# Advanced Networks

Transport layer: TCP



Dr. Wei Bao | Lecturer
School of Computer Science

THE UNIVERSITY OF SYDNEY

# Pipelined protocols: overview

## Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only s *cumulative ack*
  - does not ack packet if there is a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

## Selective Repeat:

- sender can have up to N unacked packets in pipeline
- receiver sends *individual ack* packet
- sender maintains timer for unacked packet
  - when timer expires, retransmit only that unacked packet

# Go-Back-N: sender

› "window" of up to N, consecutive unacked pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

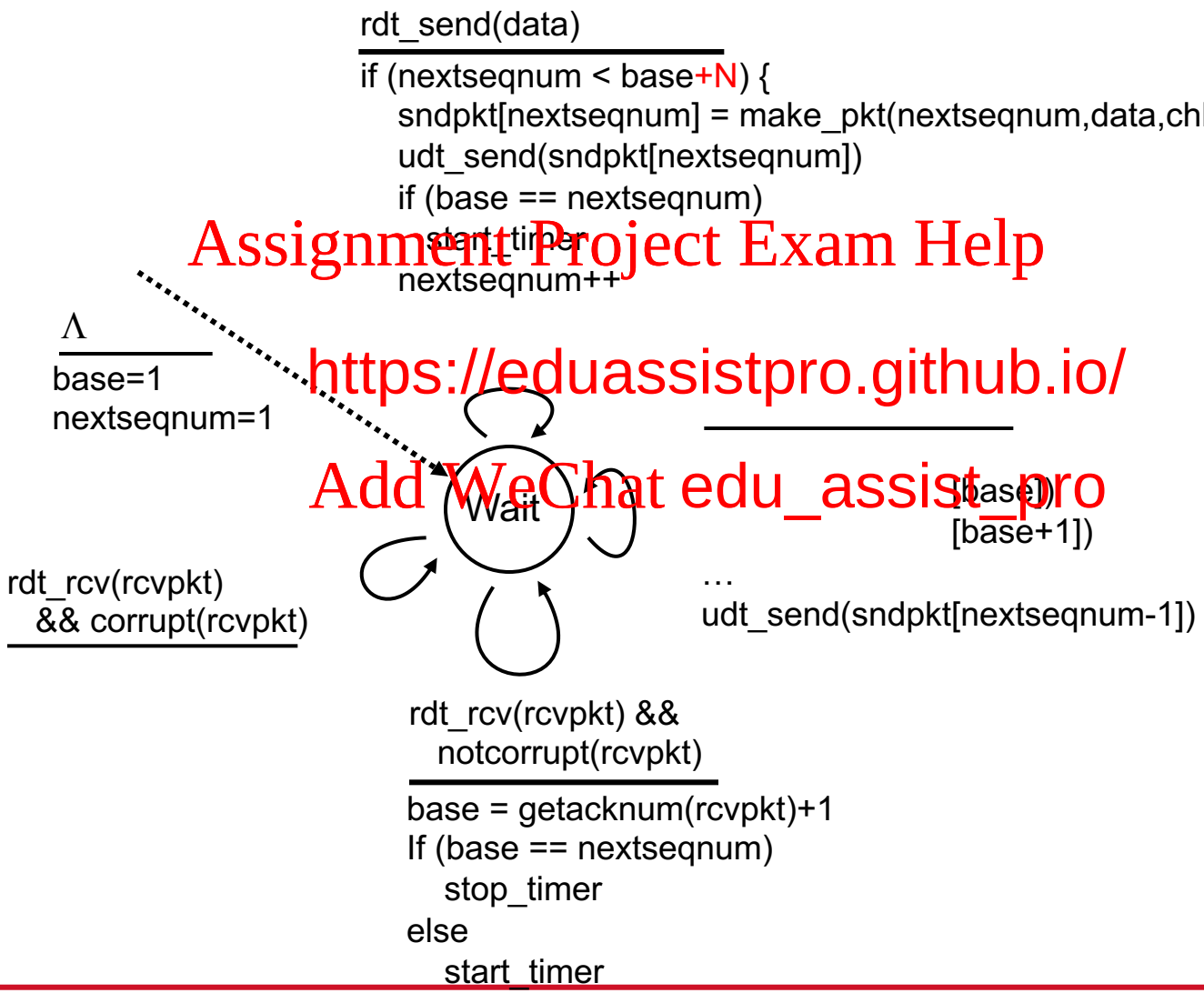
# Go-Back-N: sender

› "window" of up to N, consecutive unacked pkts allowed

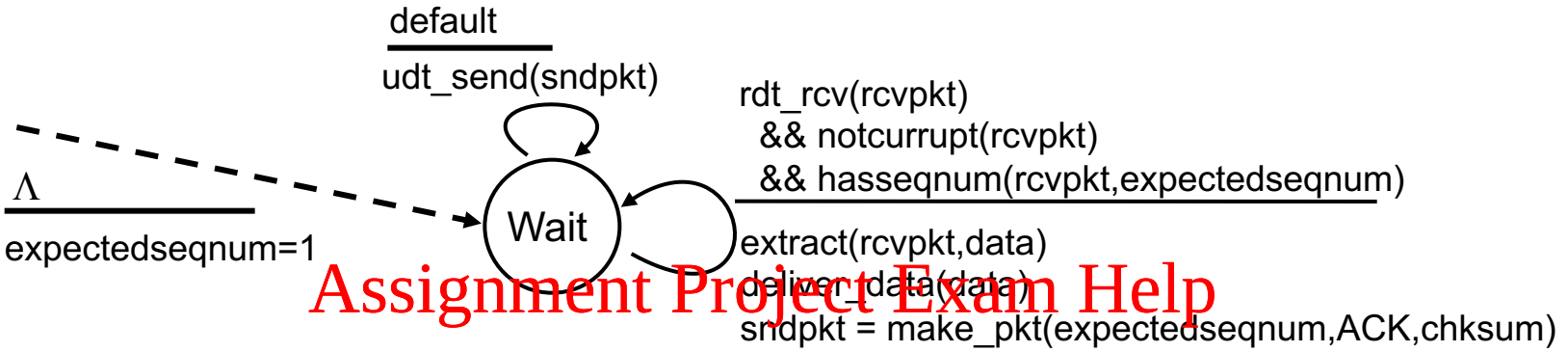


- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

# GBN: sender extended FSM



# GBN: receiver extended FSM





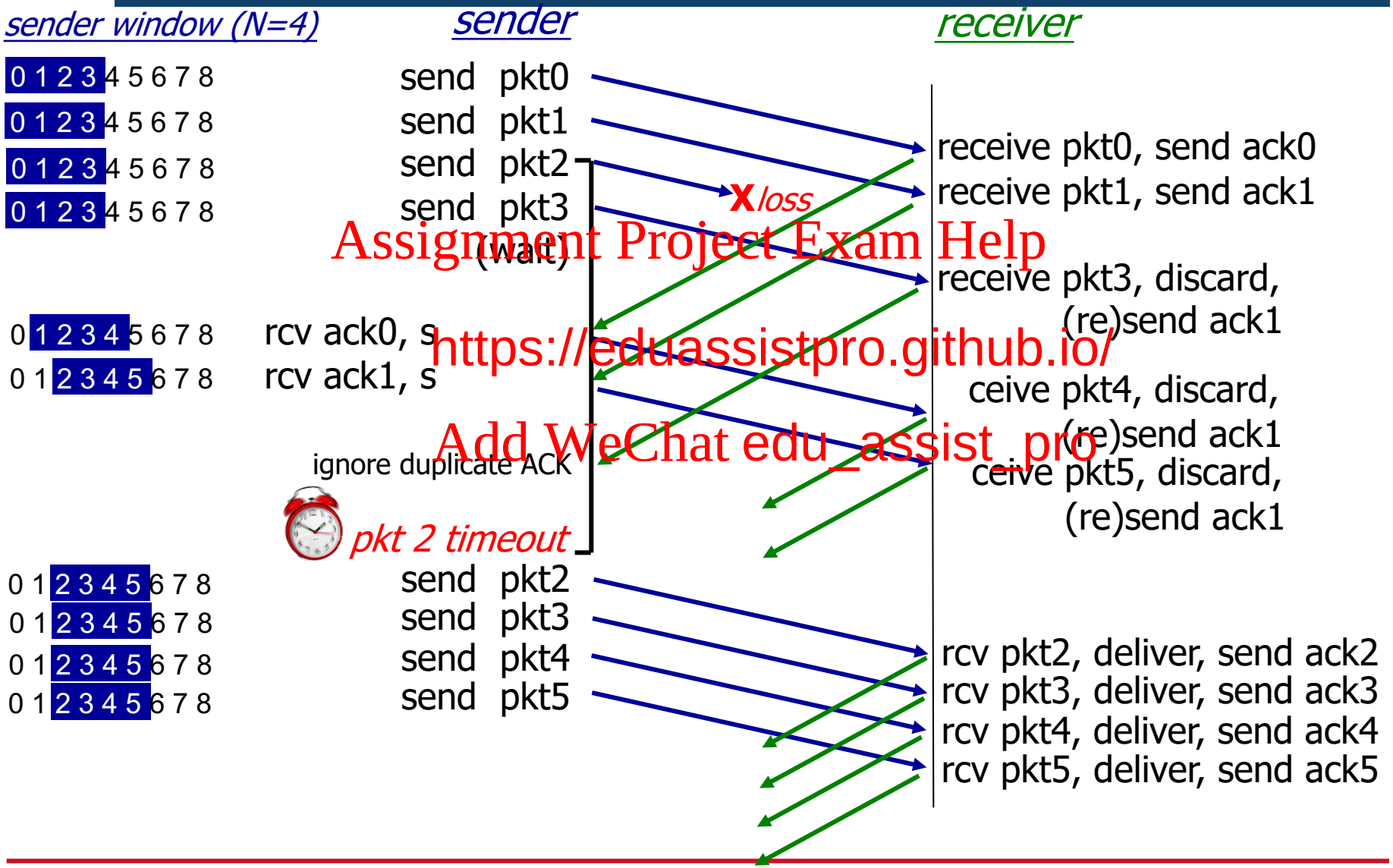

ACK-only: always send ACK ly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

› out-of-order pkt:

- discard (don't buffer): *no receiver buffering!*
- re-ACK pkt with highest in-order seq #

# GBN in action

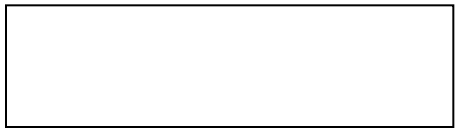


# Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts as needed for eventual in-order delivery to upper layer
- sender only re ACK not received
  - sender timer for each unACKe
- sender window
- receiver window

# Selective repeat: sender, receiver windows





# Selective repeat

## sender

data from above:

› if next available seq # in window, send pkt

timeout(n):

› resend pkt n, res

ACK(n) in [sendbase,sendbase+N-1]:

› mark pkt n as received

› if n is smallest unACKed pkt, advance window base to next unACKed seq #

## receiver

pkt n in [rcvbase, rcvbase+N-1]

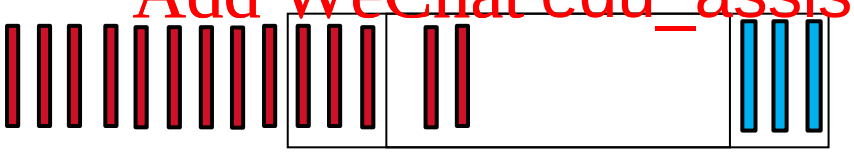
- send ACK(n)
- out-of-order: buffer
- der: deliver (also
- er buffered, in-order
- indow to
- t-yet-received pkt

[rcvbase-N,rcvbase-1]

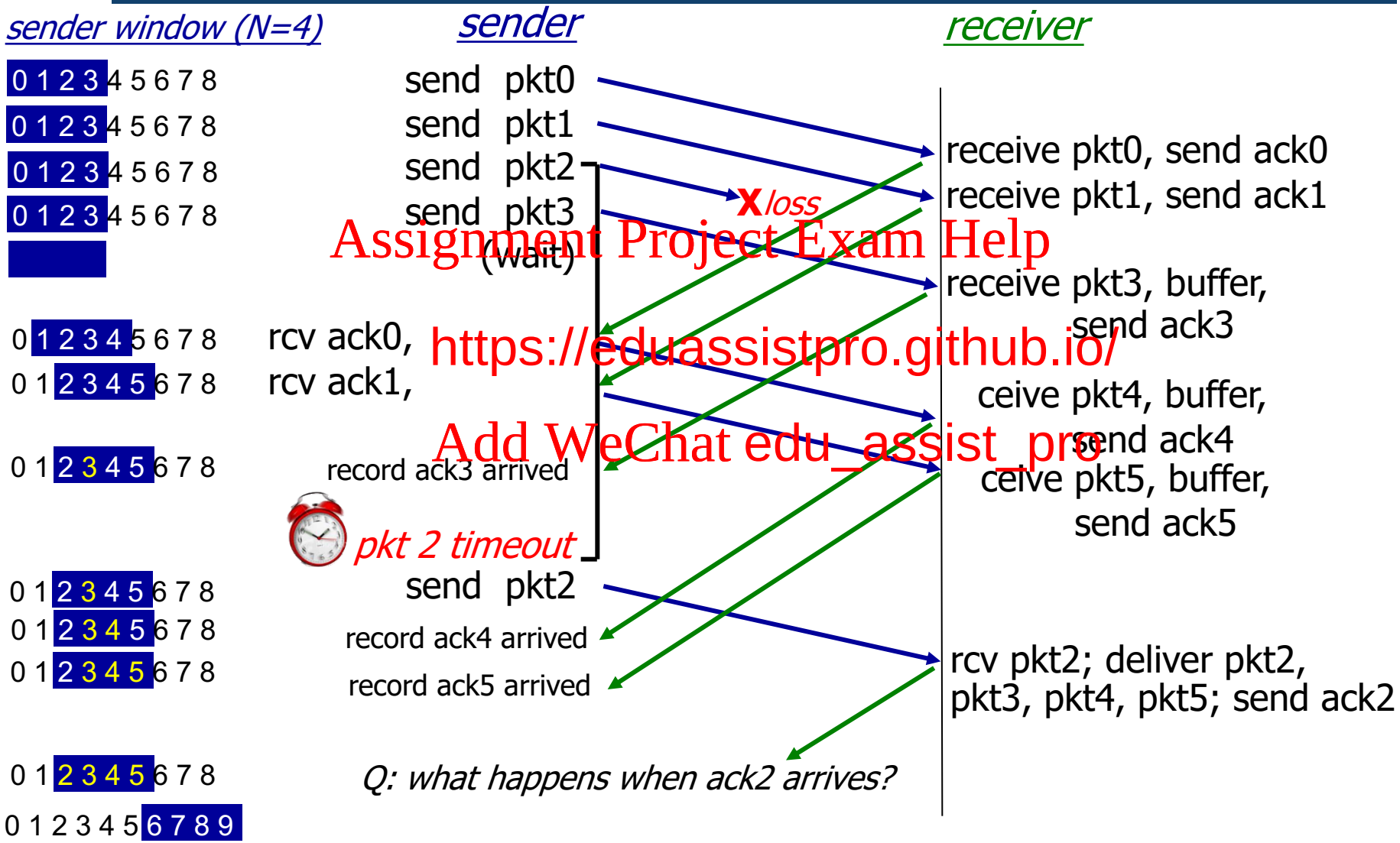
- ACK(n)

otherwise:

