COMP 3331/9331:

Comp ks and

Wee

Transport Layer (Continued)

Reading Guide: Chapter 3, Sections: 3.4, 3.5

Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexi

3.3 connectionle transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- able data transfer
- contro
- ction management

3. es of congestion control

3.7 TCP congestion control

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retran possible duplicate

handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- dds *sequence* ach pkt
- iscards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: discussion

sender:

- seq # added to pkt
- two seq. #'s (0, 1) will suffice. Why?
- must check if ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

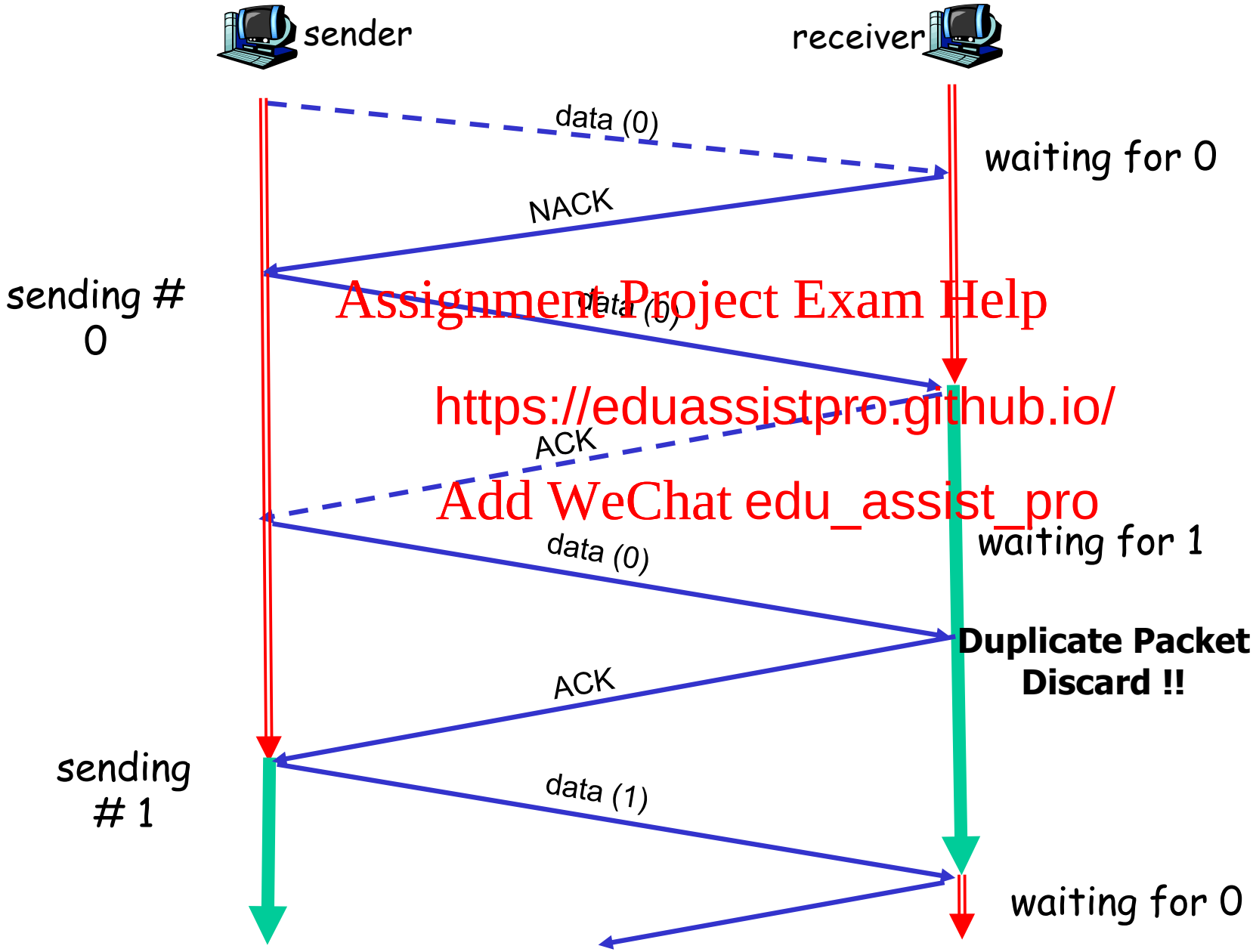
receiver:

- must check if received packet is duplicate
  - indicates whether is expected pkt
- receiver can *not* know if its last ACK/NAK received OK at sender

- New Measures: Sequence Numbers, Checksum for ACK/NACK, Duplicate detection

Another Look at rdt2.1

Dotted line: erroneous transmission
Solid line: error-free transmission

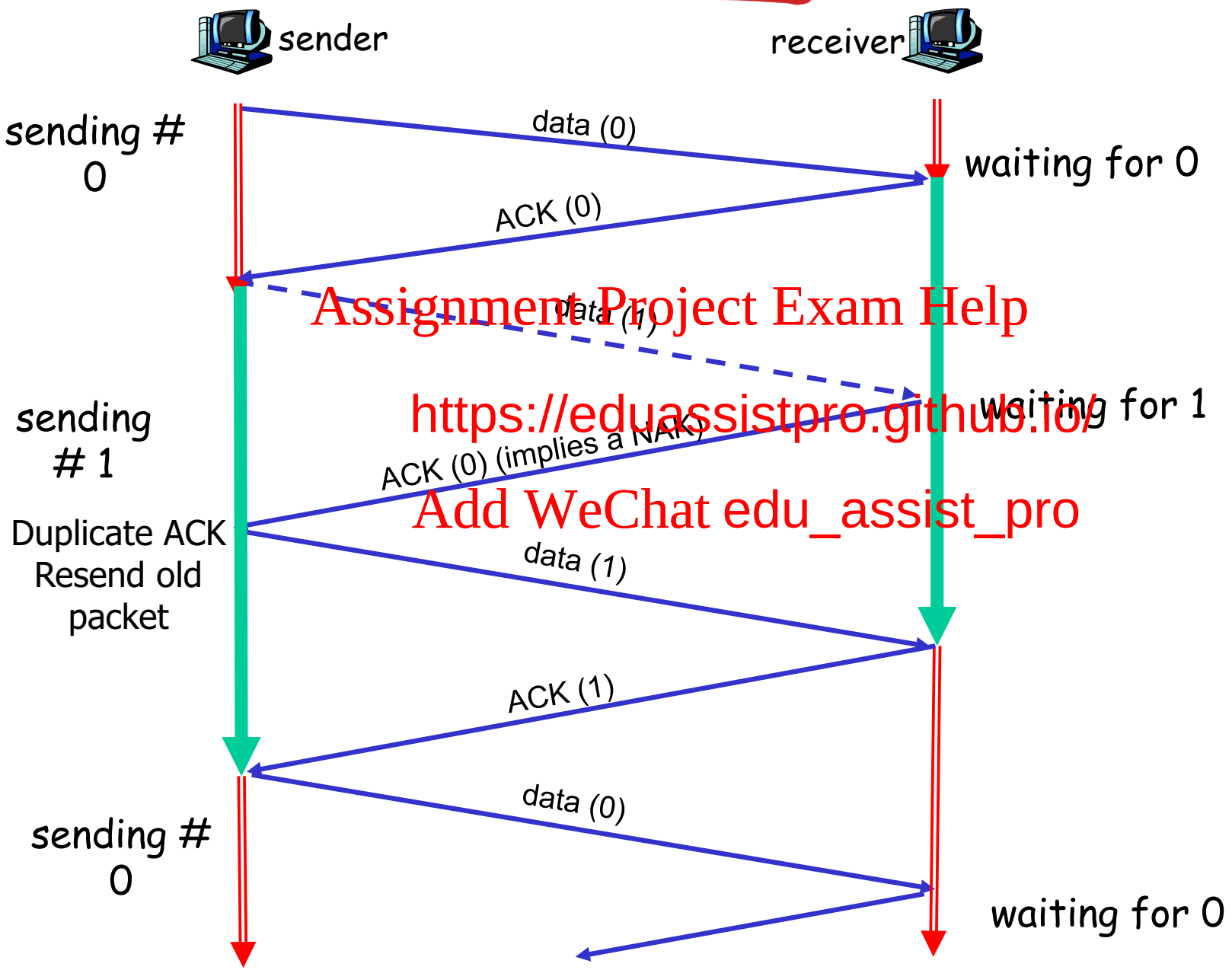


rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must f pkt being ACKed
- duplicate ACK ame action as NAK: *retransmit current pkt*

rdt2.2: Example

Dotted line: erroneous transmission
Solid line: error-free transmission



rdt3.0: channels with errors *and* loss

new assumption:

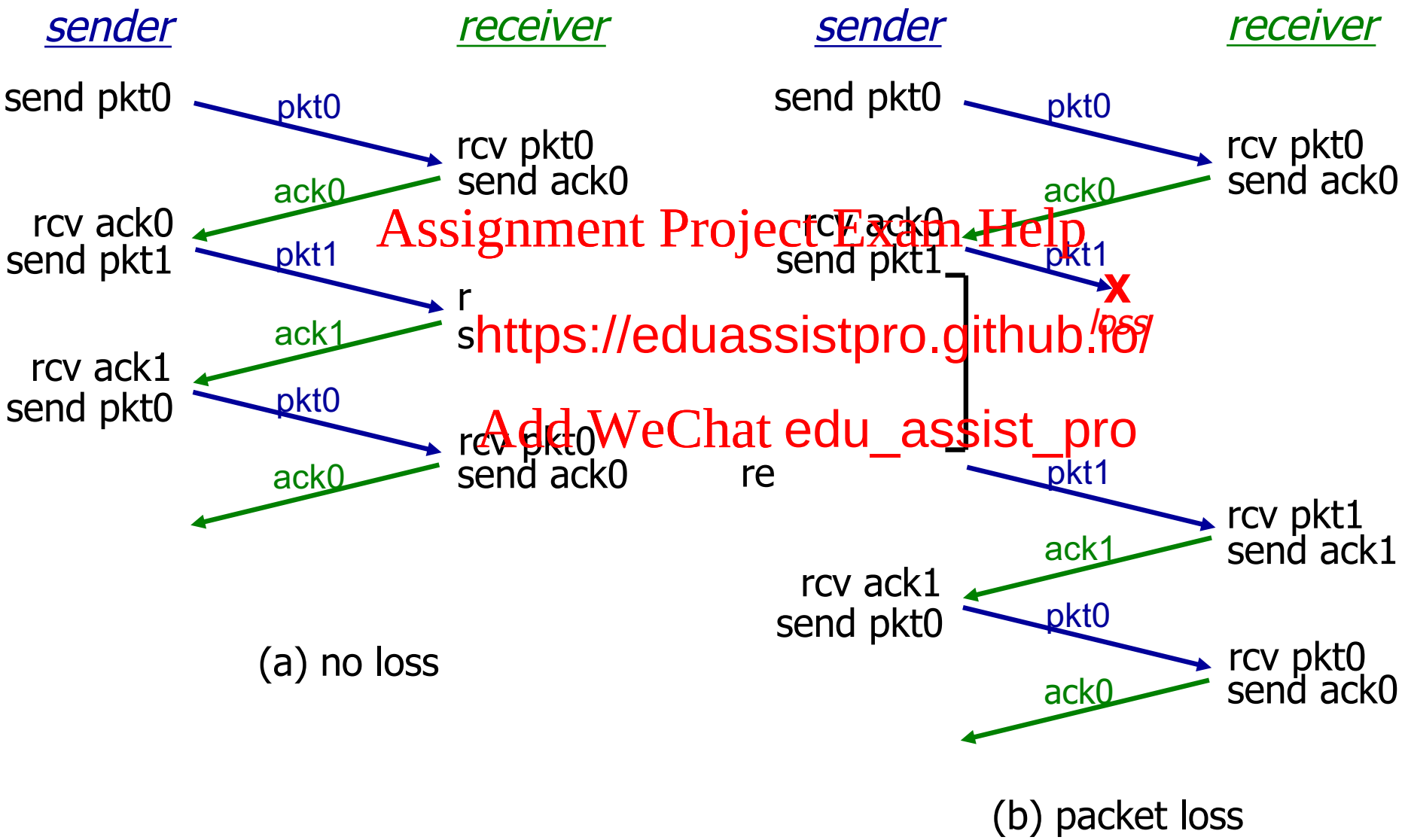
underlying channel can also lose packets (data, ACKs)

- checksum, se
  ACKs, retran
  will be of help … but not enough

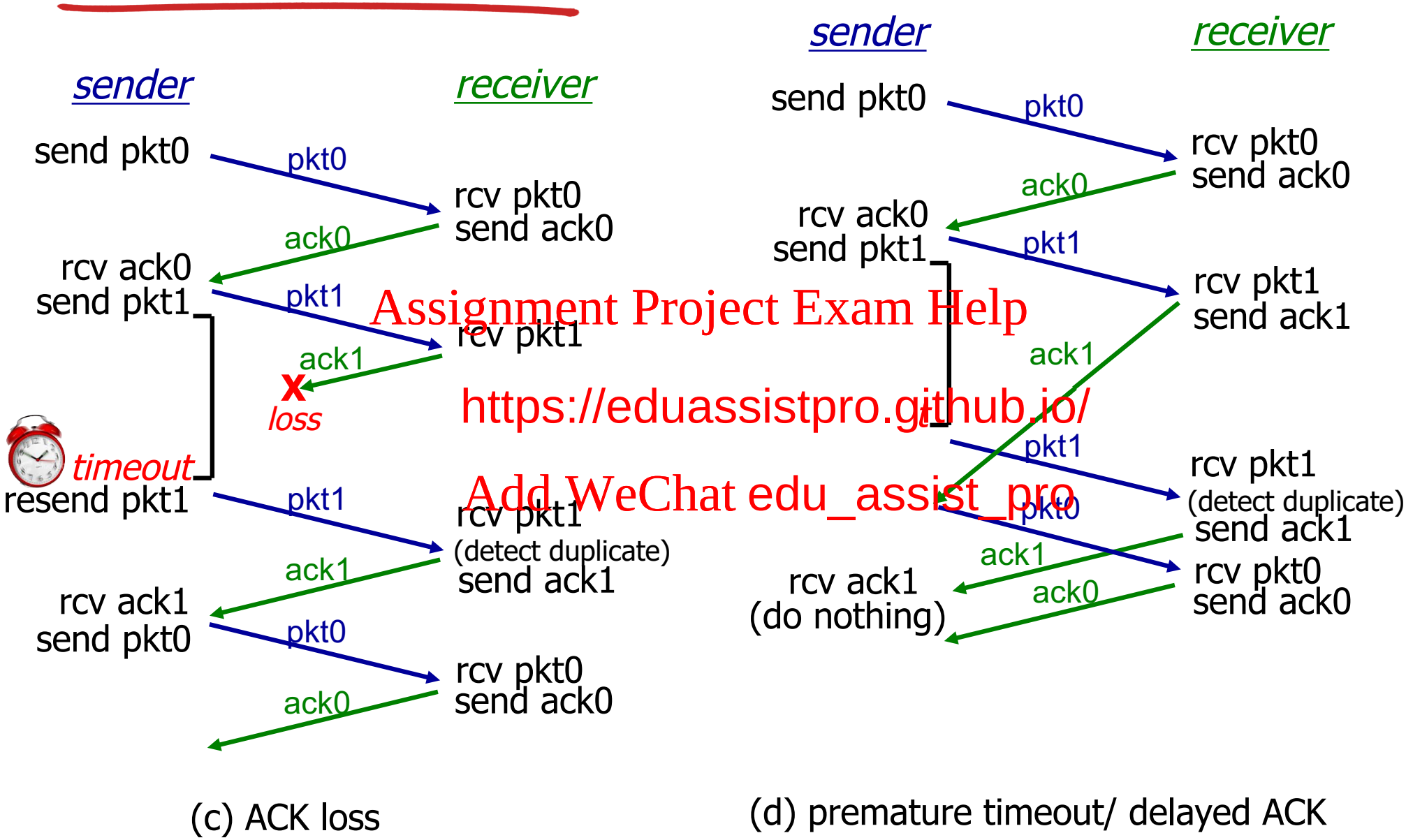
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK in this time
  r ACK d
  mission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

