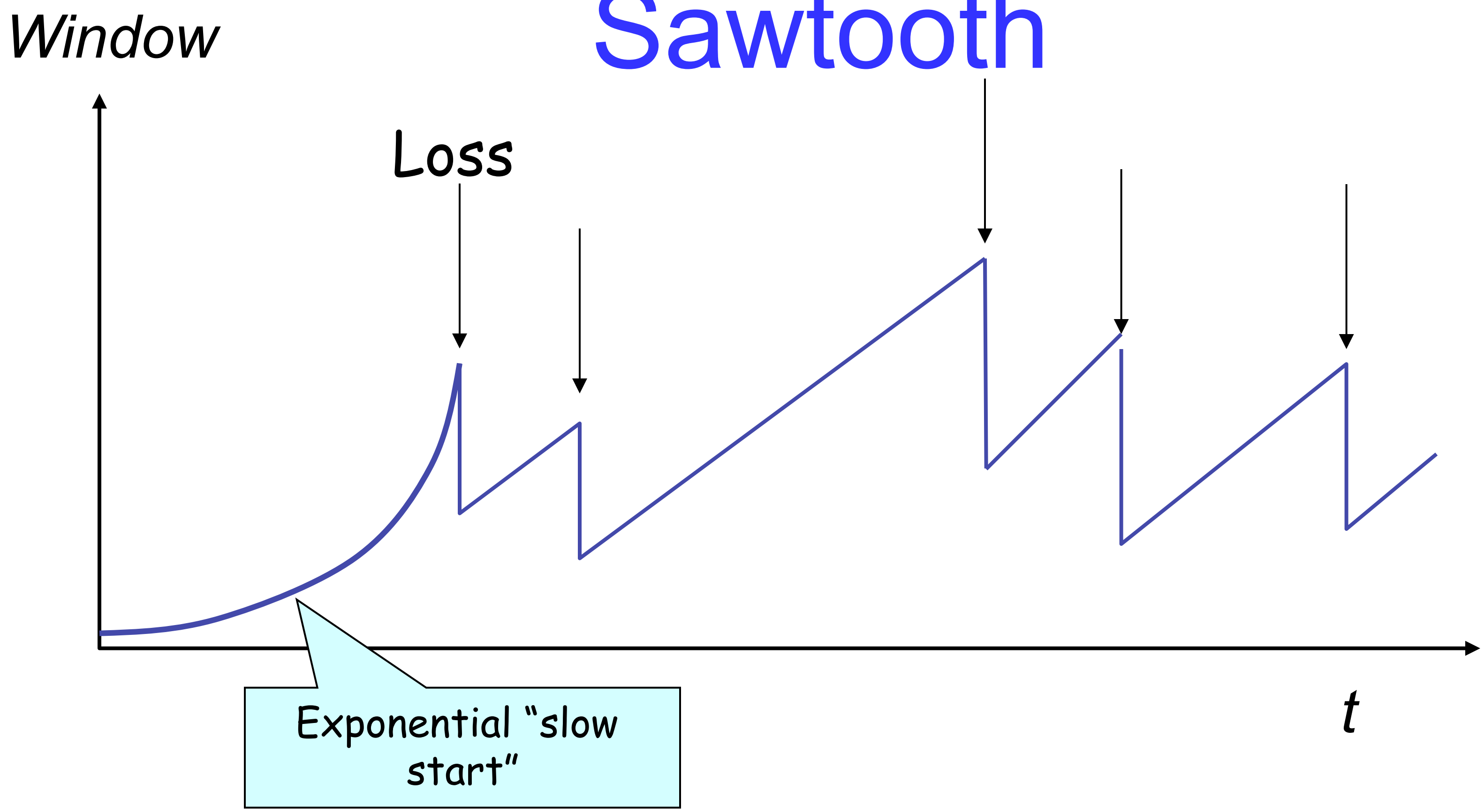
CS 457 – Lecture 24
Congestion

Fall 2011

Slow Start and the TCP Sawtooth