- ignore



# Selective repeat in action



# Connection-oriented Transport TCP

# TCP: Overview RFCs: 793,1122,1323, 2018, 2581

› point-to-point:
  - one sender, one receiver

› reliable, in-order *byte stream*

› pipelined:
  - TCP congestion set window size

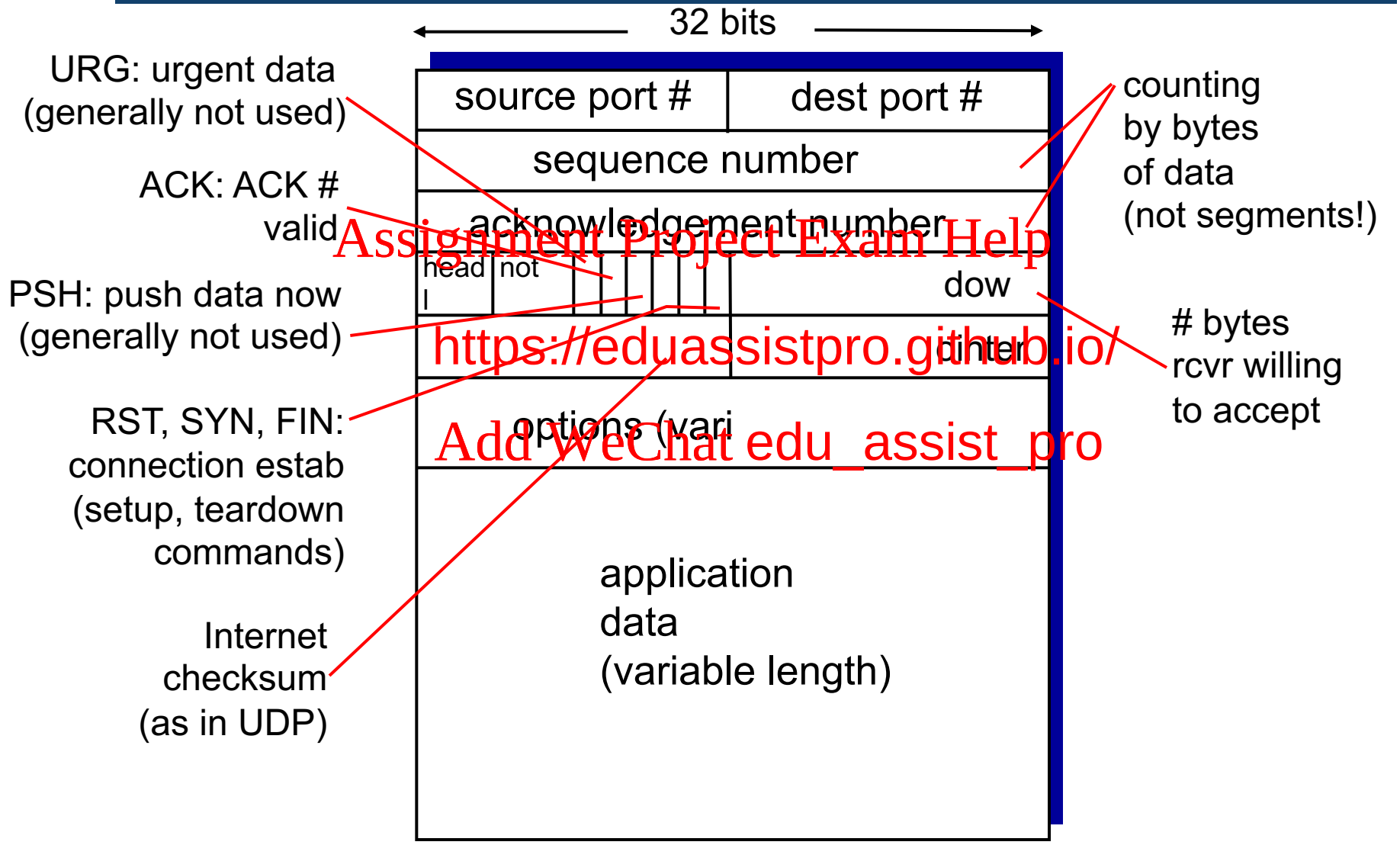
› full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment



ion-oriented:
ng (exchange of sgs) inits sender, receiver state before data exchange

› flow controlled:
  - sender will not overwhelm receiver

# TCP segment structure



# TCP seq. numbers, ACKs

**sequence numbers:**

- "number" of first byte in segment's data

**acknowledgements:**

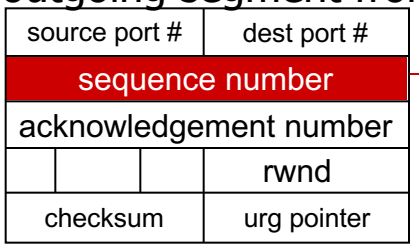
- seq # of **next by** expected from ot
- cumulative ACK

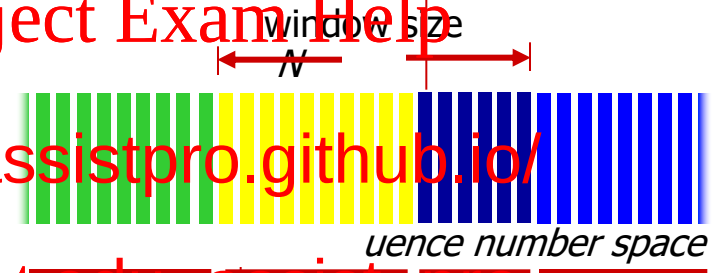
Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say,
- up to implementor
- Most will store, but still use cumulative ACK



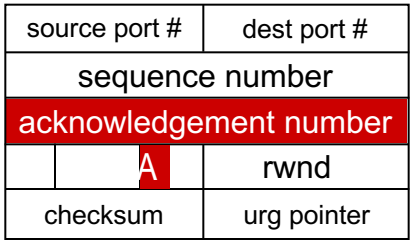
outgoing segment from sender



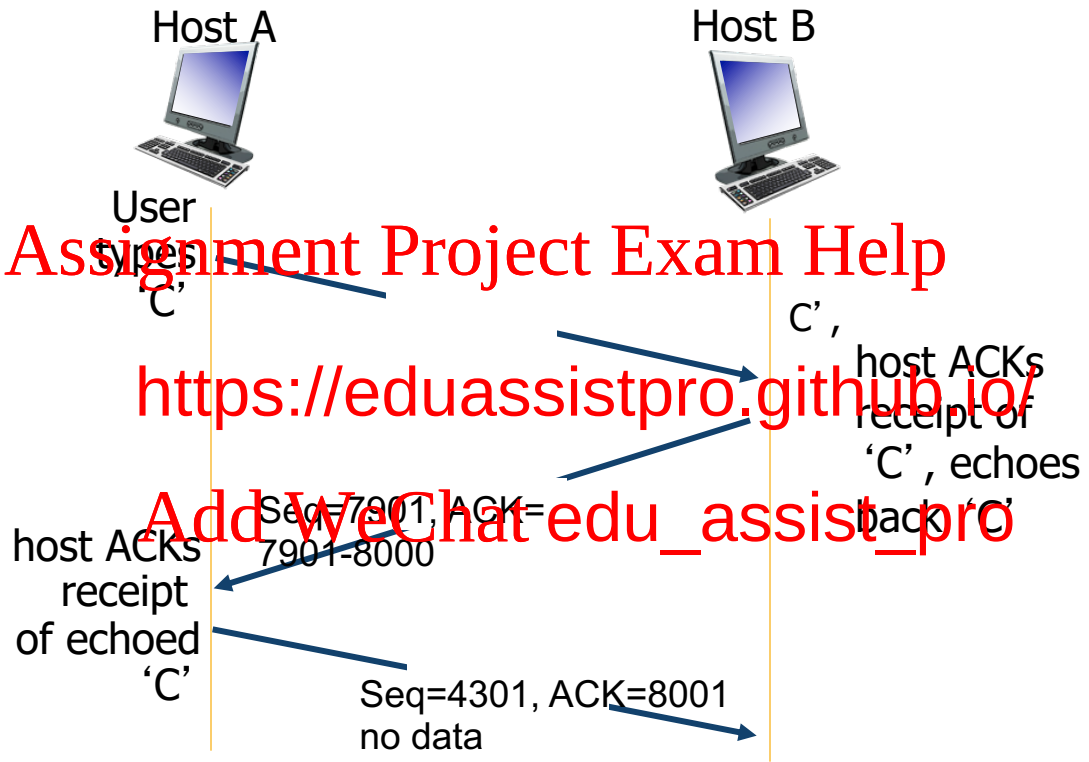




sent ACKed | sent, not-yet ACKed ("in-flight") | usable but not yet sent | not usable

incoming segment to sender



# TCP seq. numbers, ACKs




simple telnet scenario

# TCP round trip time, timeout

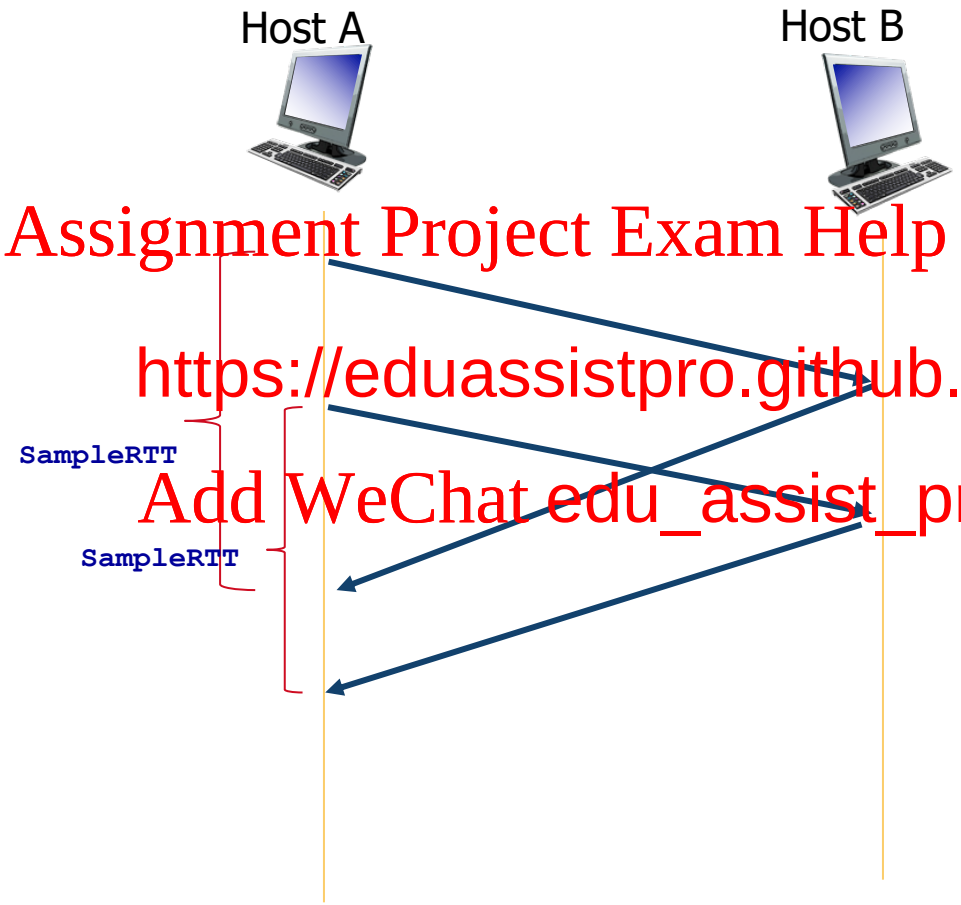
Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short*: premature timeout, unnecessary retransmissions
- *too long*: slow reaction to segment loss

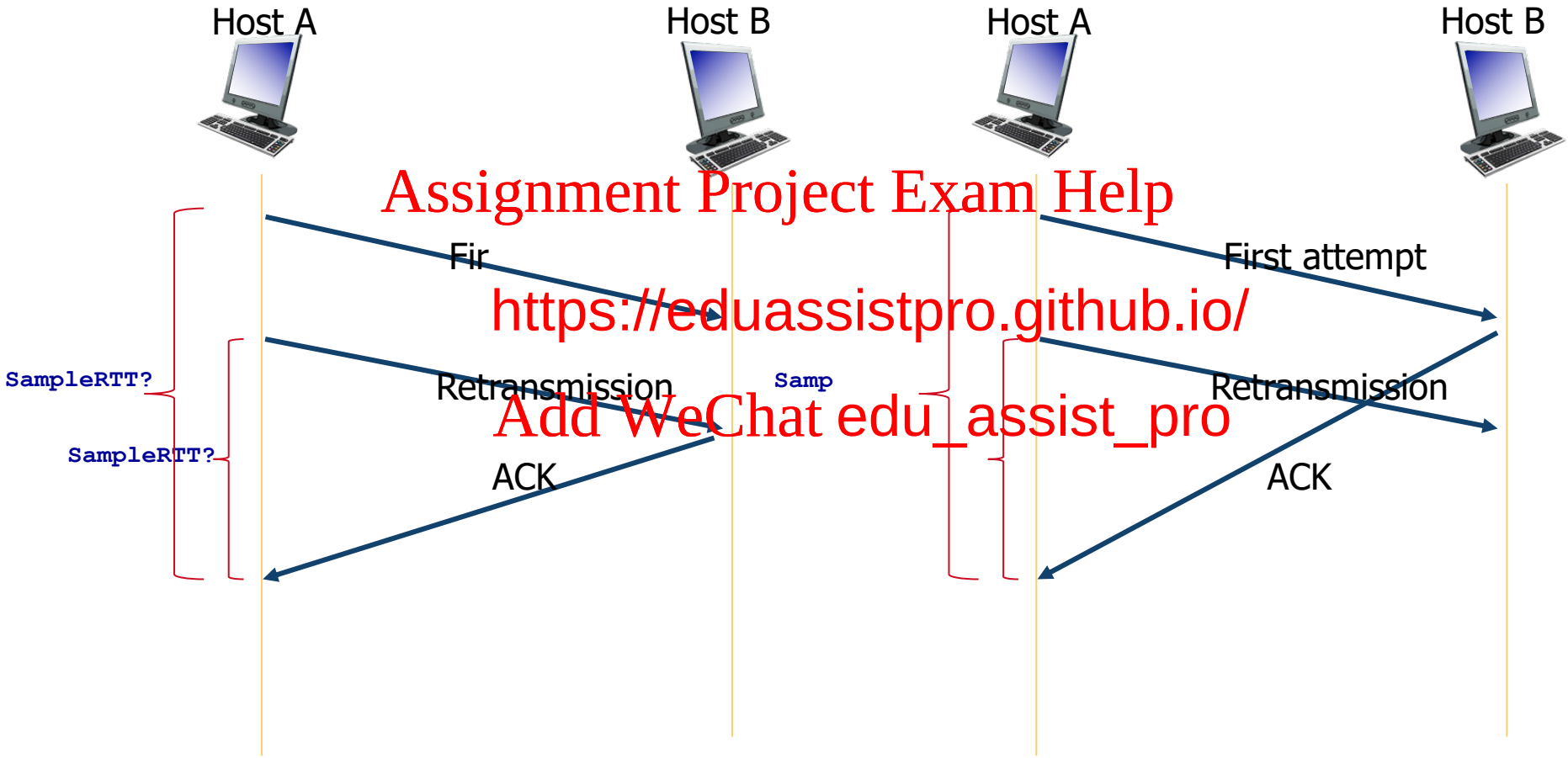
Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission receipt
  - retransmissions
- SampleRTT will vary, want estimated RTT "smoother"
  - weighted average of several *recent* measurements, not just current **SampleRTT**

