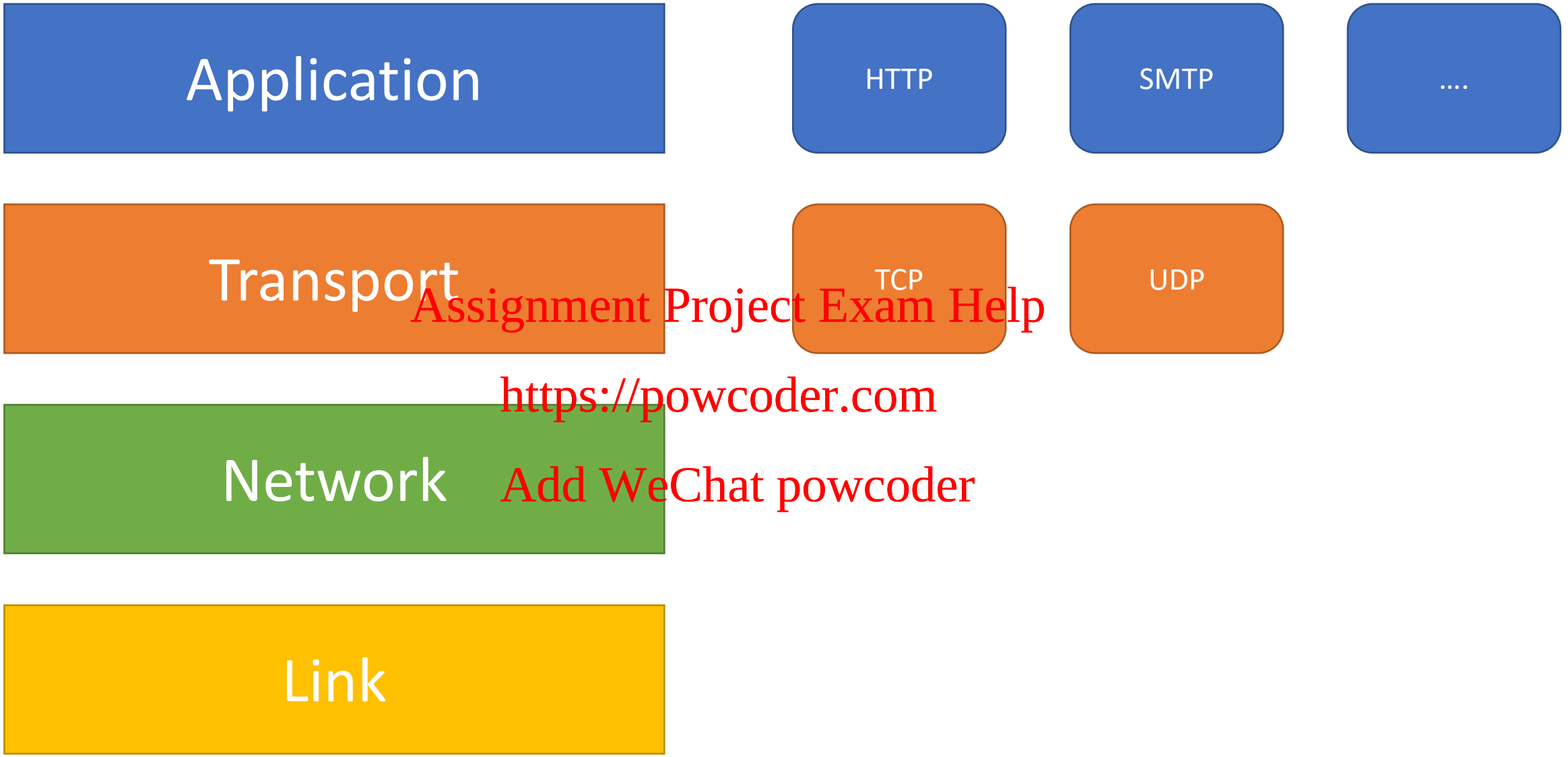
CS Discussion Week 4: The Transport Layer

Questions?

- From this week or about the HW/Project

The Transport Layer

- Provides a *logical* communication channel between processes on different hosts (“end-to-end”)
- Sender: Splits messages from application layer into “segments”
- Receiver: Merges segments back into messages for application
- In general, only two protocols used on Internet: TCP and UDP



UDP

- User Datagram Protocol
- The simpler of the two, provides less “features” than TCP
- Provides the bare minimum needed to get packets to the receiver
- Packets *not* guaranteed to be received by remote app in order
- Packets *not* guaranteed to get to remote host at all
- No connection → just send data

TCP

- Transmission Control Protocol
- Data guaranteed to be delivered and delivered *in order*
- Provides “congestion control” to allow it to “play nice” with other communication streams on the network
- Need to establish connection with a specific remote host

Should I use TCP or UDP?

- Still no guarantee of delay or throughput in either TCP or UDP
- Discuss: What applications would benefit most from each protocol?

Which applications use which protocols?

- TCP
  - HTTP (older versions)
  - SMTP, IMAP
  - FTP (file transfer)
  - SSH (remote shell)
- UDP
  - DNS (can also use TCP), DHCP (network configuration)
  - Streaming audio and video (including voice-over-IP/“VoIP”)
  - HTTP/3 and QUIC
    - Use congestion control and reliability at application layer (if needed)!

Differentiating Applications

- How do we tell which application we are communicating with?
- Identify with a numeric port!
  - Range [1, 2^{16}-1]
- Servers generally use a standard port
  - E.g., 80 for HTTP and 53 for DNS
- Clients generally use a random port
  - Usually very high up in the port ranges
- Endpoints use these ports to “multiplex” b/w multiple applications

Port Ranges

1-512	• standard services (see `/etc/services`) • super-user only
513-1023	• registered and controlled, also used for identity verification • super-user only
1024-49151	• registered services/ephemeral ports
49152-65535	• private/ephemeral ports