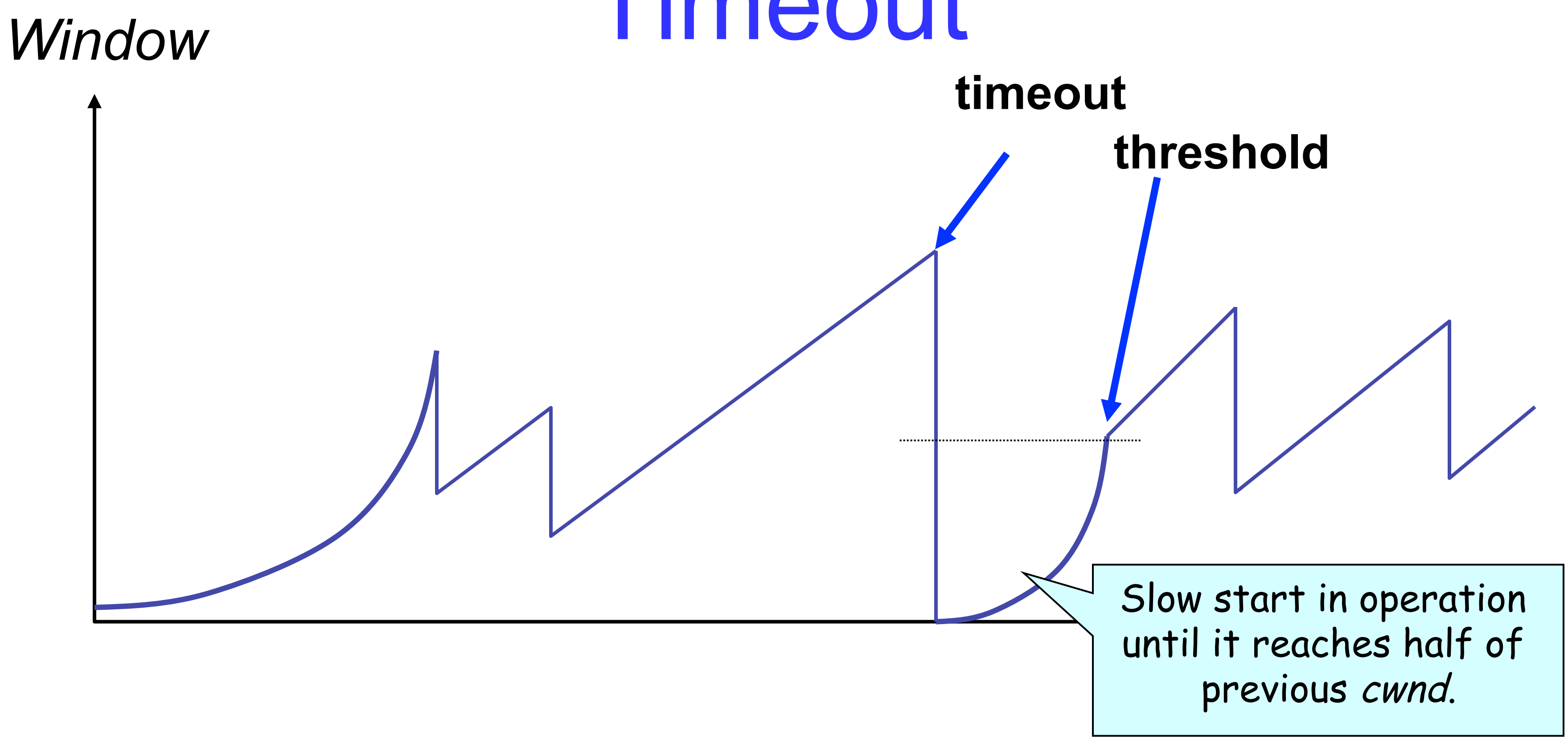


Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole window's worth of data.