# SampleRT



# Ignore retransmissions

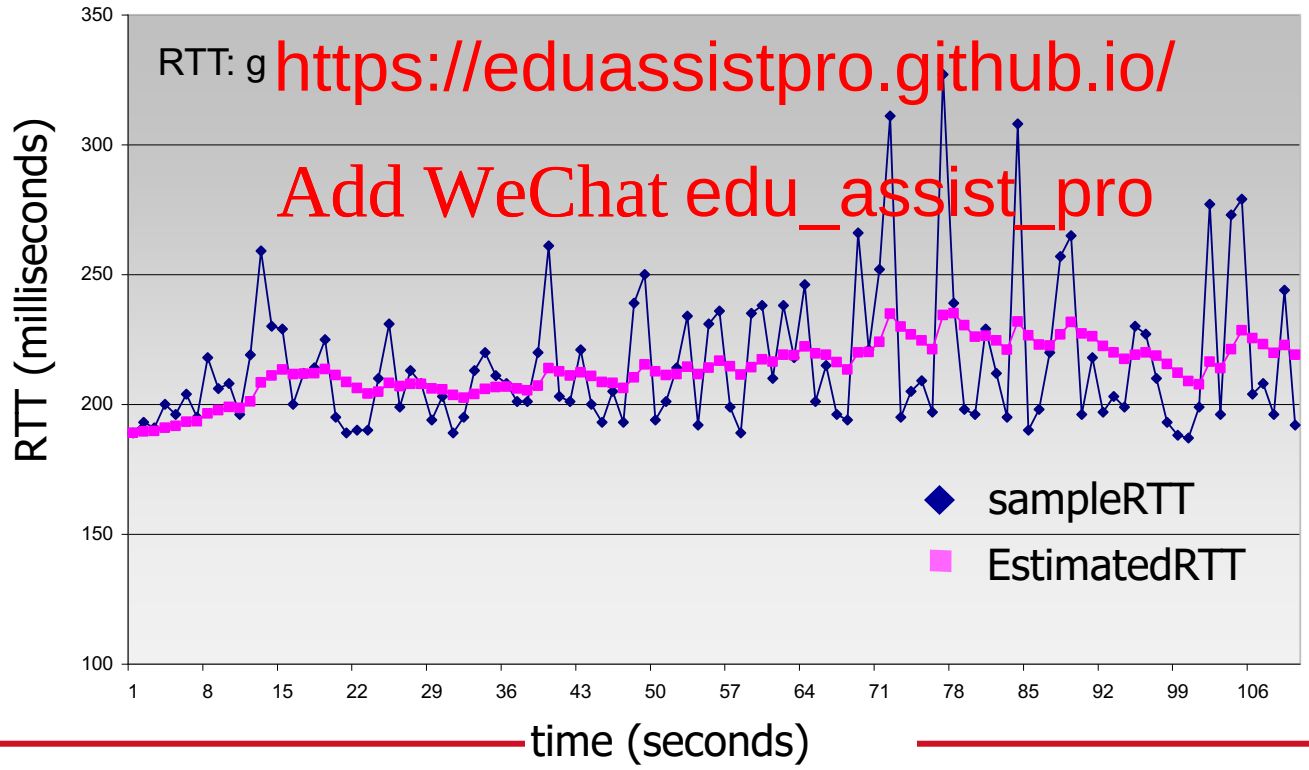


# TCP round trip time, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125





# TCP round trip time, timeout

› timeout interval: **`EstimatedRTT`** plus "safety margin"

- large variation in **`EstimatedRTT`** -> larger safety margin



› estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (
```



```
    |SampleRTT -
```

(typically,



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

estimated RTT

"safety margin"

# Reliable Data Transfer in TCP

# TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segme
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

ly consider
TCP sender:
licate acks

- ignore flow control, congestion control

# TCP sender events

*data rcvd from app:*

› create segment with seq #

› seq # is byte-stream number of first data byte in segment

› start timer if no running

- think of timer as for oldest unacked segment
- expiration interval: `TimeOutInterval`

*timeout:*

› retransmit segment that caused timeout

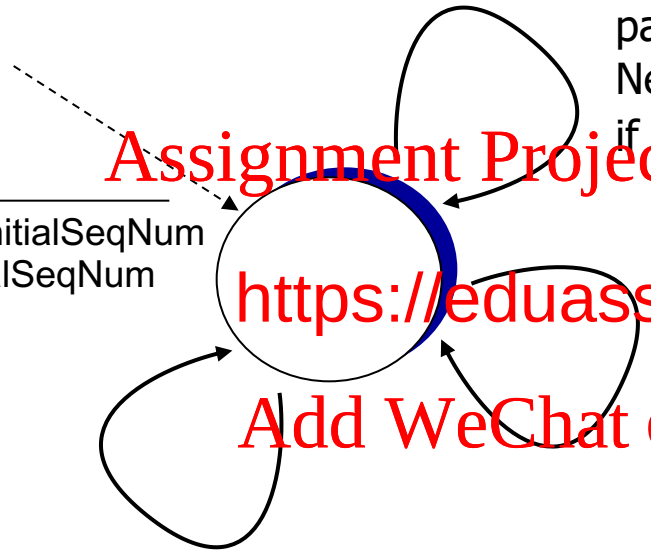
› restart timer



wledges
acked
P
se

- update what is known to be ACKed
- start timer if there are still unacked segments

# TCP sender (simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer



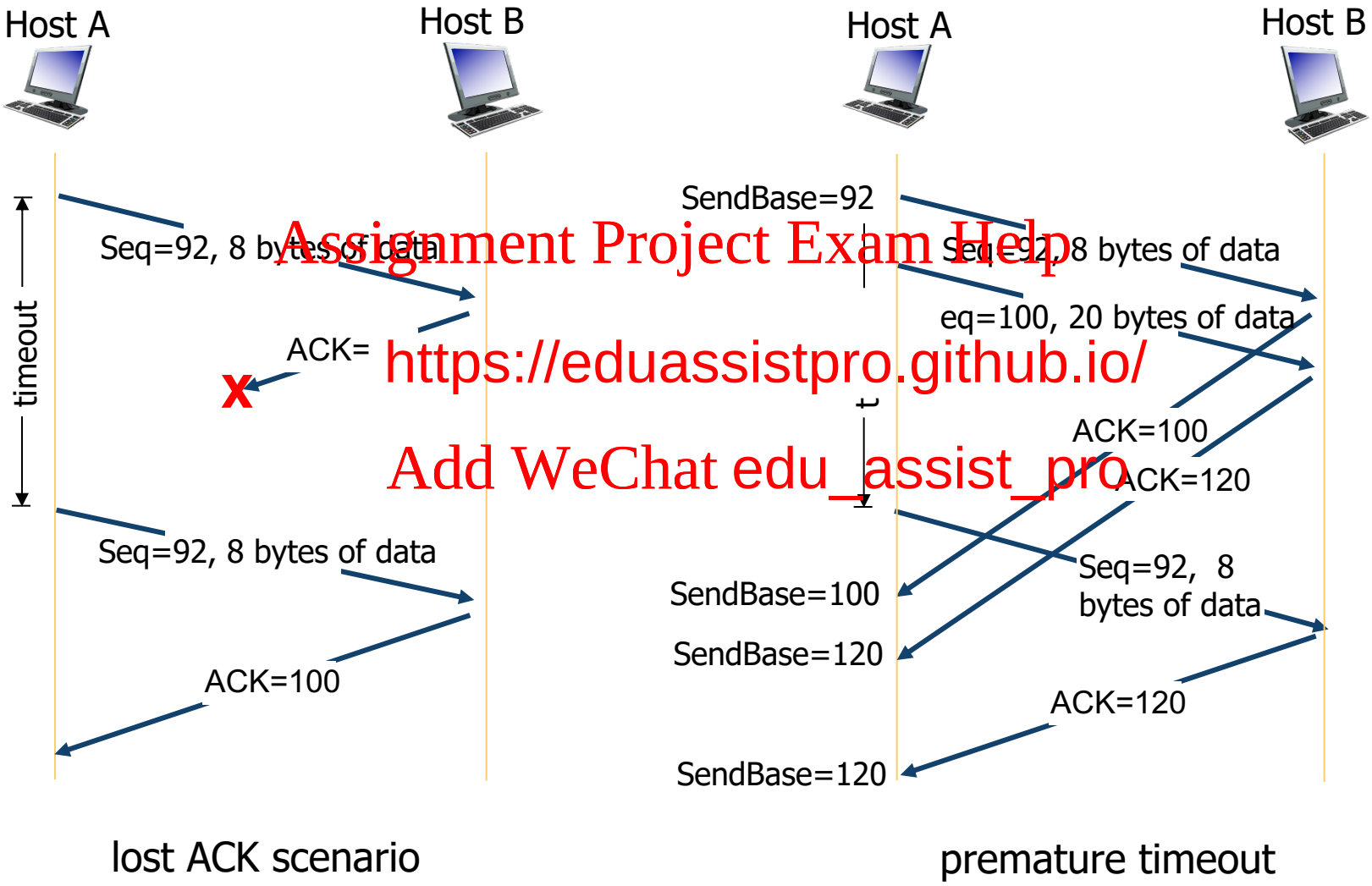
et-acked segment
st seq. #

start timer

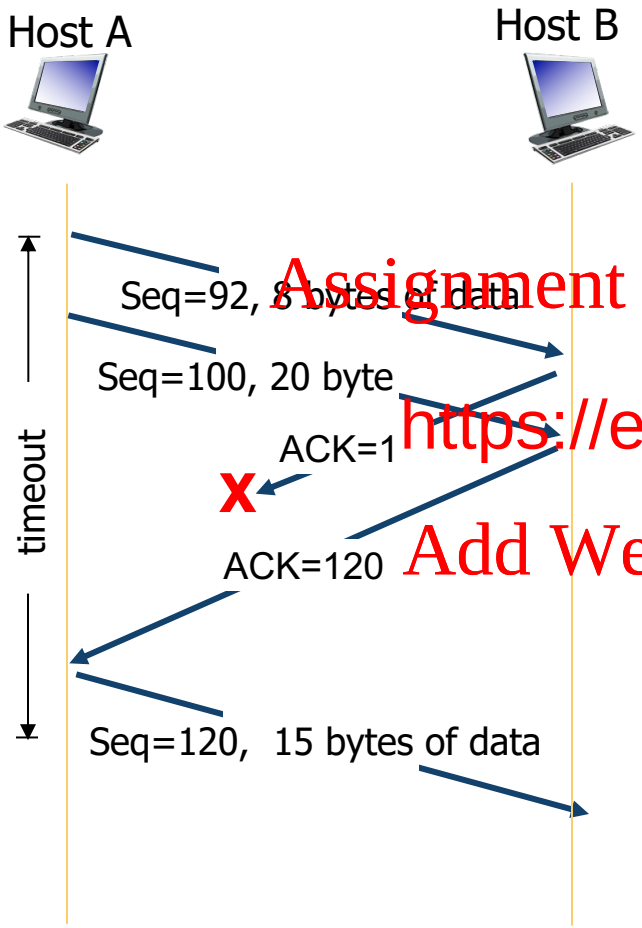
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
    }
```

# TCP: retransmission scenarios



# TCP: retransmission scenarios

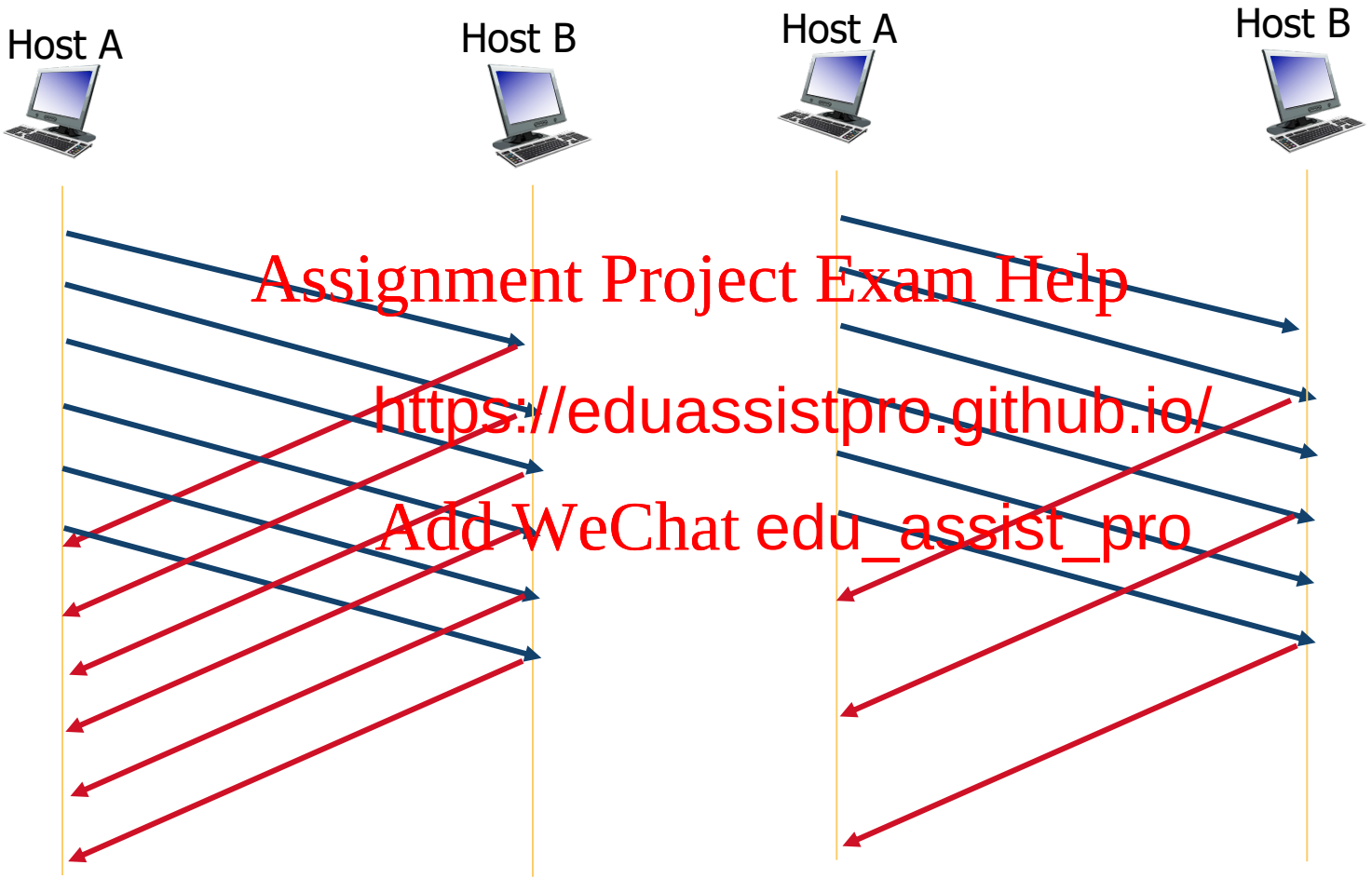



cumulative ACK

# TCP ACK generation [RFC 1122, RFC 2581]

| *event at receiver* | *TCP receiver action* |
|---|---|
| arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed | delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK |
| arrival of in-order se expected seq #. One other segment has ACK pending | end single cumulative A th in-order segments |
| arrival of out-of-order segment higher-than-expect seq. # . Gap detected | immediately send *duplicate ACK*, indicating seq. # of next expected byte |
| arrival of segment that partially or completely fills gap | immediate send ACK, provided that segment starts at lower end of gap |