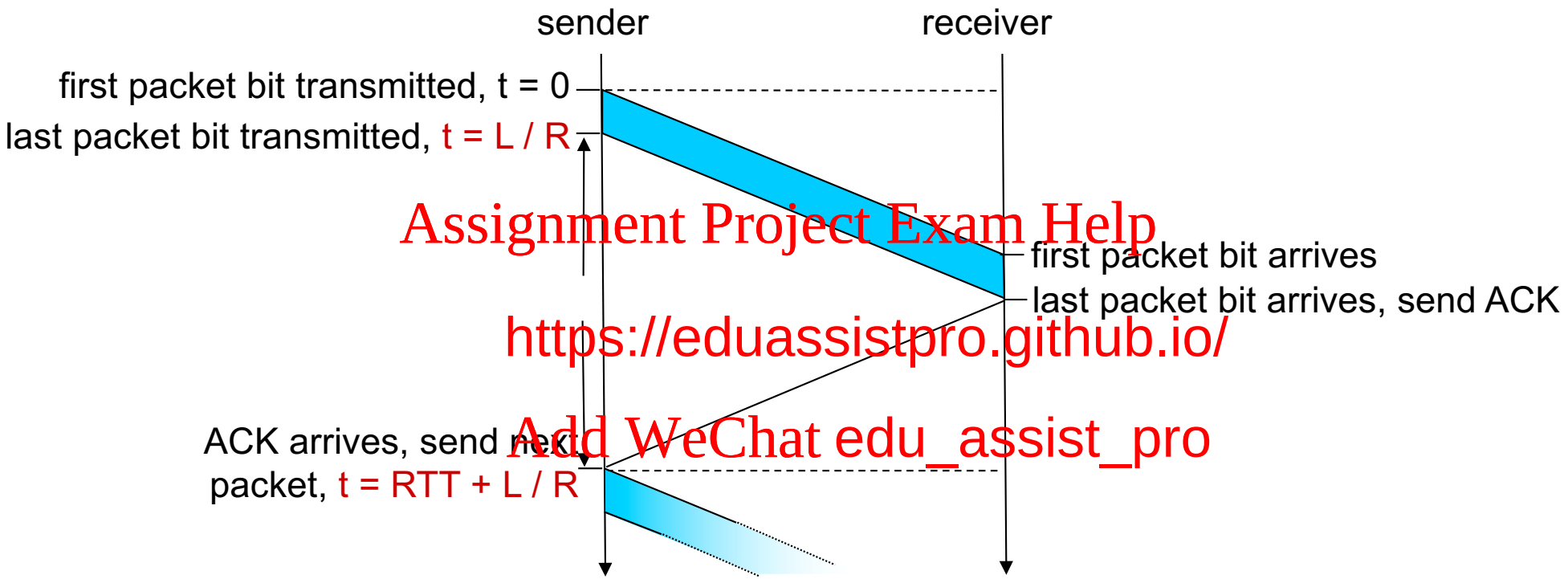
rdt3.0 in action



rdt3.0 in action



rdt3.0: stop-and-wait operation



$$U_{sender} = \frac{L / R}{RTT + L / R}$$

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 8000 bit packet and 30msec RTT:

$$D_{trans} = \frac{L}{} \quad \frac{8000\ bits}{} \quad 8\ microsecs$$

- U_{sender}: *utilization* – fraction der busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{008}{30.008} = 0.00027$$

  - RTT=30 msec, 1KB pkt every 30.008 msec: 33kB/sec thruput over 1 Gbps link
- Network protocol limits use of physical resources!

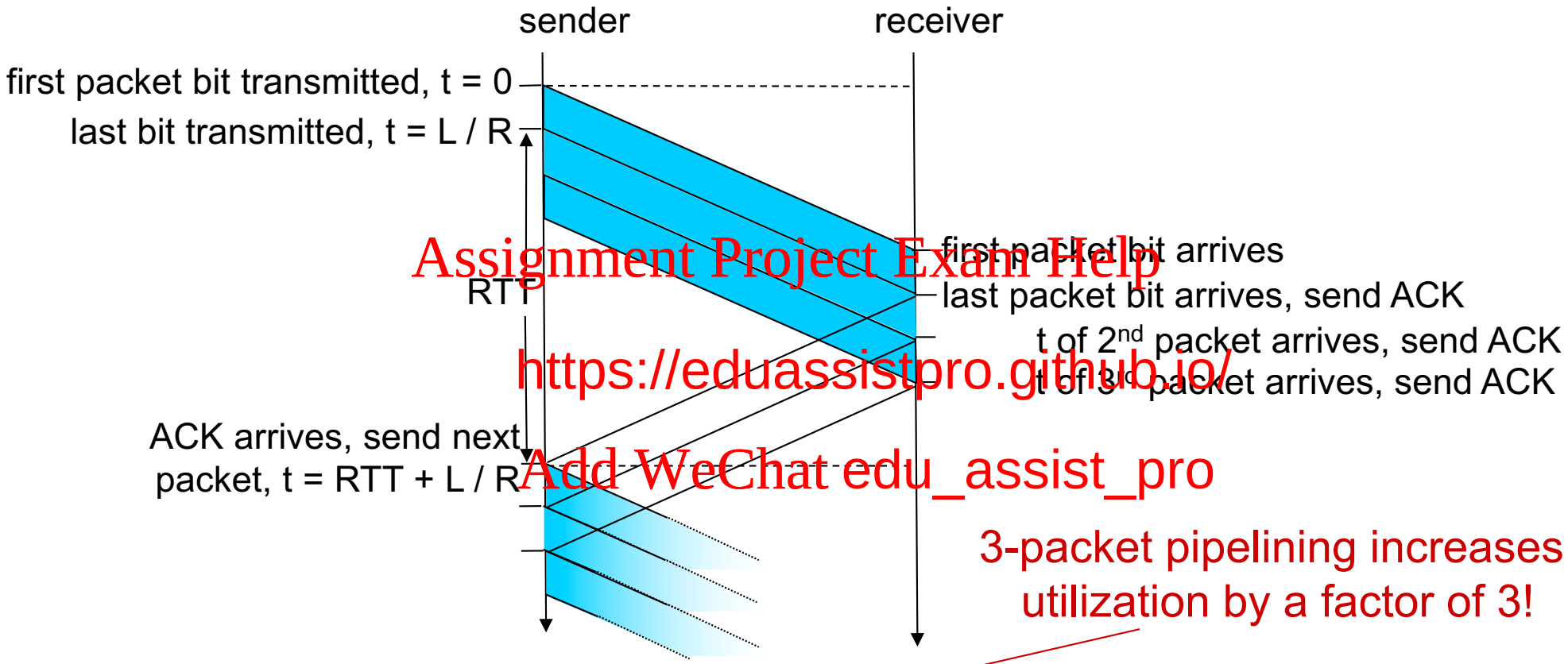
Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

- two generic forms of pipelined (sliding window) protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



$$U_{sender} = \frac{3L / R}{RTT + L / R}$$

Pipelined protocols: overview

Go-Back-N:

- Sender can have up to N unacked packets in pipeline
- Sender has single timer for oldest unacked pack timer expires, retra unacked packets
- There is no buffer available at Receiver, out of order packets are discarded
- Receiver only sends *cumulative ack,* doesn't ack new packet if there's a gap

Selective Repeat:

- Sender can have up to N unacked packets in pipeline
- Sender maintains timer for cked packet, when pires, retransmit nacked packet
- has buffer, can accept out of order packets
- Receiver sends *individual ack* for each packet

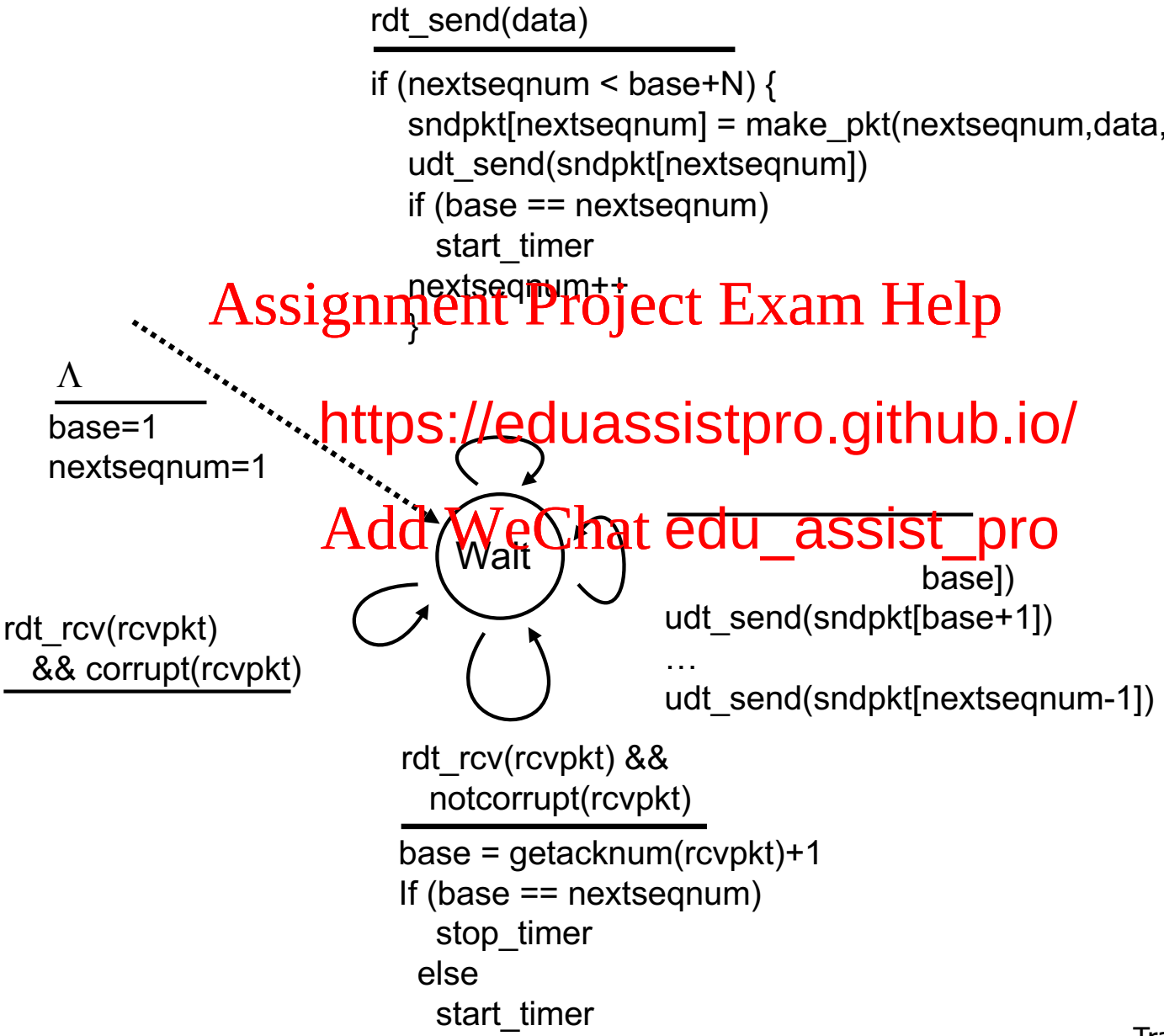
Go-Back-N: sender

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

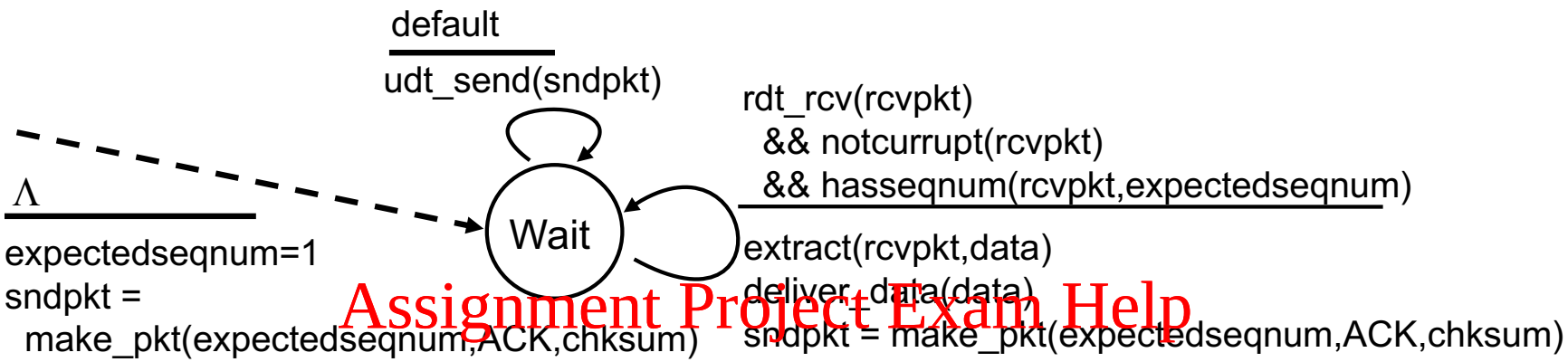
- ACK(n): ACKs all pkts up to, i q # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

Applets: http://media.pearsoncmg.com/aw/aw_kurose_network_2/applets/go-back-n/go-back-n.html
http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