Two Kinds of Loss in TCP

- Triple duplicate ACK
  - Packet n is lost, but packets n+1, n+2, etc. arrive
  - Receiver sends duplicate acknowledgments
  - … and the sender retransmits packet n quickly
  - Do a multiplicative decrease and keep going (no slow-start)
- Timeout
  - Packet n is lost and detected via a timeout
  - Could be because all packets in flight were lost
  - After the timeout, blasting away for the entire CWND
  - … would trigger a very large burst in traffic
  - So, better to start over with a very low CWND

Repeating Slow Start After Timeout



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

Repeating Slow Start After Idle Period

- Suppose a TCP connection goes idle for a while
  - E.g., Telnet session where you don't type for an hour
- Eventually, the network conditions change
  - Maybe many more flows are traversing the link
  - E.g., maybe everybody has come back from lunch!
- Dangerous to start transmitting at the old rate
  - Previously-idle TCP sender might blast the network
  - … causing excessive congestion and packet loss
- So, some TCP implementations repeat slow start
  - Slow-start restart after an idle period

Summary: TCP Congestion Control

- When `CongWin` is below `Threshold`, sender in slow-start phase, window grows exponentially.
- When `CongWin` is above `Threshold`, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, `Threshold` set to `CongWin/2` and `CongWin` set to `Threshold`.
- When timeout occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS.

Event	State	TCP Sender Action	Commentary
ACK receipt for previously unACKed data	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
ACK receipt for previously unACKed data	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
Timeout	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being ACKed	CongWin and Threshold not changed

Other TCP Mechanisms

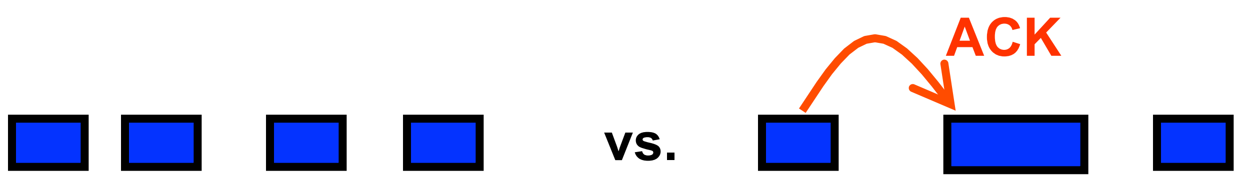
Nagle's Algorithm and Delayed ACK

Motivation for Nagle's Algorithm

- Interactive applications
  - Telnet and rlogin
  - Generate many small packets (e.g., keystrokes)
- Small packets are wasteful
  - Mostly header (e.g., 40 bytes of header, 1 of data)
- Appealing to reduce the number of packets
  - Could force every packet to have some minimum size
  - … but, what if the person doesn't type more characters?
- Need to balance competing trade-offs
  - Send larger packets to increase efficiency
  - … but at the expense of delay

Nagle's Algorithm

- Wait if the amount of data is small
  - Smaller than Maximum Segment Size (MSS)
- …and some other packet is already in flight
  - i.e., still awaiting the ACKs for previous packets
- That is, send at most one small packet per RTT
  - … by waiting until all outstanding ACKs have arrived