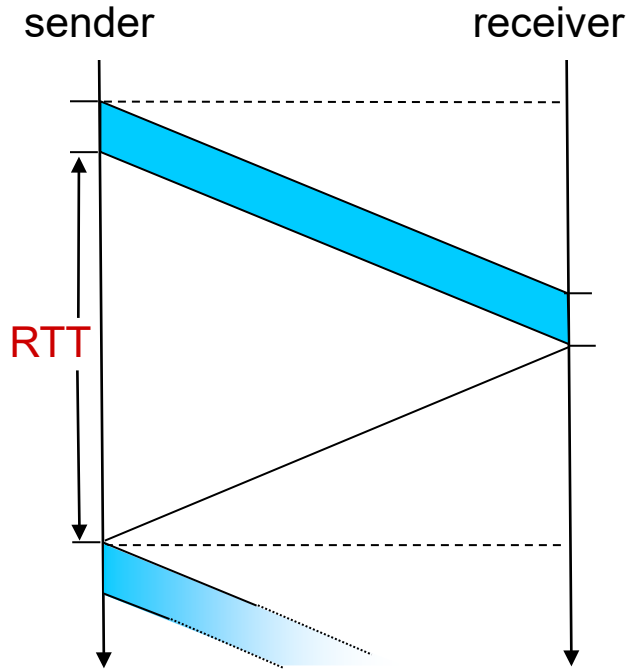
From slide by Seungbae Kim, UCLA

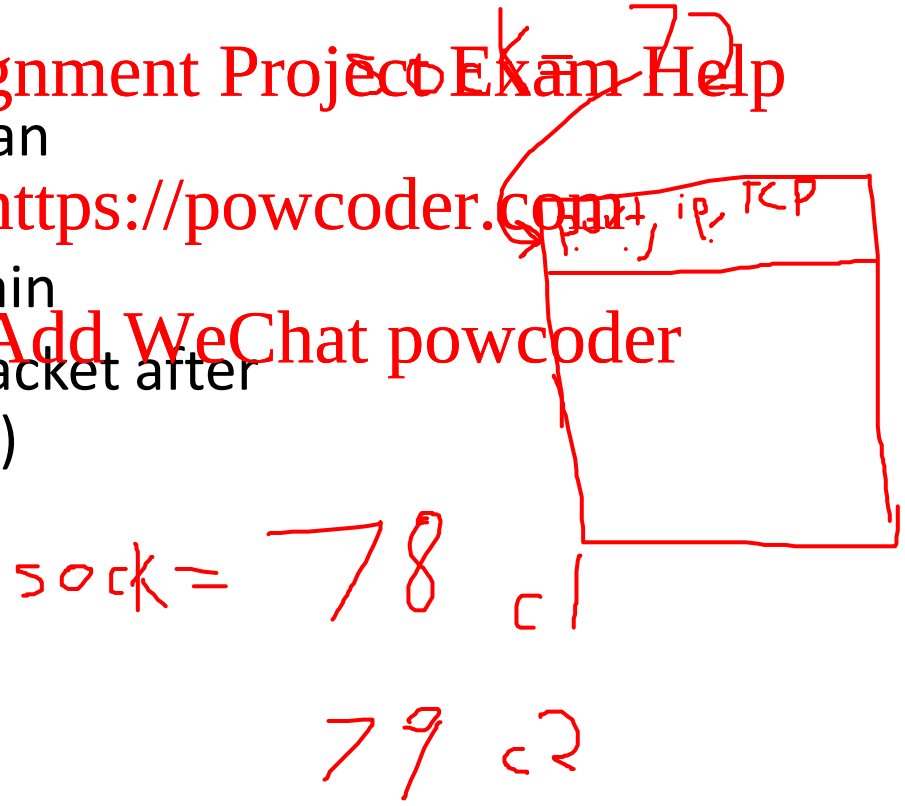
“5-Tuple”

- Each communication instance between two hosts can be identified with a “5-tuple”
- (Transport protocol, Src IP, Src port, Dst IP, Dst port)
- E.g., (UDP, 192.168.0.7, 57392, 8.8.8.8, 53)
  - This could be a DNS query from my local machine to a public DNS server!
- Related “4-tuple” is used to describe TCP connections (TCP is implicitly assumed as “Transport protocol”)

How do we send data reliably?

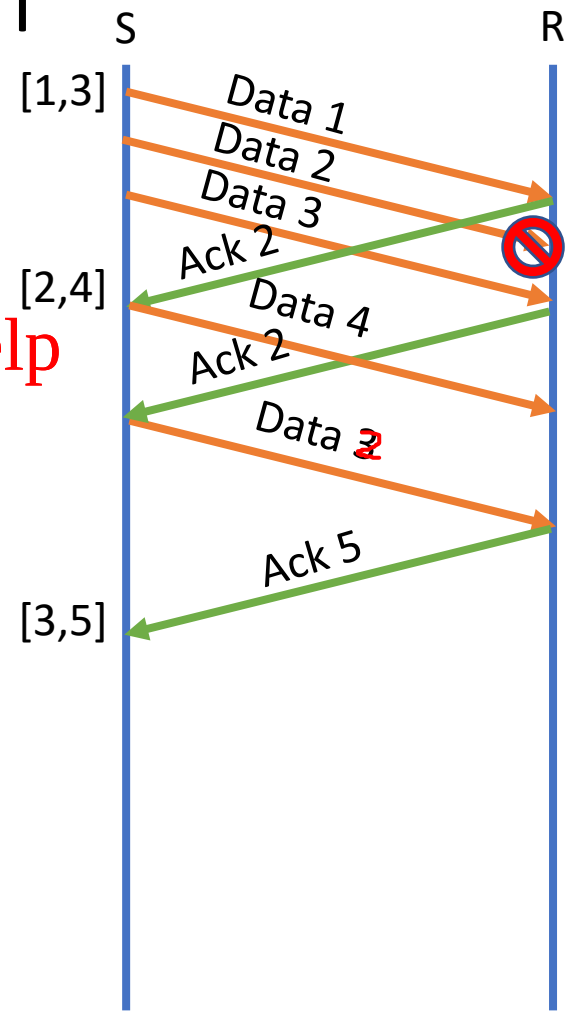
- Simple version: Stop and Wait
- As simple as possible:
  - Send a packet, wait for an acknowledgement
  - If you timeout, send again
  - Otherwise, send next packet after acknowledgement (ACK)





How do we send data reliably? II

- Sliding window!
  - Both Go-Back-N and Selective repeat are 'versions' of this technique
- Have only n bytes "in flight" at a given moment in time!
- "Slide" window when the first packet in the current window is acknowledged