GBN: sender extended FSM



GBN: receiver extended FSM

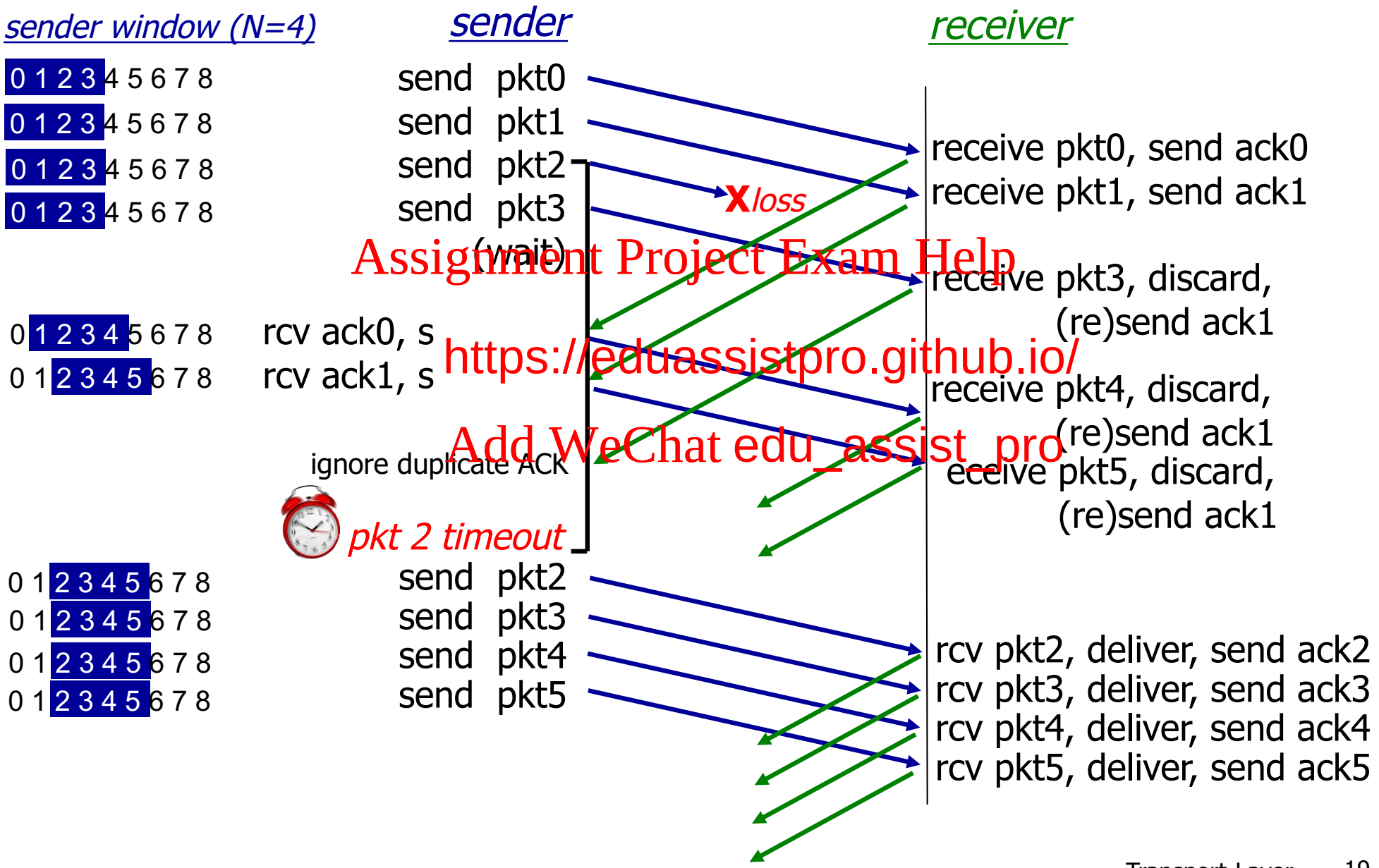


ACK-only: always send ACK for correctly-received pkt with highest *in-order*

- 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 r ACK not received
  - sender timer for each unAC
- sender window
  - *N* consecutive seq #' s
  - limits seq #s of sent, unACKed pkts

Applet: http://media.pearsoncmg.com/aw/aw_kurose_network_3/applets/SelectRepeat/SR.html

Selective repeat: sender, receiver windows

Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, re timer

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

- mark pkt n as received
- if n 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
- rder: deliver (also er buffered, in-order window to t yet-received pkt

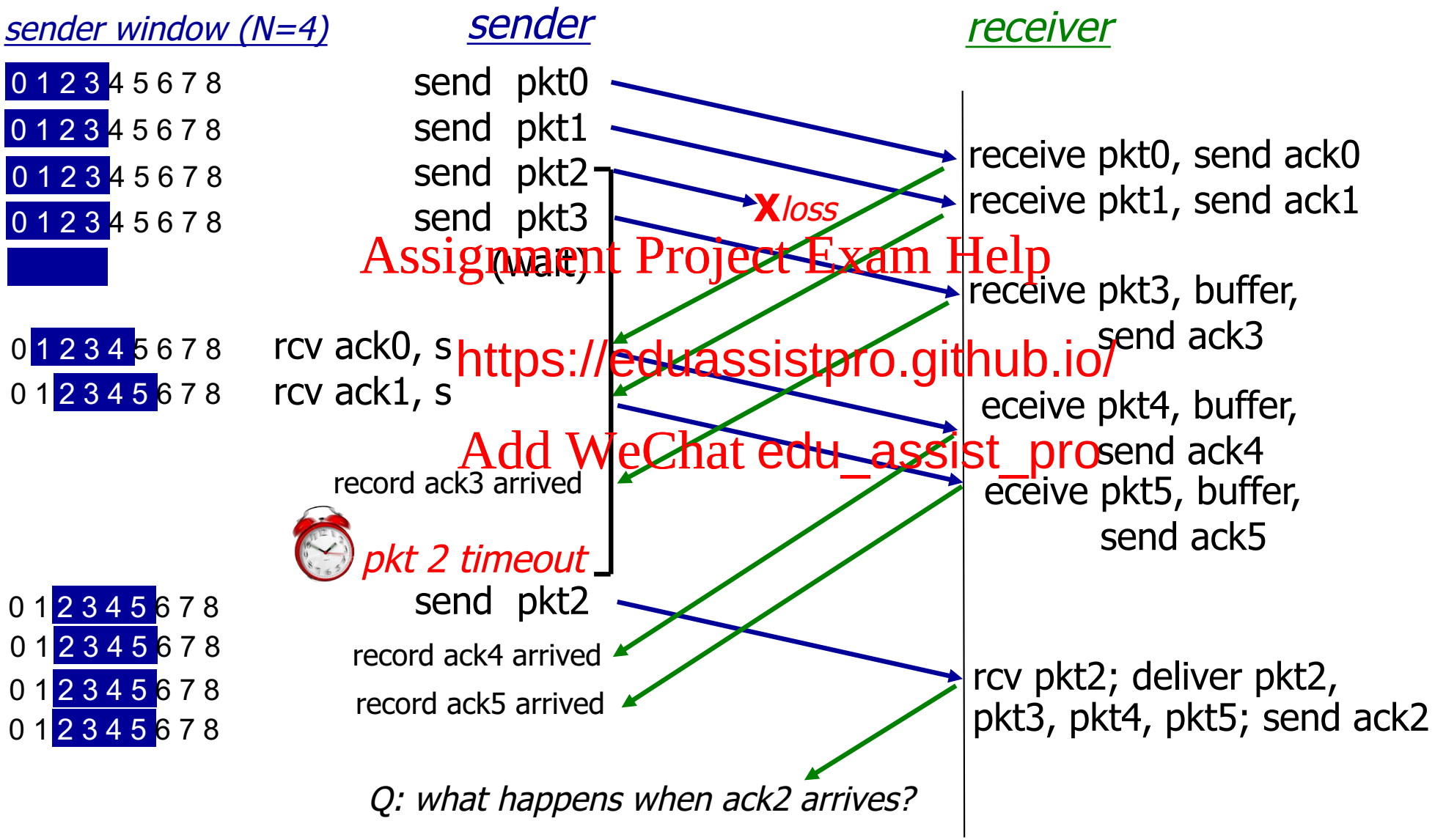
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action

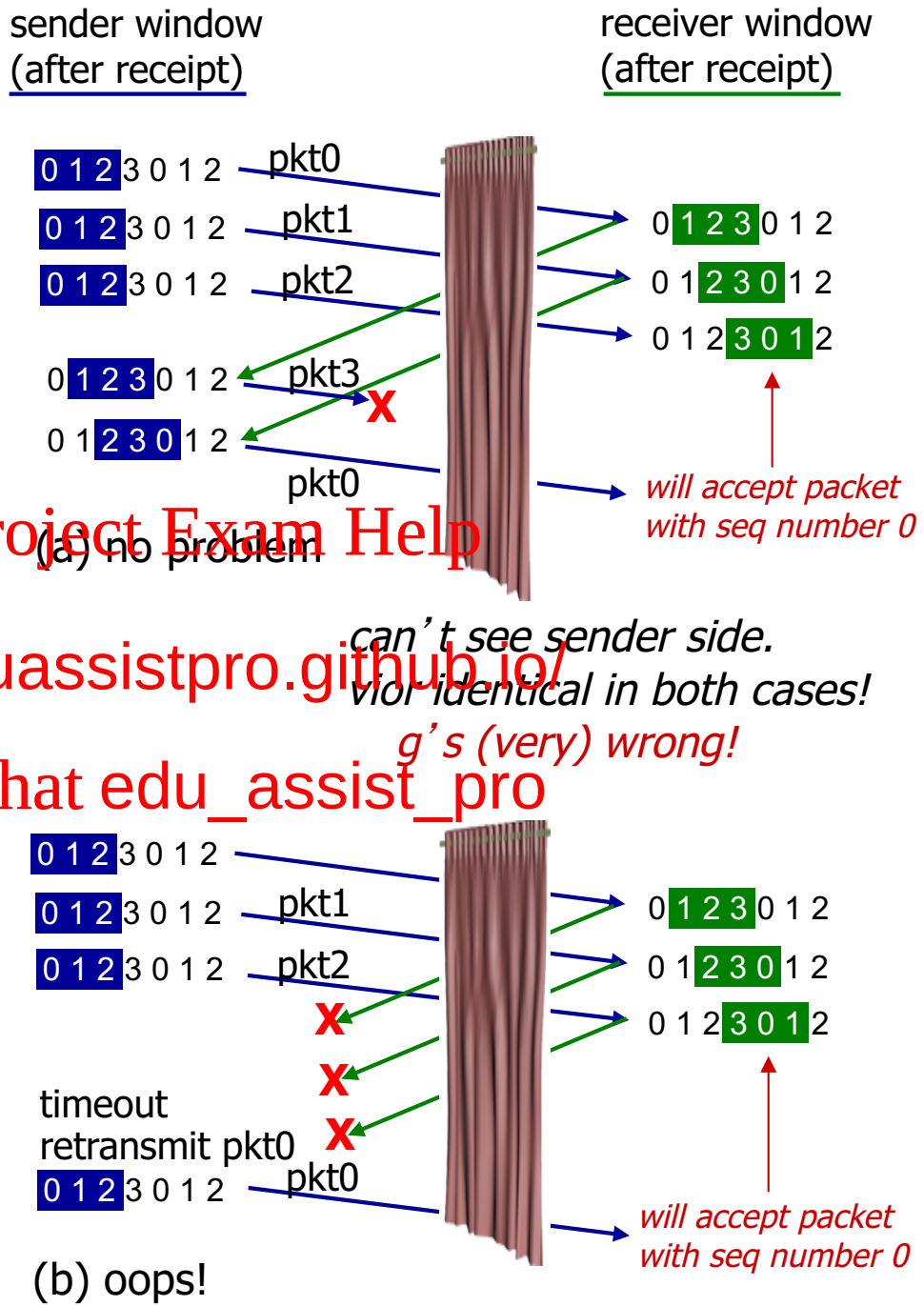


Selective repeat: dilemma

example:

- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in tw scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



Recap: components of a solution

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence nu dows)
- Sliding Windows (for effici

- Reliability protocols use the above to decide when and what to retransmit or acknowledge

Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexi

3.3 connectionle transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- able data transfer
- contro
- ction management

3. es of congestion control

3.7 TCP congestion control

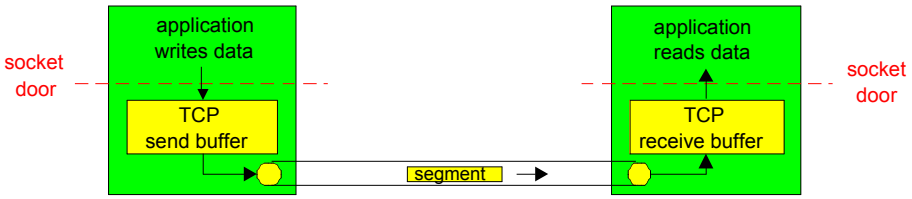
Practical Reliability Questions

- How do the sender and receiver keep track of outstanding pipelined segments?
- How many segments should be pipelined?
- How do we c bers?
- What does co nt and teardown look like?
- How should we choose timeout values?

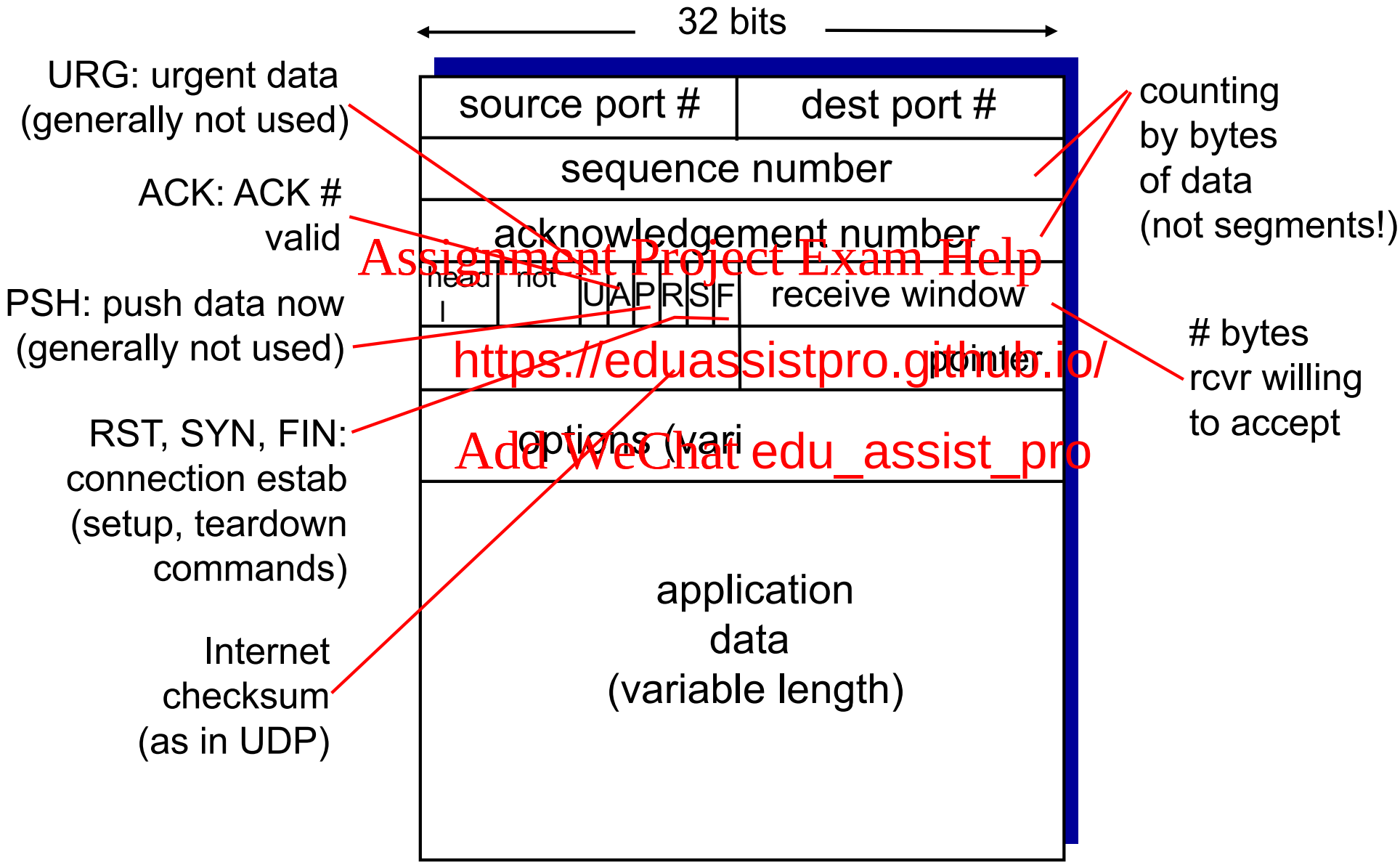
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message
- pipelined:
  - TCP congestion and flow control set window size
- send and receive buffers