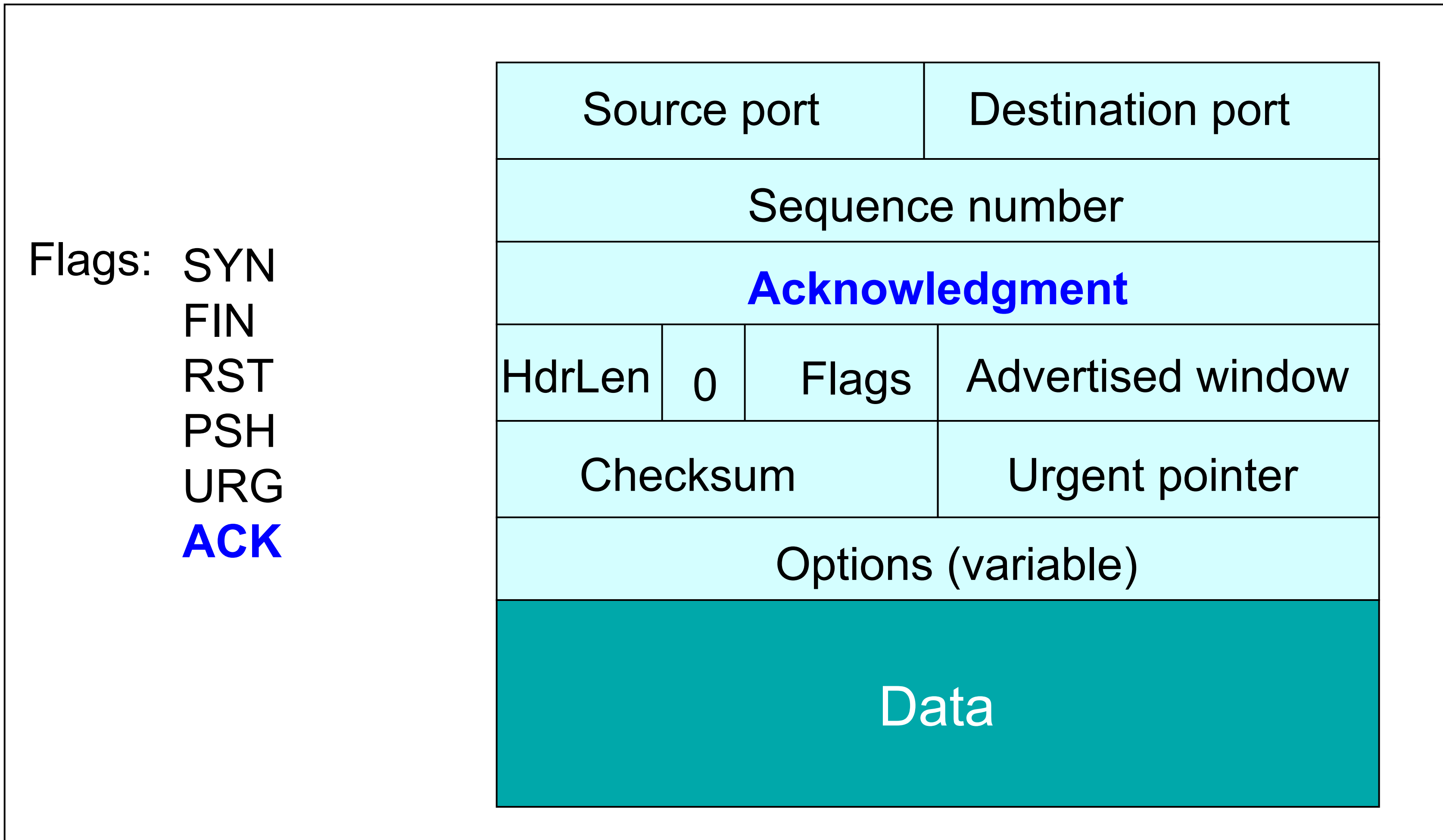


- Influence on performance
  - Interactive applications: enables batching of bytes
  - Bulk transfer: no change: transmits in MSS-sized packets anyway