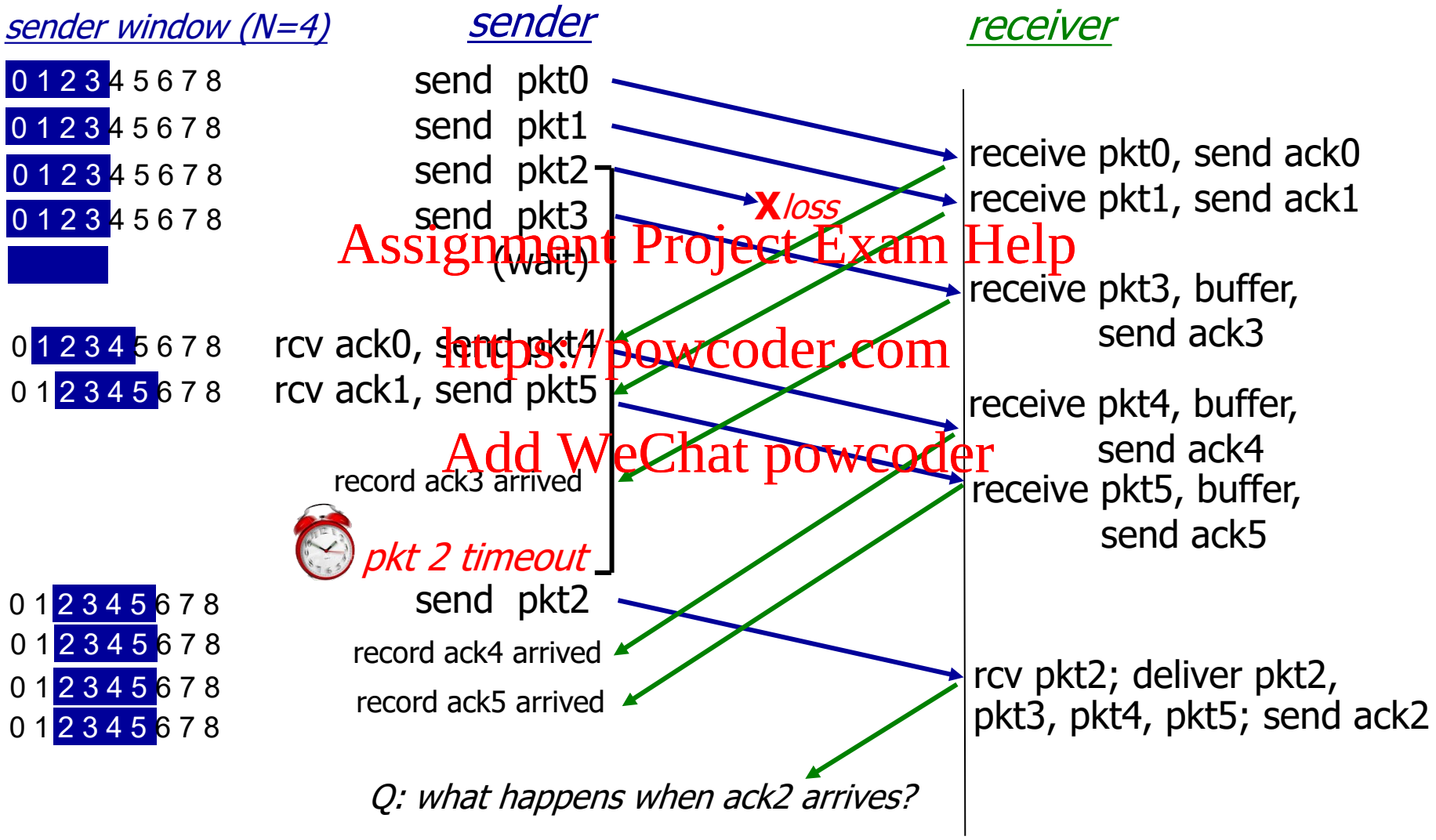
Go-Back-N

- Allow N packets to be ‘in flight’ at any given time, but send a NACK if any of them are not in order.
- So receiving 1,2,3,4,5 in a row is fine, but 1,3,4,2,5 is **not.**
- Ack (n) acknowledges all packets up t o n, because of this in-order property.

Selective Repeat

- Instead of requiring that all packets received be in order, keep a buffer of packets and 'slot them in' as necessary.
- Because the in-order property is not preserved, must ack each packet individually.
- So 1,2,3,4,5 is fine, as is 1,3,4,2,5 (which results in the receiver sending ACK(1), ACK(3), ACK(4), etc)
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - *N* consecutive seq #'s
  - limits seq #s of sent, unACKed pkts

Selective repeat in action



Slide borrowed from Giovanni Pau CS 118 Lecture Slides

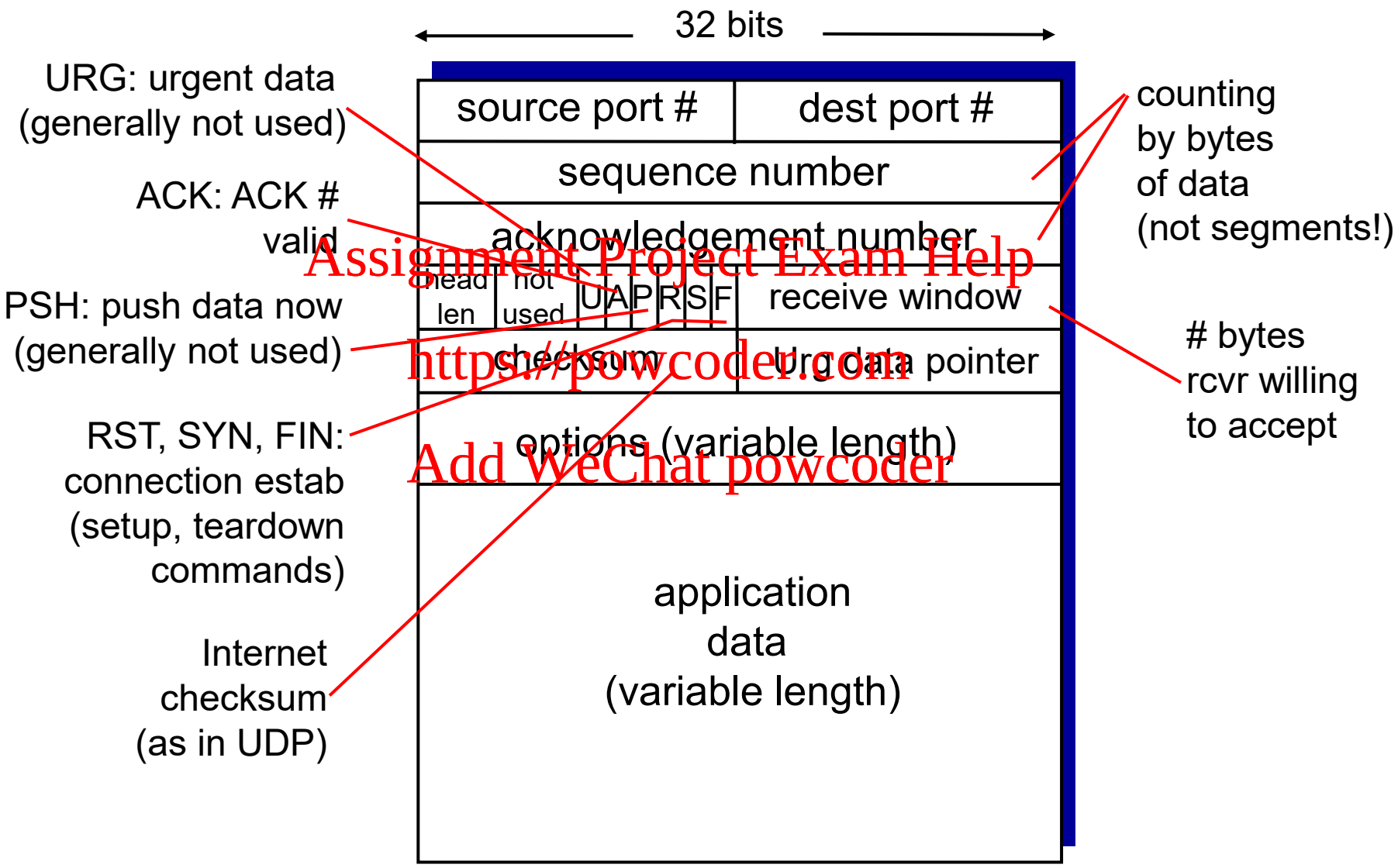
Discussion Question

- Why would someone ever use Go-Back-N vs Selective Repeat?

