COMP 3331/9331: Computer Networks and Applications

Week 6

Transport Layer (continued)

Reading Guide:

Chapter 3: Section 3.7

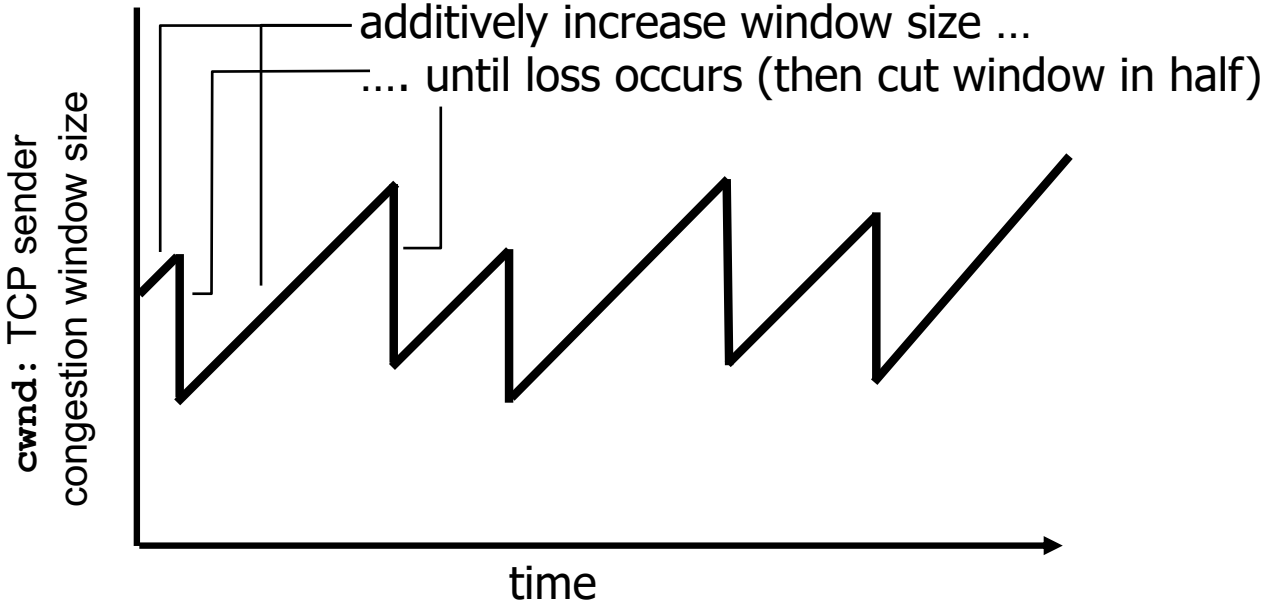
Adjusting to Varying Bandwidth

- Slow start gave an estimate of available bandwidth
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (rate decrease)
  - Known as Congestion Avoidance (CA)
- TCP uses: "Additive Increase Multiplicative Decrease" (AIMD)
  - We'll see why shortly…

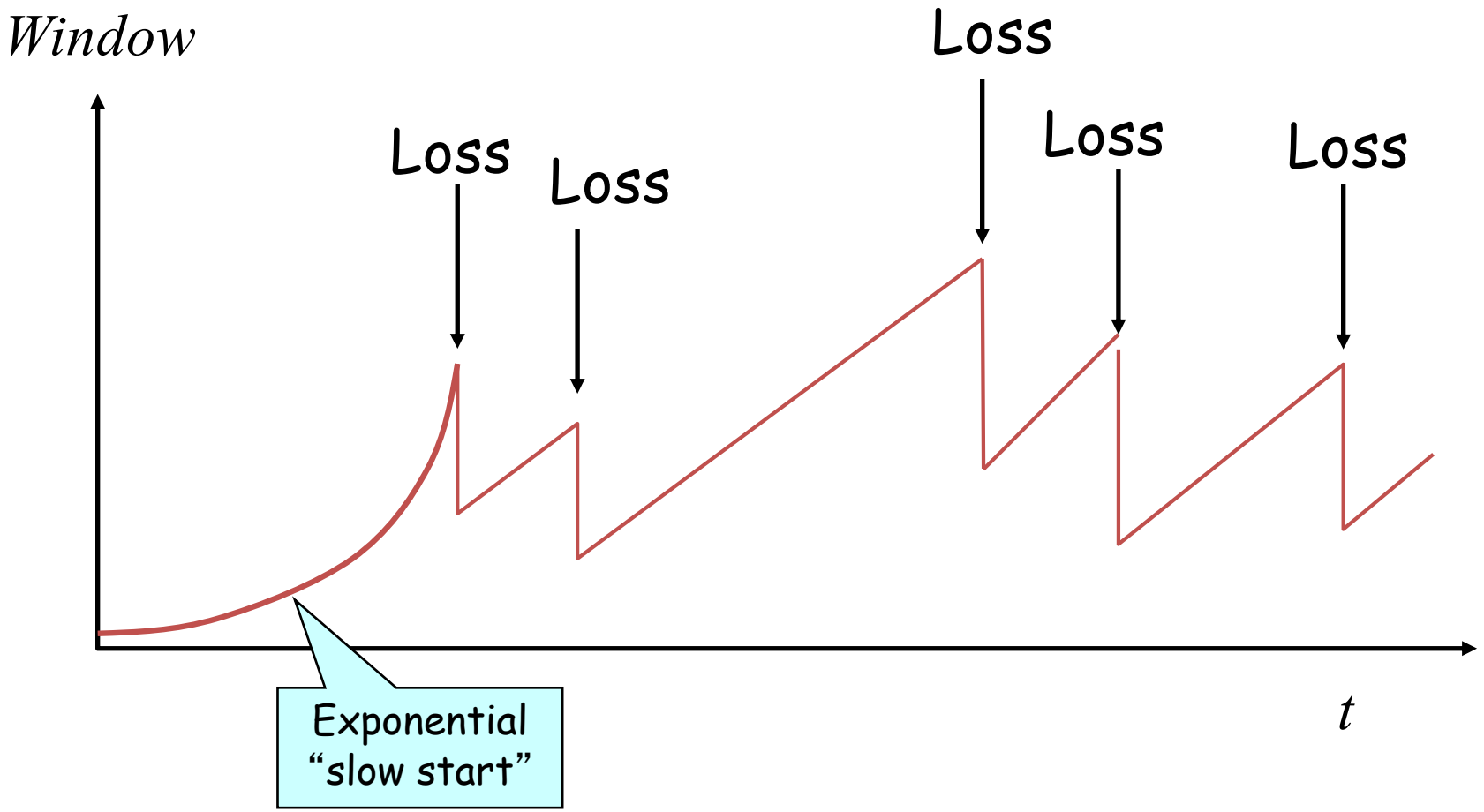
AIMD

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until another congestion event occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
    - For each successful RTT (all ACKS), **`cwnd = cwnd +1`**
    - Simple implementation: for each ACK, **`cwnd = cwnd + 1/cwnd`**
  - *multiplicative decrease:* cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



Leads to the TCP "Sawtooth"



Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?