# TCP ACK

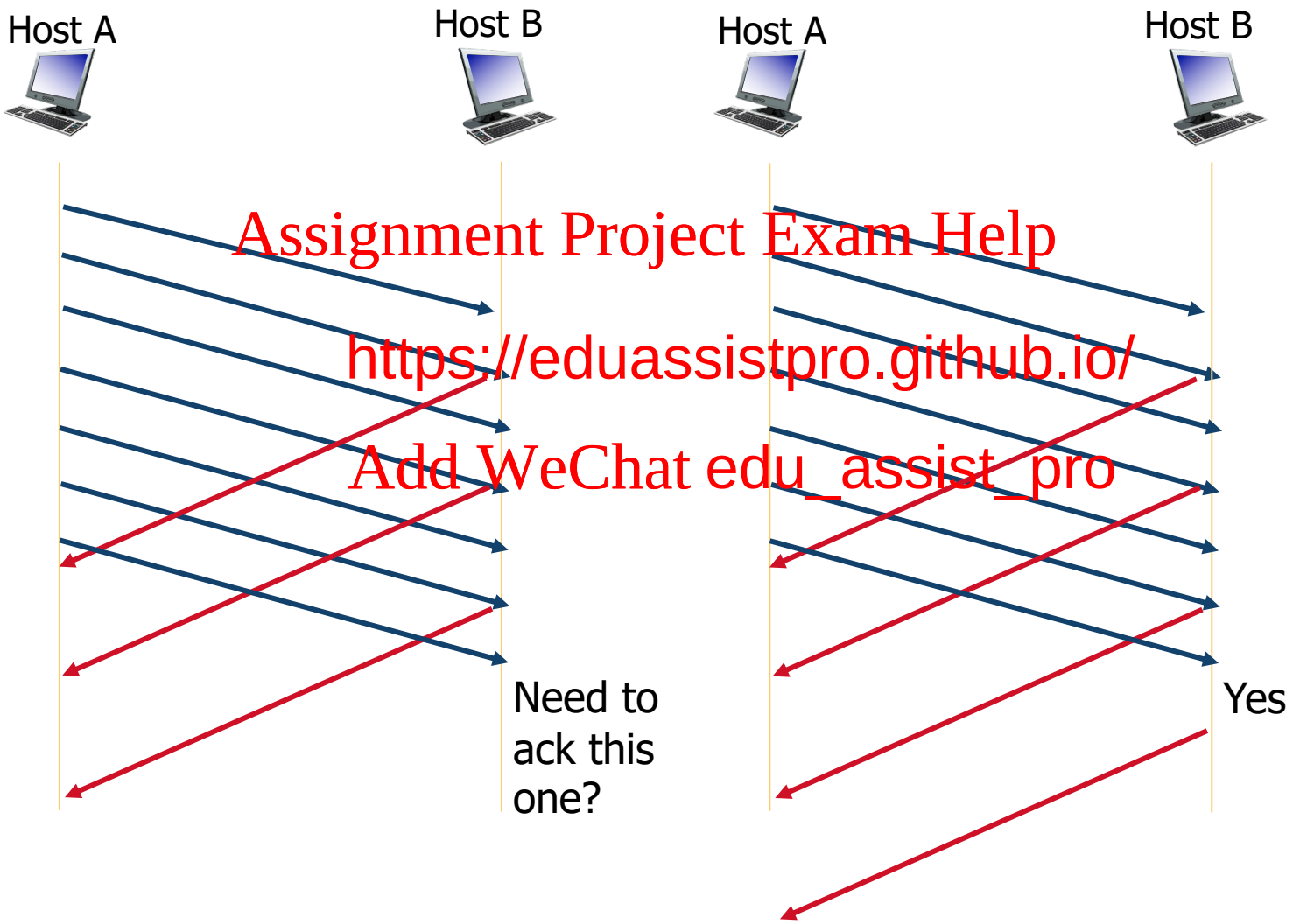





Too many ACKs

cumulative ACK

# TCP ACK



# TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost seg duplicate ACK
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

*TCP fast retransmit*

if sender receives 3 duplicate ACKs for same

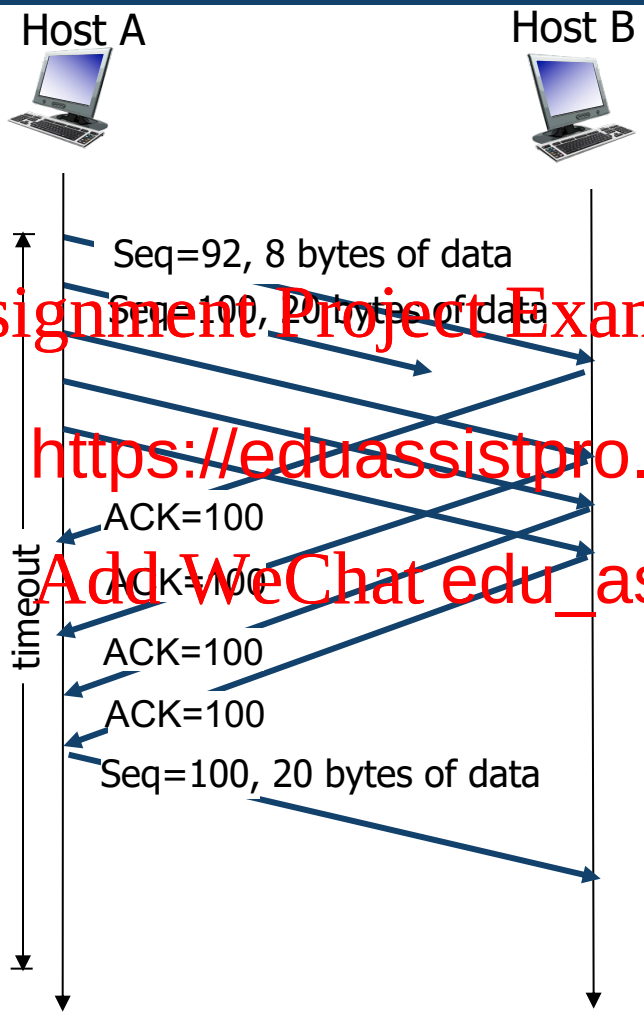
idate ACKs"),

segment

t seq #

- likely that unacked segment lost, so don't wait for timeout

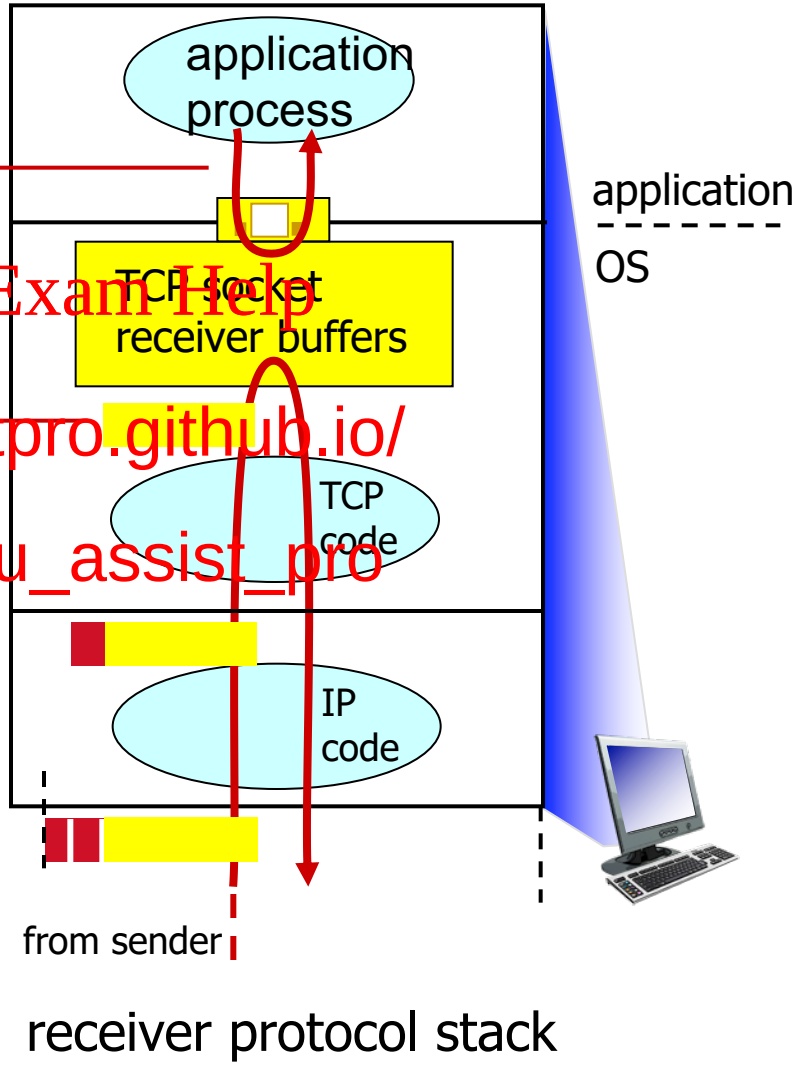
# TCP fast retransmit




fast retransmit after sender receipt of triple duplicate ACK

# Flow Control in TCP

# TCP flow control



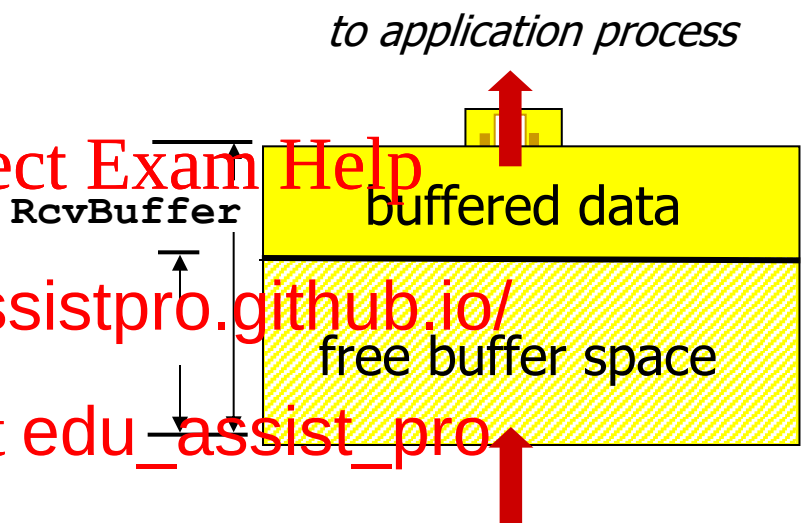



*flow control*

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 se
    options (typical defa
    bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow




*receiver-side buffering*

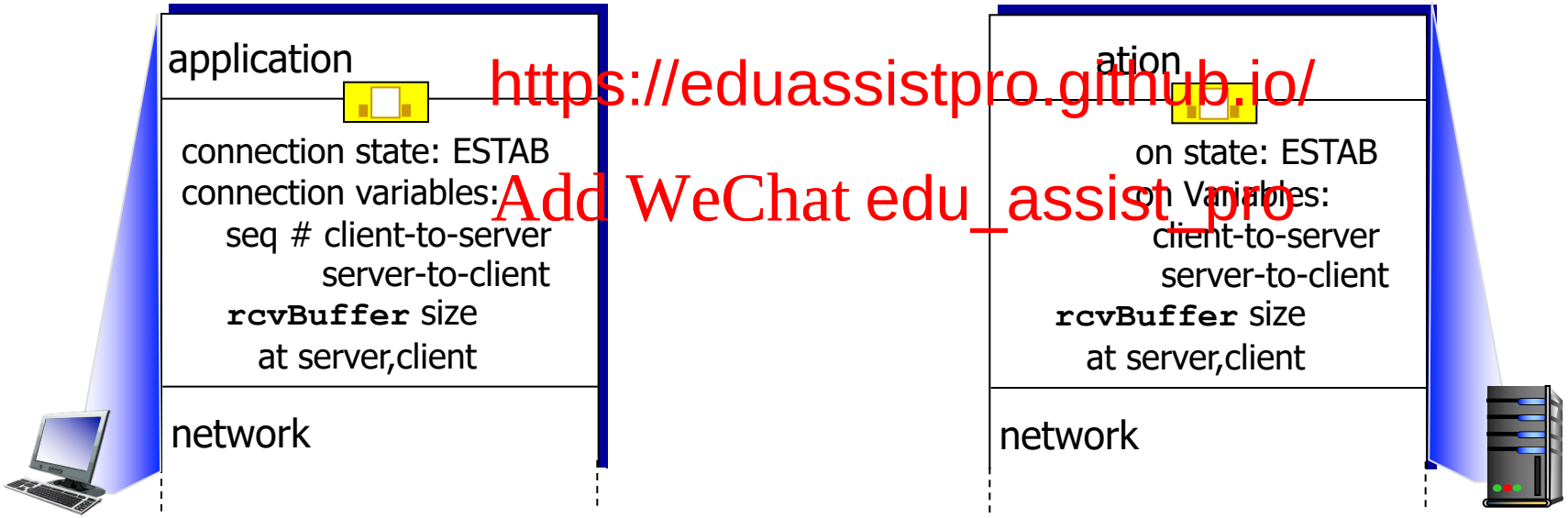
# Connection Management in TCP

# Connection Management

before exchanging data, sender/receiver "handshake":

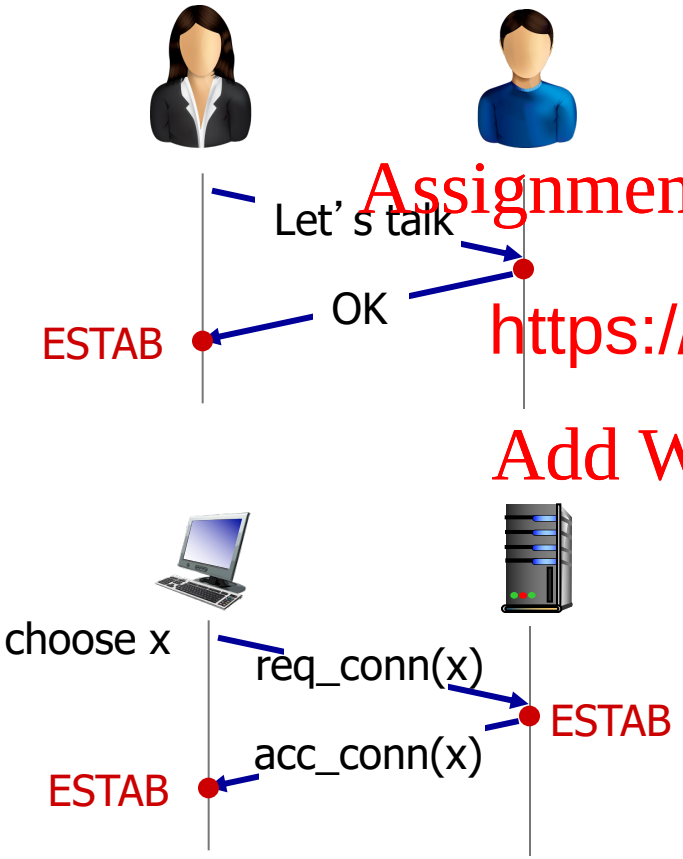
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters





# Agreeing to establish a connection

2-way handshake:

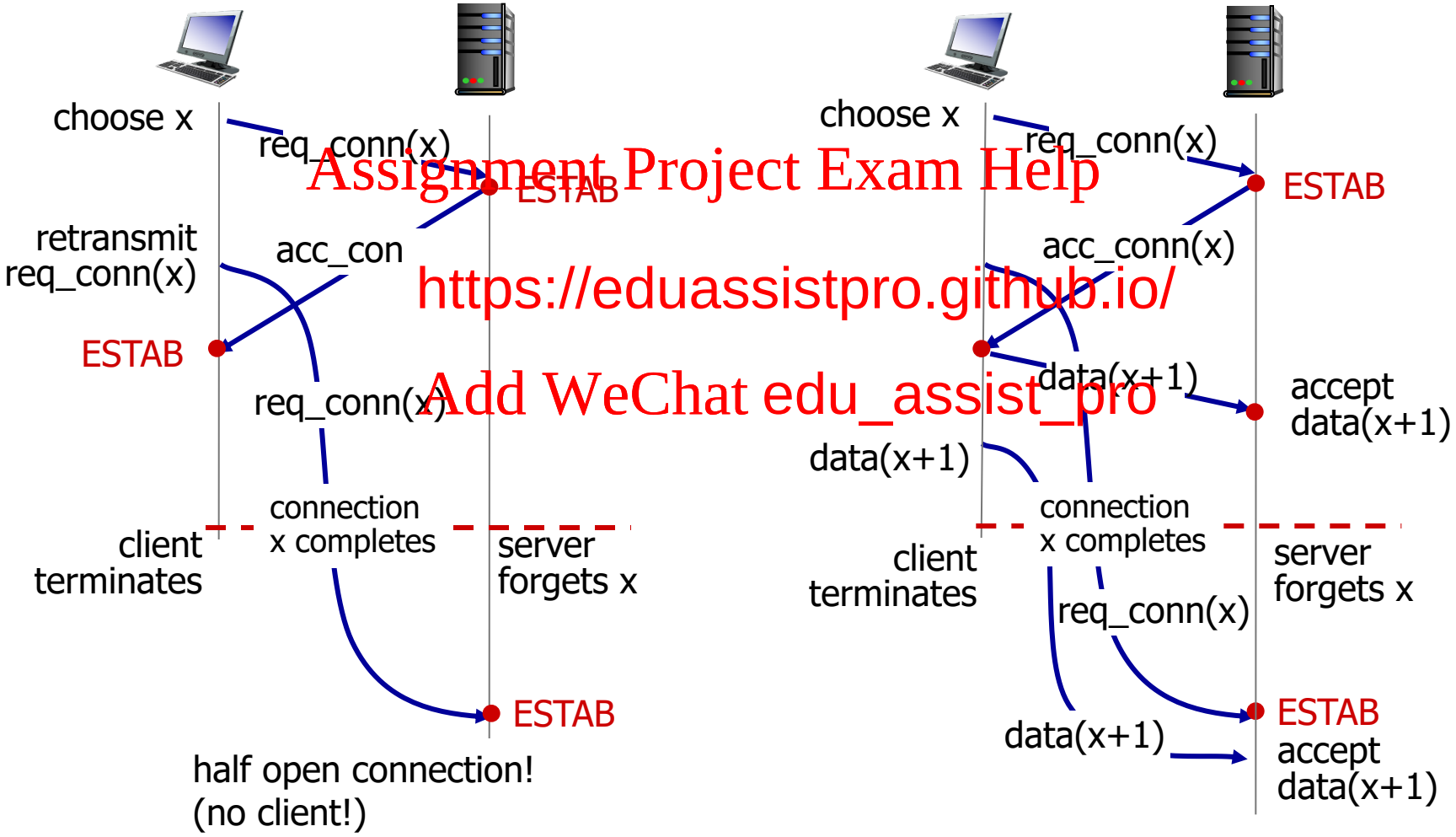



*Q:* will 2-way handshake always work in network?

