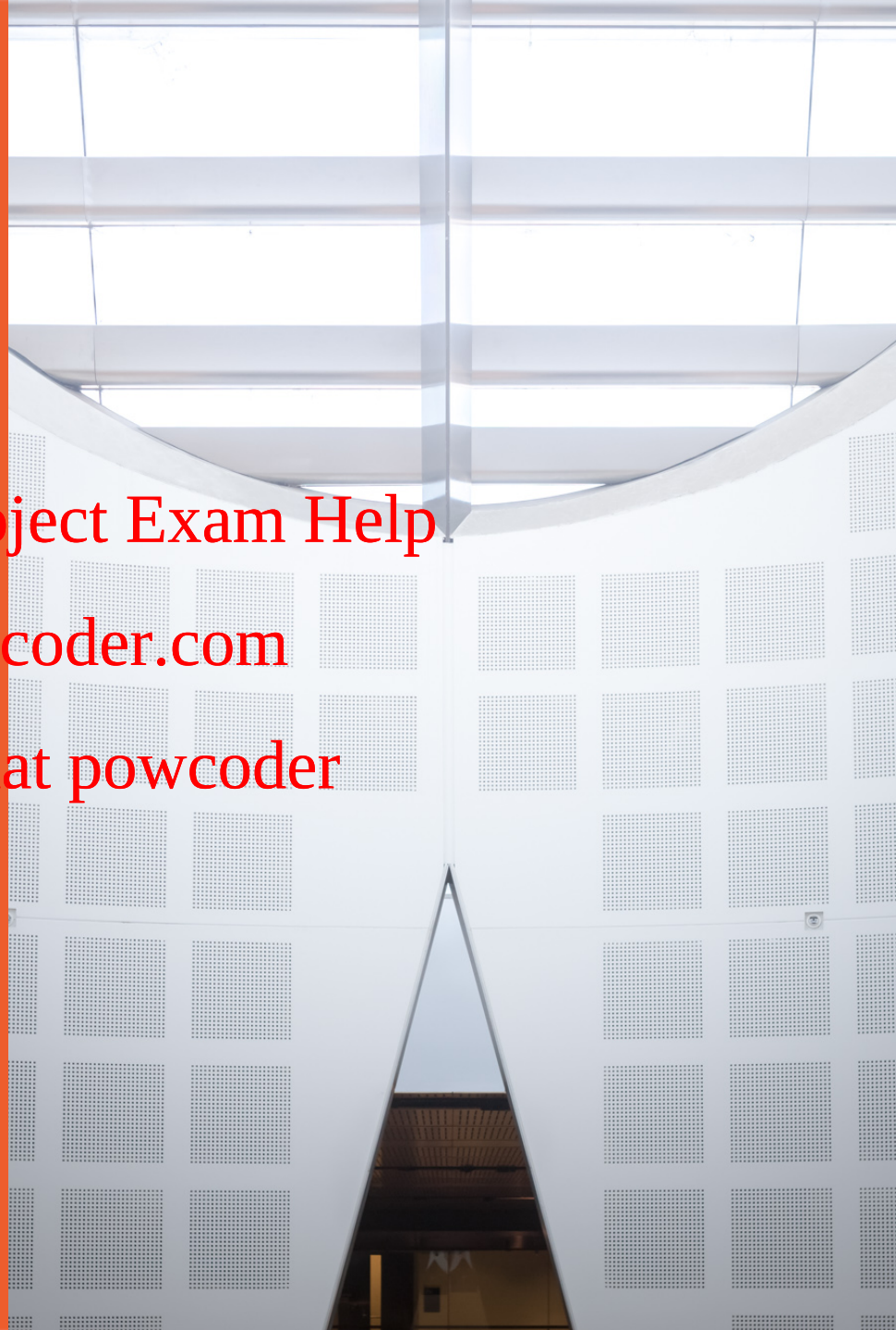
COMP3221: Distributed Systems

Communication (TCP)

Dr Nguyen Tran

School of Computer Science



THE UNIVERSITY OF SYDNEY

Previously…

- Previous lecture:
  - Message-based communication is complex (e.g., routing towards destination, subject to message losses)
- Today's lecture:
  - How to avoid message losses?
  - How to give the impression that everything happens locally (not remotely)?
  - One to many communication

Outline

- The Problem of Message Loss
- The TCP/IP Solution
- Multicast communication

The Problem of Message Loss

Message loss

Cause

- Networks are in general unreliable
- Messages can be lost (never been delivered even if sent)

- Examples:
  - A server receives too many requests simultaneously so it cannot treat all
  - A router drops the message because its queue is full

Message losses may impact the computation of a distributed system

Message loss

Coordinated Attack Problem

- Constraints of the problem
  - Two armies, each led by a general on separate mountains surrounding a battlefield (distributed system)
  - Can only communicate via messengers (message passing)
  - Messengers can be killed before reaching destination (message losses)

- Goal: they want to coordinate an attack
  - If they attack at different times, they both die
  - If they attack at the same time, they win

Message loss

Coordinated Attack Problem (con't)

- There is no protocols to make sure they will win!

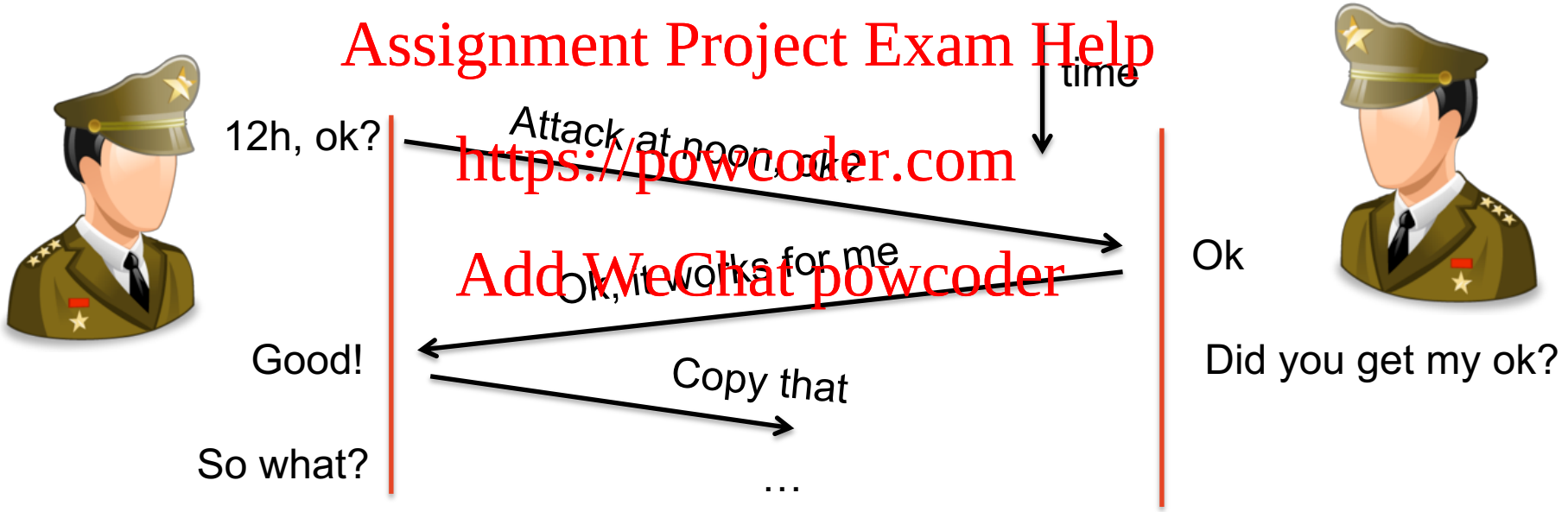
12h!

time

Message loss

Coordinated Attack Problem (con't)

- There is no protocols to make sure they will win!