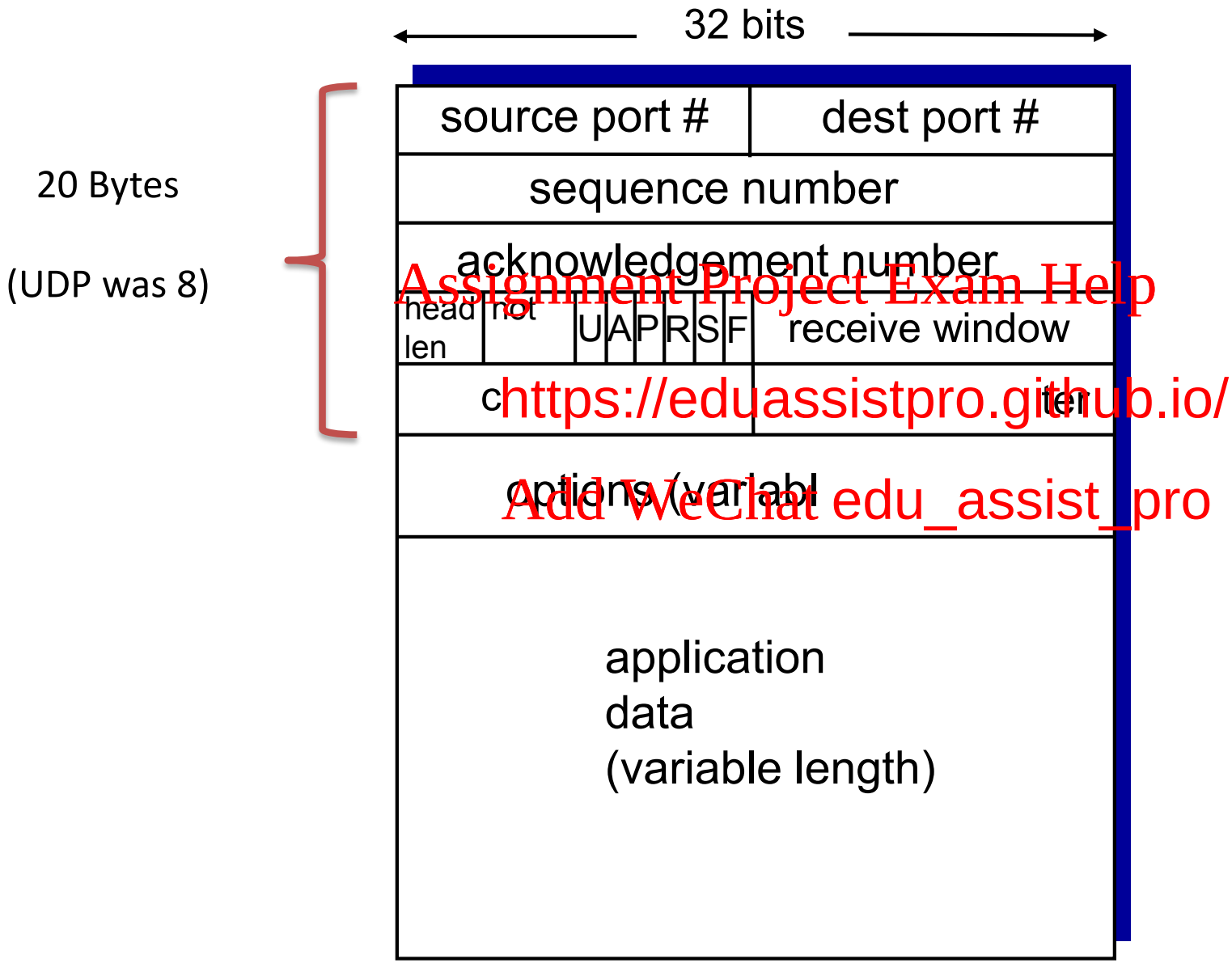
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment
    e
- ec -oriented:
  - shaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver



TCP segment structure



TCP segment structure



Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexi

3.3 connectionle transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure

able data transfer

control

ction management

3. es of congestion control

3.7 TCP congestion control

Recall: Components of a solution for reliable transport

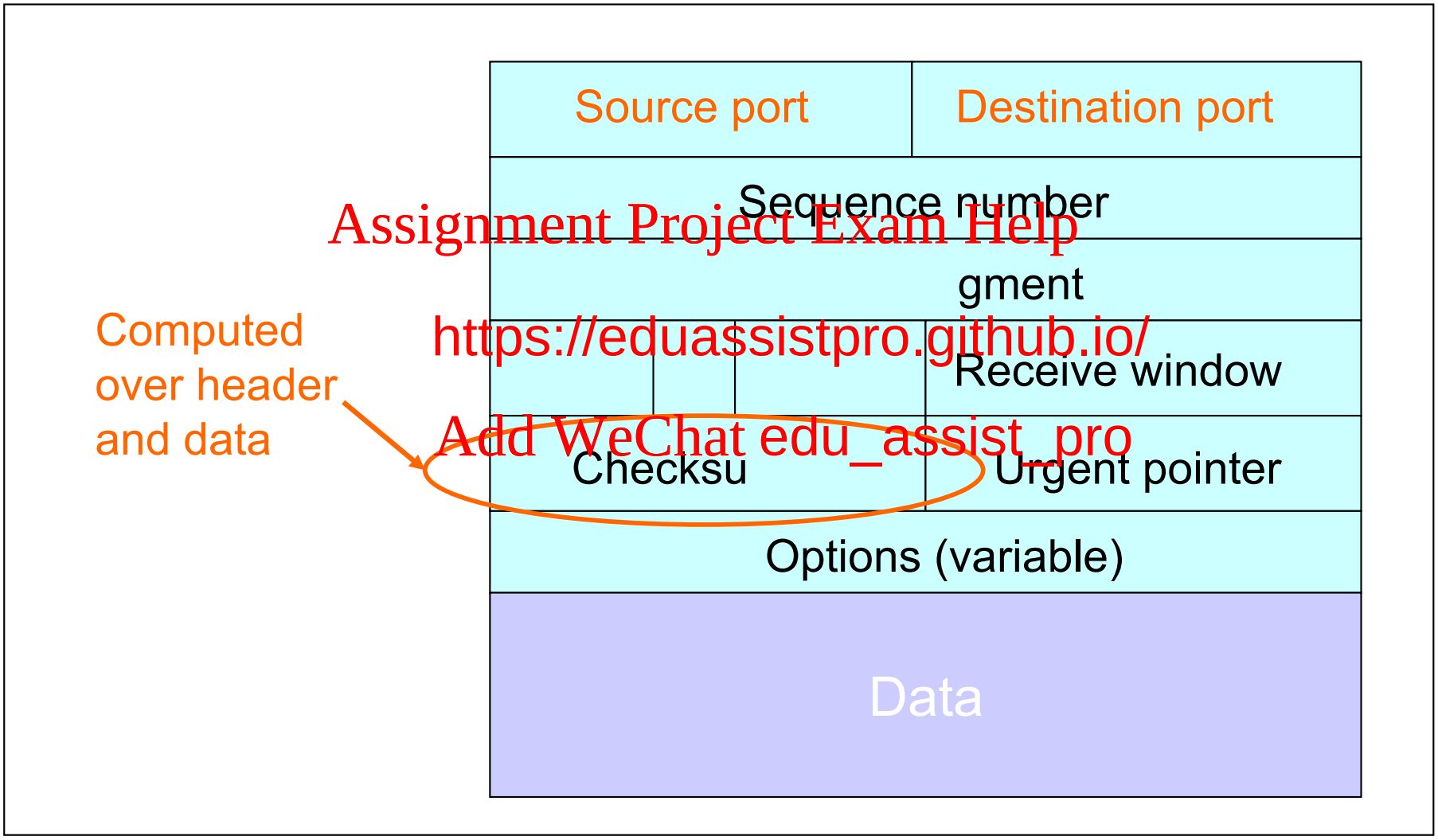
- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledg
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
  - Go-Back-N (GBN)
  - Selective Repeat (SR)

What does TCP do?

Many of our previous ideas, but some key differences

- Checksum

TCP Header



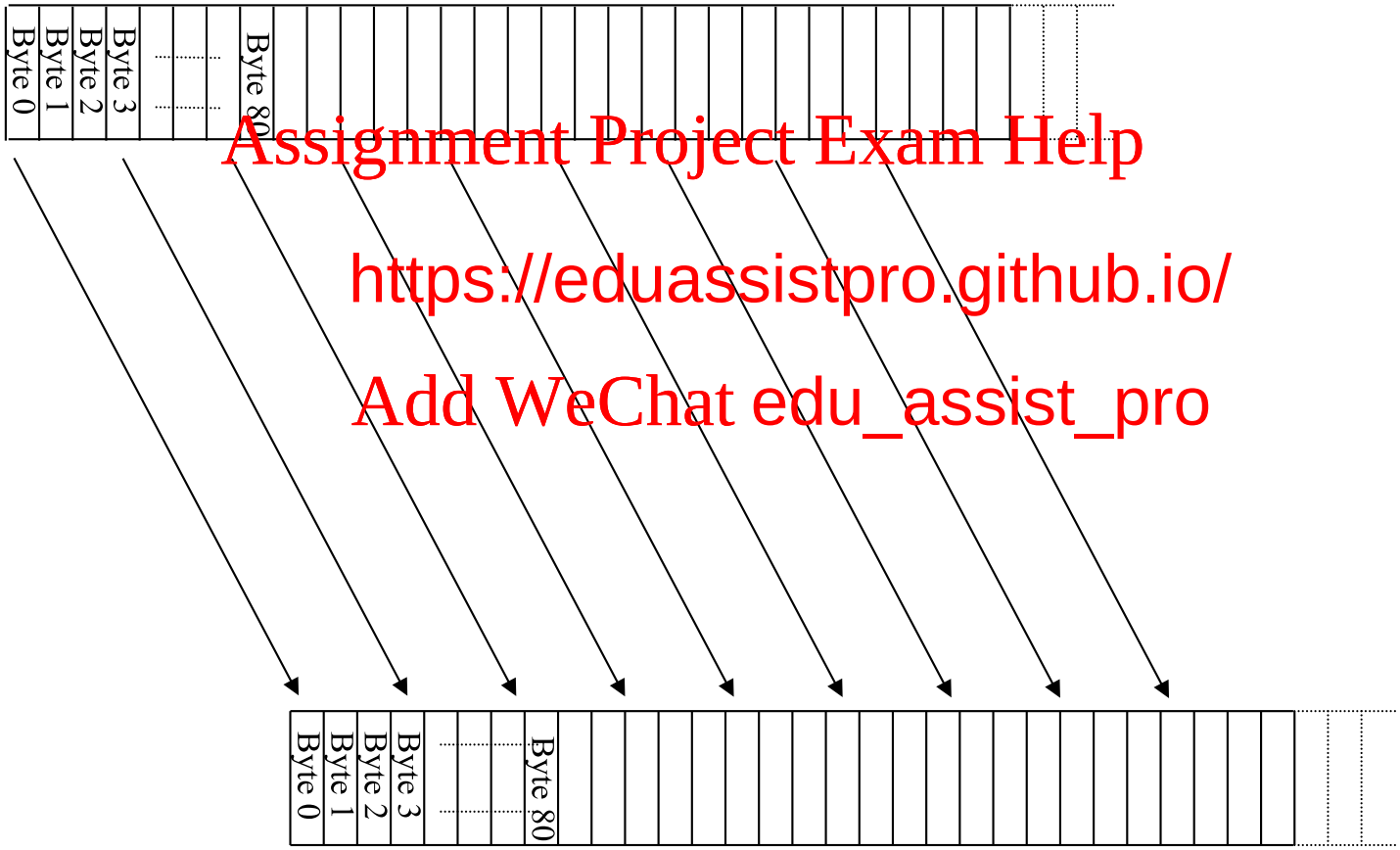
What does TCP do?

Many of our previous ideas, but some key differences

- Checksum
- **Sequence numb**

TCP "Stream of Bytes" Service ..

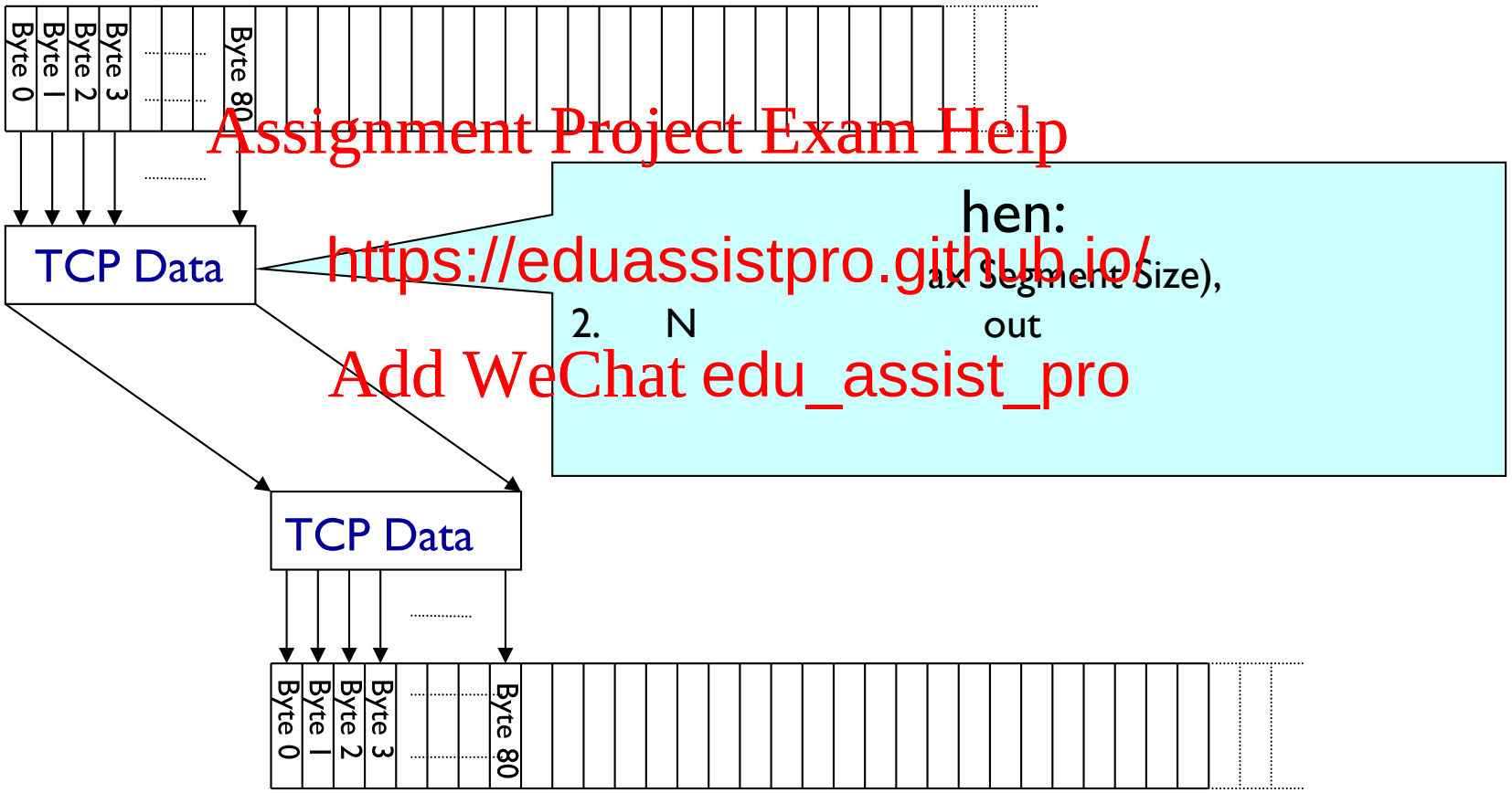
Application @ Host A



Application @ Host B

.. Provided Using TCP "Segments"