- Introduce a “slow start threshold” (ssthresh)
  - Initialized to a large value

- Convert to AI when cwnd = ssthresh, sender switches from slow-start to AIMD-style increase
  - On timeout, ssthresh = CWND/2

Implementation

- State at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - [Also dupACKcount and timer, as before]

- Events
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

- *2*MSS packets per ACK*
- *Hence after one RTT (All ACKs with no drops):*
  CWND = 2xCWND

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

Slow start phase

- Else
  - CWND = CWND + 1/CWND

"Congestion Avoidance" phase (additive increase)

- *Hence after one RTT (All ACKs with no drops):*
  CWND = CWND + 1

Event: dupACK

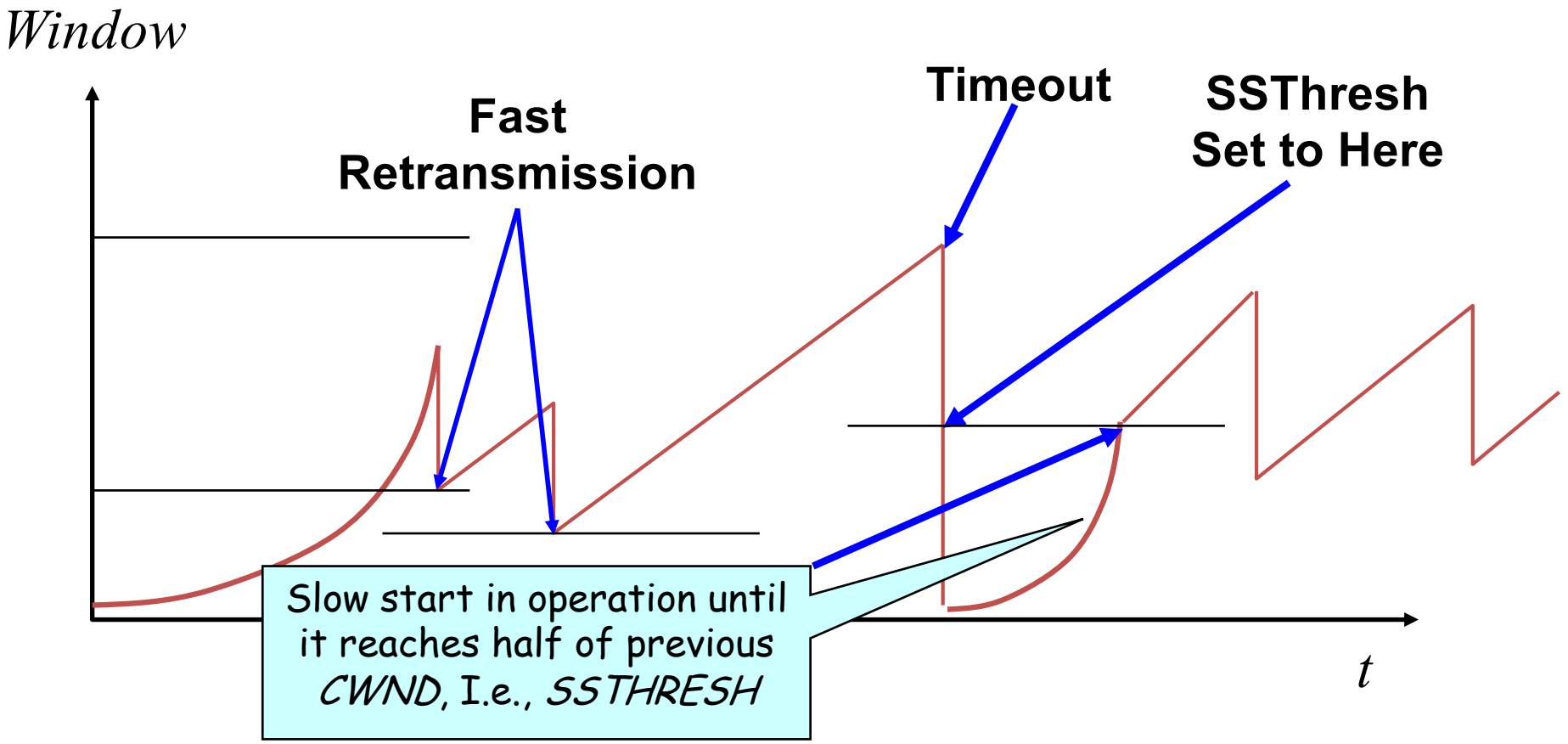
- dupACKcount ++

- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2

Event: TimeOut

- On Timeout
  - ssthresh = CWND/2
  - CWND = 1

Example



Slow-start restart: Go back to CWND = 1 MSS, but take advantage of knowing the previous value of CWND

One Final Phase: Fast Recovery (improved)

- The problem: Fast retransmit slow in recovering from an isolated loss

Example (window in units of MSS, not bytes)

- Consider a TCP connection with:
  - CWND=10 packets (of size MSS, which is 100 bytes)
  - Last ACK was for byte # 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 201, 301,…, 1001] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

Timeline

- ACK 101 (due to 201) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 301) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 401) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 501) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 601) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 701) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 801) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 901) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 1001) cwnd=6 + 1/6 (no xmit)
- ACK 1101 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK “clocking” (to increase CWND) stalls for another RTT

Solution: Fast Recovery (improved)

Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

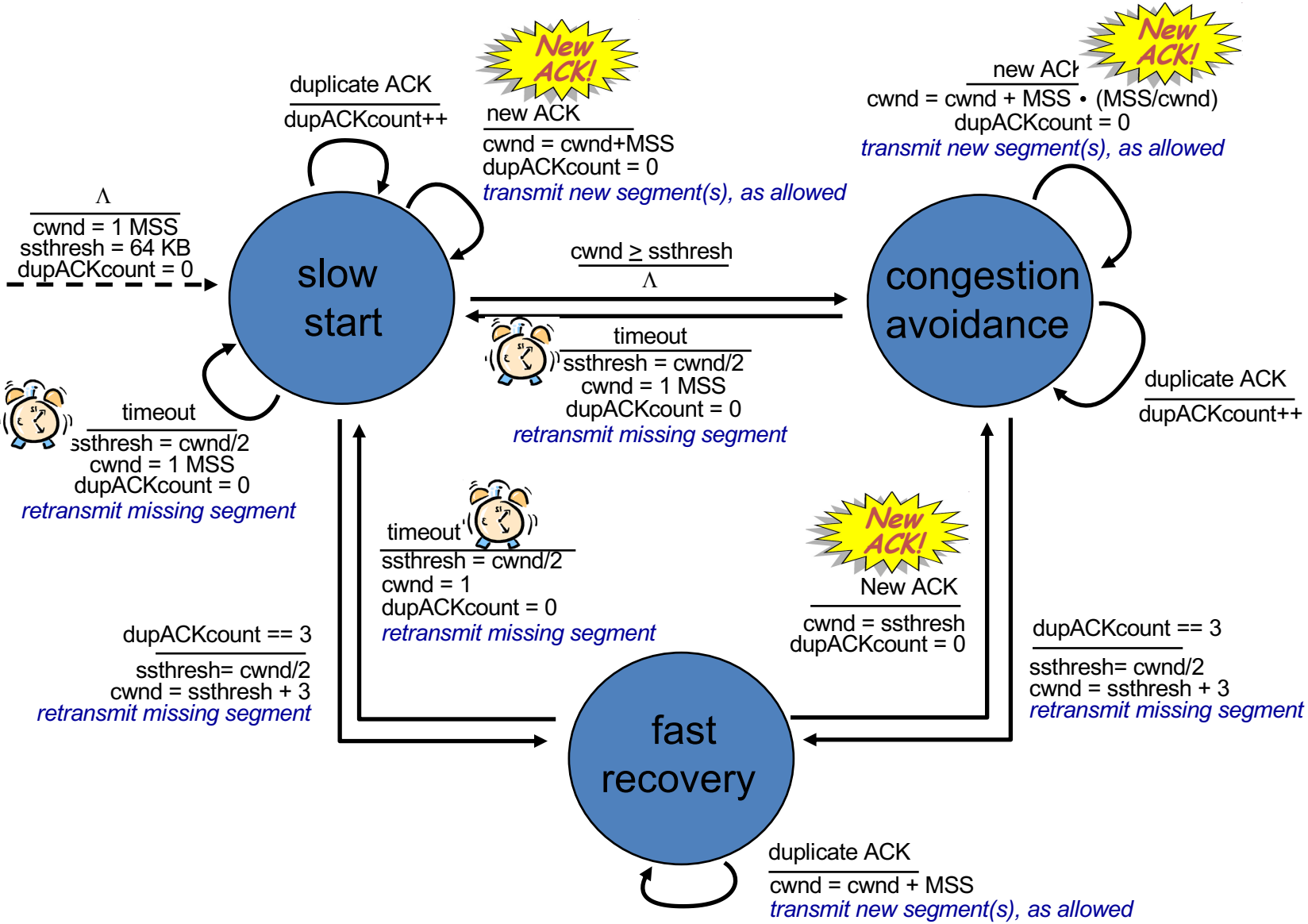
Example

- Consider a TCP connection with:
  - CWND=10 packets (of size MSS = 100 bytes)
  - Last ACK was for byte # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 201, 301,…, 1001] are in flight
  - Packet 101 is dropped