Message loss

Analogy in networking

- Constraints of the problem
  - Two remote entities of a distributed system
  - Can only communicate through messages
  - The network is unreliable: messages can be dropped
- Goal: they want to make sure to do something simultaneously

This is impossible, even if all messages go through

TCP/IP

Communication 2/2
Week 4, COMP3221

THE UNIVERSITY OF SYDNEY

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

- Point-to-point:
  - one sender, one receiver
- Connection-oriented: In order byte stream:
- Full duplex data:
  - bi-directional data flow in same connection
- Flow controlled:
  - sender will not overwhelm receiver
- Congestion controlled:
  - TCP congestion and flow control set window size

TCP seq. numbers, ACKs

sequence numbers:

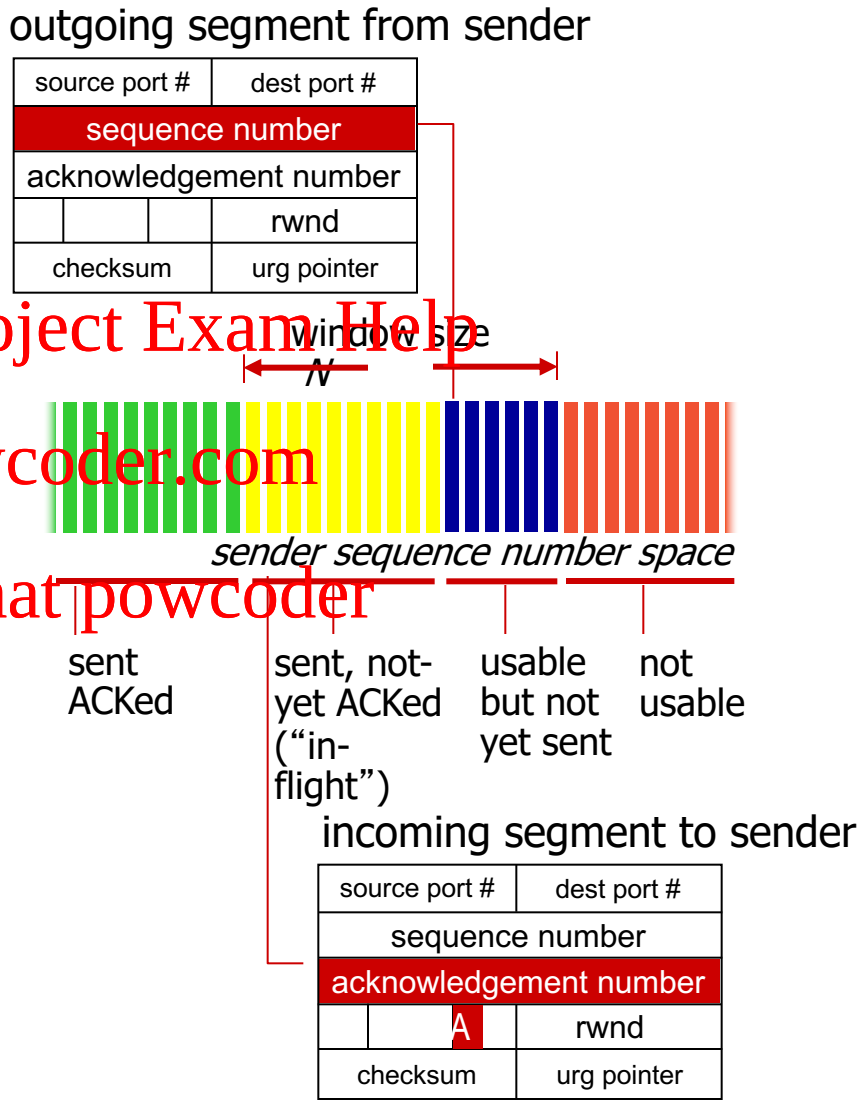
- byte stream "number" of first byte in segment's data

acknowledgements:

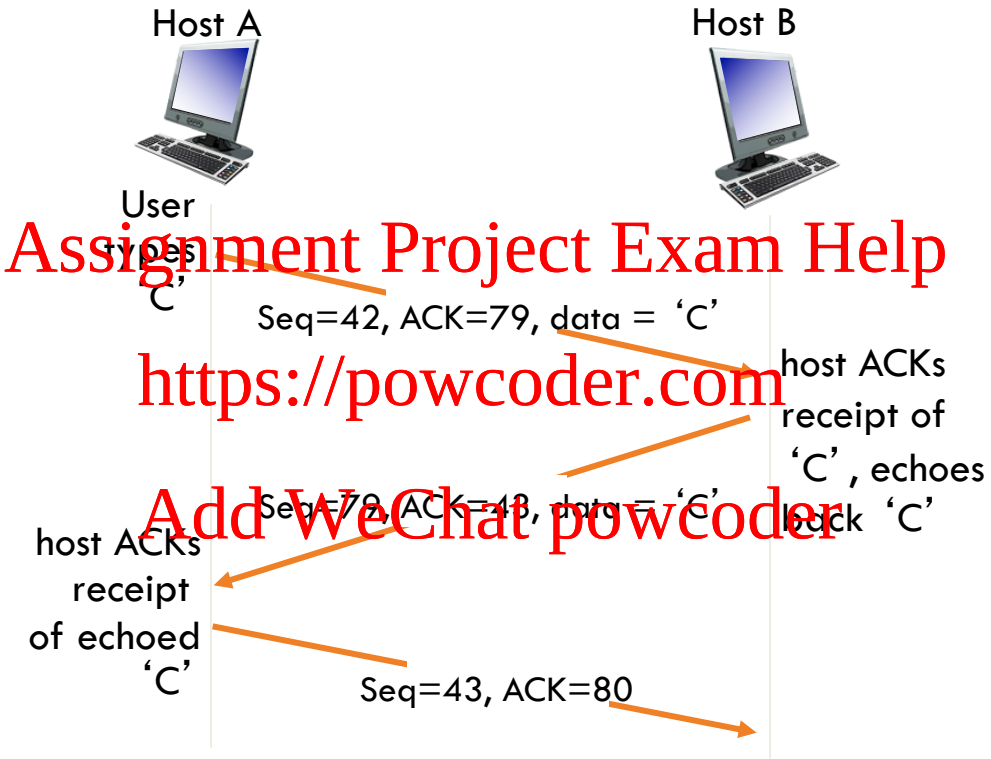
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