TCP: An In-Depth Look

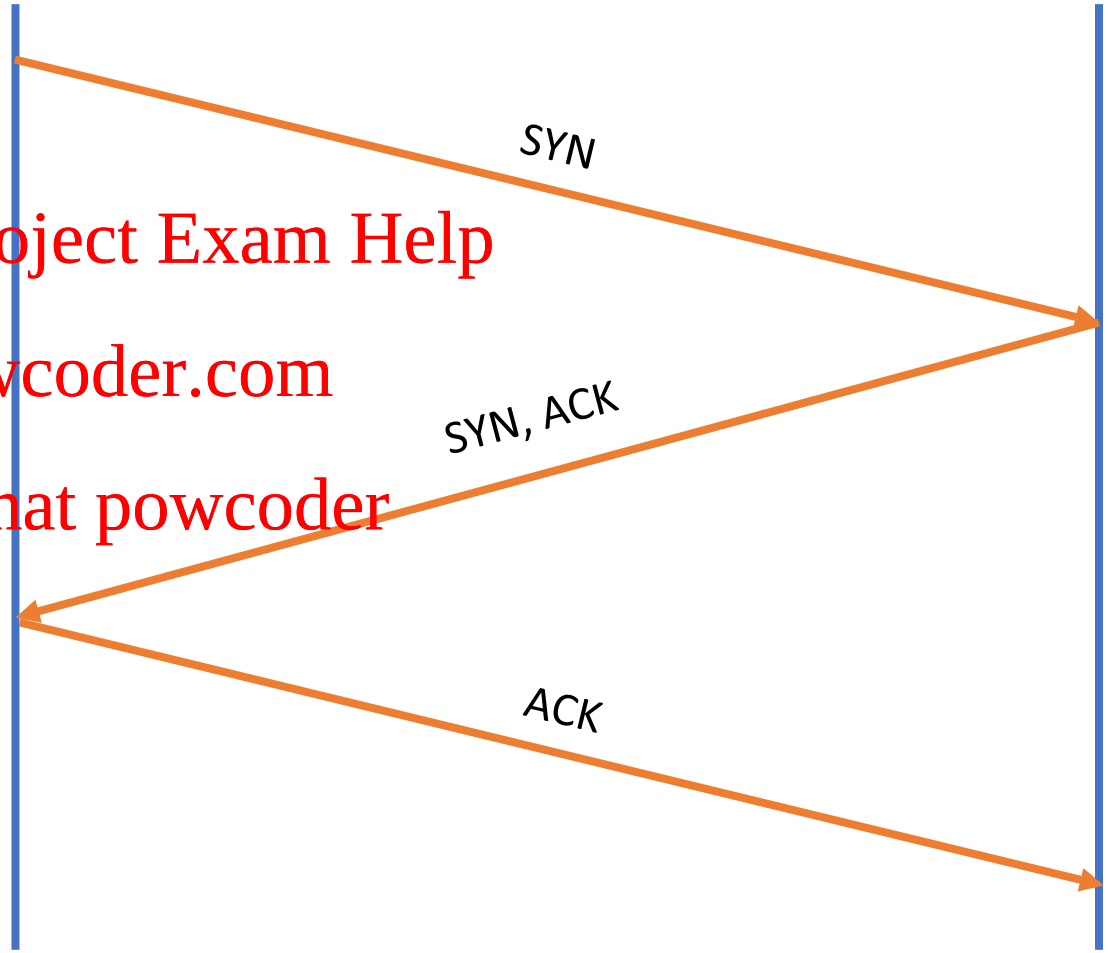
- We'll take a more careful look at TCP, including some actual details of the protocol.