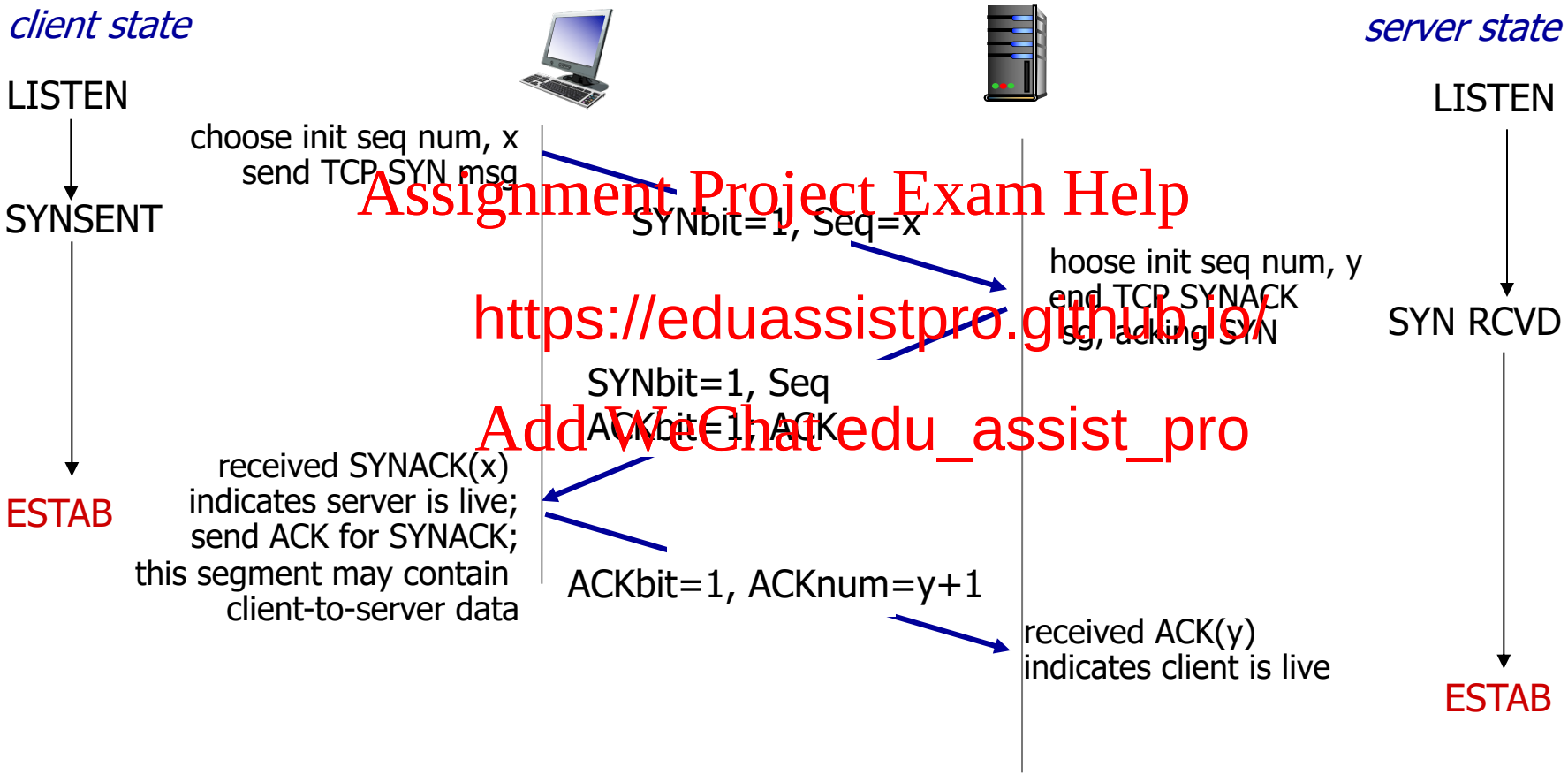
- variable delays
- retransmitted messages (e.g. due to message loss
- dering
- r side

# Agreeing to establish a connection

## 2-way handshake failure scenarios:



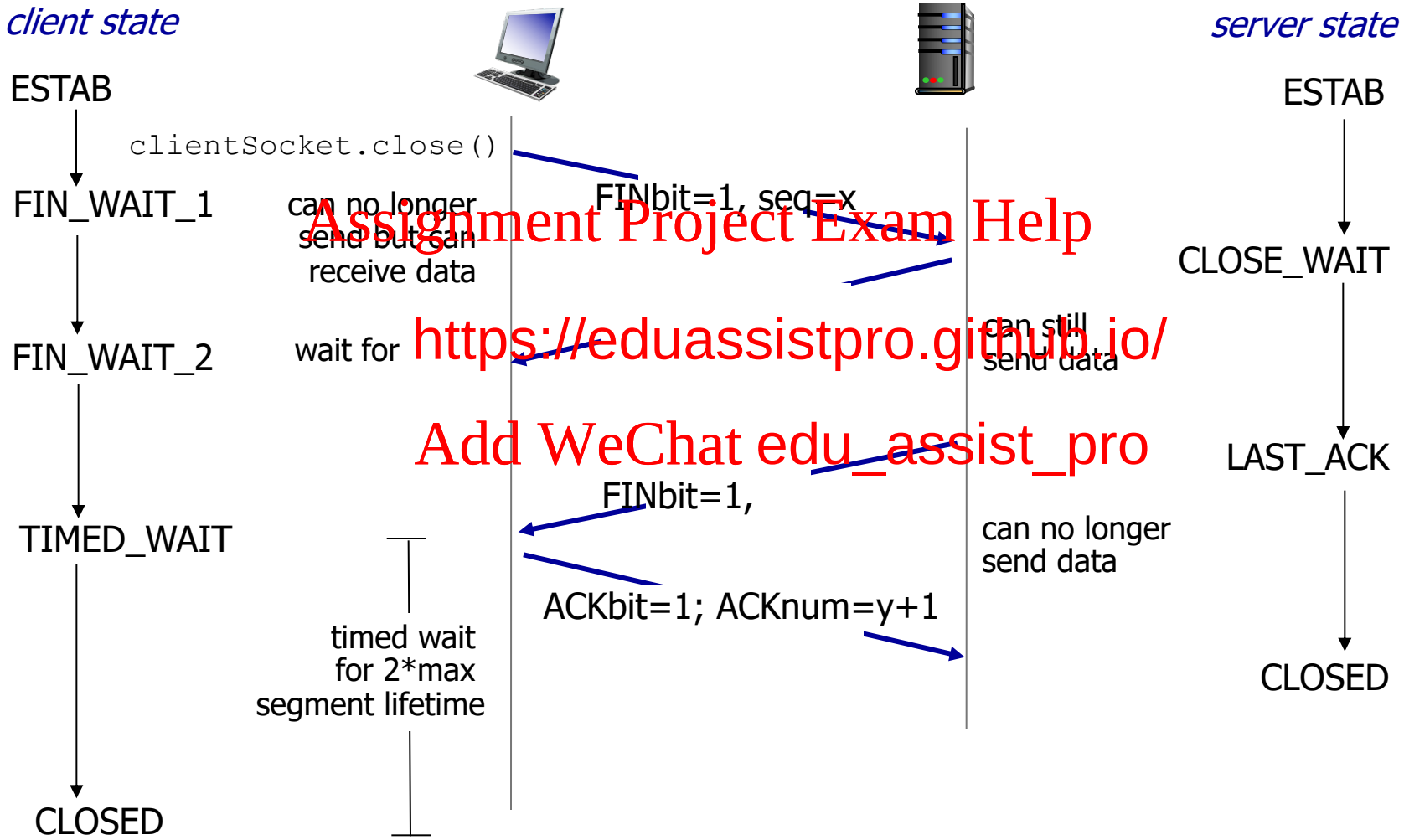
# TCP 3-way handshake



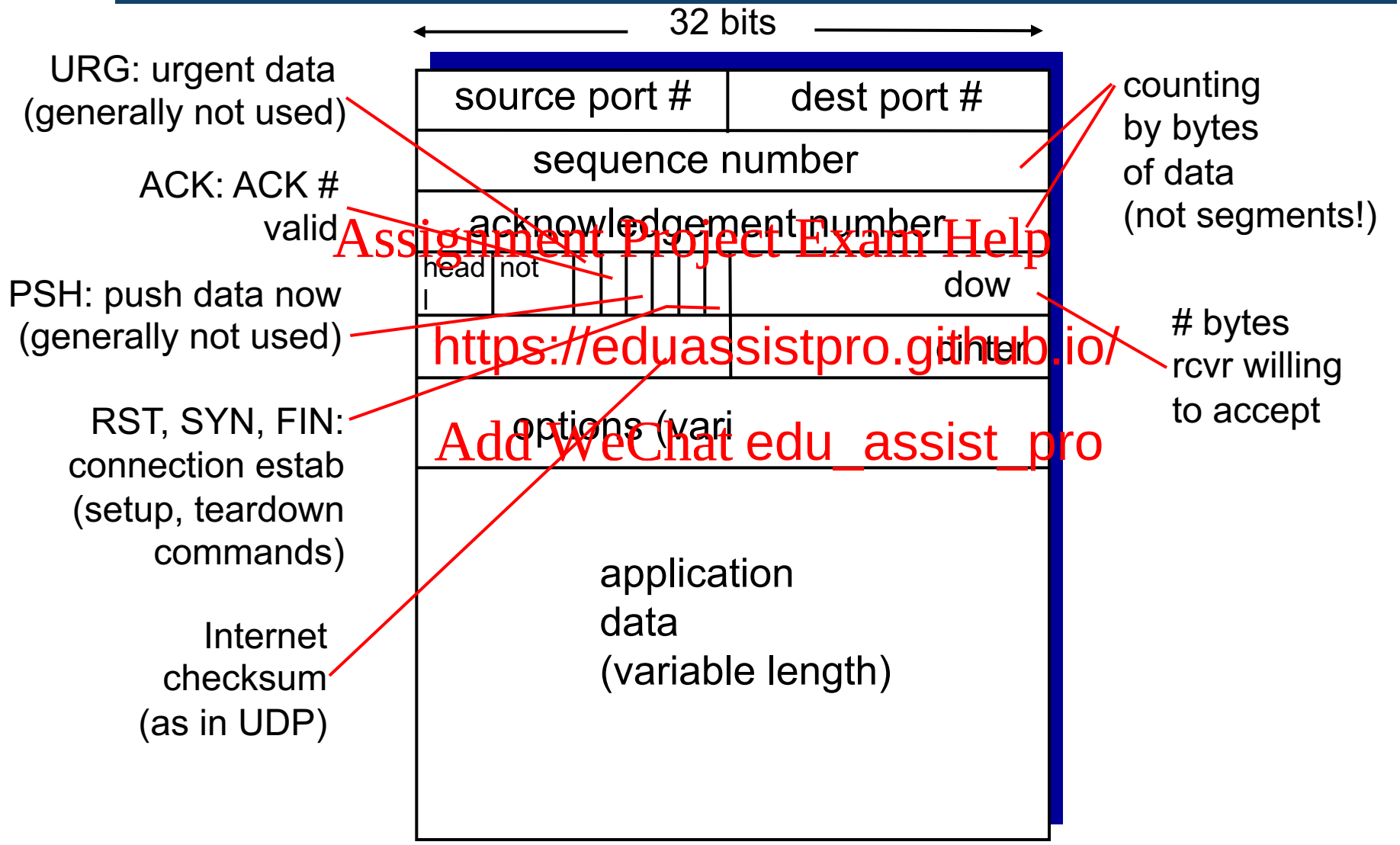
# TCP: closing a connection

- client, server each closes their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK

# TCP: closing a connection



# TCP segment structure



# Principles of Congestion Control

# Principles of congestion control

*congestion:*

› informally: "too many sources sending too much data too fast fo



› different from f



› manifestations:

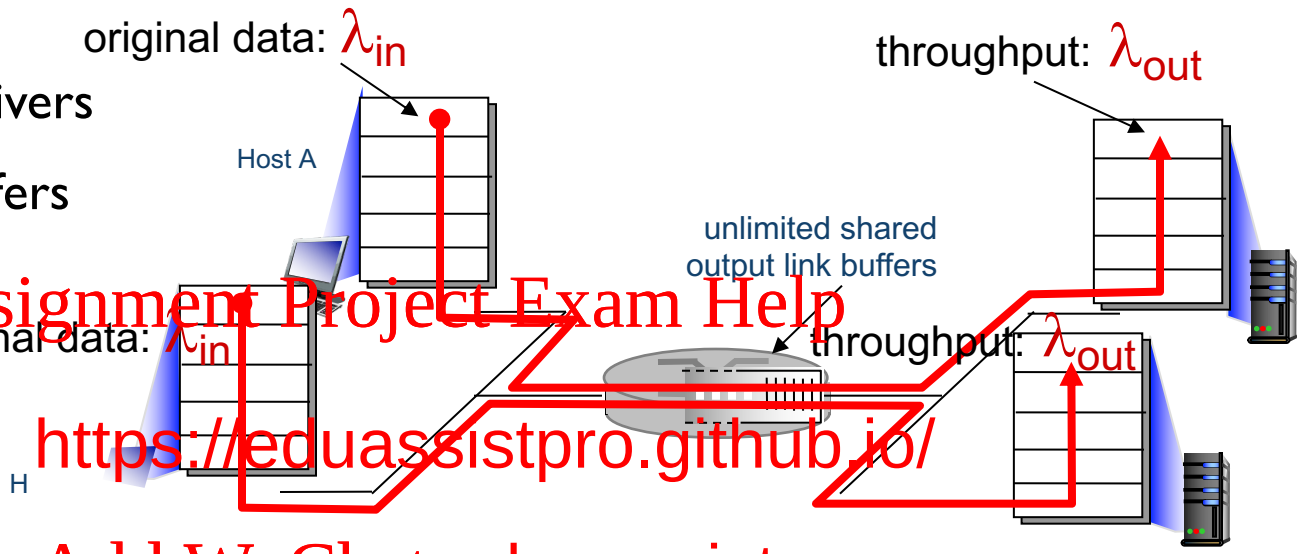


- lost packets (buffer overflow at routers)
- long delays (queueing in router buffers)

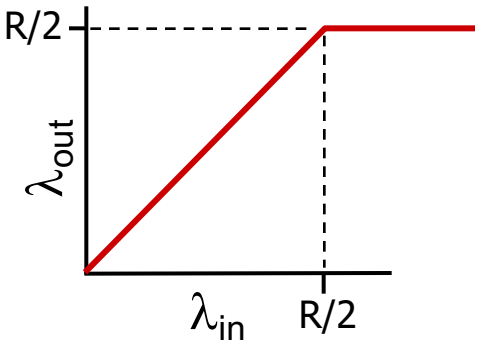
› a top-10 problem!