TCP seq. numbers, ACKs



simple telnet scenario

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 not already running
  - think of timer as for oldest unacked segment
  - expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

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

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

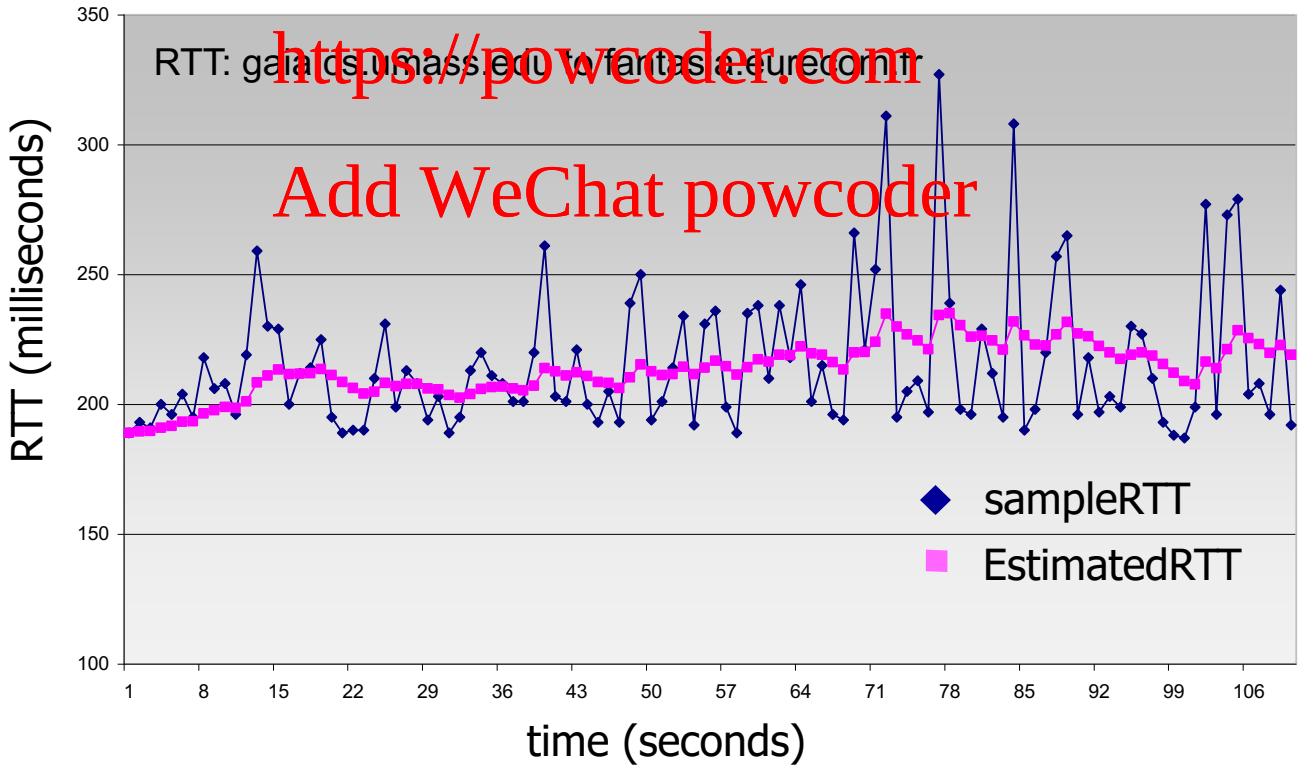
Q: how to estimate RTT?

- **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-β)*DevRTT +

β*|SampleRTT-EstimatedRTT|

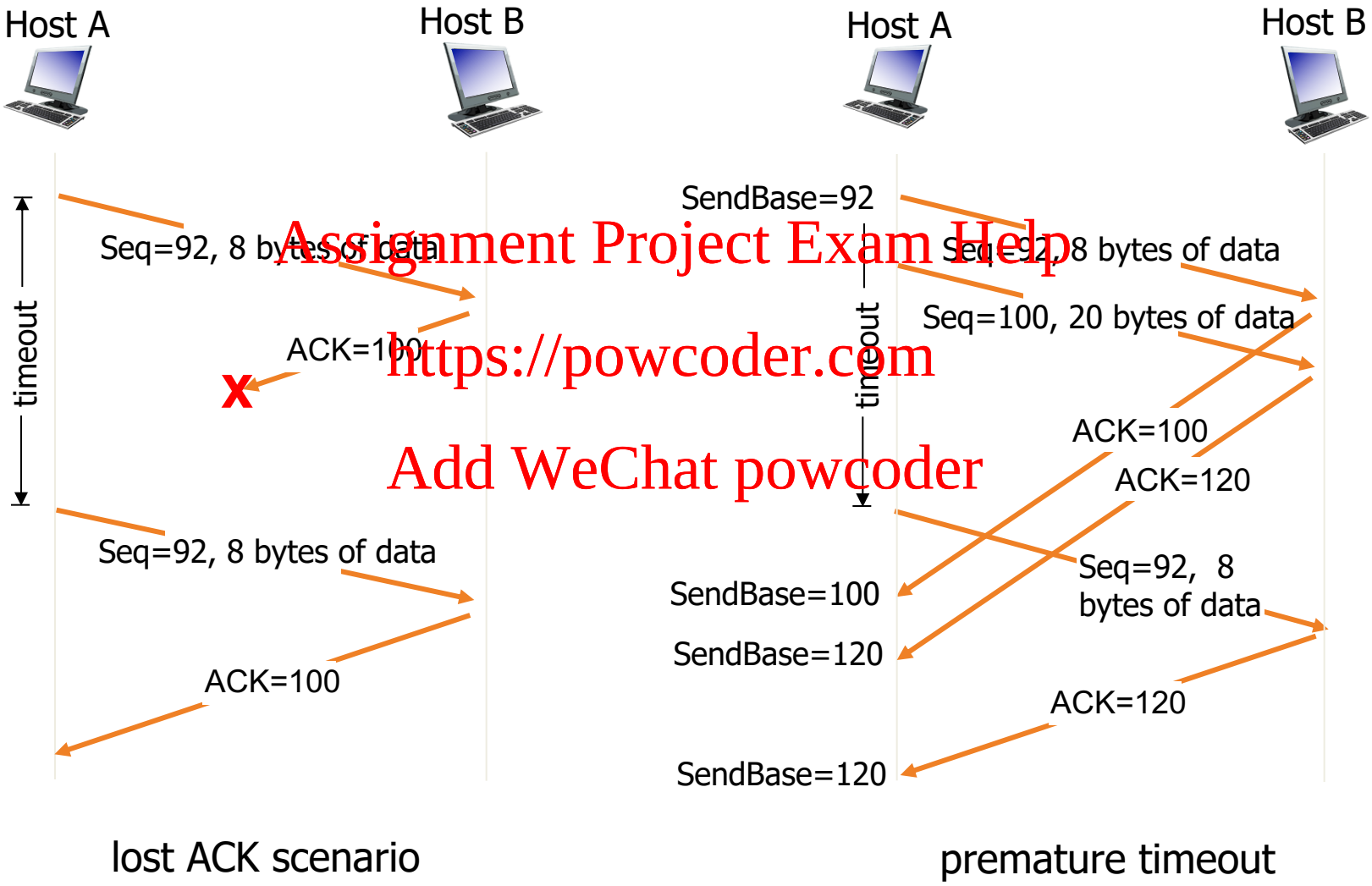
(typically, β = 0.25)