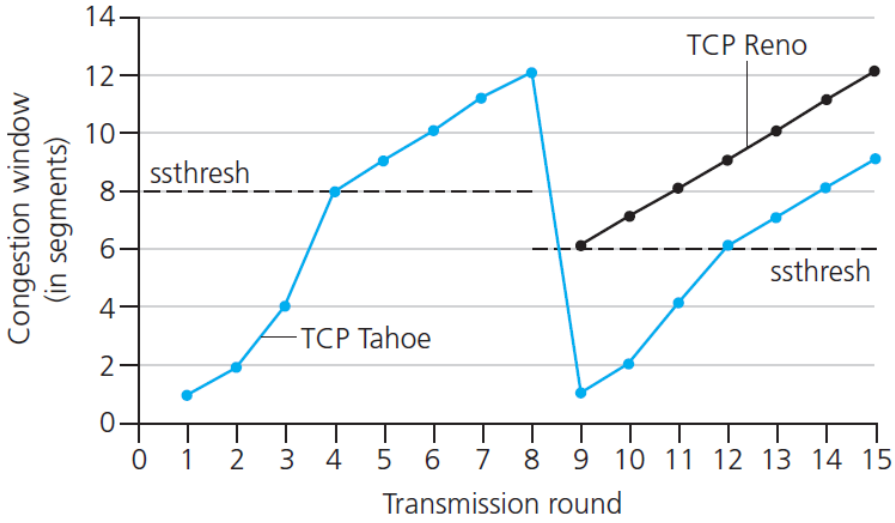
Timeline

- ACK 101 (due to 201) cwnd=10 dup#1
- ACK 101 (due to 301) cwnd=10 dup#2
- ACK 101 (due to 401) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 501) cwnd= 9 (no xmit)
- ACK 101 (due to 601) cwnd=10 (no xmit)
- ACK 101 (due to 701) cwnd=11 (xmit 1101)
- ACK 101 (due to 801) cwnd=12 (xmit 1201)
- ACK 101 (due to 901) cwnd=13 (xmit 1301)
- ACK 101 (due to 1001) cwnd=14 (xmit 1401)
- ACK 1101 (due to 101) cwnd = 5 (xmit 1501) ← exiting fast recovery
- Packets 1101-1401 already in flight
- ACK 1201 (due to 1101) cwnd = 5 + 1/5 ← back in congestion avoidance

Summary: TCP Congestion Control



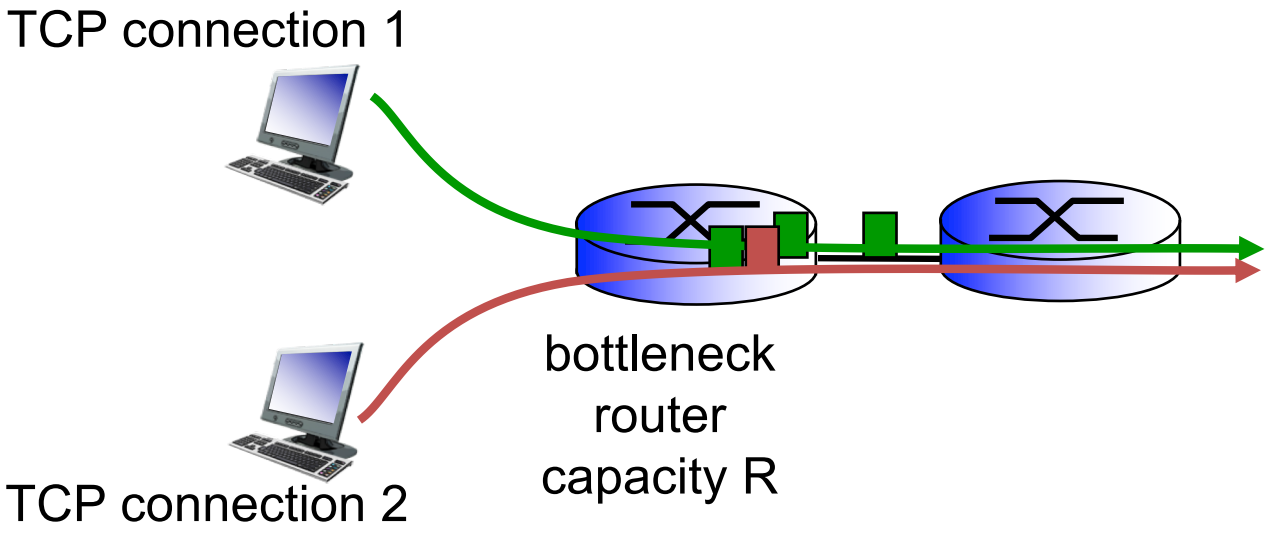
TCP Flavours



- TCP-Tahoe
  - cwnd =1 on triple dup ACK & time
- TCP-Reno
  - cwnd =1 on timeout
  - cwnd = cwnd/2 on triple dup ACK
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK (NOT COVERED IN THE COURSE)
  - incorporates selective acknowledgements

TCP Fairness

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



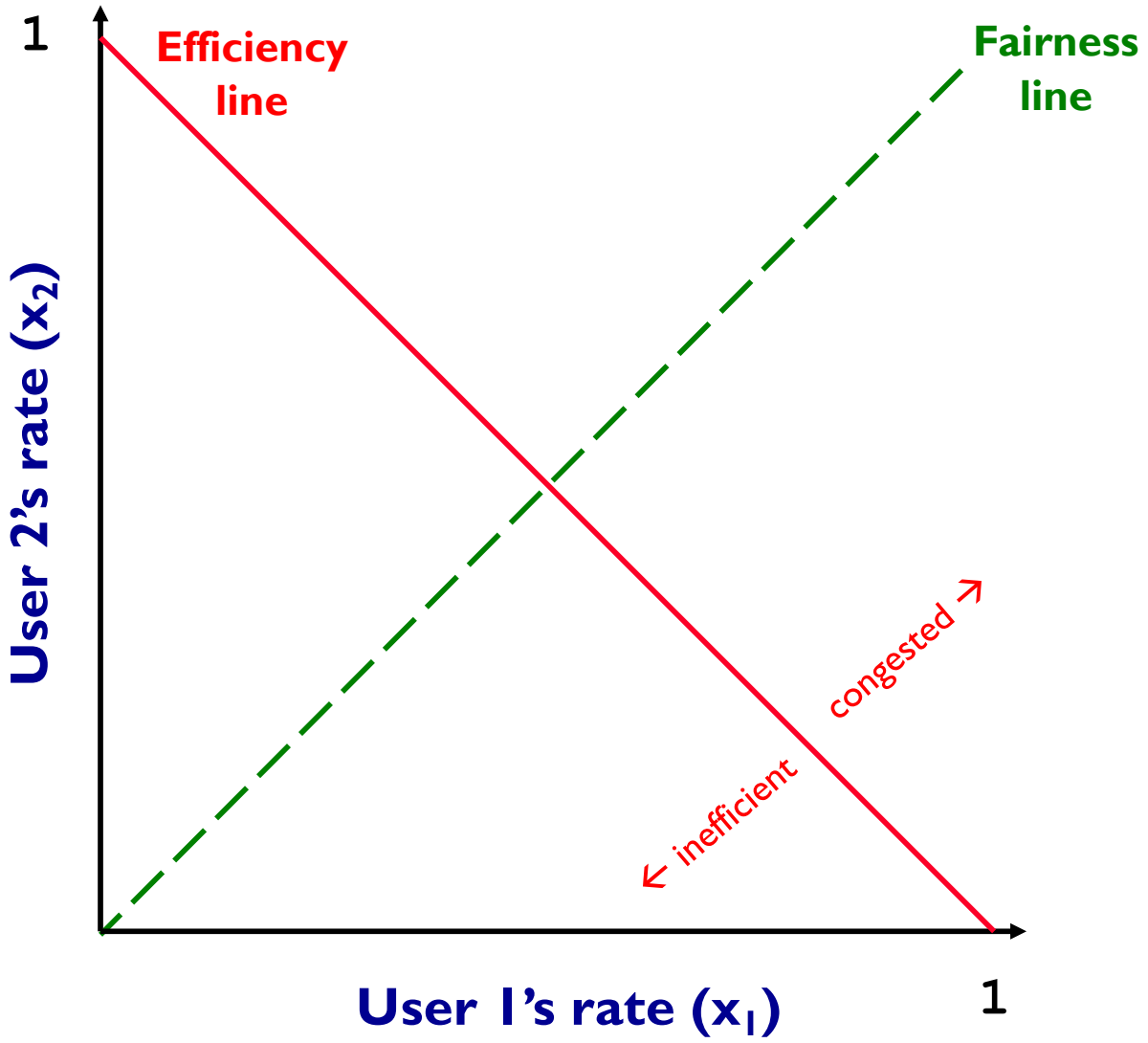
Why AIMD?