TCP: In-Depth Overview I: Protocol Structure

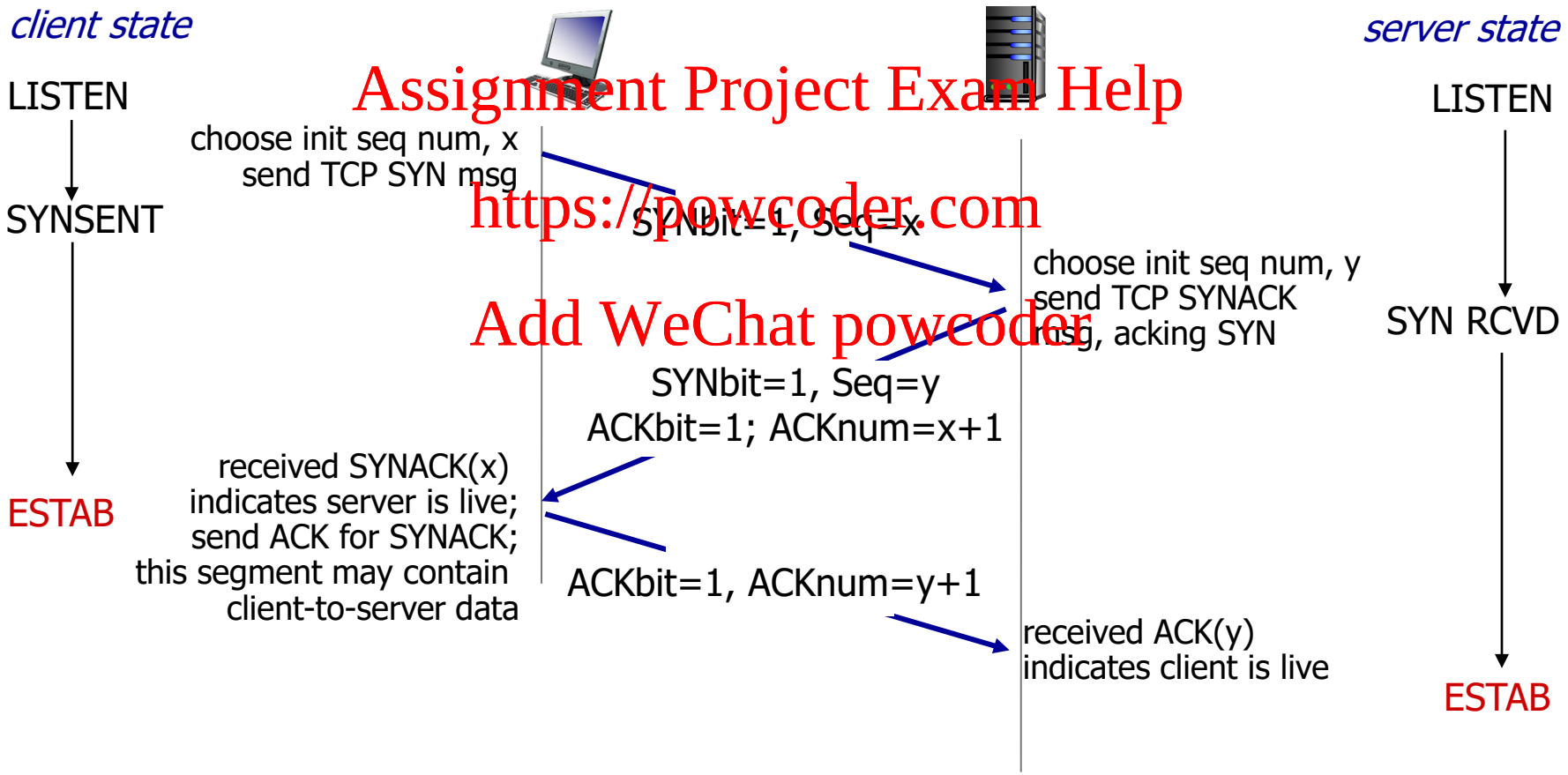


TCP: Establishing a Connection

- "Three-way handshake"
- initialize TCP control variables:
- Initial seq. # used in each direction
- Flow control window size
- **Three way handshake**
  - 1: client host sends TCP SYN segment to server
  - specifies initial seq # Does *not* carry data
  - 2: server receives SYN, replies with <**SYN_ACK**, **SYN>** segment
  - 3: client sends SYN_ACK
    - may carry data in this segment



Establishing a Connection II

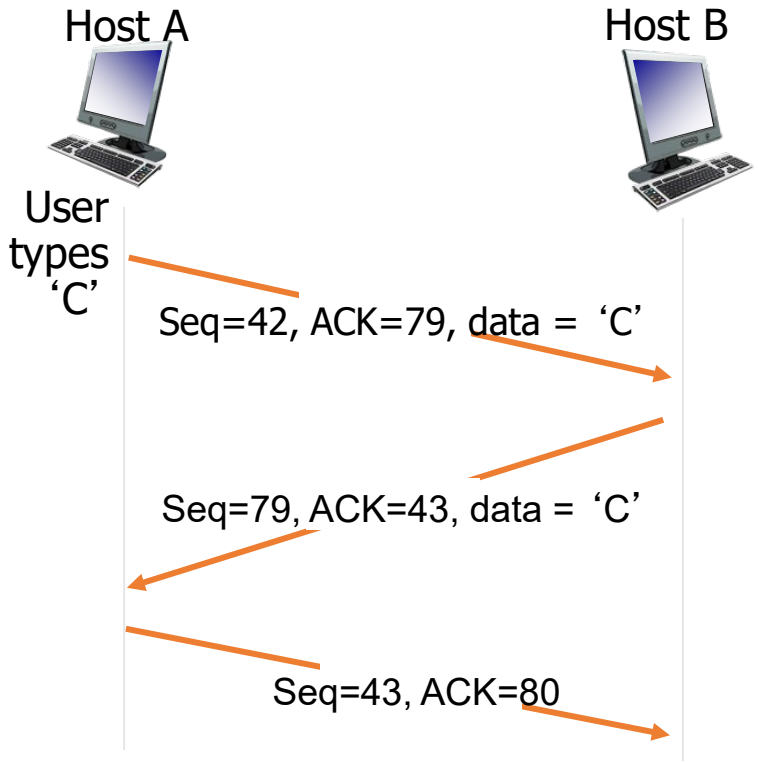


TCP: Closing a Connection

- Either end can initiate the close of ***its end*** of the connection any time
- 1: one end (A) sends TCP FIN control segment to the other
- 2: the other end (B) receives FIN, replies with FIN_ACK; when it's ready to close too, send FIN
- 3: A receives FIN, replies with FIN-ACK.
- 4: B receives FIN_ACK, close connection
- A Enters "timed wait", waits for 2xMSL (maximum segment lifetime) before deleting the connection state

TCP Sequence Numbers

- Byte stream ‘number’ of first byte in segments data
- Related note: Acknowledgements have a seq # of next byte expected from other side