TimeoutInterval = EstimatedRTT + 4*DevRTT

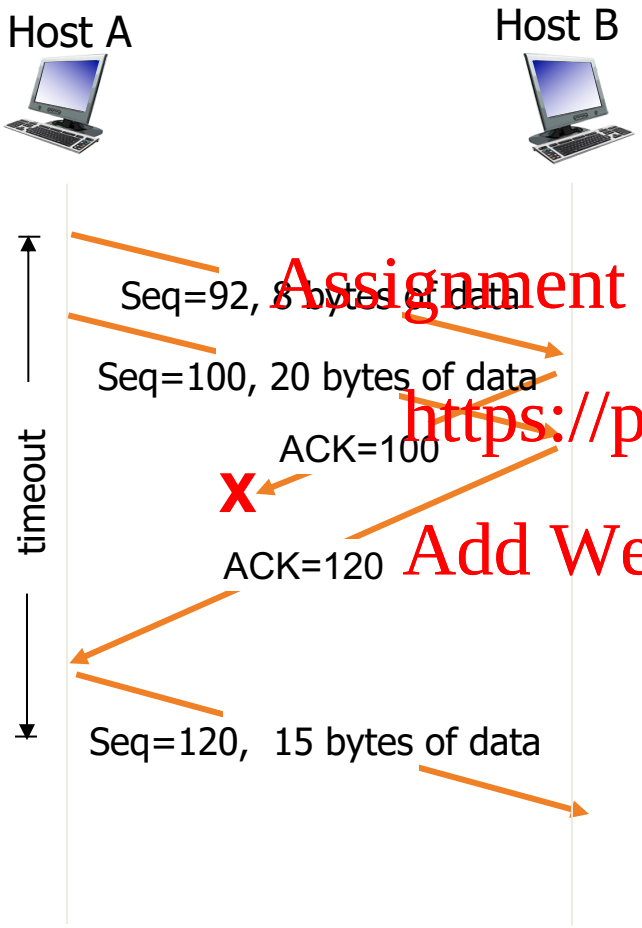
estimated RTT

"safety margin"

TCP: retransmission scenarios



TCP: retransmission scenarios



cumulative ACK

TCP fast retransmit

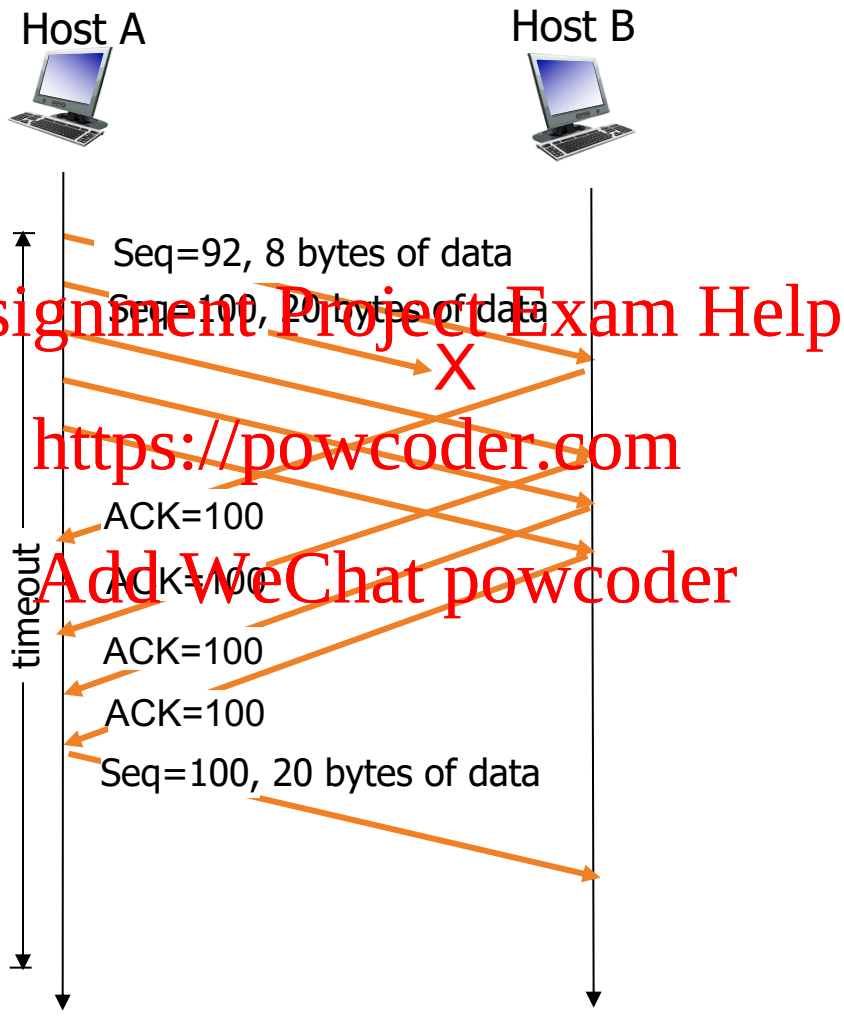
- time-out period often relatively long:
  - long delay before resending a lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