- Some rate adjustment options: Every RTT, we can
  - Multiplicative increase or decrease: CWND→ a*CWND
  - Additive increase or decrease: CWND→ CWND + b

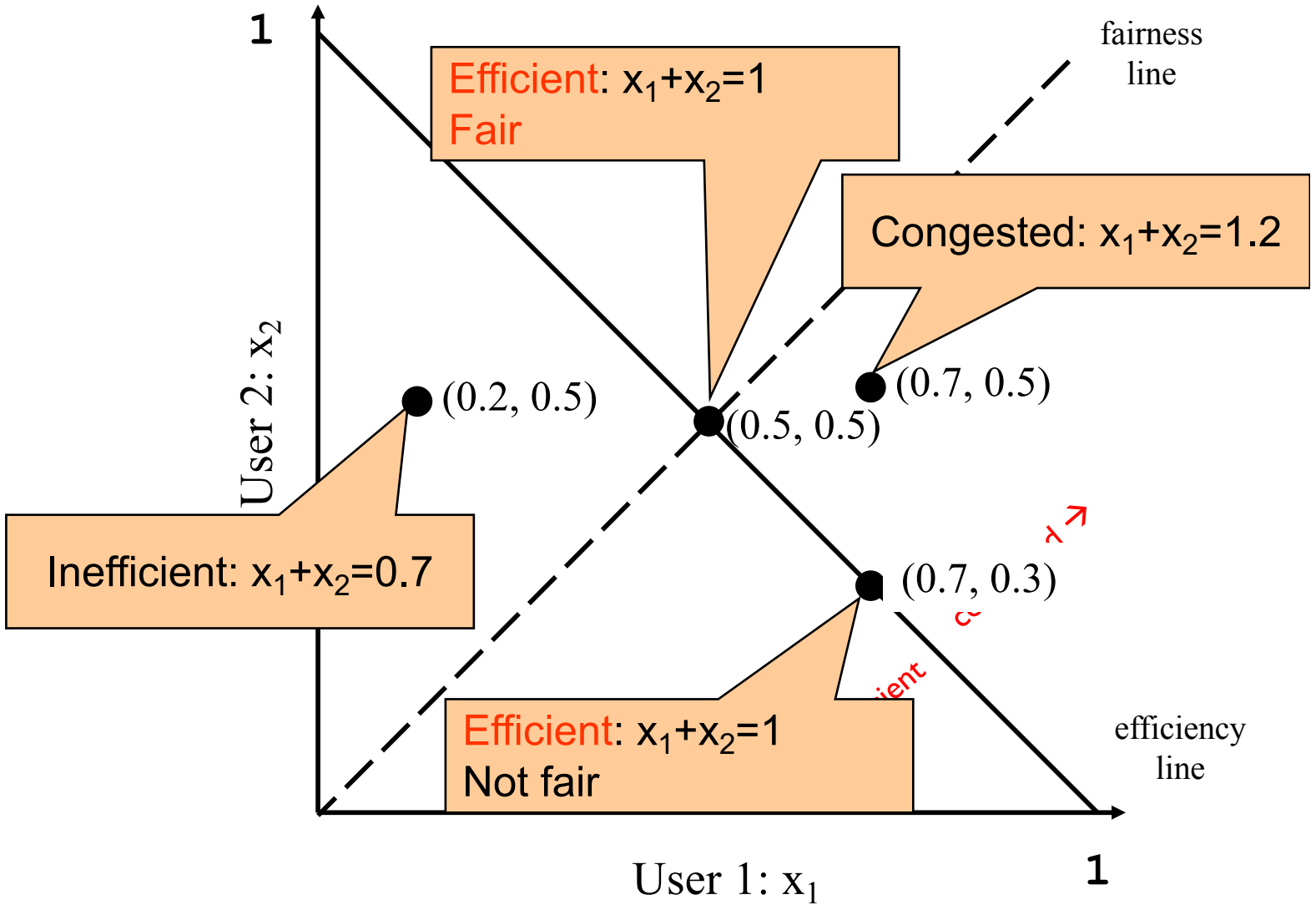
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase and decrease

Simple Model of Congestion Control

- Two users
  - rates x_1 and x_2
- Congestion when $x_1+x_2 > 1$
- Unused capacity when $x_1+x_2 < 1$
- Fair when $x_1 = x_2$