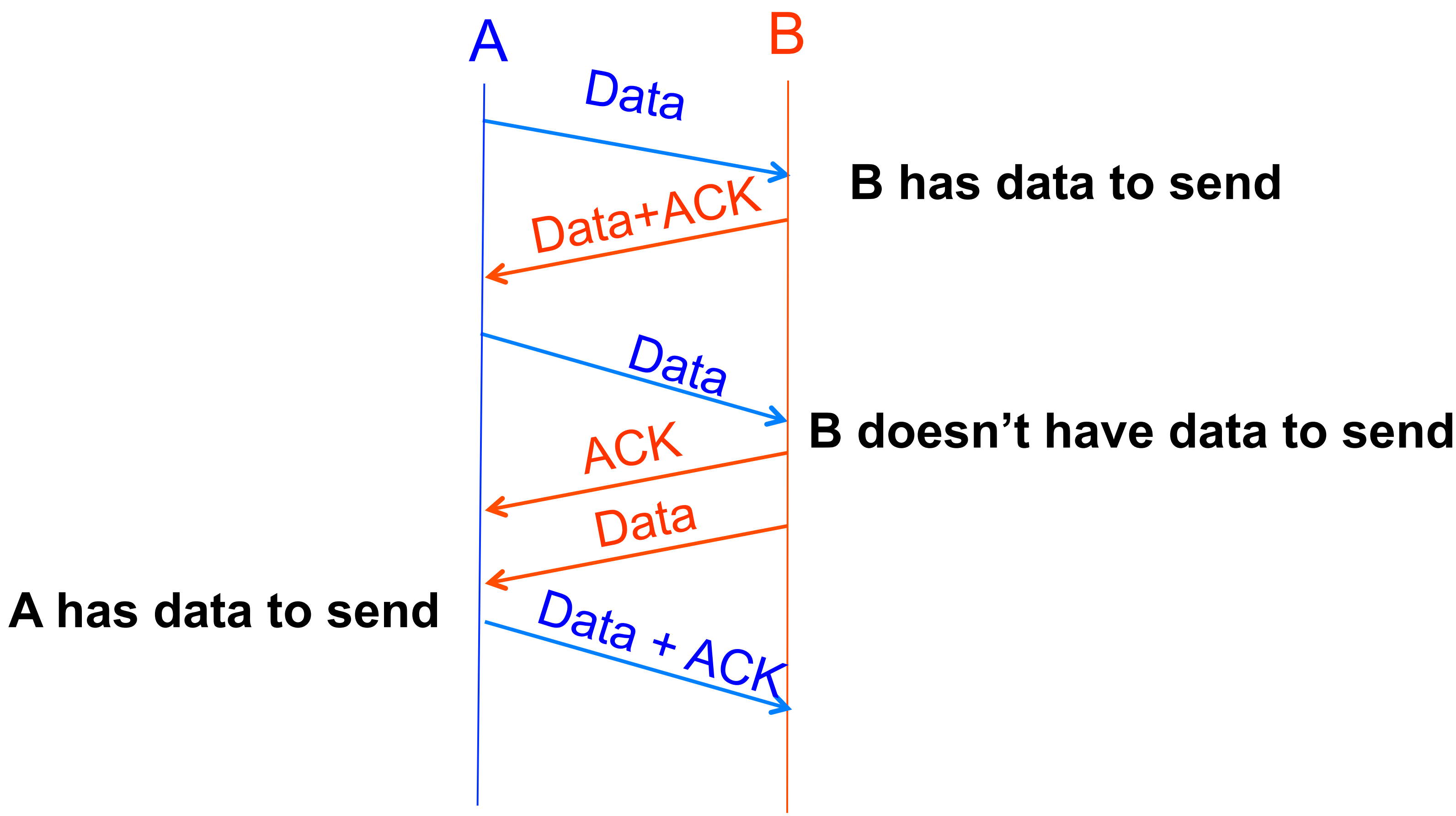
Delayed ACK - Motivation

- TCP traffic is often bidirectional
  - Data traveling in both directions
  - ACKs traveling in both directions
- ACK packets have high overhead
  - 40 bytes for the IP header and TCP header
  - … and zero data traffic
- Piggybacking is appealing
  - Host B can send an ACK to host A
  - … as part of a data packet from B to A