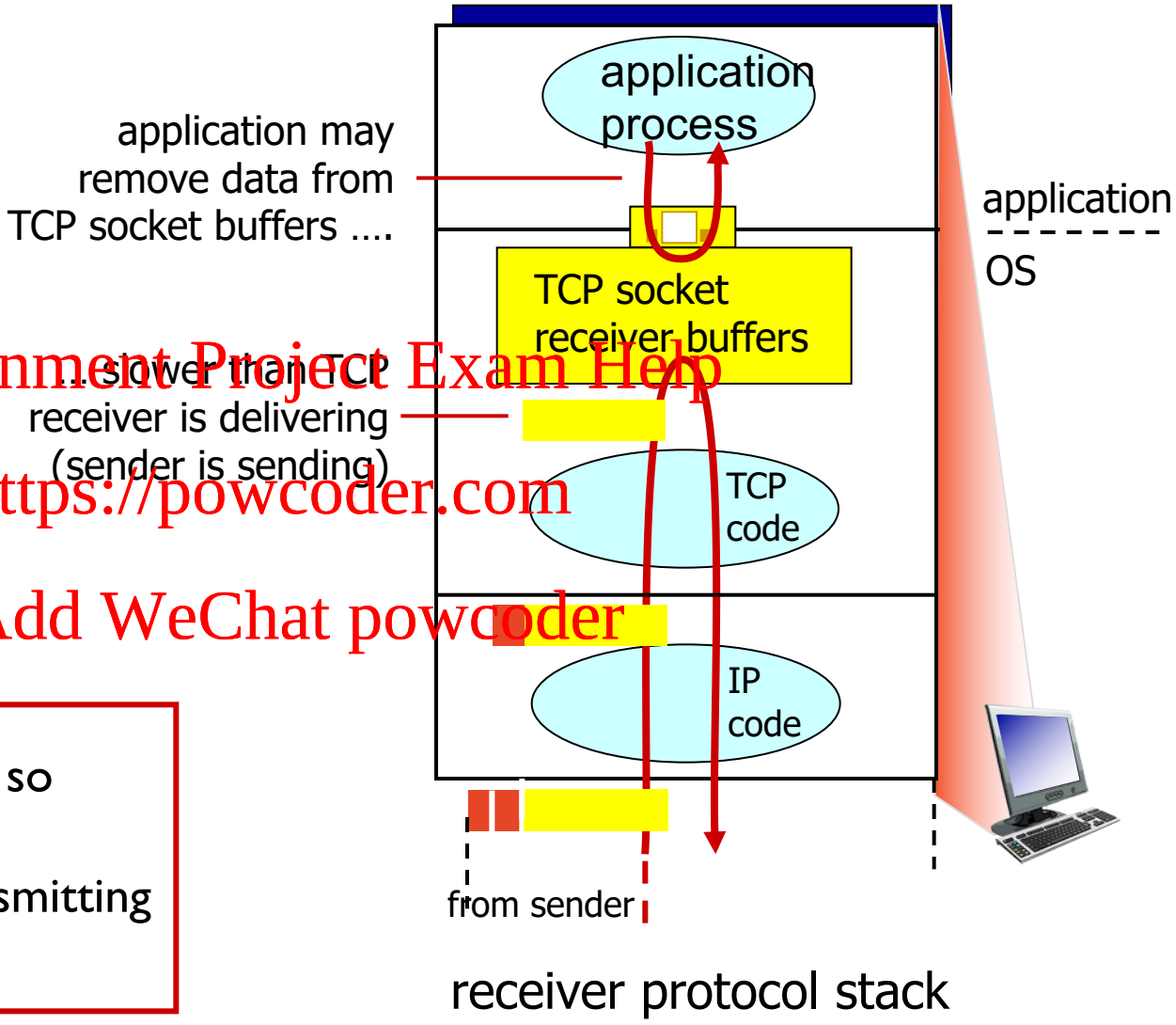
- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP flow control

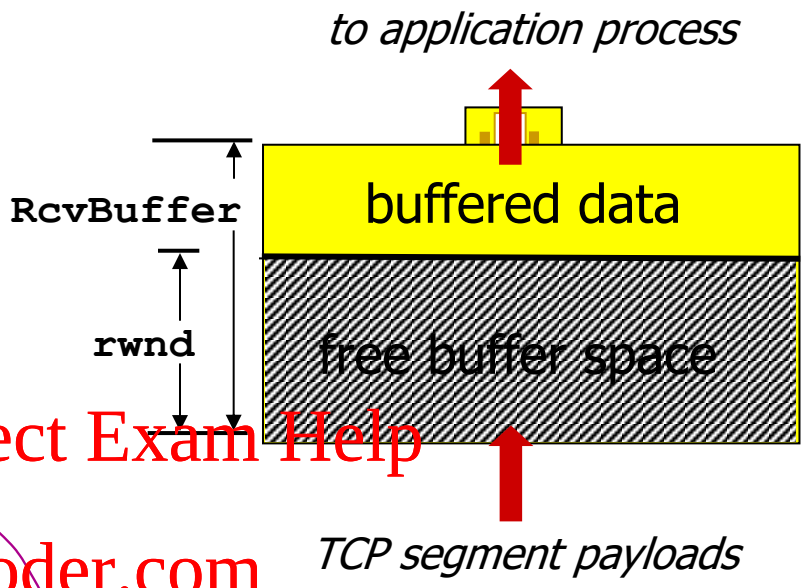


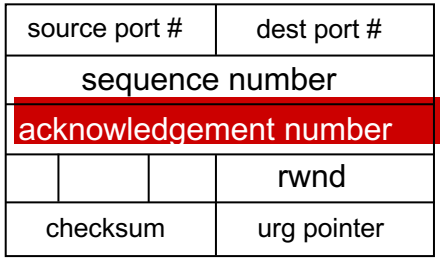
flow control

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 set via socket options (typical default is 4096 bytes)
  - many operating systems auto adjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow





Principles of congestion control

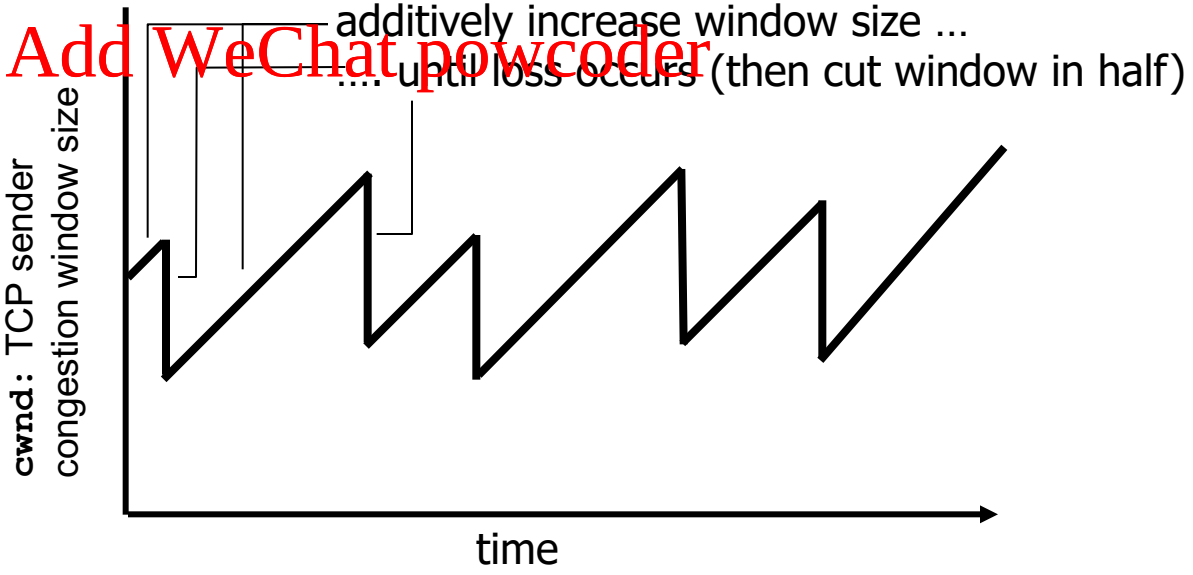
congestion:

- informally: "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!