Host A

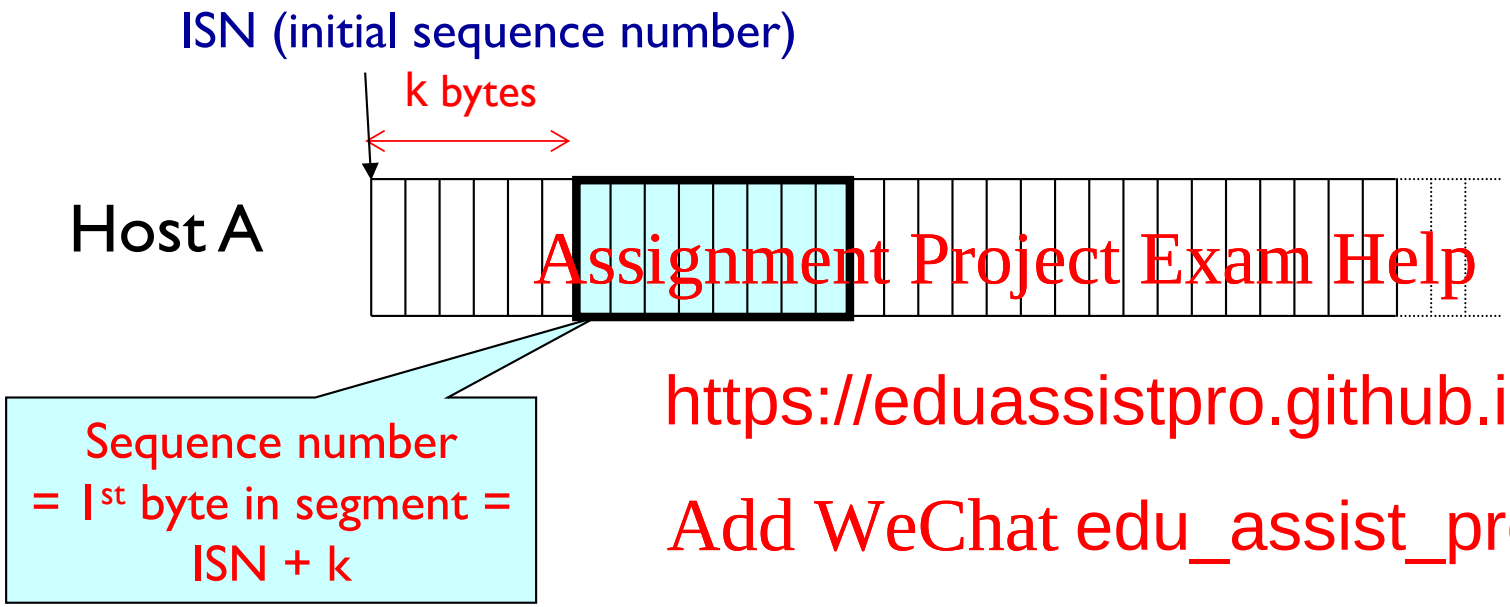


Host B

TCP Segment Size

- IP packet
  - No bigger tha n Unit (MTU)
  - E.g., up to 150
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)

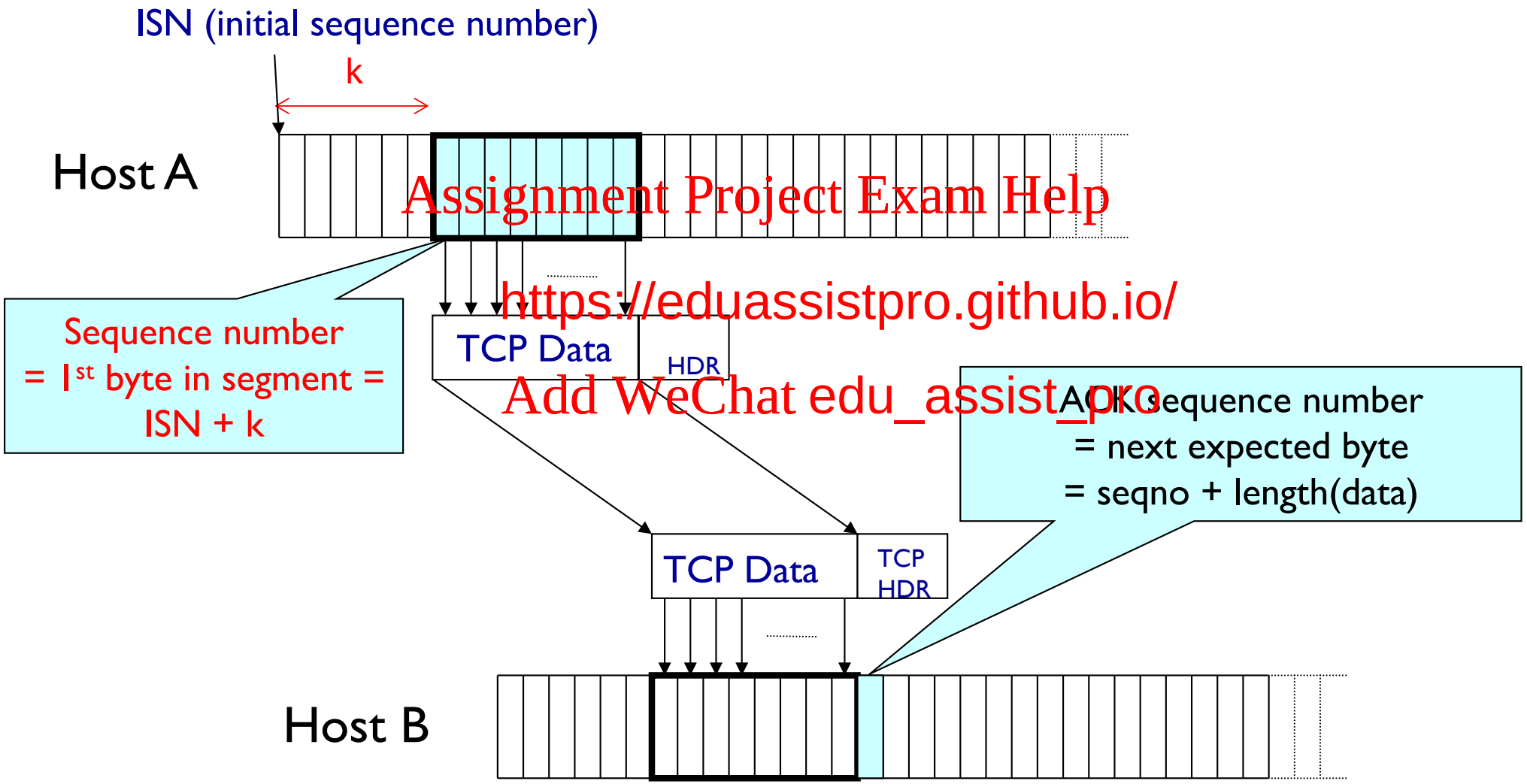
Sequence Numbers



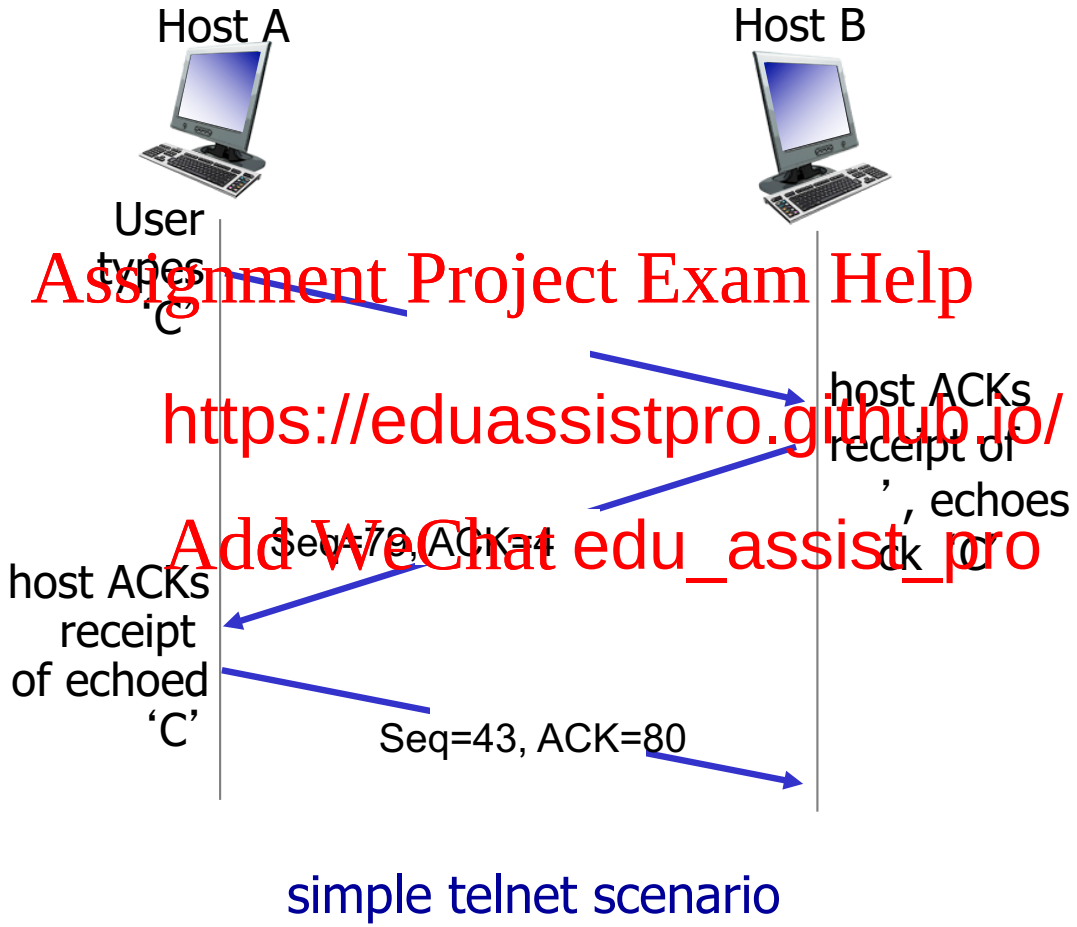
Sequence numbers:

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

Sequence & Ack Numbers



TCP seq. numbers, ACKs



simple telnet scenario

What does TCP do?

Most of our previous tricks, but a few differences

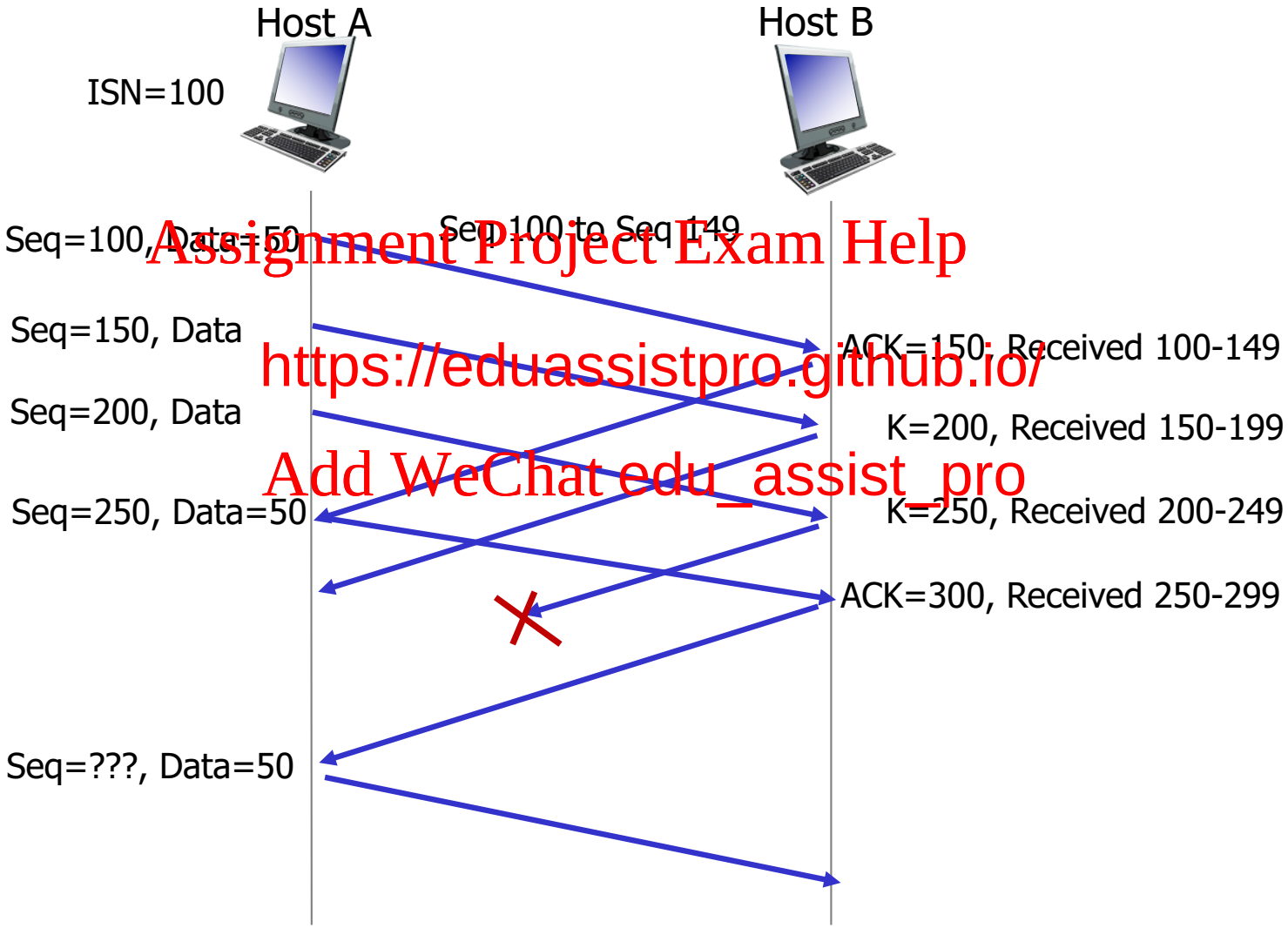
- Checksum
- Sequence numbers are byte offsets
- Receiver sends cu nts (like GBN)

