**, by J.F. Kuros and K.W. Ross, 6th Eds, 2013, Pearson Education.

2) **Data Communications and Networking**, by B. A. Forouzan , 5th Eds, 2012, McGraw-Hill.

3) **Chapter 3 : Transport Layer,** PowerPoint slides of "Computer Networking: A Top Down Approach", 6th Eds, J.F. Kurose, K.W. Ross