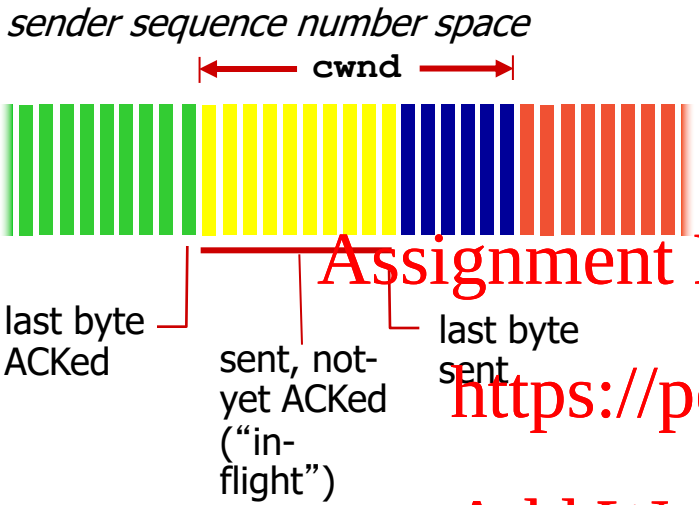
TCP congestion control

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

- sender limits transmission:

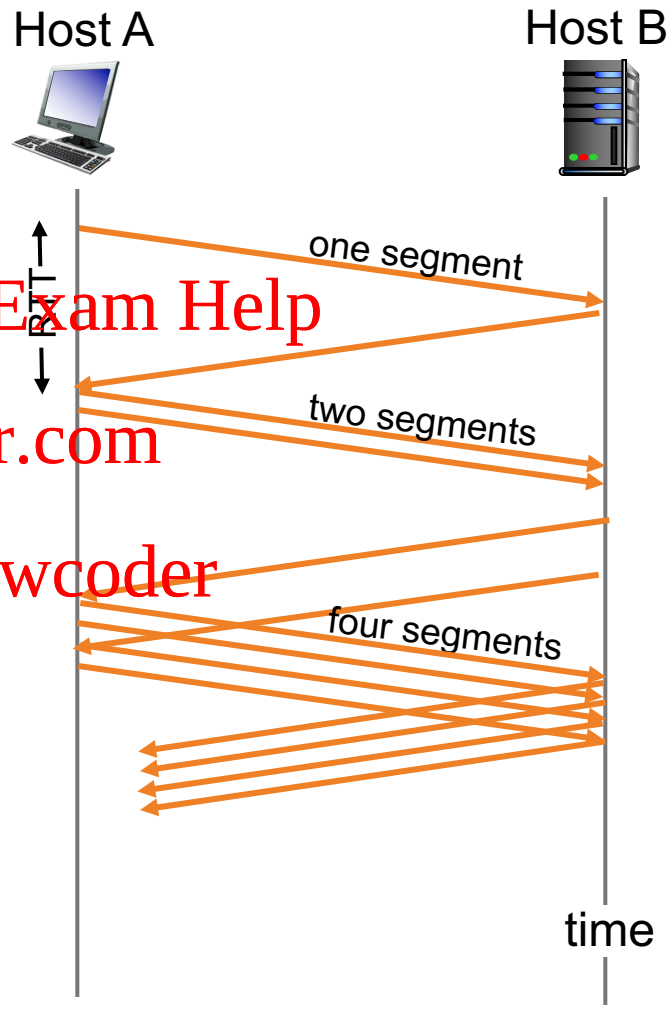
$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}$$

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

- Congestion window **(cwnd)** is dynamic function of perceived network congestion

TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS (Maximum Segment Size)
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



TCP: detecting, reacting to loss

- Loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

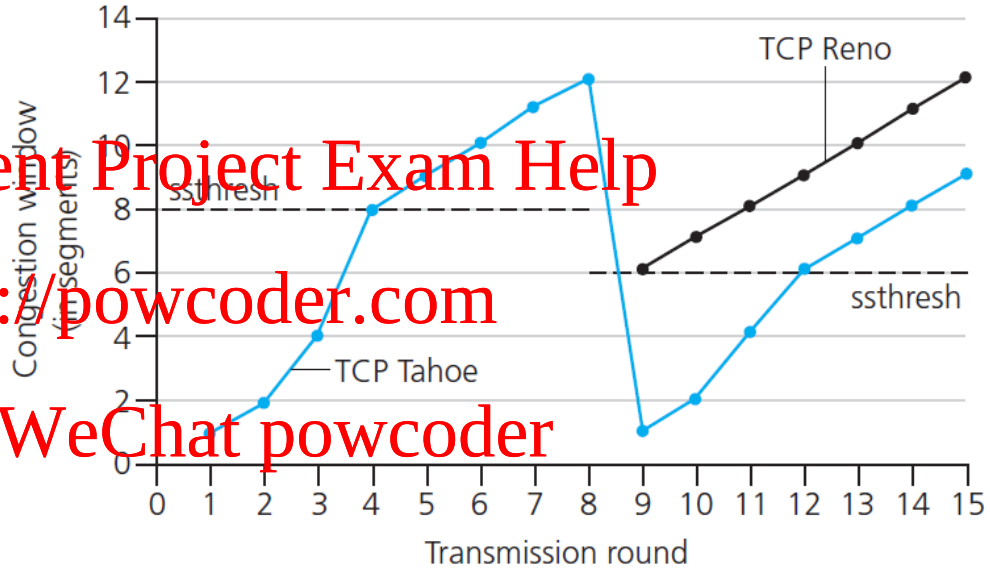
- Loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - **`cwnd`** is cut in half window then grows linearly

- TCP Tahoe always sets **`cwnd`** to 1 (timeout or 3 duplicate acks)

TCP: switching from slow start to Congestion Avoidance

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.