# Causes/costs of congestion: scenario 1

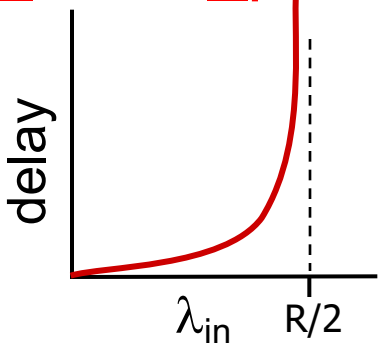
- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission






- maximum per-connection throughput: R/2

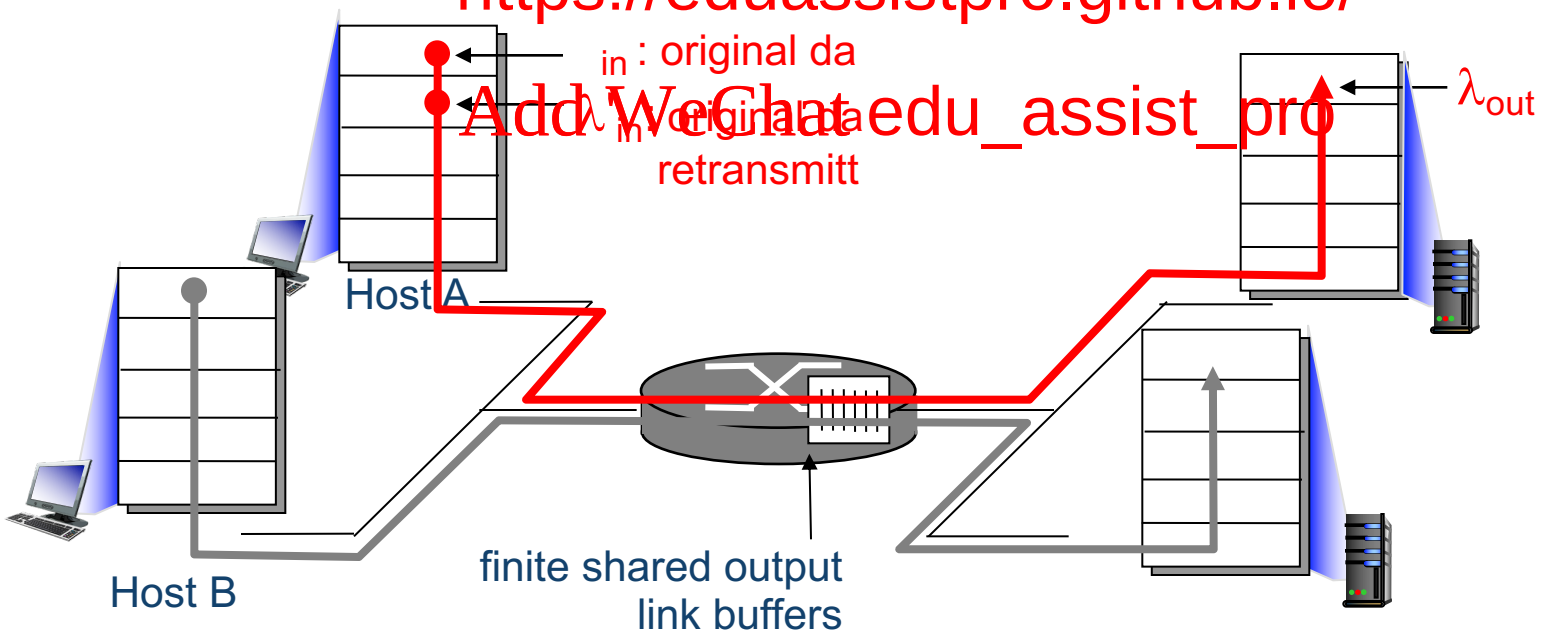



- large delays as arrival rate, $\lambda_{in}$, approaches capacity

# Causes/costs of congestion: scenario 2

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

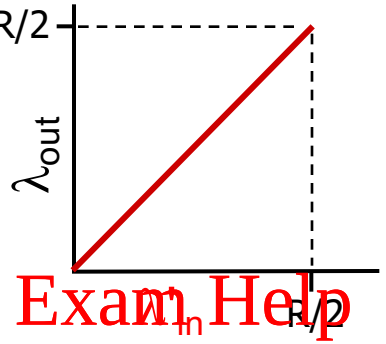




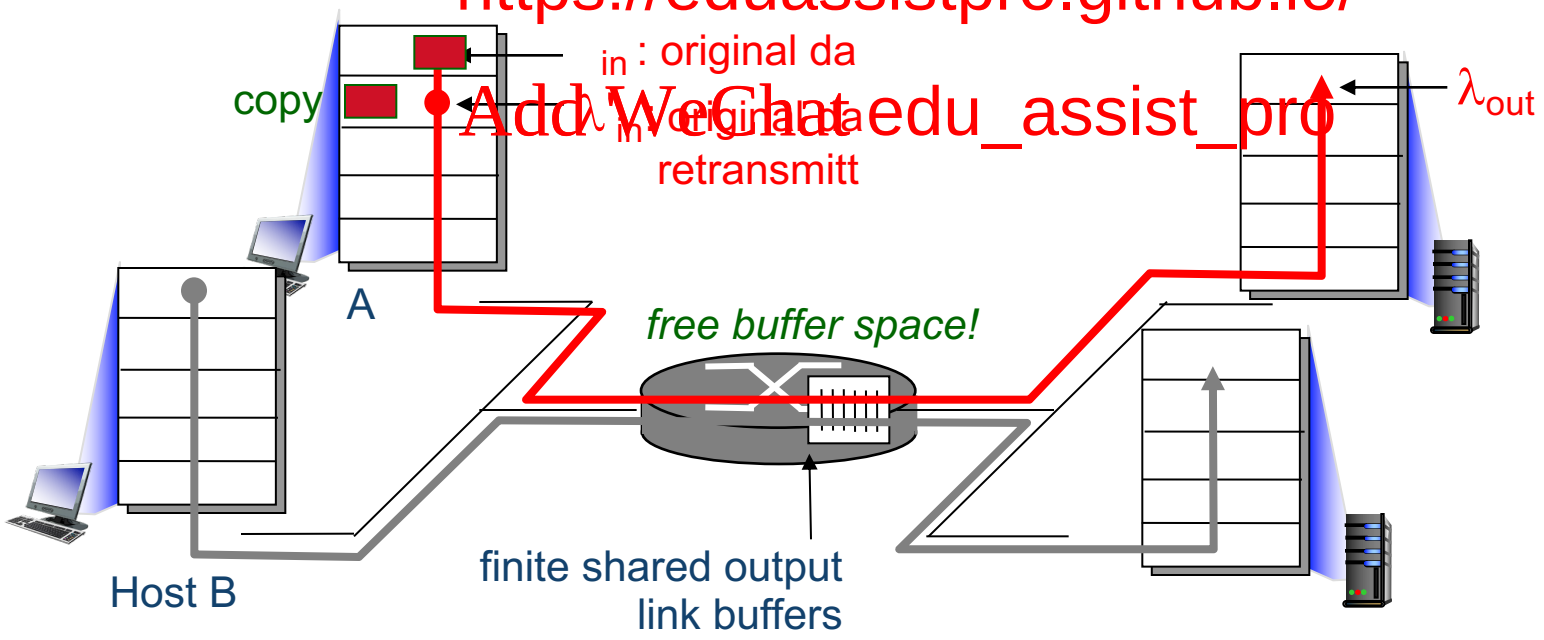
# Causes/costs of congestion: scenario 2

## idealization: perfect knowledge

- sender sends only when router buffers available





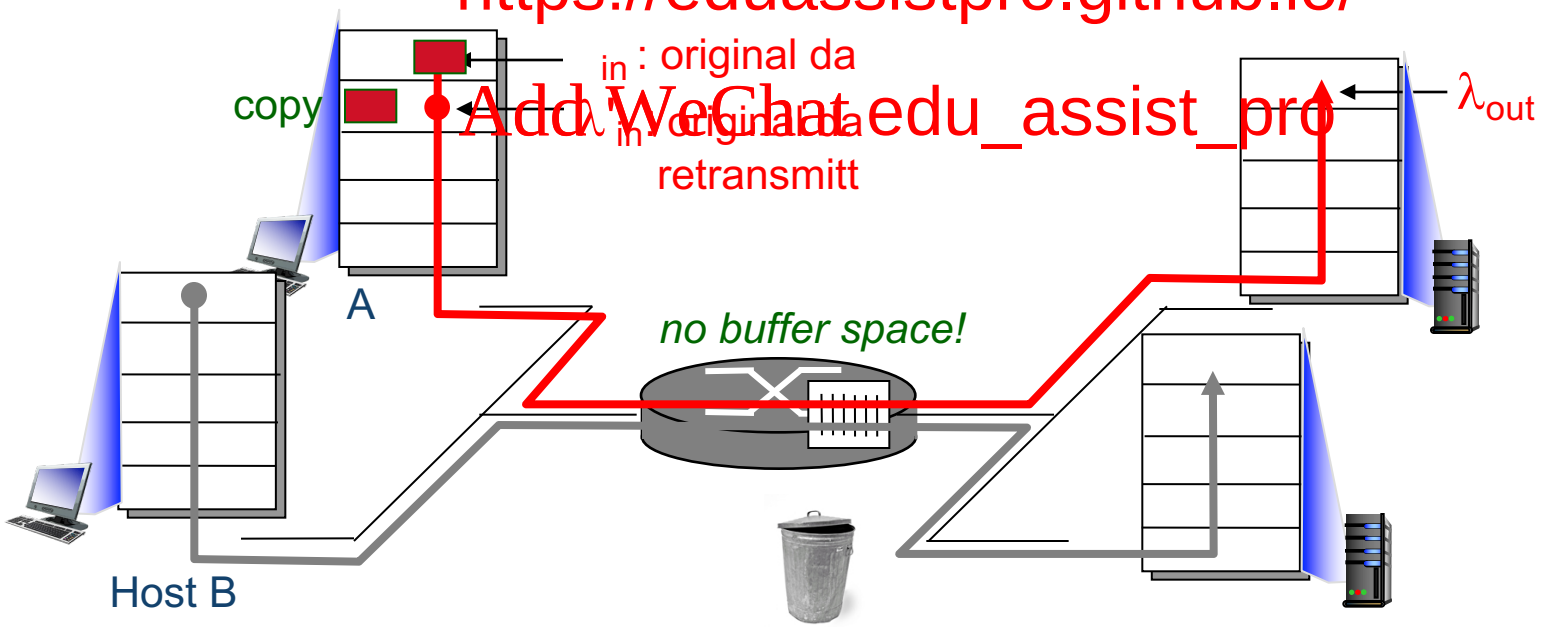
# Causes/costs of congestion: scenario 2