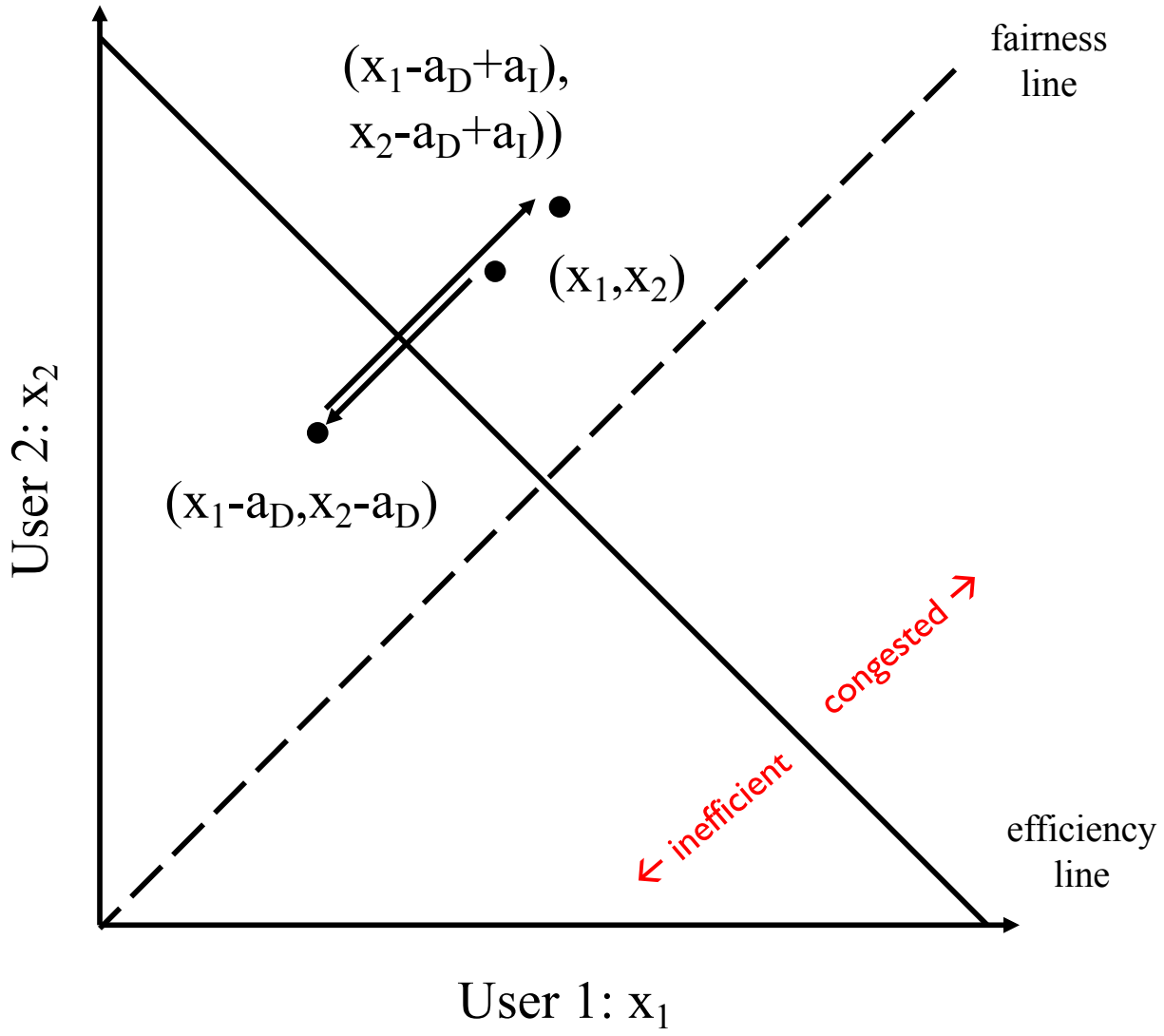


Example

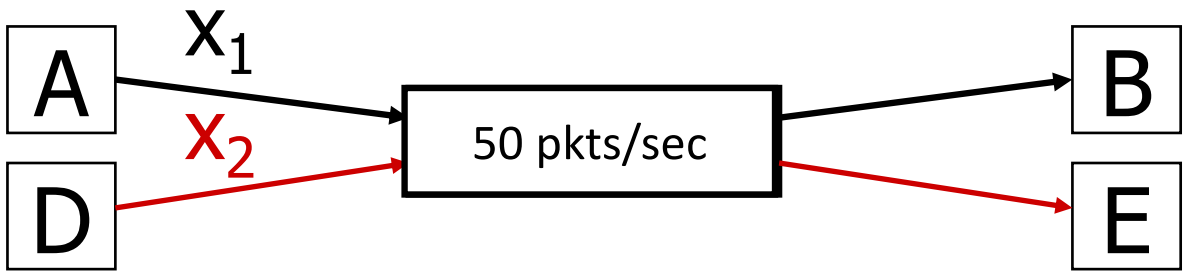

1
fairness line
Efficient: $x_1+x_2=1$
Fair
Congested: $x_1+x_2=1.2$
User 2: x_2
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
Inefficient: $x_1+x_2=0.7$
(0.7, 0.3)
Efficient: $x_1+x_2=1$
Not fair
efficiency line
User 1: x_1
1