ACKing and Sequence Numbers

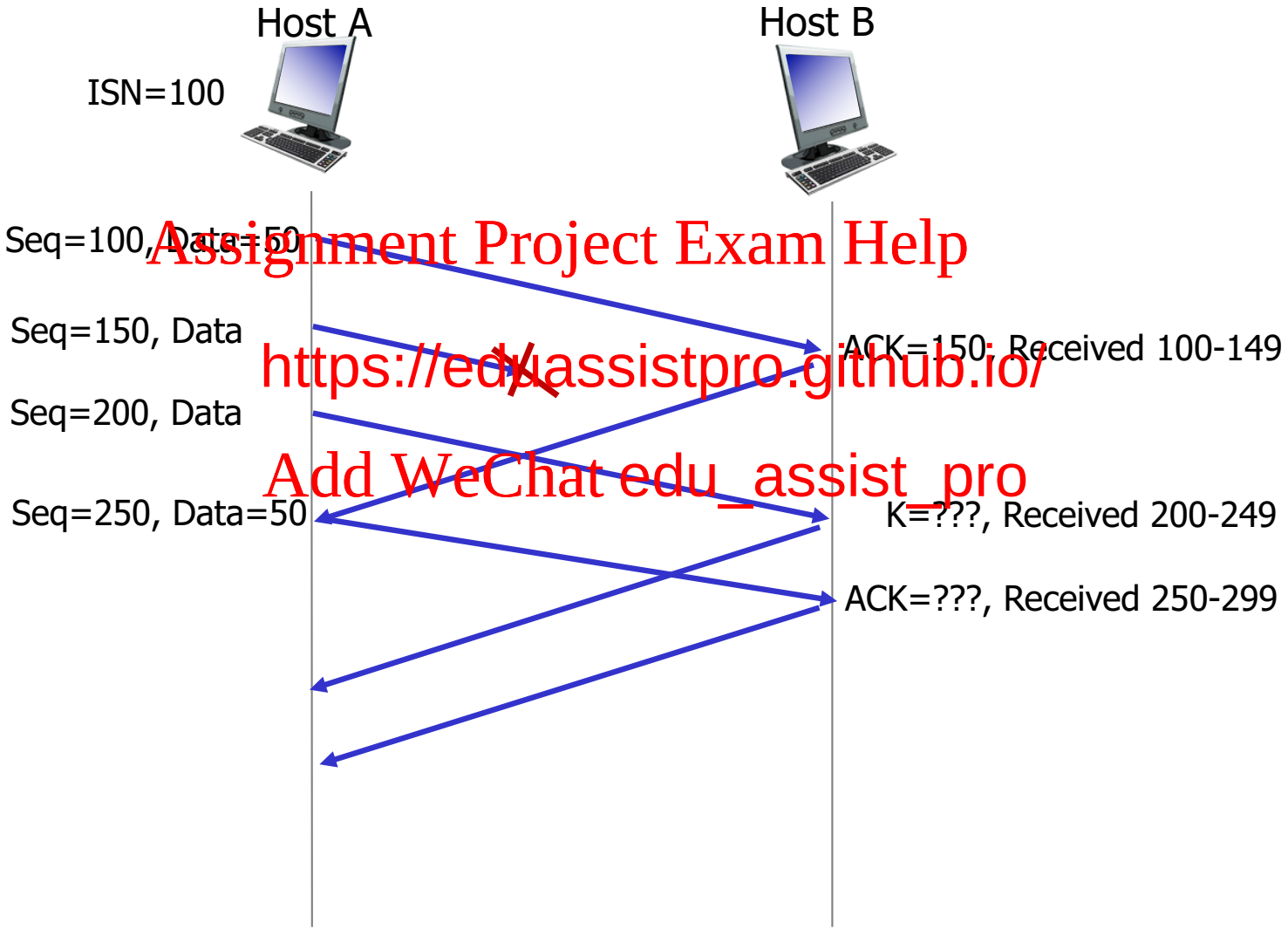
- Sender sends packet
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, ….X+B-1]

- Upon receipt o s an ACK
  - If all data prior
    - ACK acknowledges X+B (bec xt expected byte)
  - If highest in-order byte receive 1) < X
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

TCP seq. numbers, ACKs



TCP seq. numbers, ACKs



Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=
- Seqno of next packet is same a field

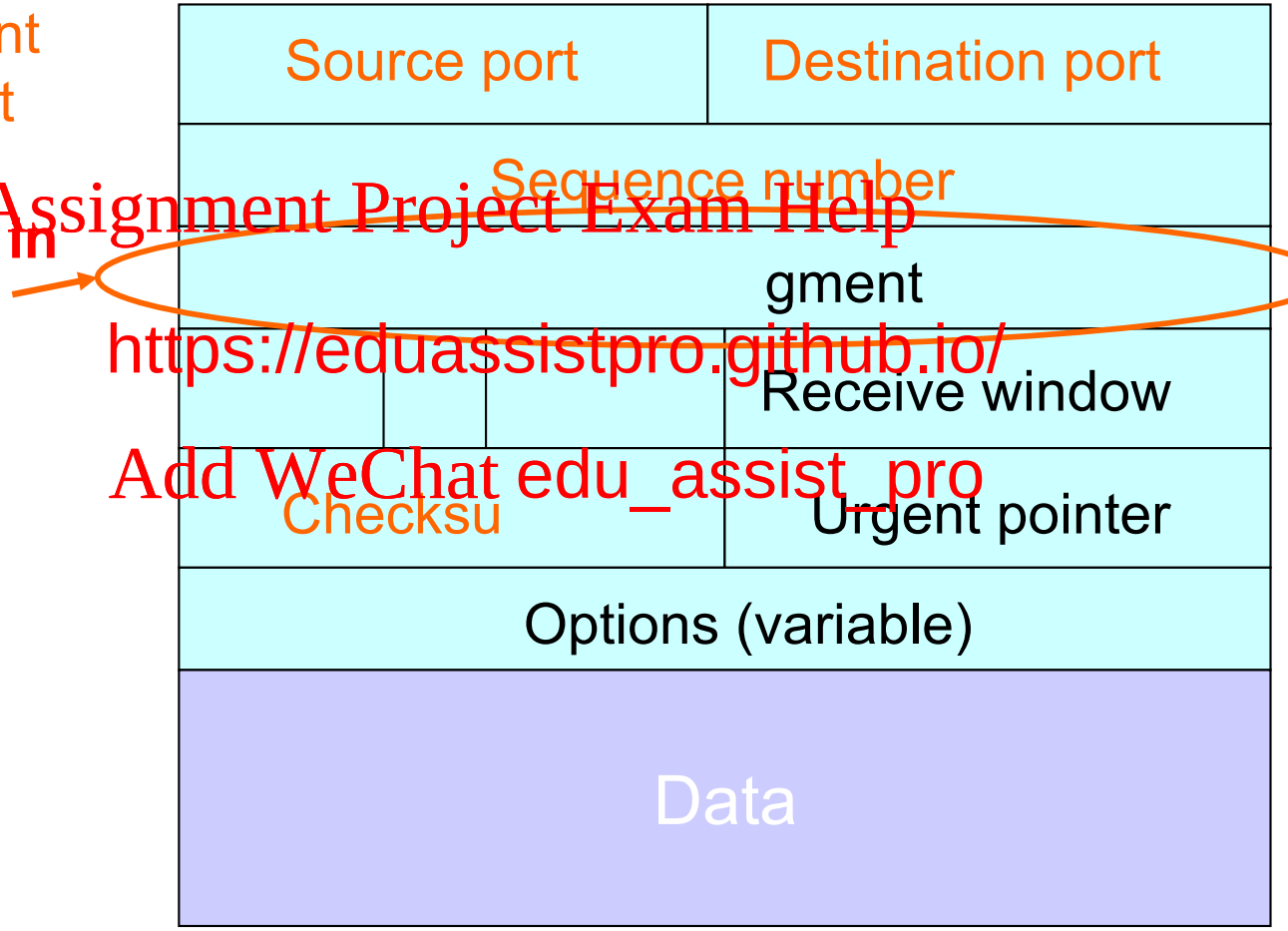
Packet Loss

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: ~~seqno=X+B, length=B~~ LOST

- Sender: seqno=
- Receiver: ACK = X+B

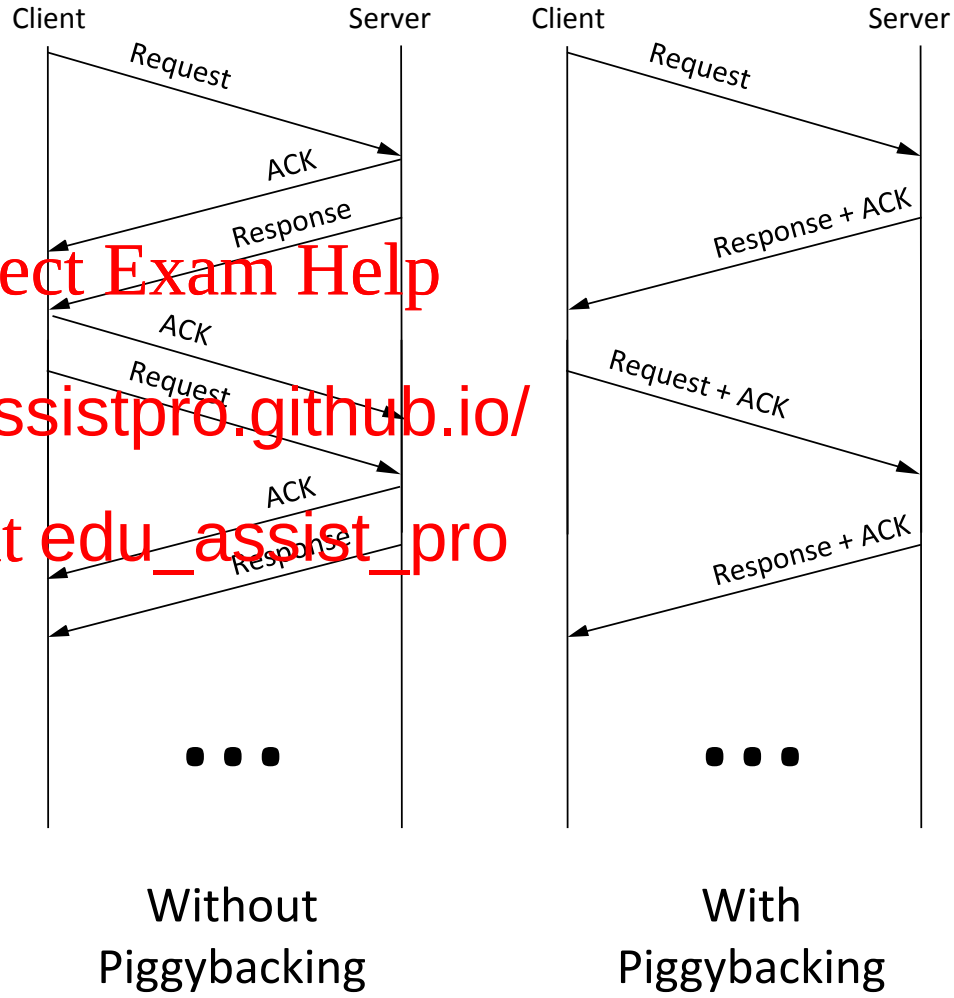
TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order** (*"What Byte is Next"*)



Piggybacking

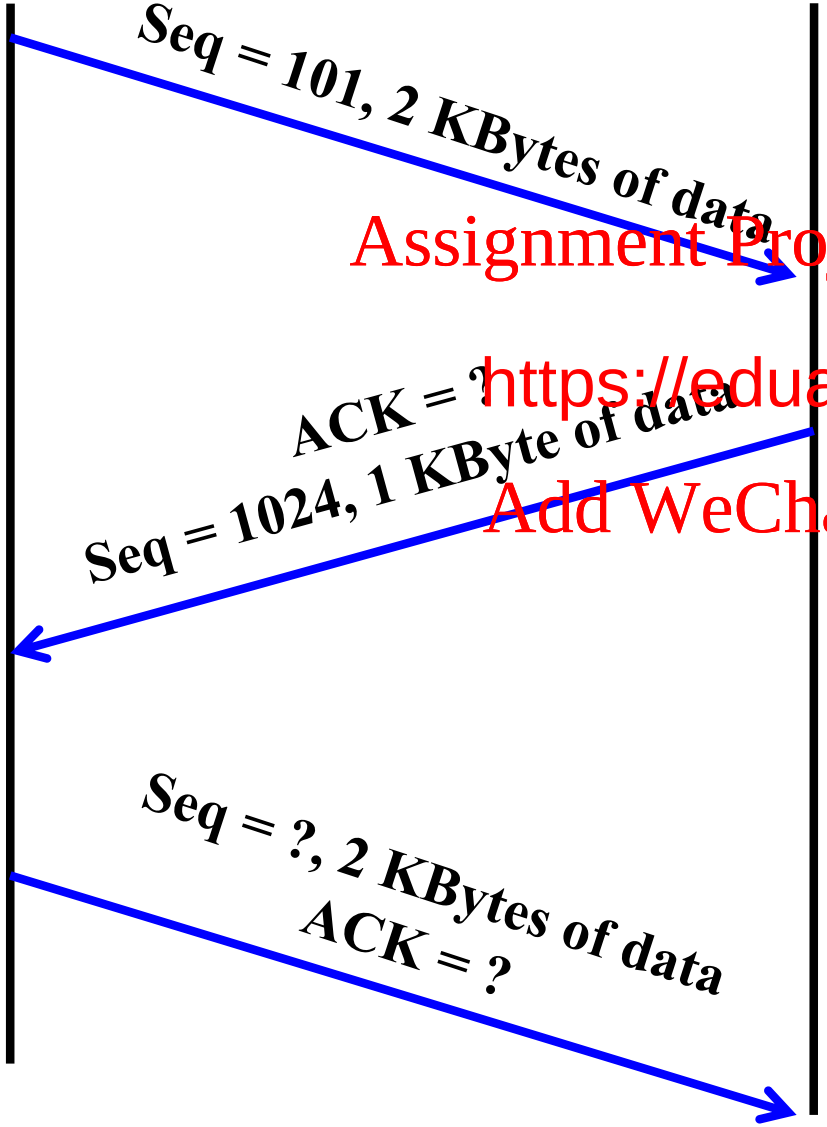
- So far, we've assumed distinct "sender" and "receiver" roles
- In reality, usu sides of a connection send some data



Without Piggybacking

With Piggybacking

Quiz



What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cu nts (like GBN)
- Receivers can buff ets (like SR)

Loss with cumulative ACKs

- Sender sends packets with 100Bytes and sequence numbers:
  - 100, 200, 30 , 800, 900, …
- Assume the fifth packet (500) is lost, but no others
- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500,…

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cu nts (like GBN)
- Receivers do not d ckets (like SR)
- Sender maintains a single retrans r (like GBN) and retransmits on timeout

TCP round trip time, timeout

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT vari
- *too short:* pre timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss and connection has lower throughput