*Idealization: known loss*

packets can be lost, dropped at router due to full buffers

› sender only resends if packet *known* to be lost



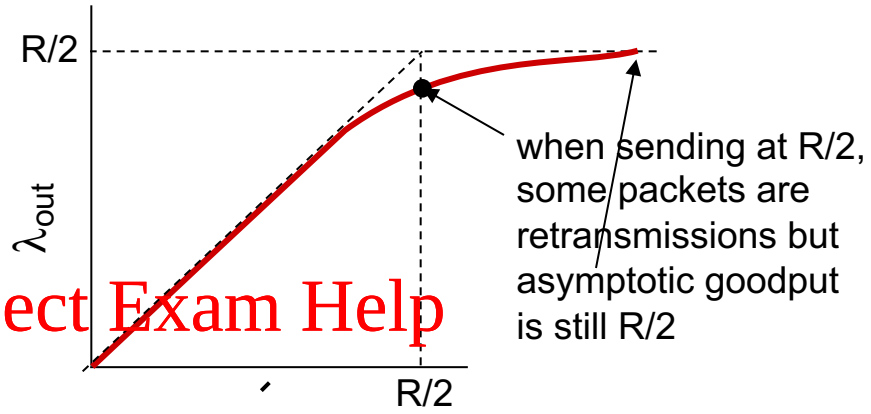


# Causes/costs of congestion: scenario 2

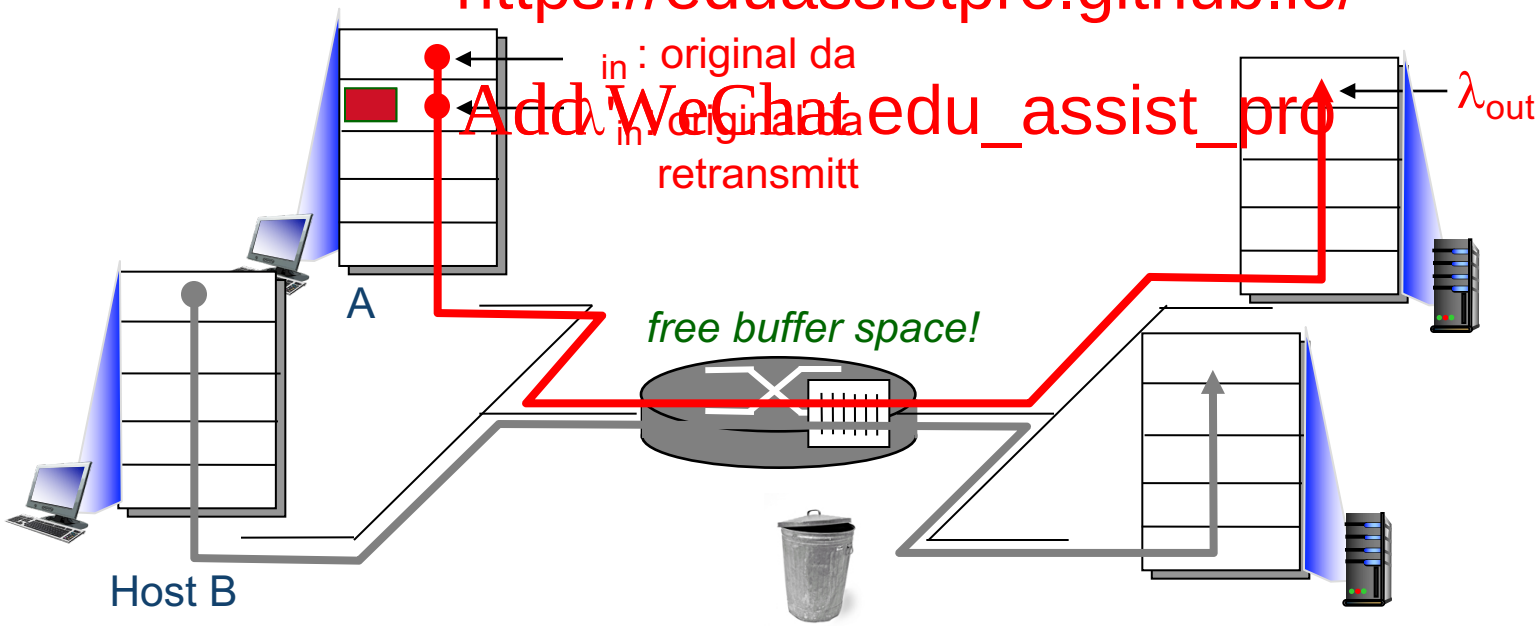
*Idealization: known loss*

packets can be lost, dropped at router due to full buffers

› sender only resends if packet *known* to be lost



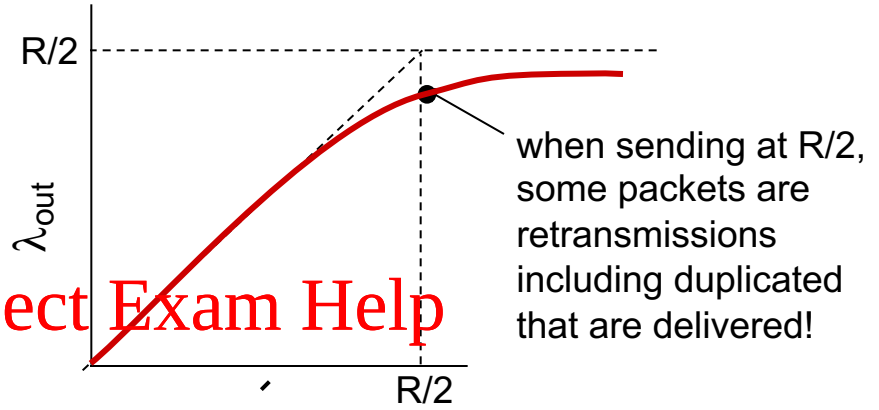


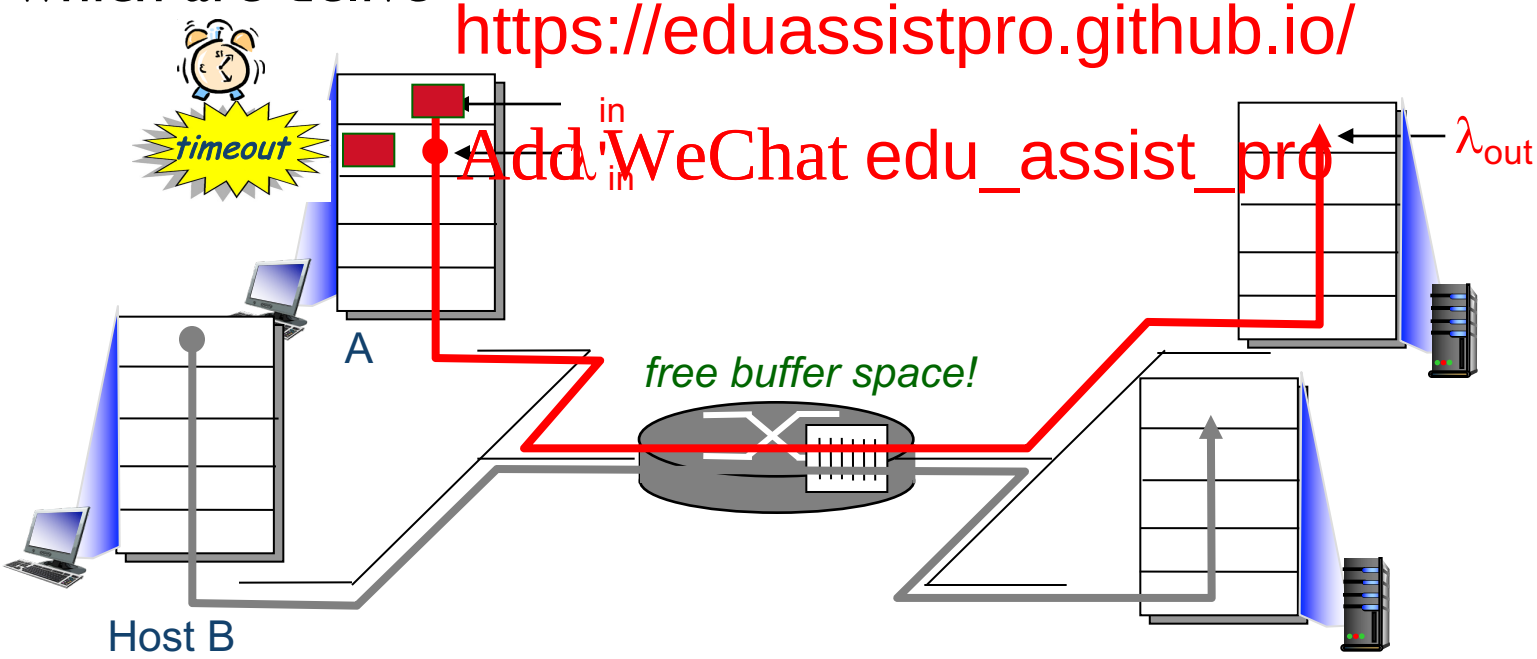
# Causes/costs of congestion: scenario 2

## *Realistic: duplicates*

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



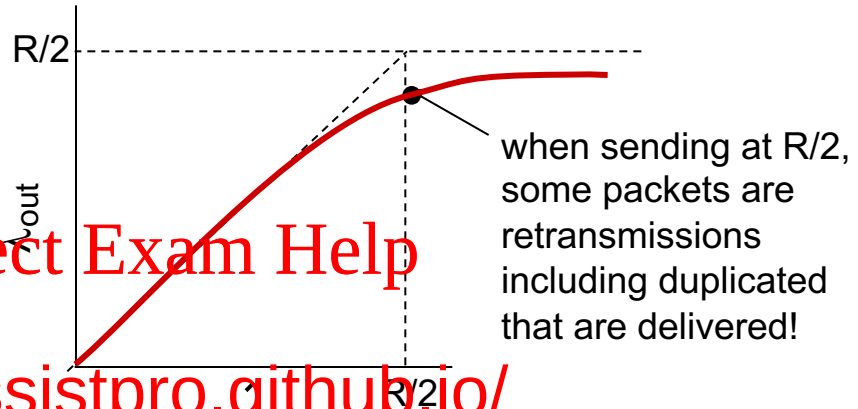


# Causes/costs of congestion: scenario 2

## *Realistic: duplicates*

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copi which are delive






## "costs" of congestion:

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

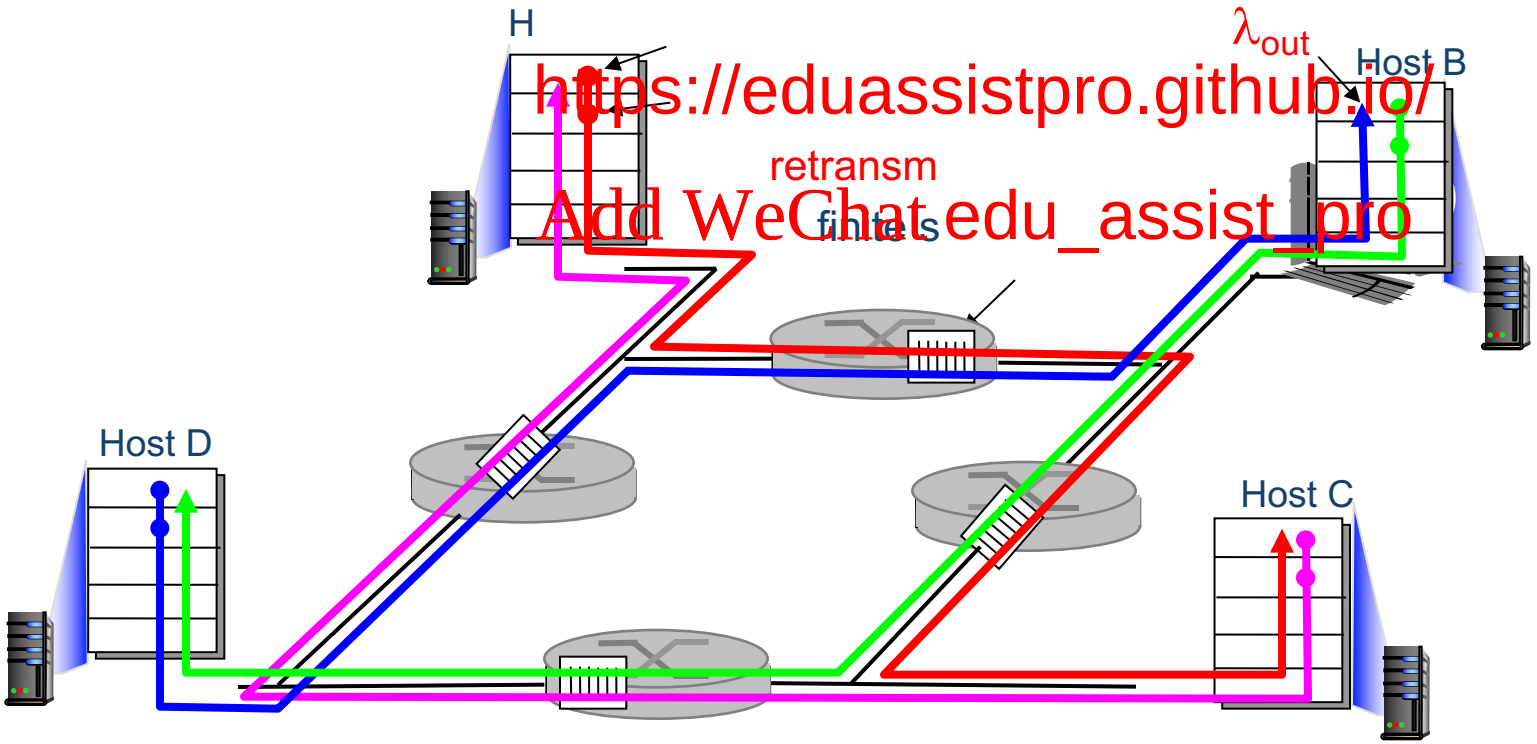
# Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

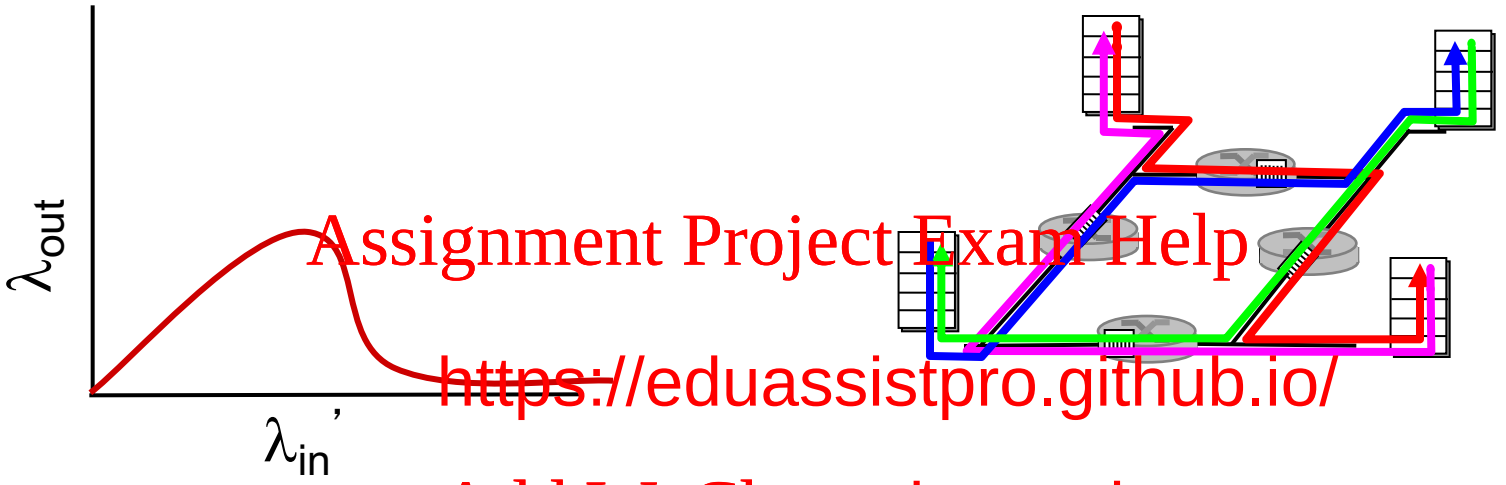
Q: what happens as $\lambda_{in}'$ increases ?

A: as red $\lambda_{in}'$ 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 used for that packet was wasted!

# Approaches towards congestion control

two broad approaches towards congestion control:

**end-end congestion control:**

- no explicit feed from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

**network-assisted congestion control:**

- single bit indicating congestion
- explicit rate for sender to send at

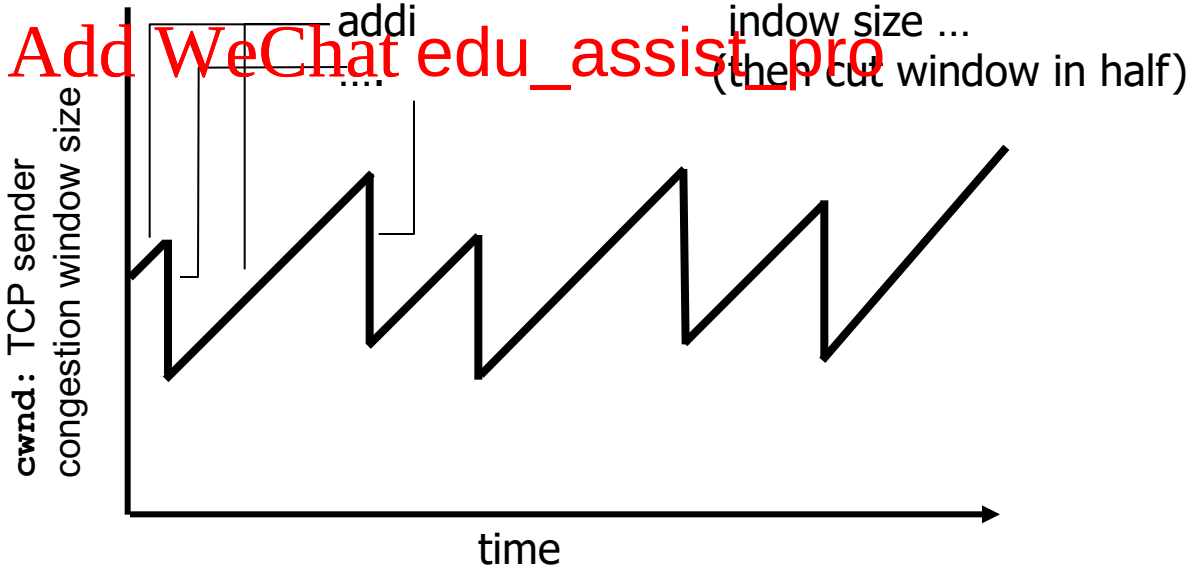
# TCP Congestion Control

# Additive increase multiplicative decrease (AIMD)

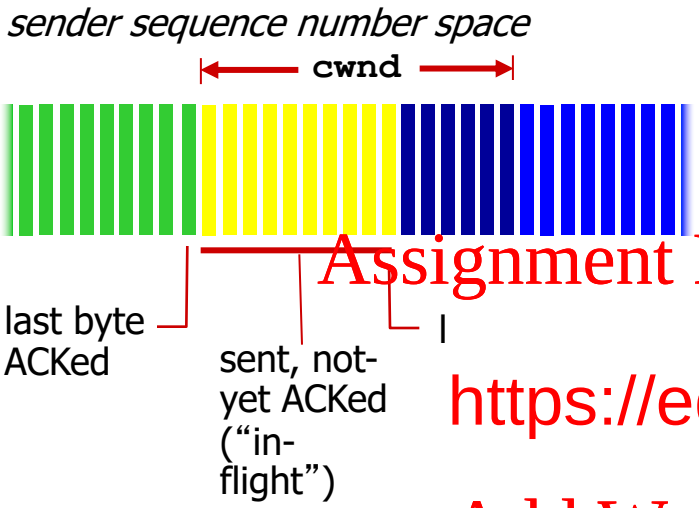
- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase cwnd by 1 MSS (maximum segment siz
  - *multiplicative* detected half after loss



AIMD saw tooth behavior: probing for bandwidth



# TCP Congestion Control: details




*TCP sending rate:*

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



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

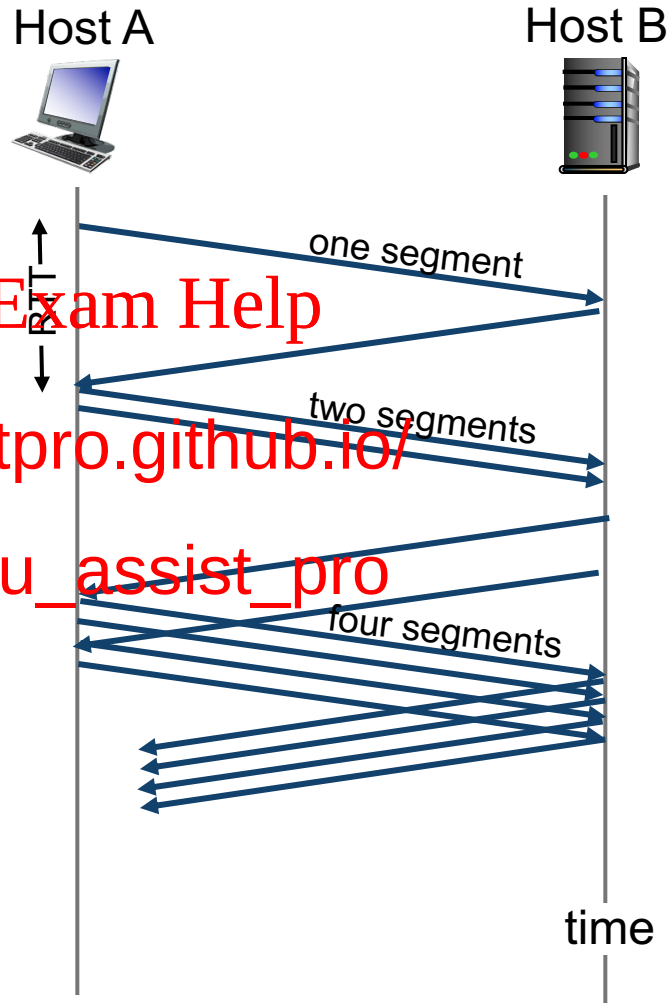
› sender limits transmission:

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}$$

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

# TCP Slow Start

- when connection begins, increase rate exponentially:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by increme every ACK receiv
- *summary:* initial rate is slow but ramps up exponentially fast
- when should the exponential increase switch to linear (additive increase)?



# TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: cwnd reaches **`ssthresh`**

**`cwnd=12`** **`ssthresh=6`**

loss!



Implementation:

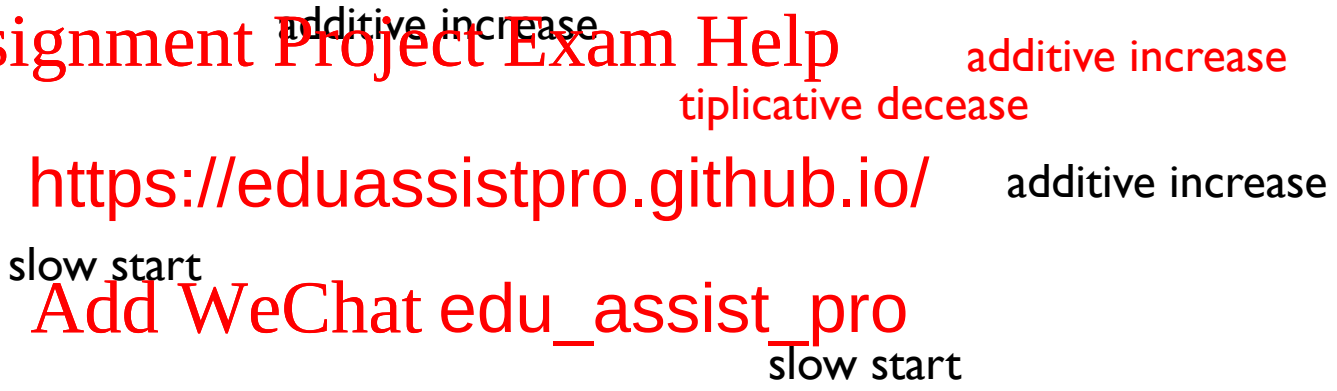
- At beginning **`ssthresh`**, specified in different versions of TCP
- (In this example **`ssthresh=8 segment`**)
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

# TCP: detecting, reacting to loss

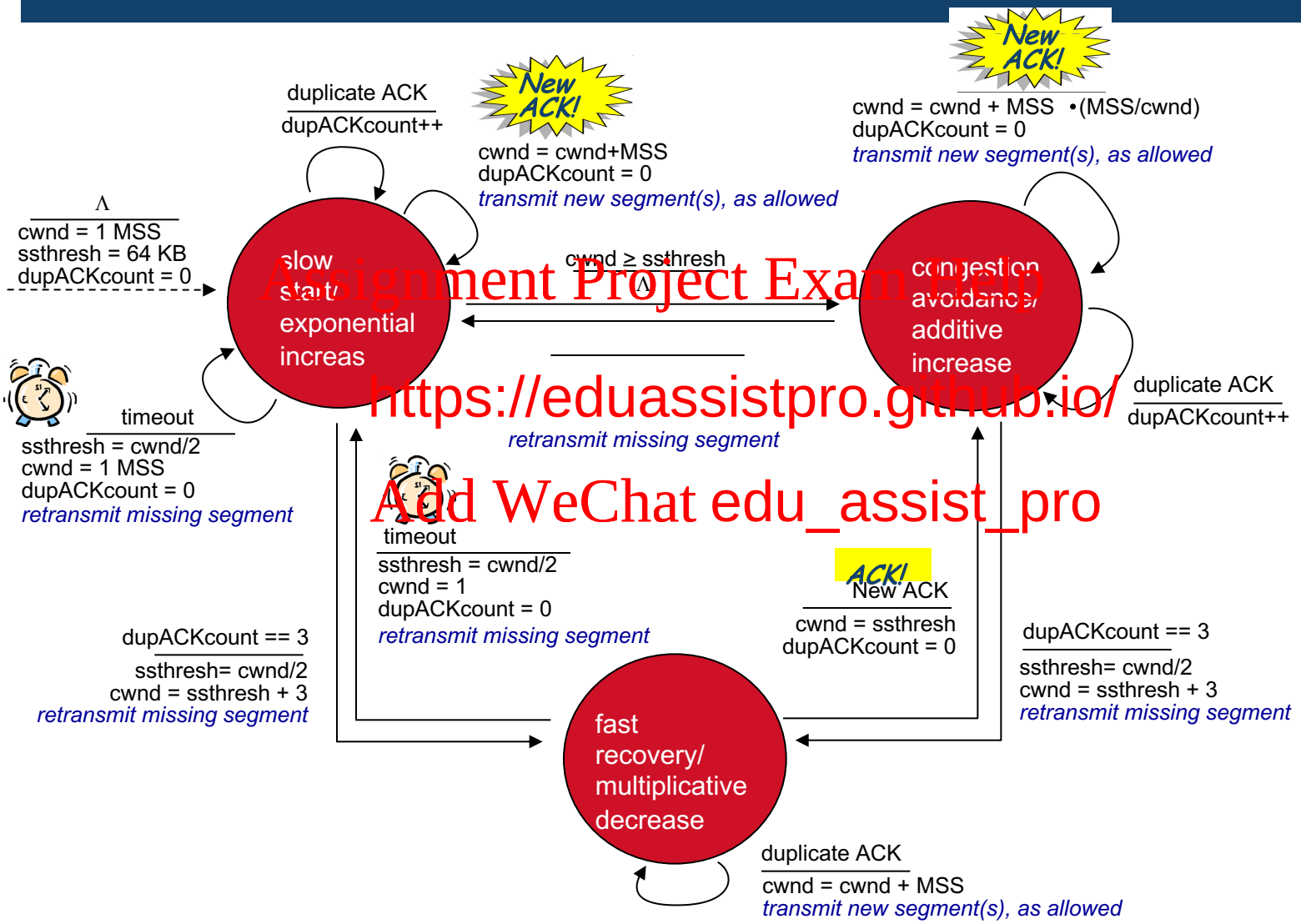
- loss indicated by timeout:
  - **cwnd** set to 1 MSS;
  - window then grows exponentially (as in slow start) to **ssthresh**, then grows linearly
- loss indicated by
  - TCP Tahoe, same as loss indicated by ti ets **cwnd** to 1 (timeout or 3 duplicate acks)
  - TCP RENO
    - **cwnd** is cut in half window then grows linearly (additive increase)
    - fast recovery

# TCP: switching from slow start to CA





# Summary: TCP Reno Congestion Control

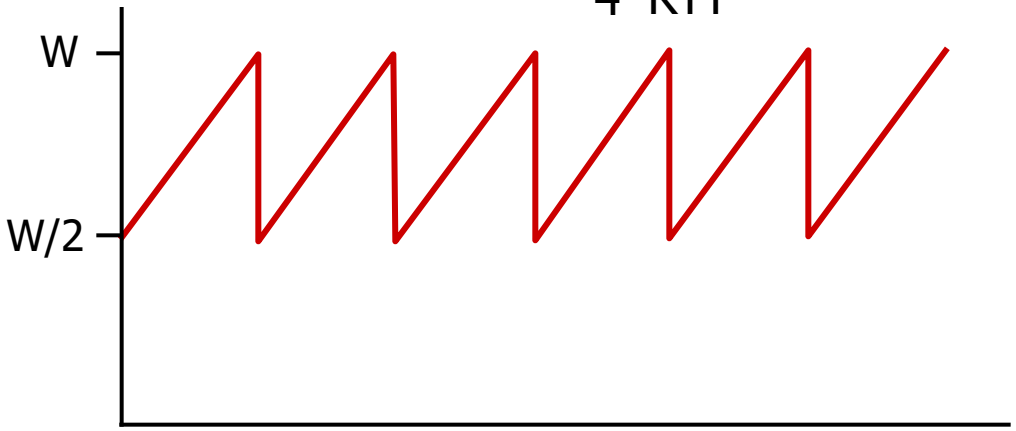


# TCP Reno throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window si
  - avg. thruput is 3/4W per RTT



$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



# TCP Futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in te bability, L [Mathis 1997]:



$$\text{TCP throughput} = \frac{1.}{\sqrt{}}$$

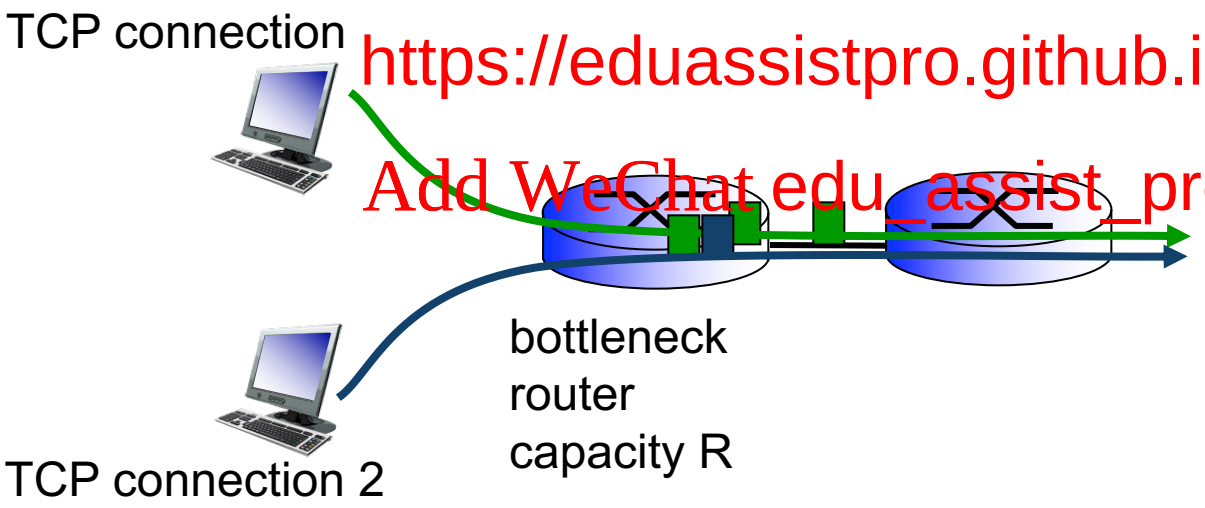
➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$
*– a very small loss rate!*

- new versions of TCP for high-speed
  - Vegas, Westwood, CUBIC, etc.

# TCP Fairness

*Fairness:* K TCP sessions share same bottleneck link of bandwidth R, each has average rate of R/K



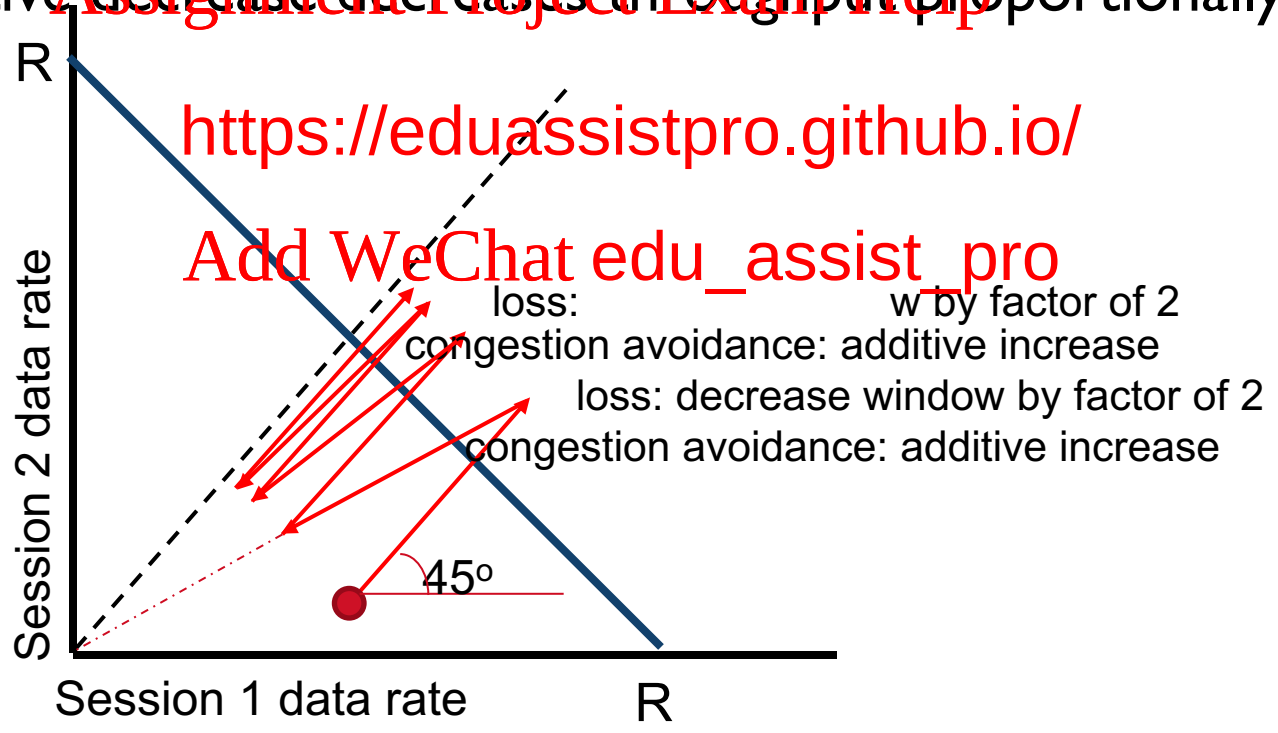


# Why is TCP fair?

two competing sessions:

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





# Fairness (more)

## *Fairness and UDP*

- multimedia apps often do not use TCP
  - do not want rate by congestion co
- 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
  - f rate R
  - 1 TCP, gets 0.1R
  - 9 TCPs, gets 0.9R

# Final word

*Window size = min (rwnd, cwnd)*



*receive window* *n window*

*flow control* *congestion control*

# Conclusion

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - connection setup, teardown
  - flow control
  - congestion control
- instantiation, implementation i net
  - UDP
  - TCP