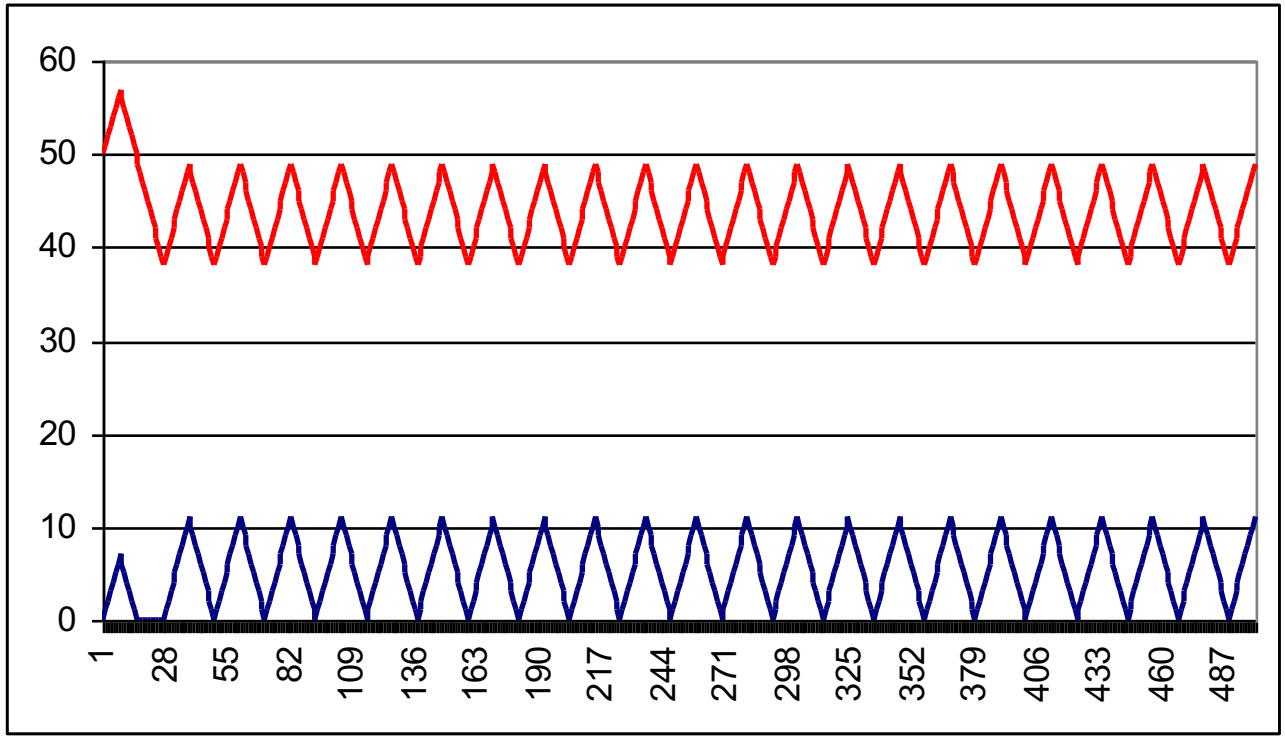
AIAD

- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness



AIAD Sharing Dynamics

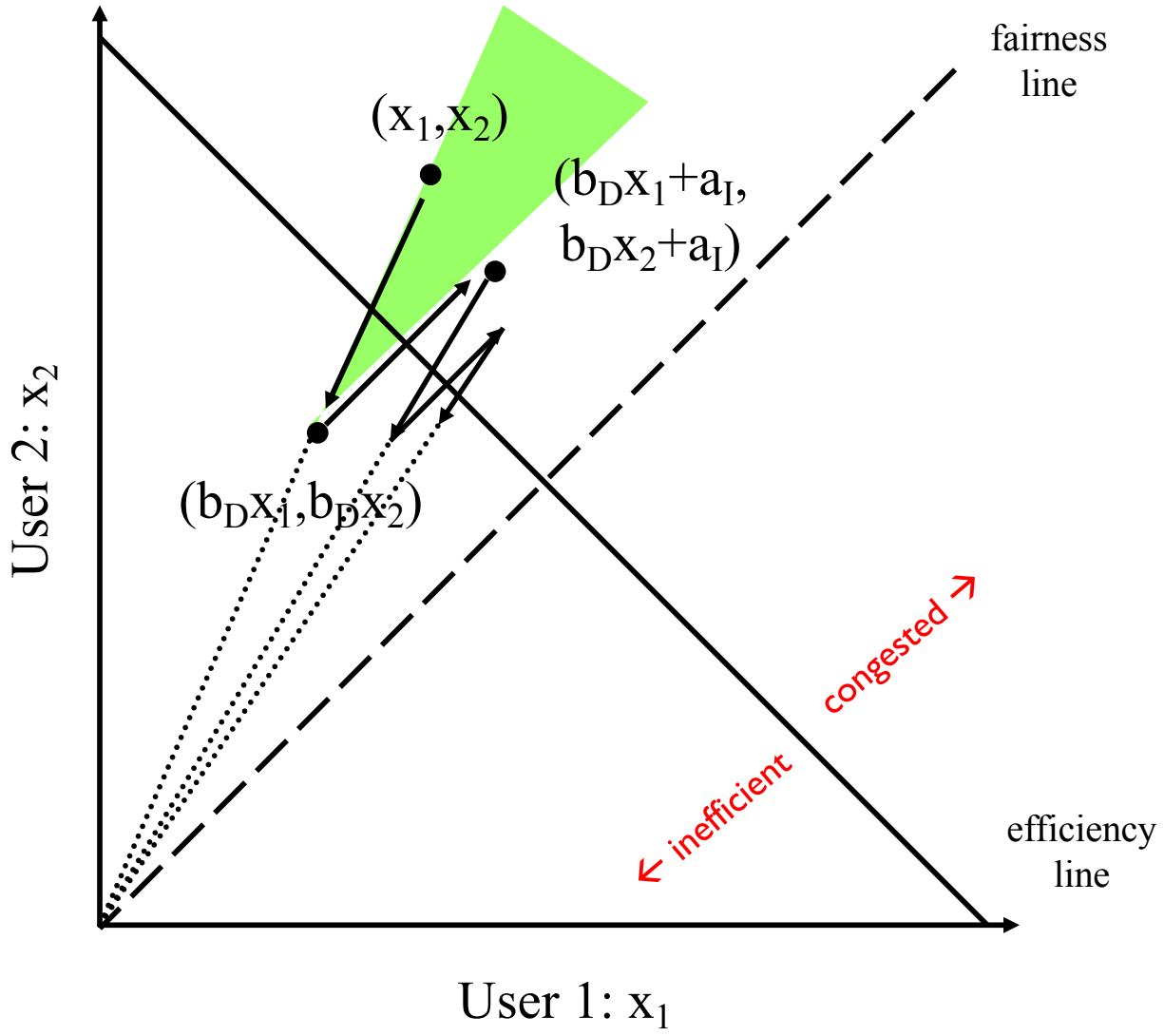




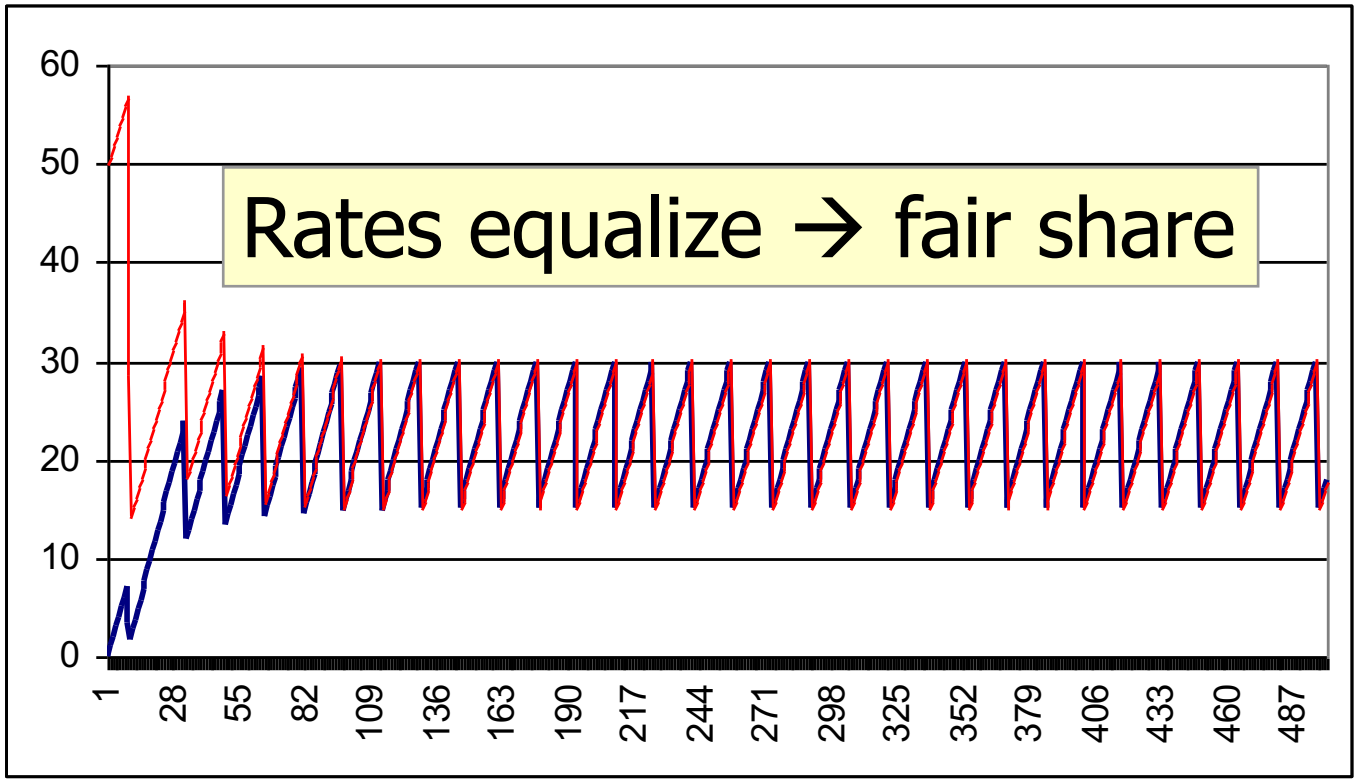
25

AIMD

- Increase: $x+a_I$
- Decrease: $x*b_D$
- Converges to fairness

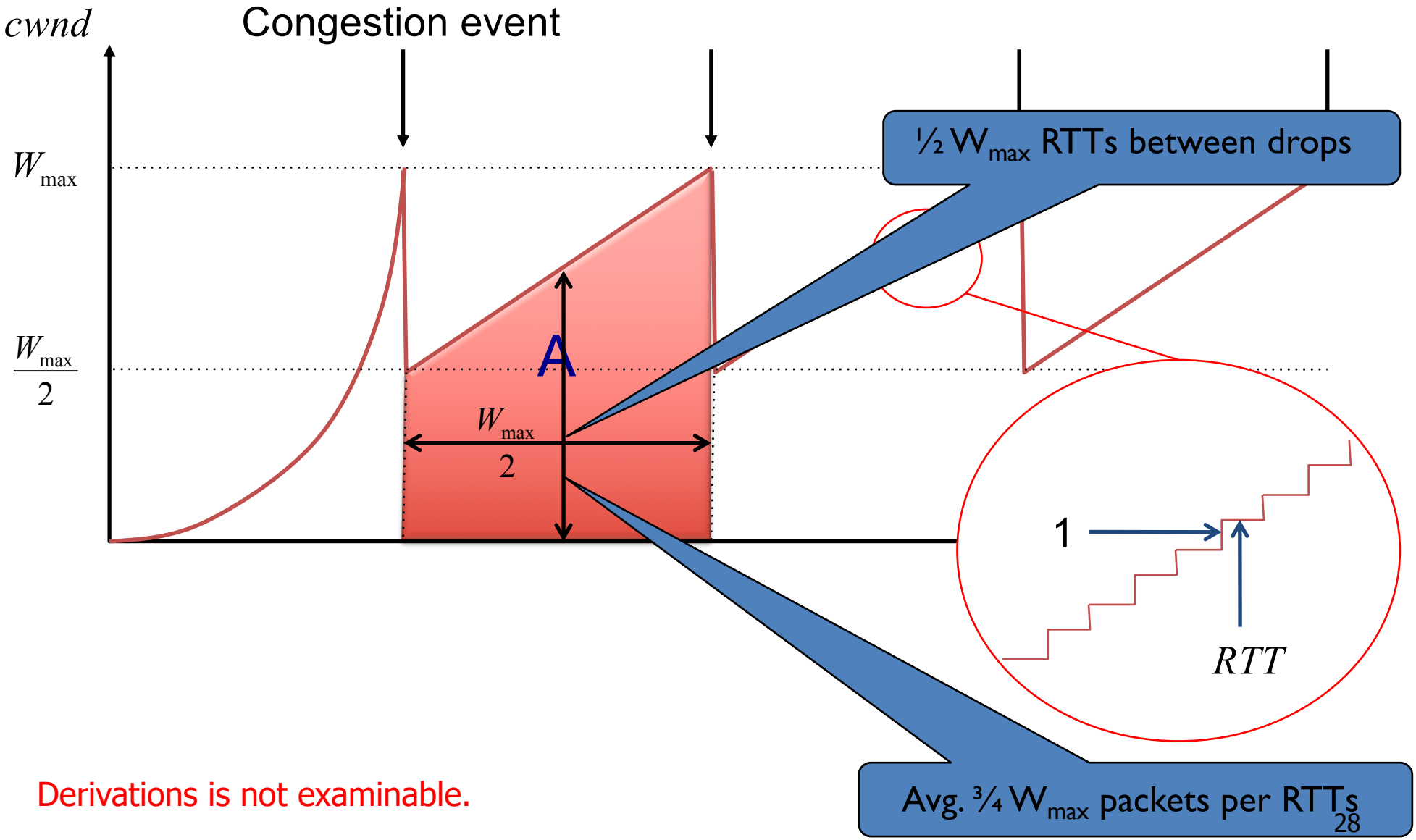


AIMD Sharing Dynamics



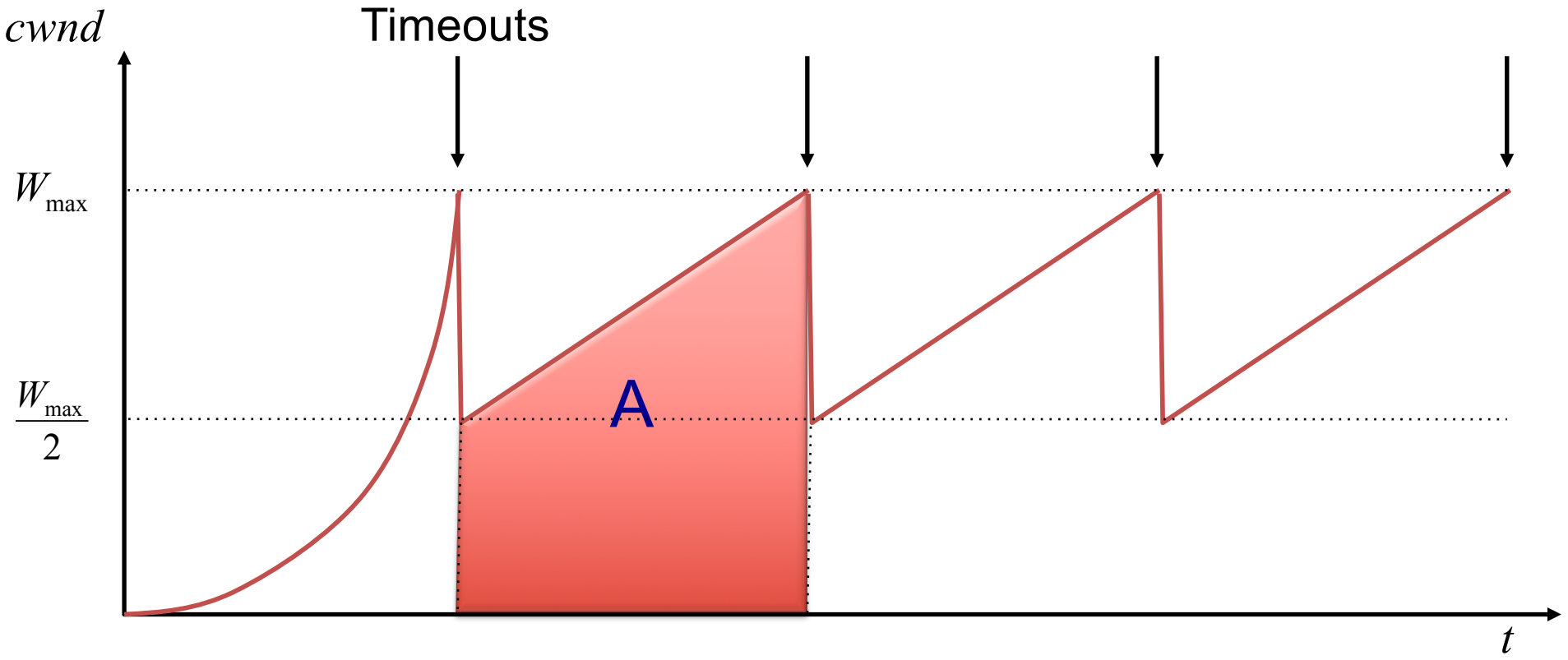
27