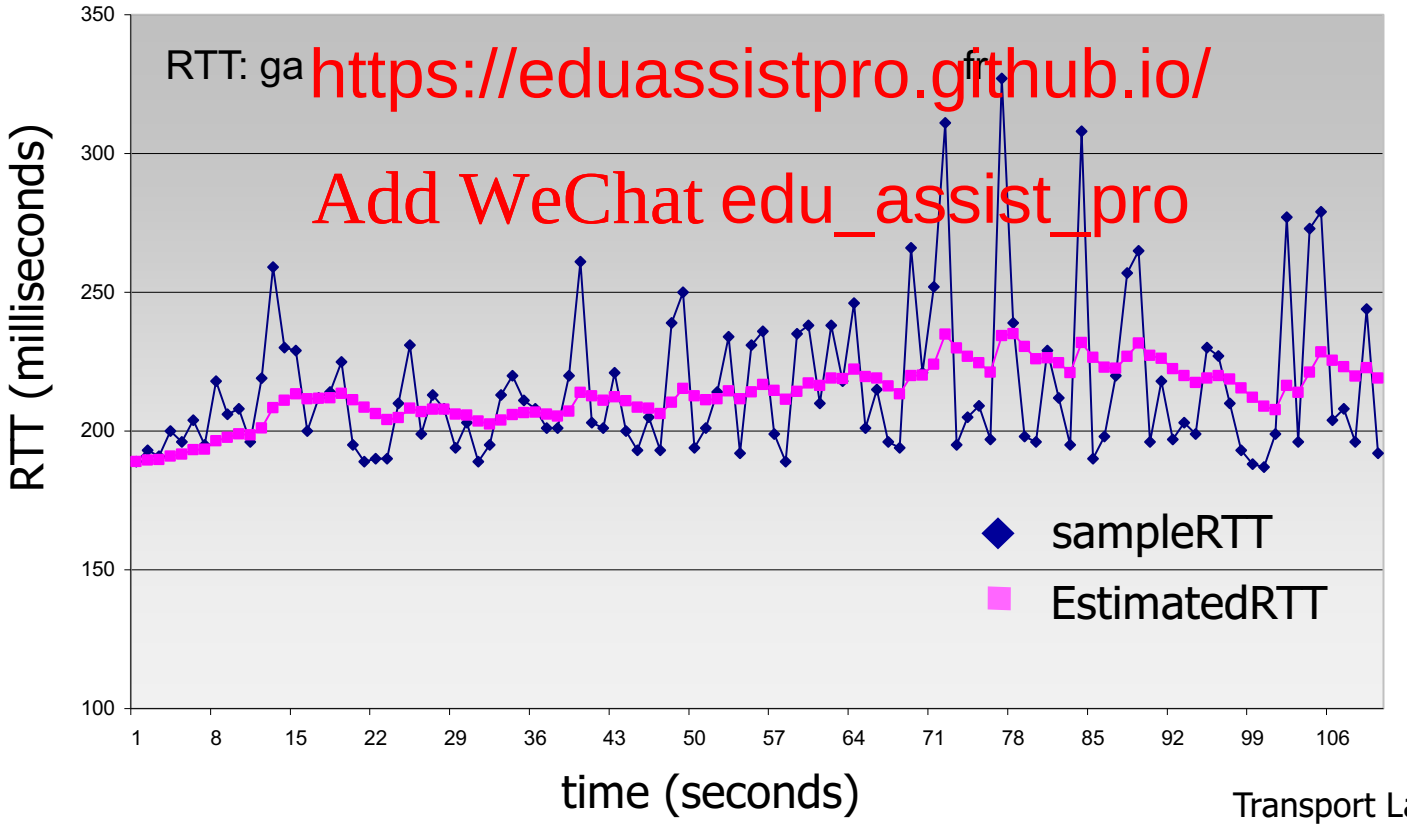
Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK
  - re retransmissions
  - **RTT** will vary, want 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
- 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 = (1                    )*DevRTT +
                 |SampleRTT-EstimatedRTT|
          (typically,                  )
```

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

estimated RTT

"safety margin"

Practice Problem:
http://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198700.cw/index.html

TCP round trip time, timeout

(EstimatedRTT+4*DevRTT)

RTT
EstimatedRTT

DevRTT

TimeoutInterval = EstimatedRTT + 4*DevRTT

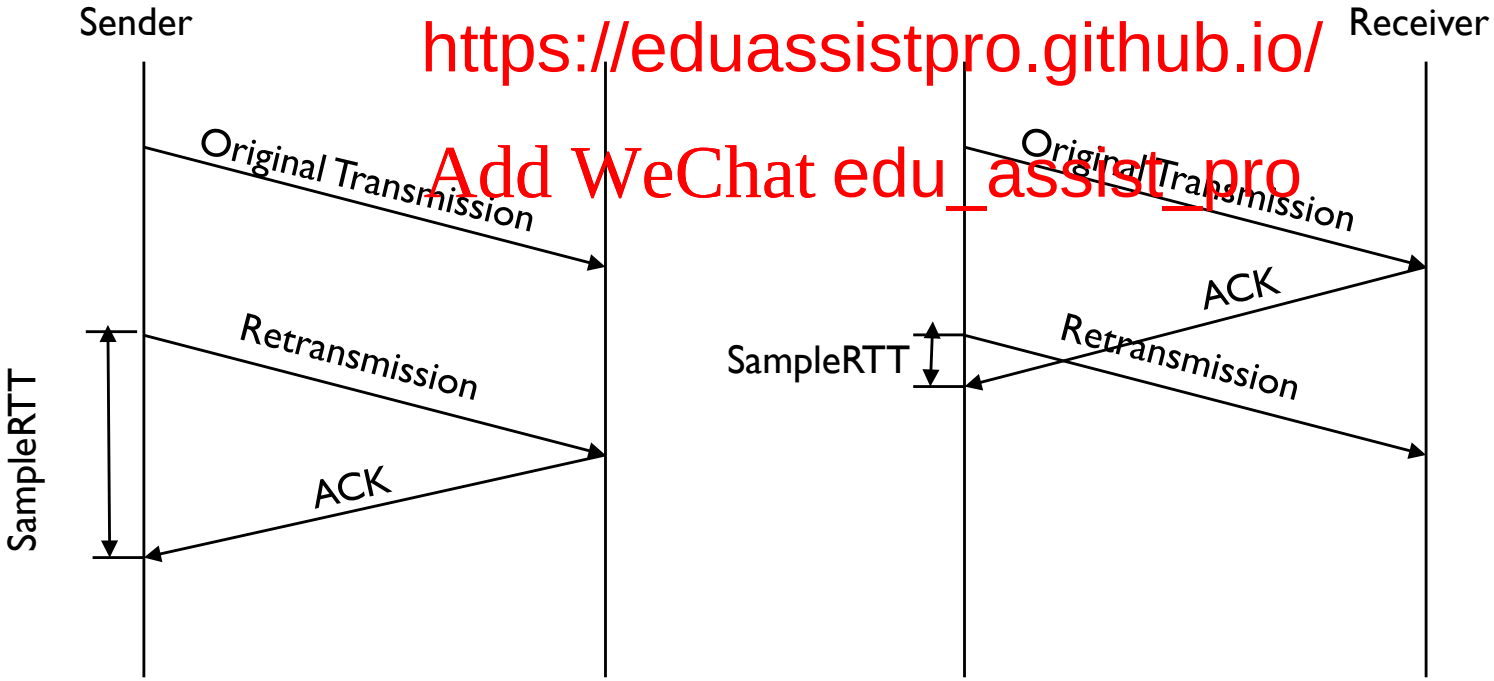
estimated RTT

"safety margin"

Figure: Credits Prof David Wetherall UoW

Why exclude retransmissions in RTT computation?

- How do we differentiate between the real ACK, and ACK of the retransmitted packet?



TCP sender events:

PUTTING IT TOGETHER

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of firs byte in segm
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: `TimeOutInterval`

timeout:

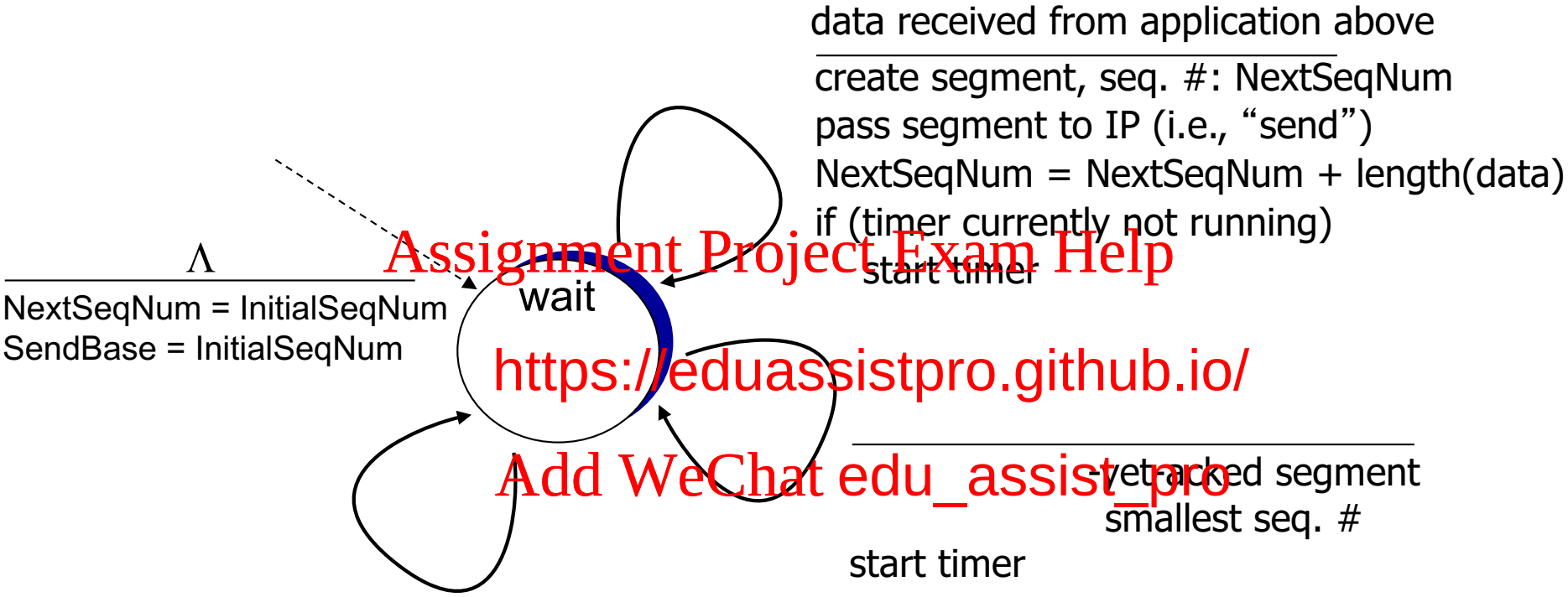
- retransmit segment that caused timeout
- restart timer

knowle
y unacked

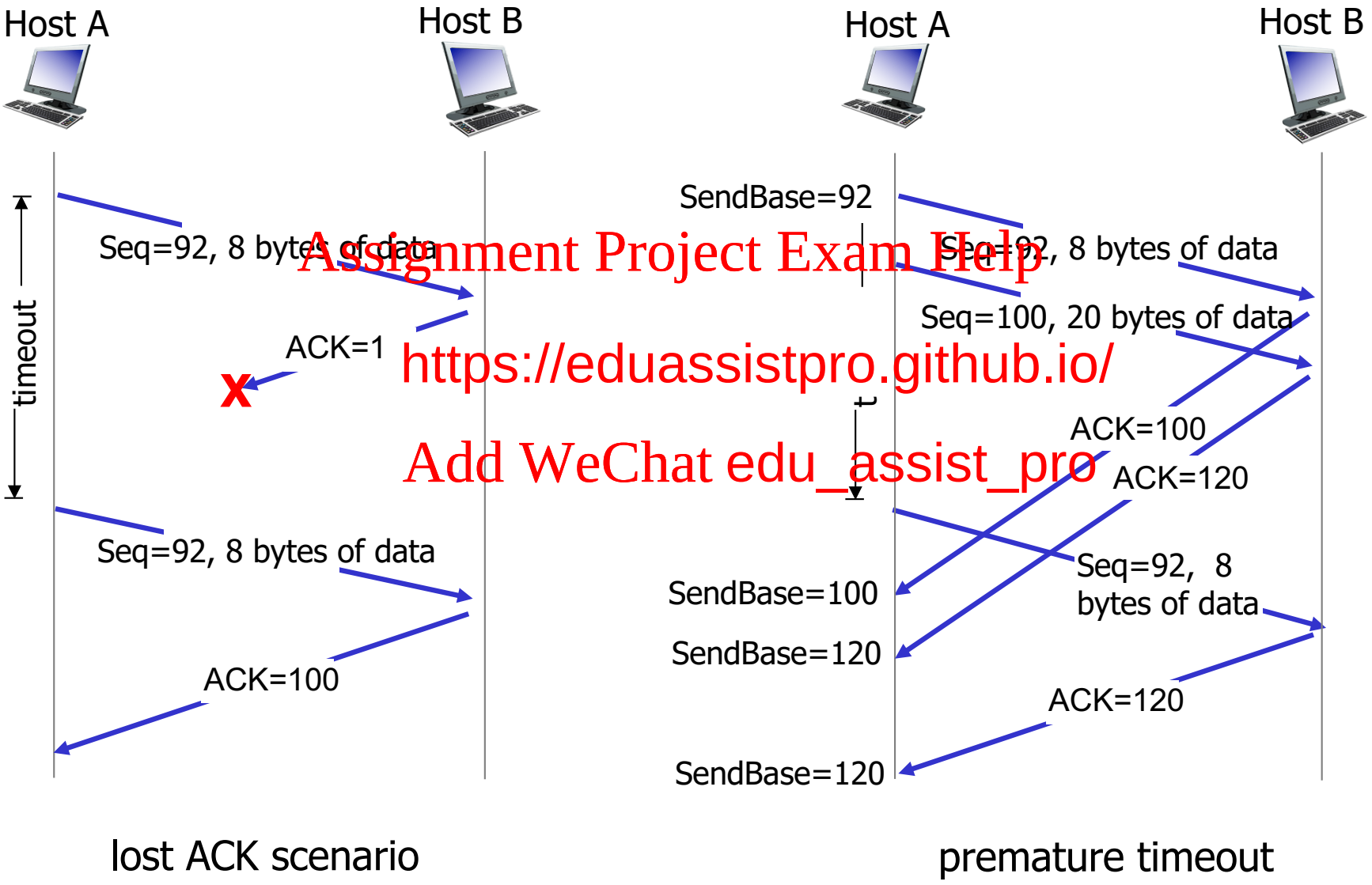
- update what is known to be ACKed
- start timer if there are still unacked segments

TCP sender (simplified)

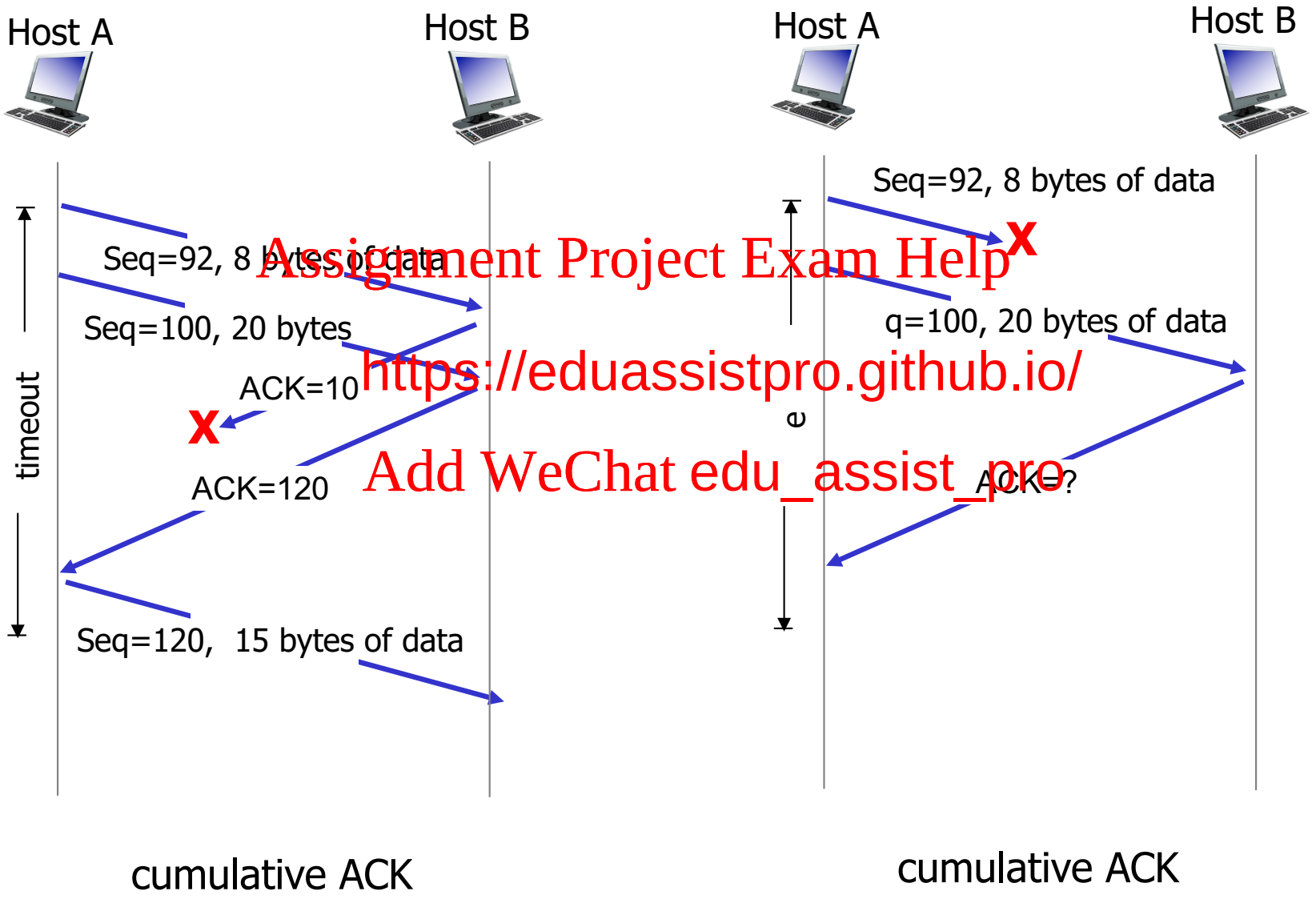
PUTTING IT TOGETHER



TCP: retransmission scenarios



TCP: retransmission scenarios



Quiz

Suppose host A sends two TCP segments back to back to Host B over a TCP connection. The first segment has sequence number 90; the second has sequence number 110.

- Q1: How much da?

- Q2: Suppose that the first segment is lost but the second segment arrives at B. In the acknowledgement that Host B sends to Host A, what will be the acknowledgement number?

TCP ACK generation [RFC 1122, RFC 2581]

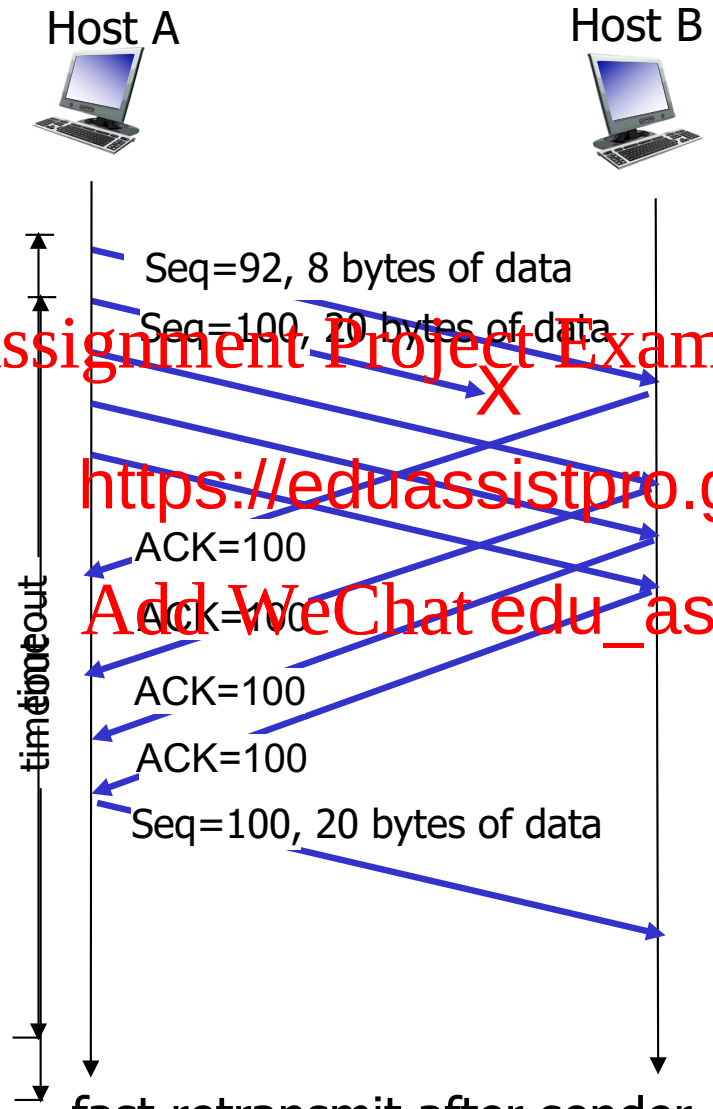
event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order seg expected seq #. One other segment has ACK pending	end single cumulative A oth 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