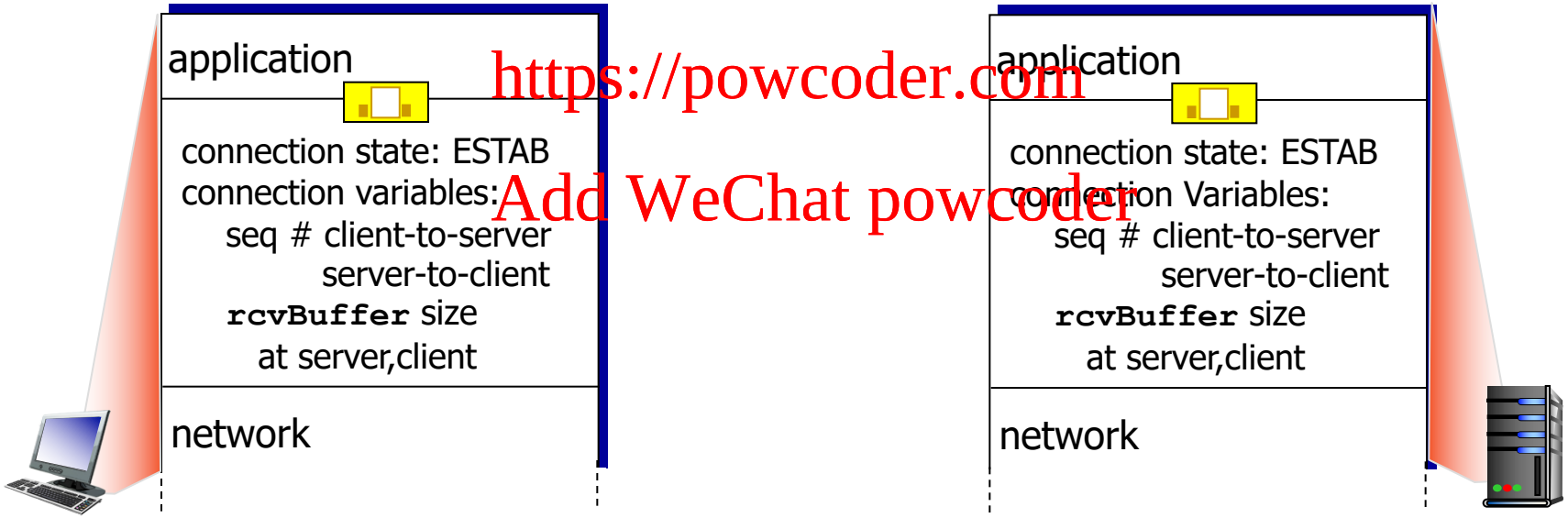
Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Connection Management

before exchanging data, sender/receiver "handshake":

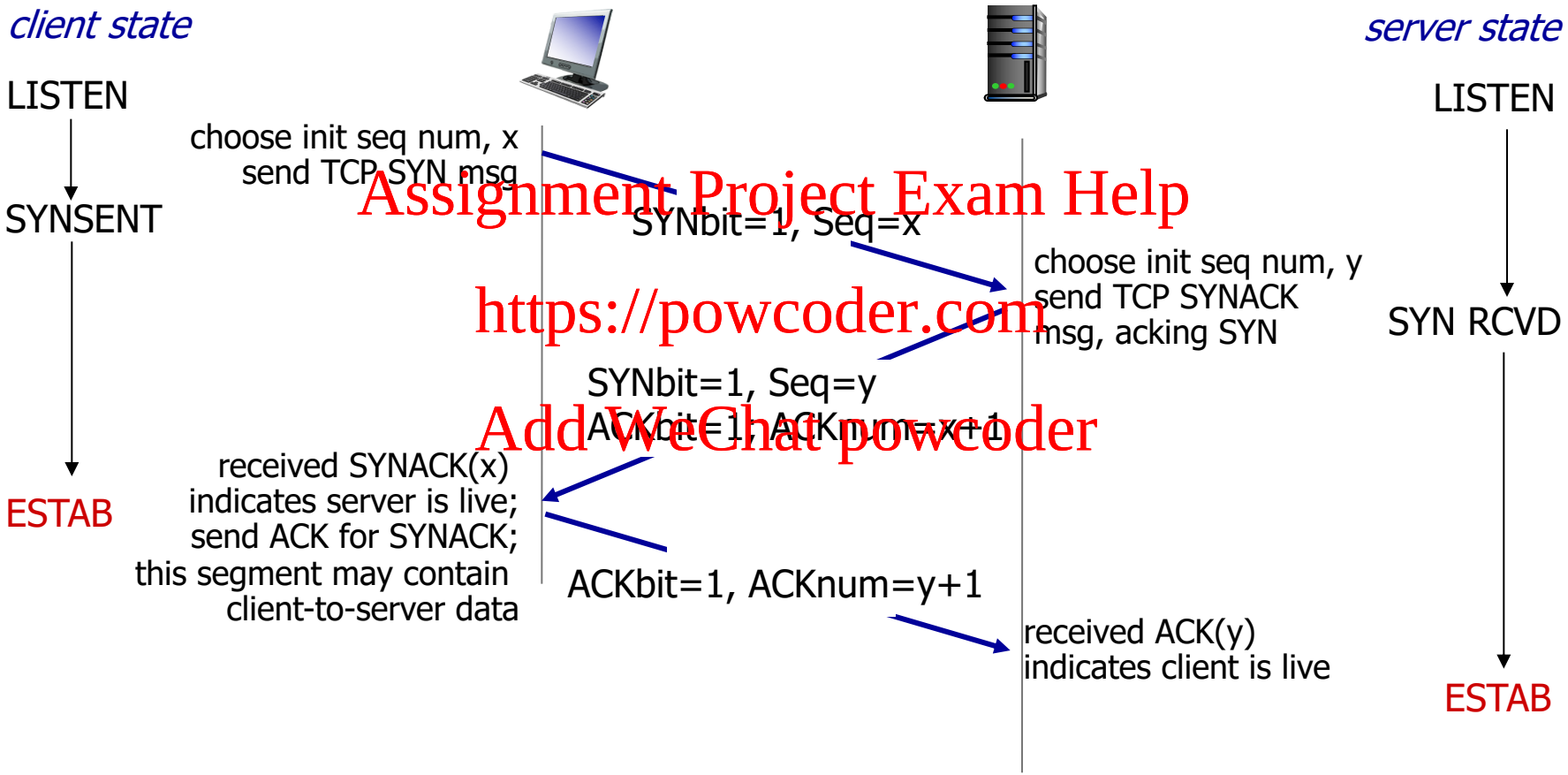
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters, e.g. MSS, rwnd, etc.
- connection-oriented