A Simple Model for TCP Throughput



Derivations is not examinable.

A Simple Model for TCP Throughput



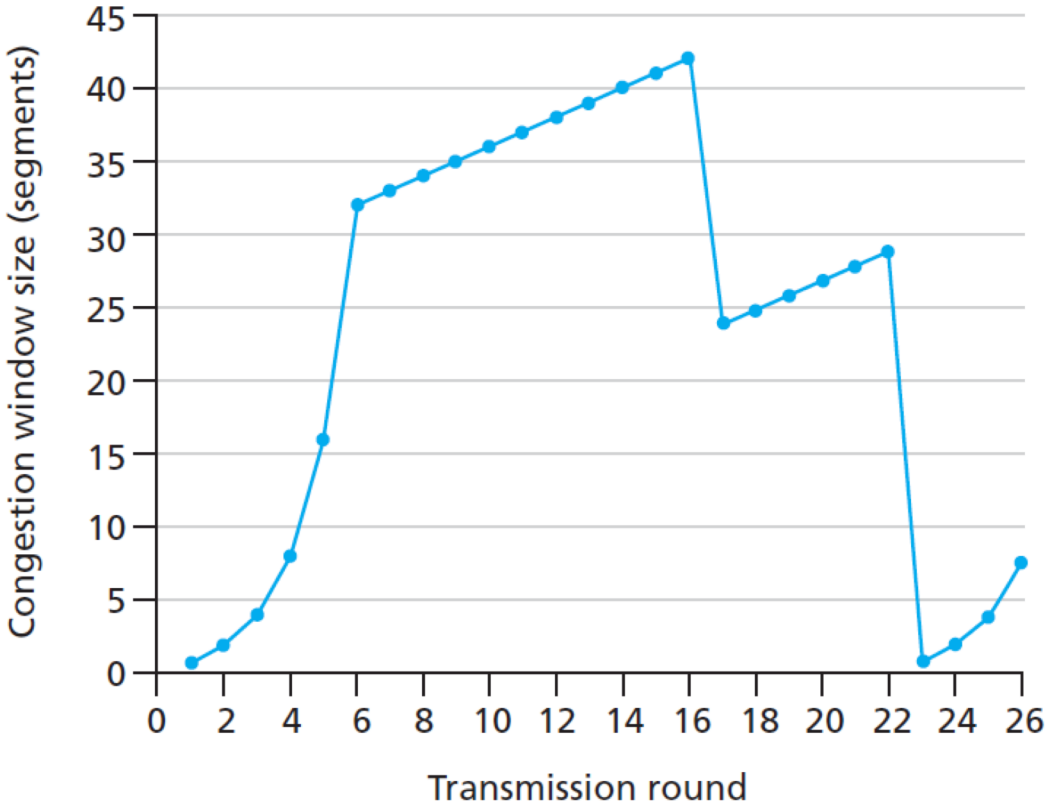
Packet drop rate, $p = 1/A$, where $A = \frac{3}{8} W_{max}^2$

Throughput, $B = \frac{A}{\left(\frac{W_{max}}{2}\right) RTT} = \sqrt{\frac{3}{2}} \frac{1}{RTT\sqrt{p}}$

Quiz: TCP Congestion Control?

In the figure how many congestion avoidance intervals can you identify?