What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cu nts (like GBN)
- Receivers may not ackets (like S
- Sender maintains a single retrans r (like GBN) and retransmits on timeout
- Introduces fast retransmit: optimisation that uses duplicate ACKs to trigger early retransmission

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- "Duplicate ACKs" are a sign of an isolate
  - The lack of ACKs means that pack been delivered
  - Stream of ACKs means some packets are being delivered
  - Could trigger resend on receiving "k" duplicate ACKs (TCP uses k = 3)

TCP fast retransmit

if sender receives 3 duplicate ACKs for ata duplicate ACKs"), acked with smallest seq #

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

What does TCP do?

Most of our previous ideas, but some key differences

- Checksum
- Sequence numbers
- Receiver sends cu nts (like GBN)
- Receivers do not drop out-of-seq ets (like SR)
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission

Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexi

3.3 connectionle transport: UDP

3.4 principles of reliable data transfer

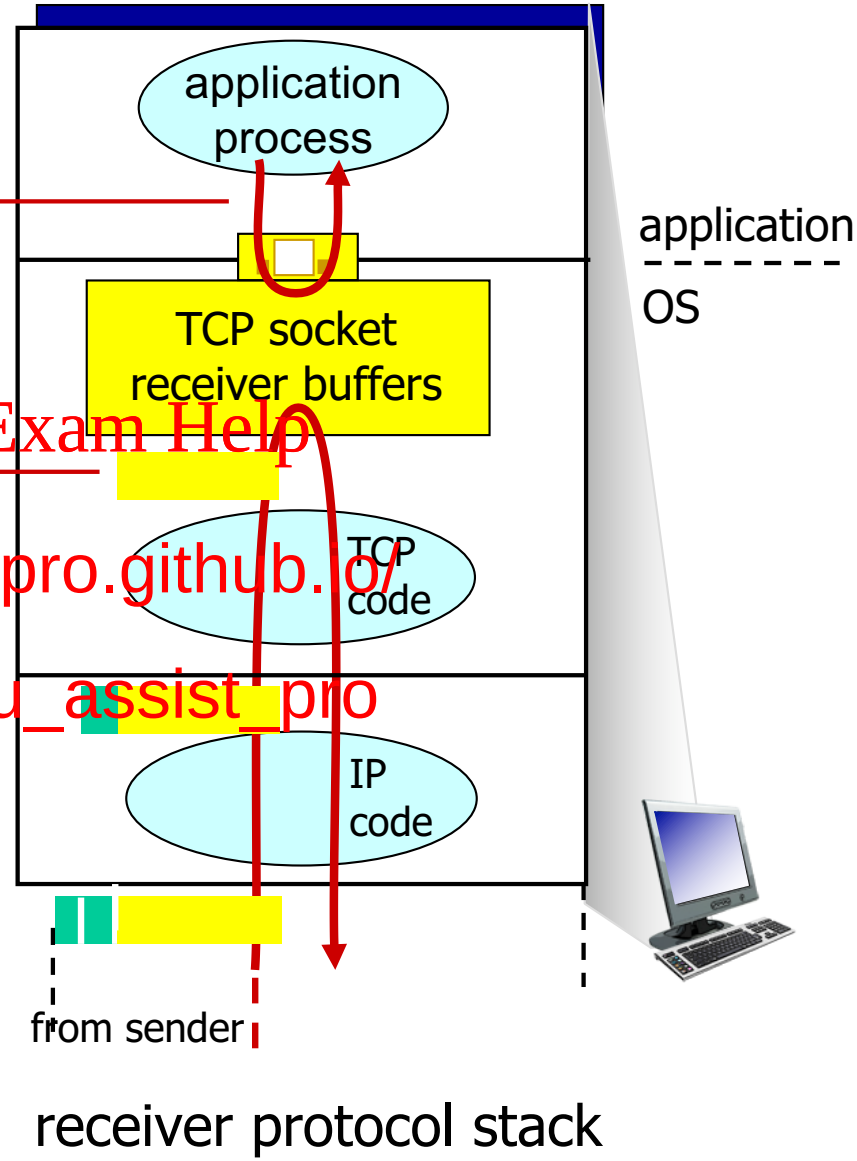
3.5 connection-oriented transport: TCP

- segment structure
- able data transfer
- ontrol
- ction management

3. es of congestion control

3.7 TCP congestion control

TCP flow control

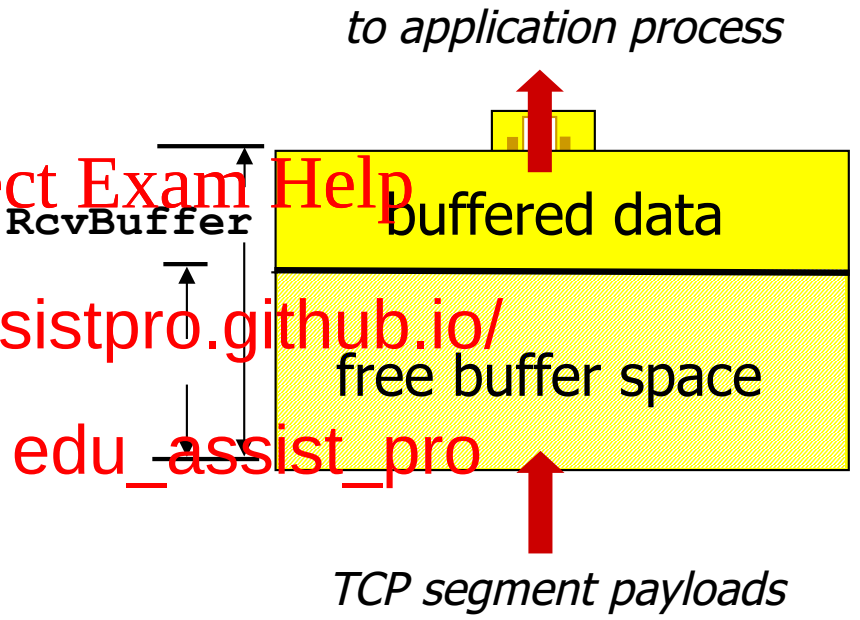


flow control

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

TCP flow control

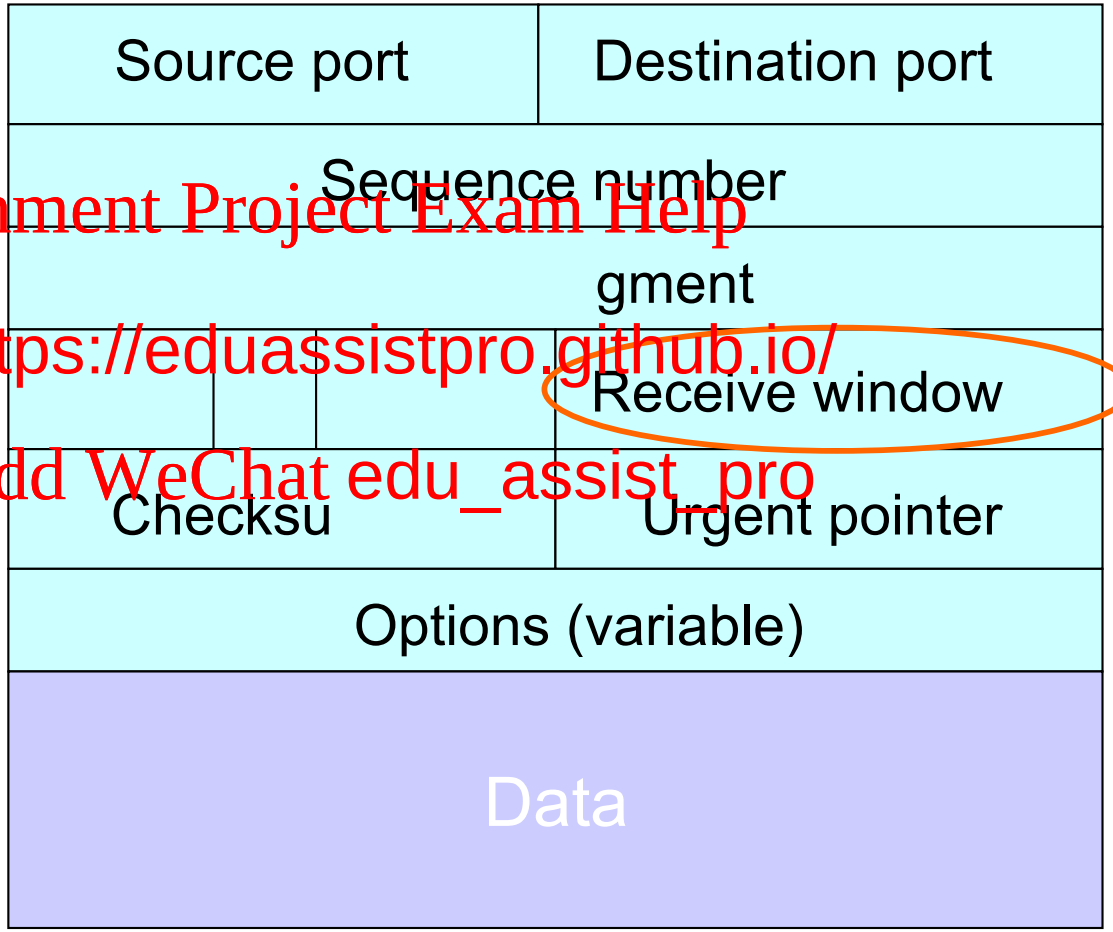
- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** si socket options (is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow



receiver-side buffering

http://media.pearsoncmg.com/aw/aw_kurose_network_4/applets/flow/FlowControl.htm

TCP Header



Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexi

3.3 connectionle transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure

able data transfer

control

ction management

3. es of congestion control

3.7 TCP congestion control