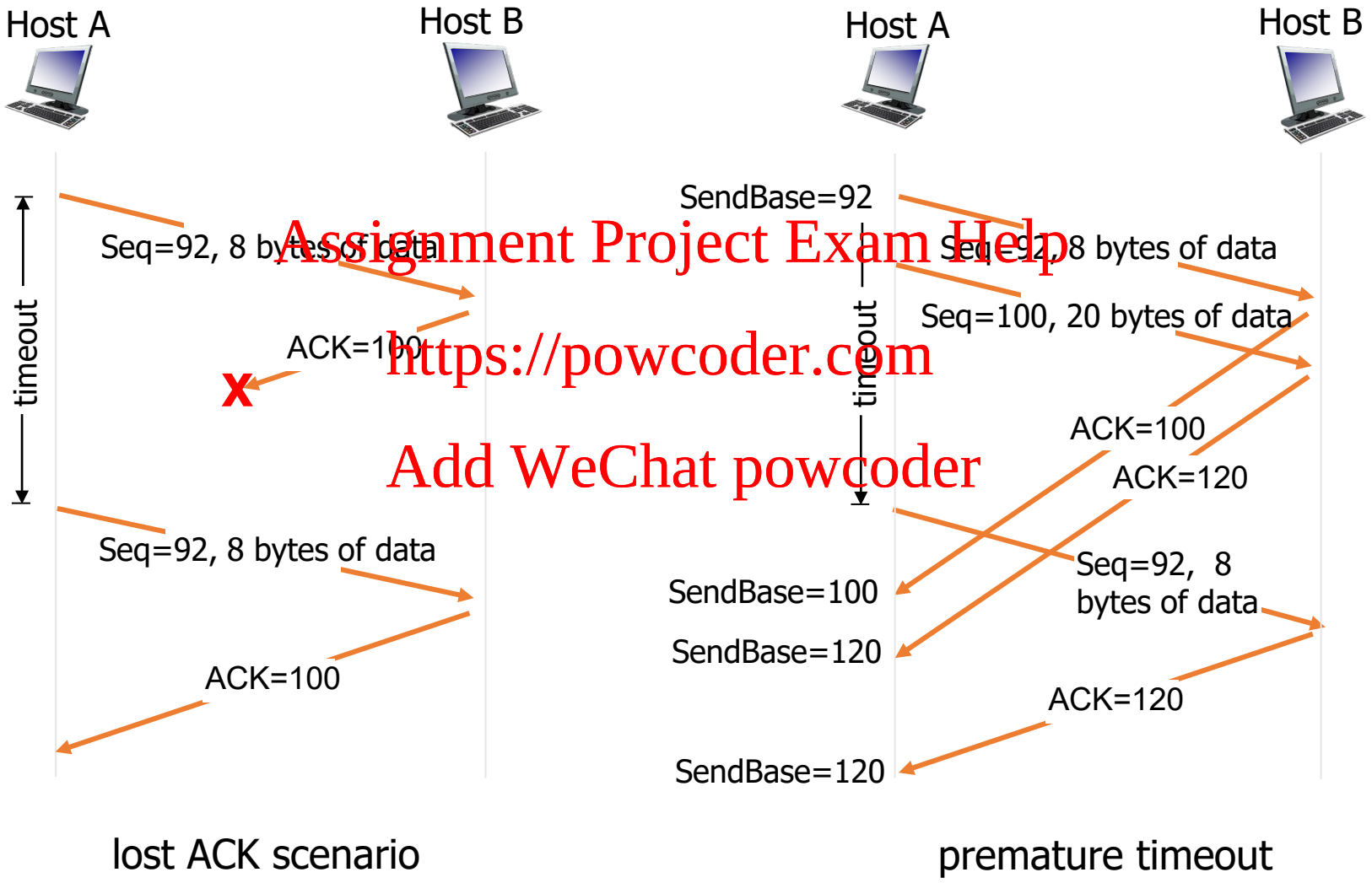


simple telnet scenario

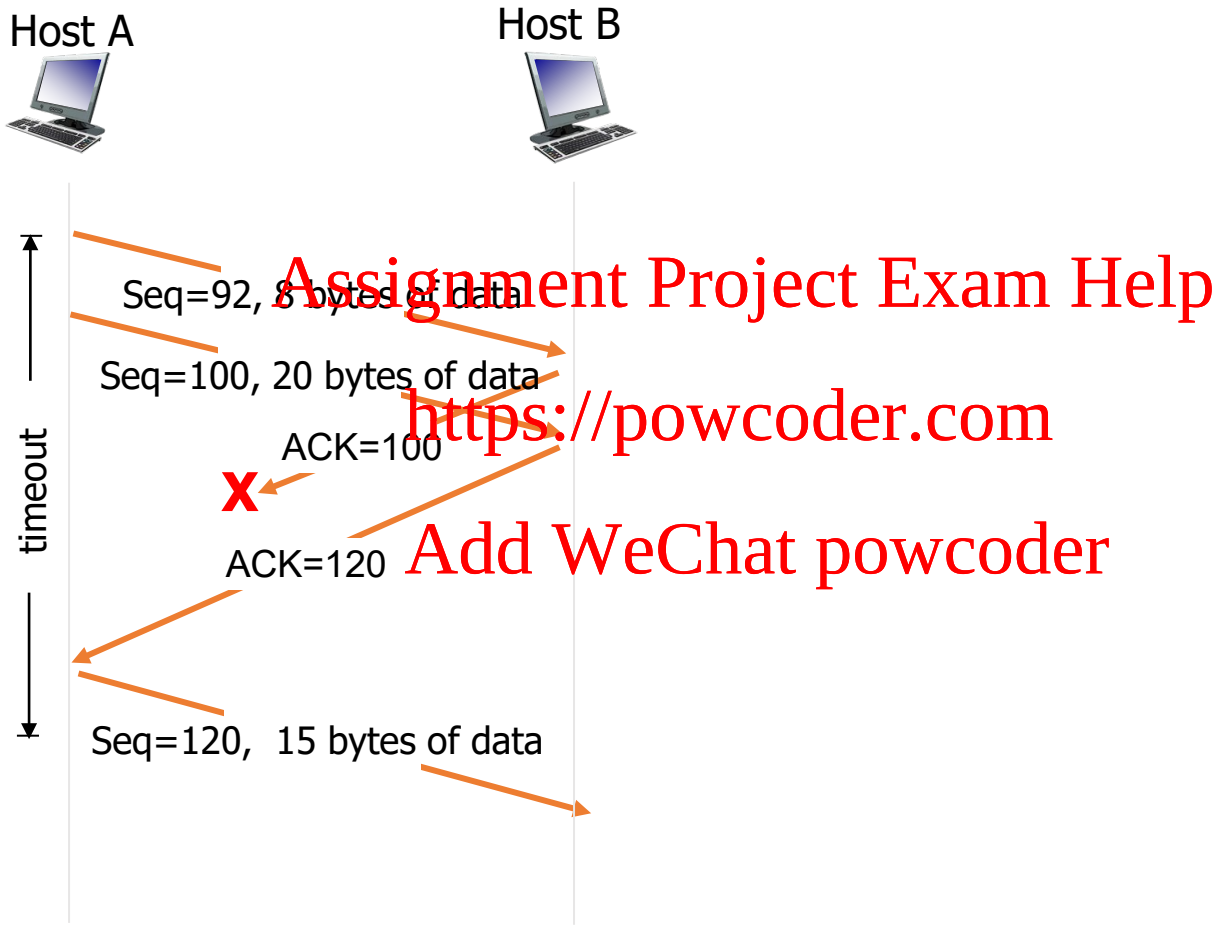
TCP: Reliability

- As you would expect, essentially running (a somewhat modified) Selective Repeat algorithm.
  - Here we have cumulative ACKs
- TCP provides its own reliability because the underlying layer (IP) does not make any reliability guarantees

TCP: retransmission scenarios



TCP: retransmission scenarios



cumulative ACK

TCP Acks

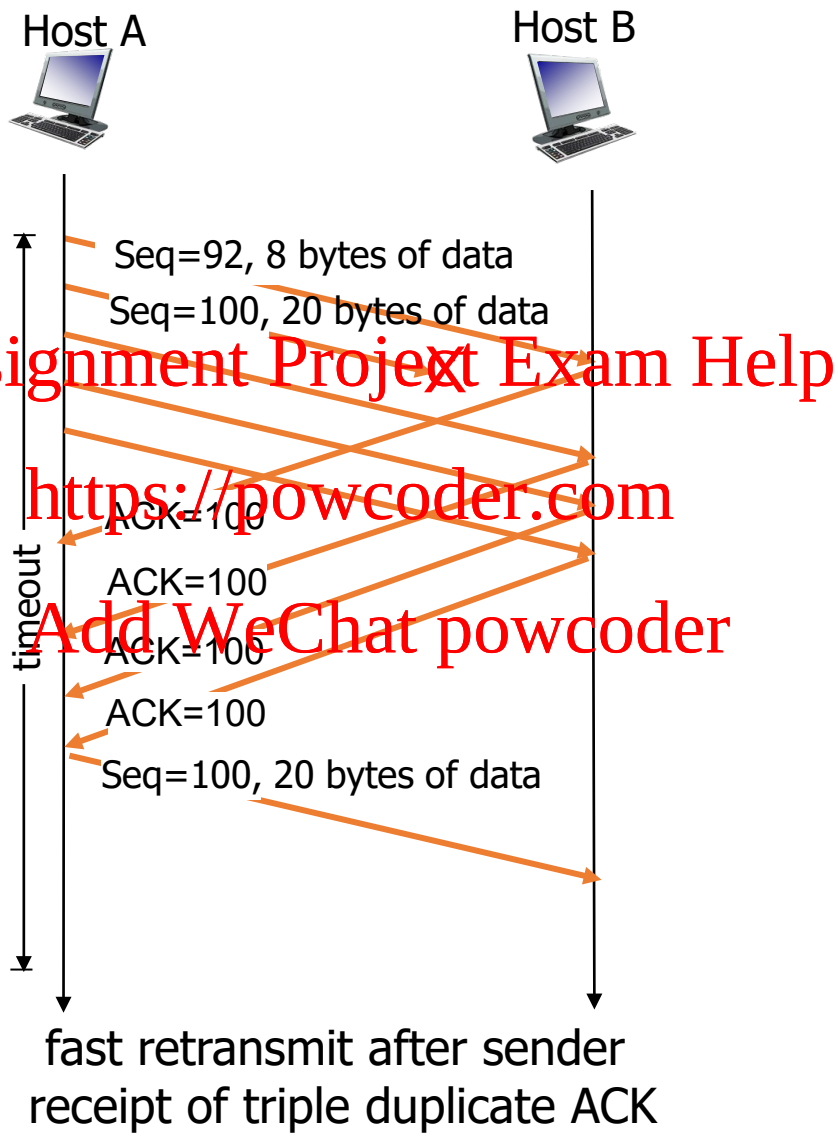
- Why are ACKs generated?