A. 0
B. 1
C. 2
D. 3
E. 4

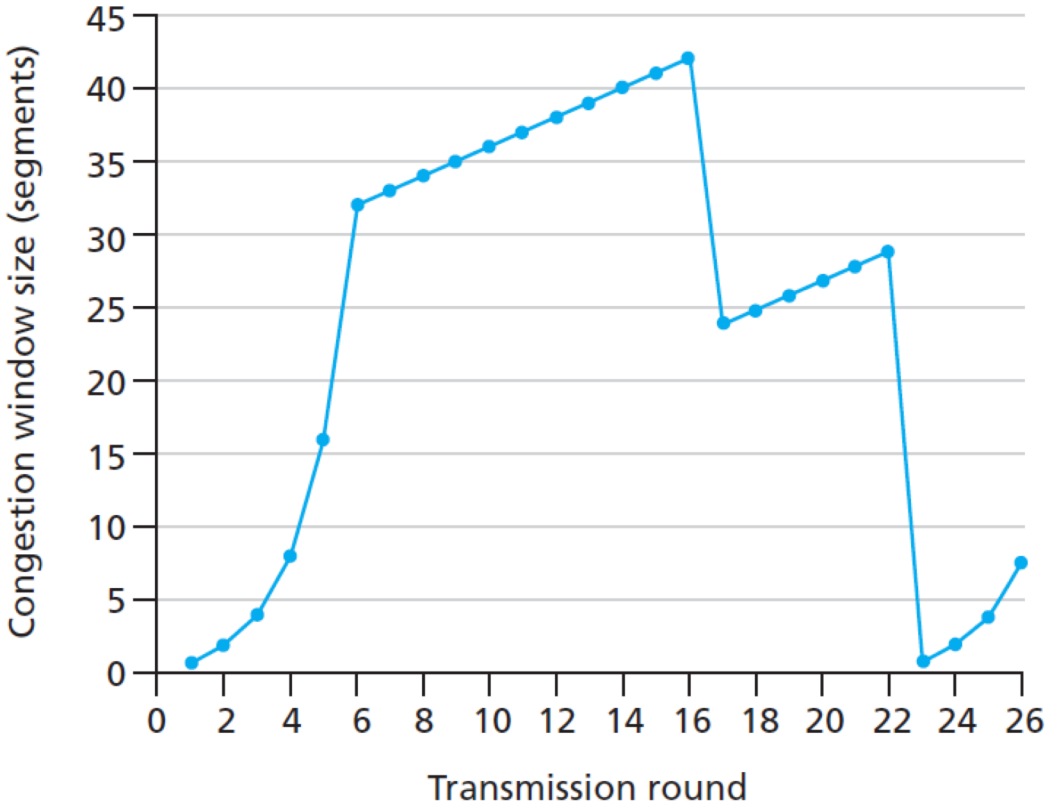


www.zeetings.com/salil

Quiz: TCP Congestion Control?

In the figure how many slow start intervals can you identify?

A. 0
B. 1
C. 2
D. 3
E. 4

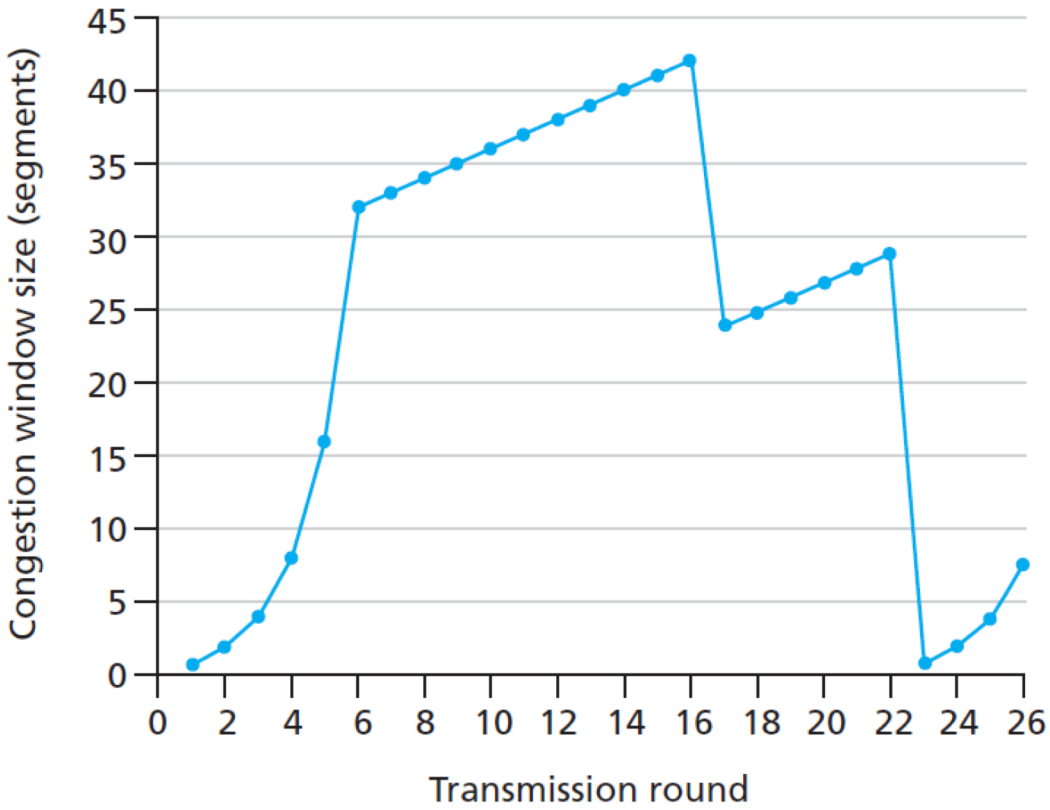


www.zeetings.com/salil

Quiz: TCP Congestion Control?

In the figure after the 16th transmission round, segment loss is detected by ?

A. Triple Dup Ack
B. Timeout

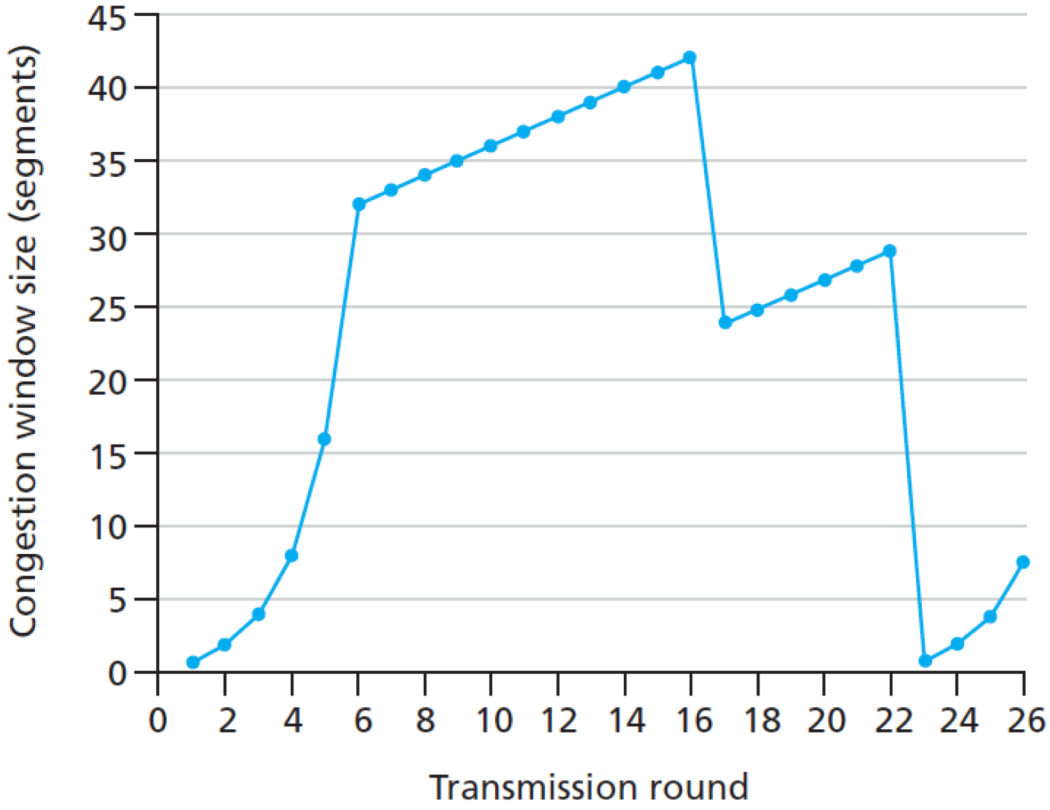


www.zeetings.com/salil

Quiz: TCP Congestion Control?

In the figure what is the initial value of ssthresh (slow start threshold)?

A. 0
B. 28
C. 32
D. 42
E. 64



www.zeetings.com/salil

Transport Layer: Summary

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

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"