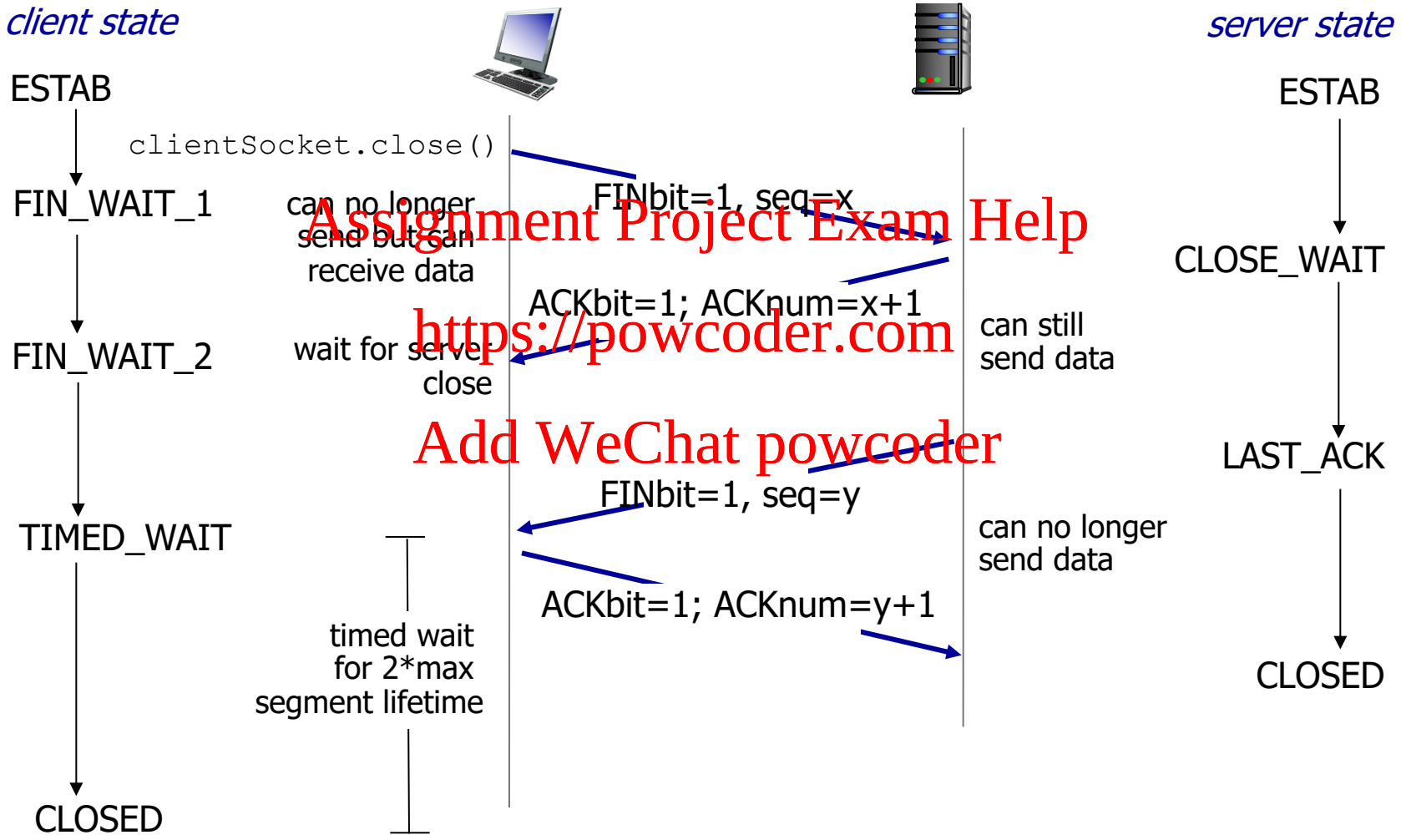

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
  welcomeSocket.accept();
```

TCP 3-way handshake



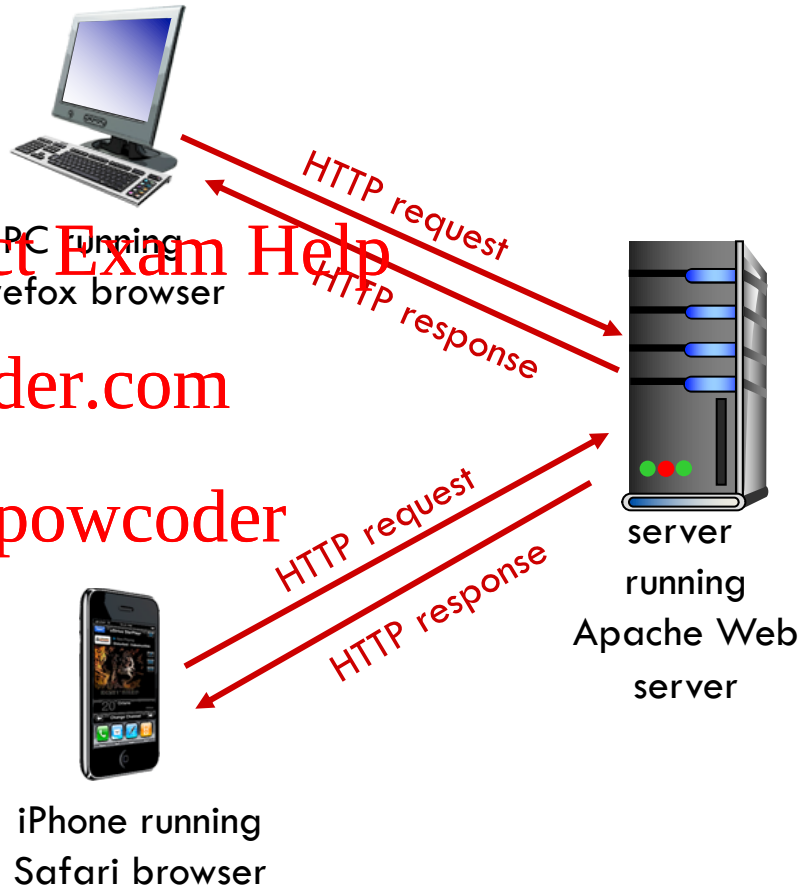
TCP: closing a connection



HTTP Overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



HTTP overview

Uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

What is HTTPS ?

- HTTP over TLS (Transport Layer Security), e.g. over SSL

Related terminology

- Broadcast: one-to-all communication
  - Action of sending a message to all nodes of the system
  - Typically used for relatively small systems, like IP broadcast as part of Ethernet in LANs
- Unicast: one-to-one communication
  - In contrast with broadcast, describe a special form of distribution to a single receiver
  - Used generally in the context of multimedia or streaming applications
- Anycast: one-to-random-one communication
  - Send a message to an IP address range from to obtain a response from any potential receiver, whose IP address belongs to this range
  - Used with UDP and TCP
- Multicast: one-to-many communication
  - Action of sending a message to multiple nodes of the system (not necessarily all nodes)
  - This term is used in many contexts (network, algorithm)

Conclusion (Transportation layer)

- Message losses can have dramatic consequences
- TCP/IP protocol suite hides these losses from the application level
- Socket, RPC, use TCP/IP
  - Sockets are complex
  - RPC is more transparent for the client
- Multicast communication scales well for large distributed systems