1. Arrival of an in-order, expected packet, all data up to that packet ACK'd
   1. Delayed ACK – wait 500 ms then ACK
2. Arrival of an in-order, expected packet
   1. Cumulative ACK
3. Arrival of out of order packet (higher than expected seq # (Gap)
   1. Immediately send duplicate ACK, indicating seq # of next expected byte
4. Arrival of a segment that partially or completely fills a gap
   1. Immediately send ACK (could be duplicate ACK)

TCP Fast Retransmit

- What happens in the (common) scenario of one packet being dropped and the next packets going through?
- if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #
  - likely that unacked segment lost, so don't wait for timeout

TCP Fast Retransmit



Flow Control in TCP

- More or less as simple as possible – if confused, just think of the easiest way to do this.
- If the sender sends too quickly, it might overwhelm the receiver & ensure that none of the data being sent can actually be stored and processed.
- Thus, receiver advertises how much space they actually have left in the **rwnd** value of the TCP header
- Sender never allows more than rwnd amount of bytes of un-ack'd data to be in flight

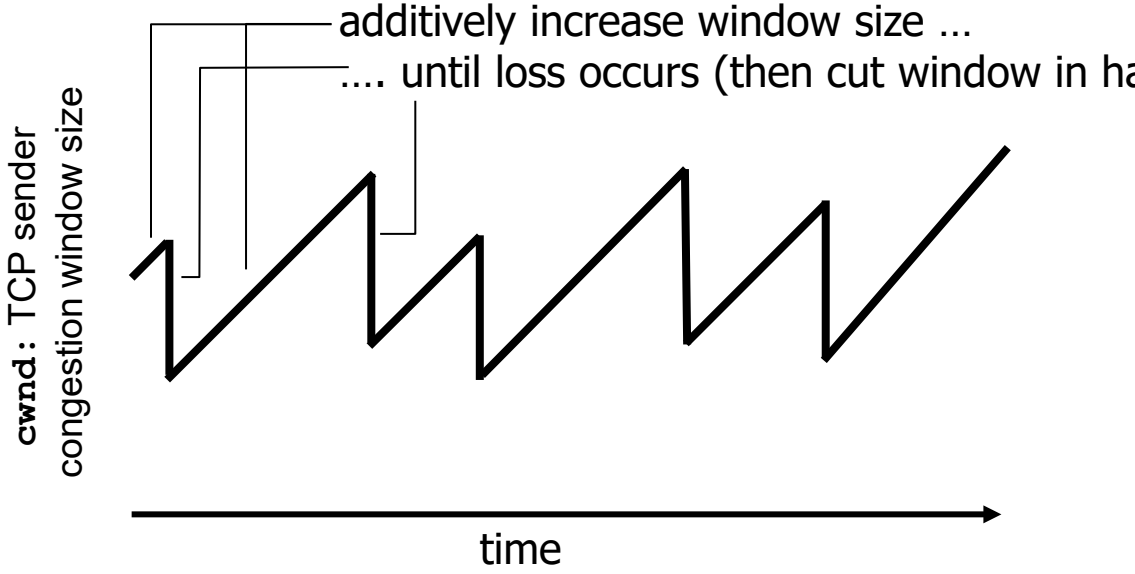
Congestion Control

- Constraining amount of packets sent to preserve the Network
  - Not the same thing as flow control (which tries to keep you from overwhelming the other host you're connected to)
- If not dealt with, get lost packets and long delays
- If *really* not dealt with, Internet Meltdown (Van Jacobson)

TCP Congestion Control: AIMD

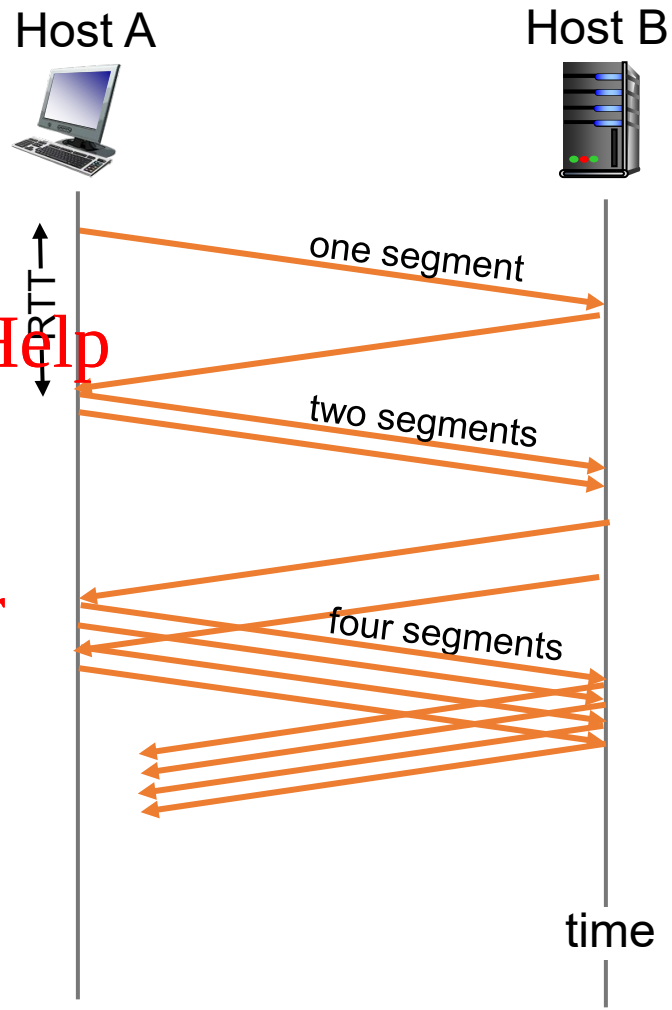
- AIMD = Additive Increase, Multiplicative Decrease
- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** (congestion window) by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **cwnd** in half after loss

AIMD General Behavior:



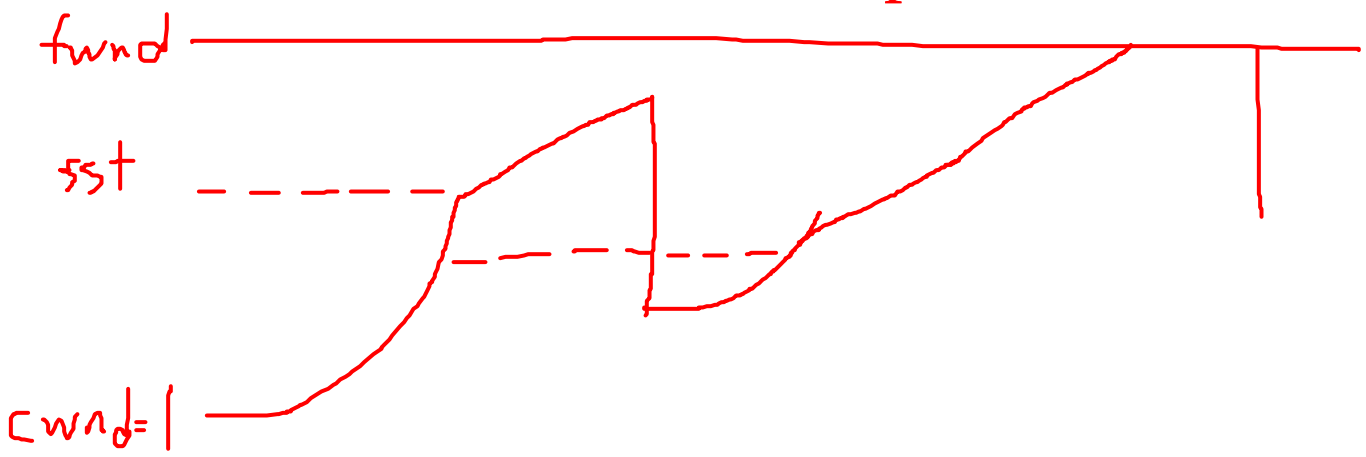
TCP Slow Start

- When first connecting, increase rate exponentially until first loss event.
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- In short: Start slow, exponential ramp up



Switching from Slow Start to AIMD

- When cwnd gets to half its value before timeout, switch from the former to the latter
- Implementation-wise, we create a variable called ssthresh, and set it to ½ of cwnd just before the loss event.
- In general, the throughput of these combined techniques is quite good.



TCP Fairness

- Between two competing TCP sessions, we will asymptotically approach fairness.
- Property: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

The End (of TCP)

Project 1: Select

- Problem: read ( ) and recv( ) work in **blocking mode** by default
  - Unless a new data arrives or remot ehost closes connection, read( ) or recv( ) will not return
- **Non-blocking mode**: use select ( )

select ( )

- Makes your code much simpler/easier
- **int select (int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);**
  - **numfds**: Greatest file descriptor + 1
  - **readfds**, **writefds**, **exceptfds**: Set of sockets to watch for reads/writes/exceptions
  - **timeout**: Timeout after which select will return if no sockets ready
  - Returns when at least one socket ready, or if timeout
  - Return value: number of sockets that are ready

fd_set

- A set of sockets (or file descriptors) that will be monitored by select ( )
- Macros of set operation
  - FD_SET(int fd, fd_set *set); // add fd to set
  - FD_CLR(int fd, fd_set *set); // remove fd from set
  - FD_ISSET(int fd, fd_set *set); // is fd in set?
  - FD_ZERO(fd_set *set); // clear all entries from set

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

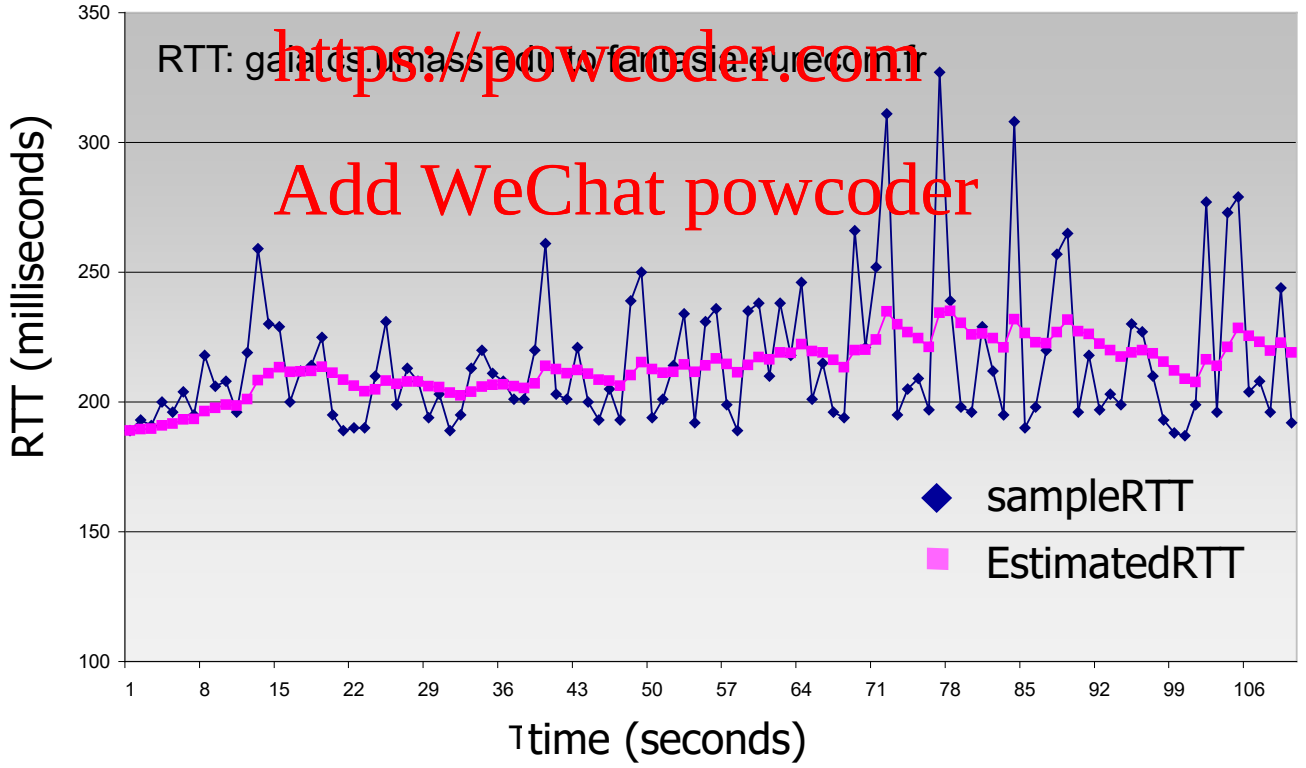
Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission 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
- 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:

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

(typically, $\beta = 0.25$)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

estimated RTT

"safety margin"

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/