TCP Header Allows Piggybacking

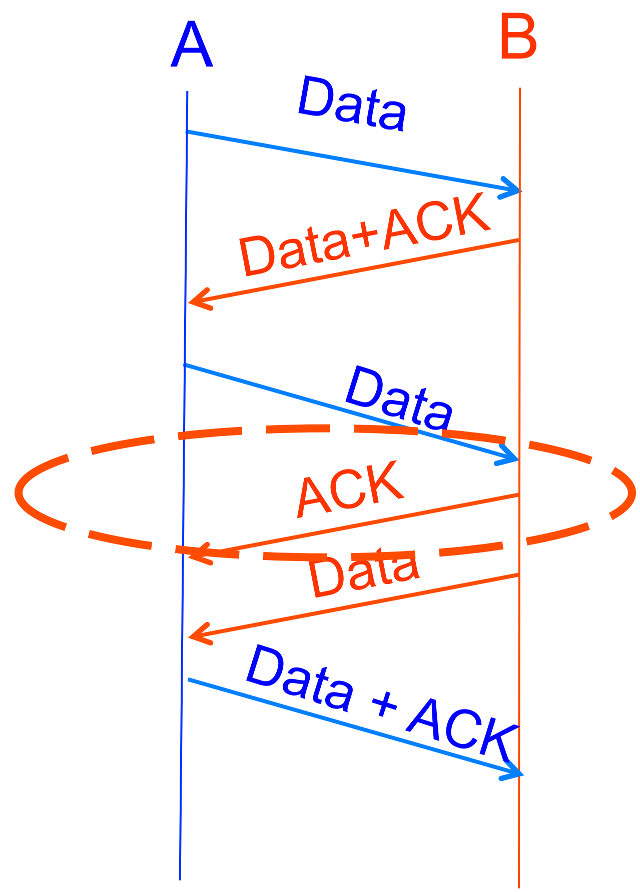


Example of Piggybacking



Increasing Likelihood of Piggybacking

- Increase piggybacking
  - TCP allows the receiver to *wait* to send the ACK
  - … in the hope that the host will have data to send
- Example: rlogin or telnet
  - Host A types characters at a UNIX prompt
  - Host B receives the character and executes a command
  - … and then data are generated
  - Would be nice if B could send the ACK with the new data



Delayed ACK

- Delay sending an ACK
  - Upon receiving a packet, the host B sets a timer
  - If B's application generates data, go ahead and send
    - And piggyback the ACK bit
  - If the timer expires, send a (non-piggybacked) ACK
- Limiting the wait
  - Timer of 200 msec or 500 msec
  - ACK every other full-sized packet

TCP Throughput and Fairness

Recall Fixed Window Delay

Assume

- Sender requests a file
- Receiver accepts request and replies with D bit file
- No congestion
- File Request and ACK messages a very small
  - small enough to ignore their transmission time

How much time will elapse before the file is completely transfered?

Case 1: "A Big Enough Window"

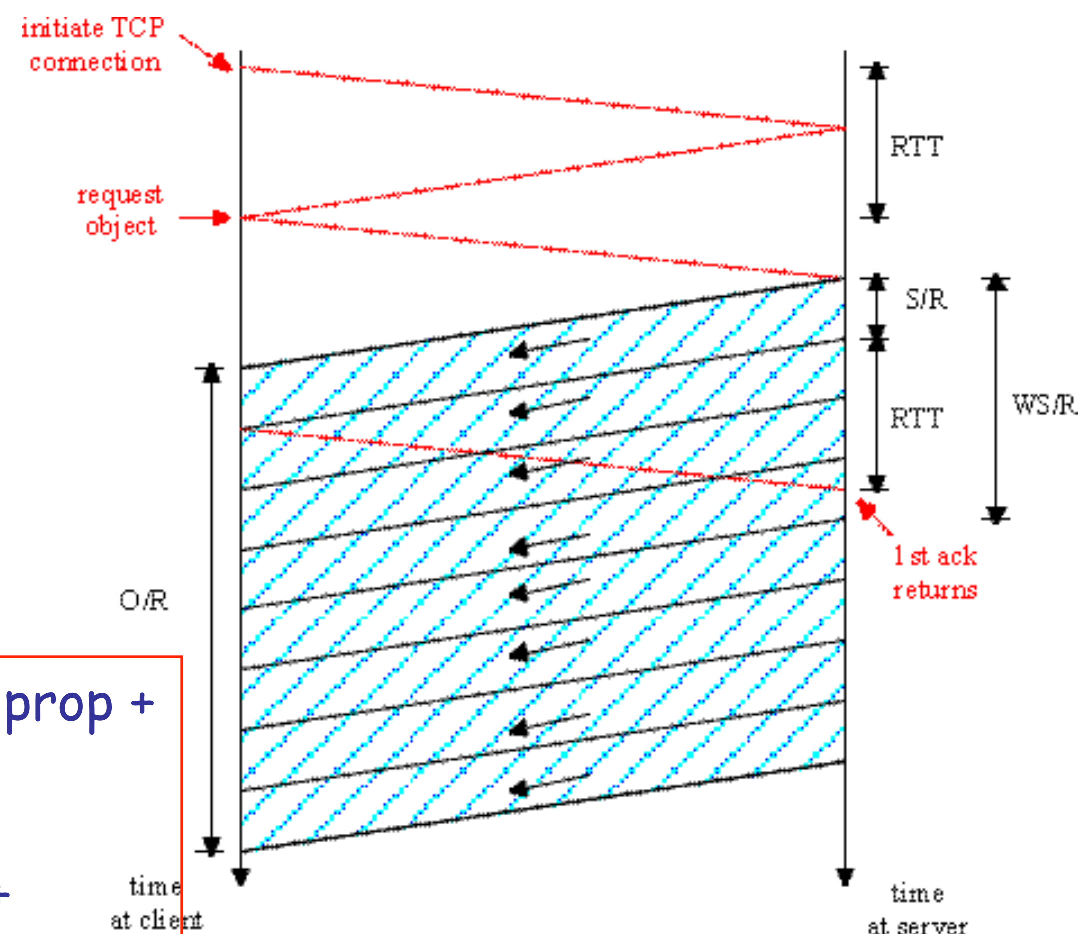
"Big enough" means that time to send window is bigger than time to get first ACK:

More precisely:

$(W*S)/R > RTT + S/R$:

delay = handshake + request + prop + transmt

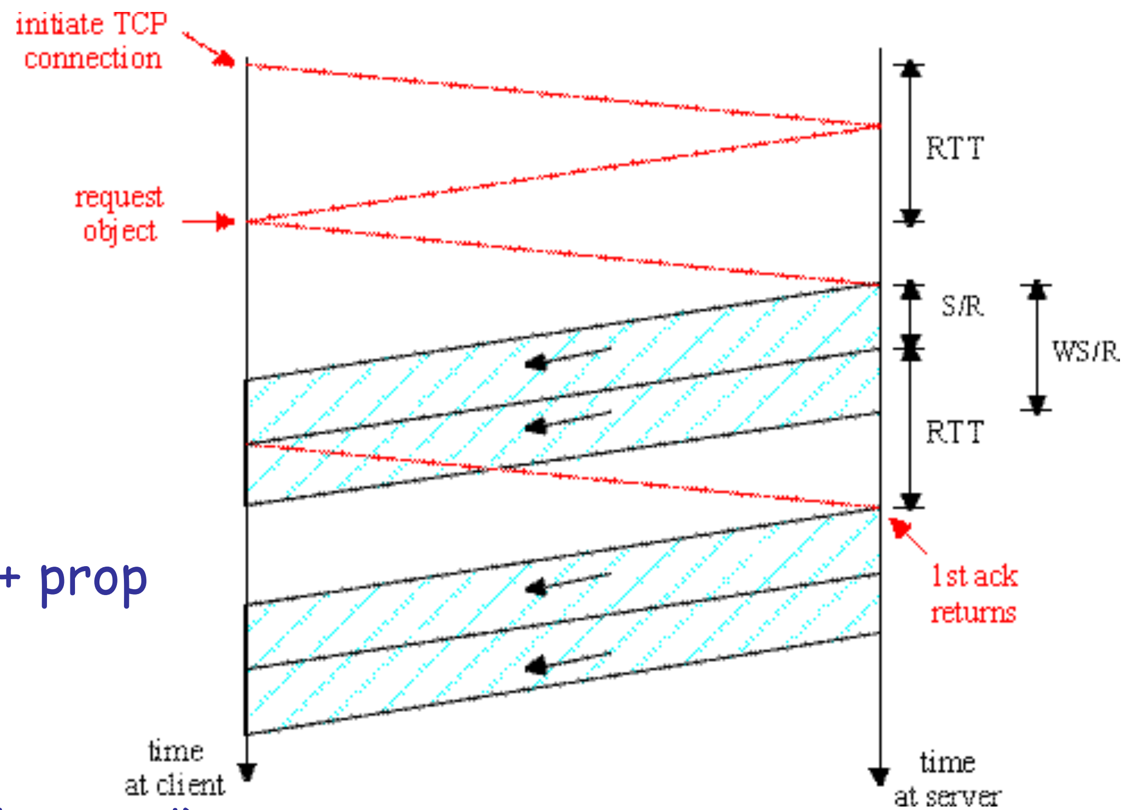
$= RTT + \frac{1}{2} RTT + \frac{1}{2} RTT + D/R$



Case 2: Window is "Too Small"

"Too small" means that time to send window is smaller than time to get first ACK:

More precisely:
$(W*S)/R < RTT + S/R$:



delay = handshake + request + prop + transmt + waiting time

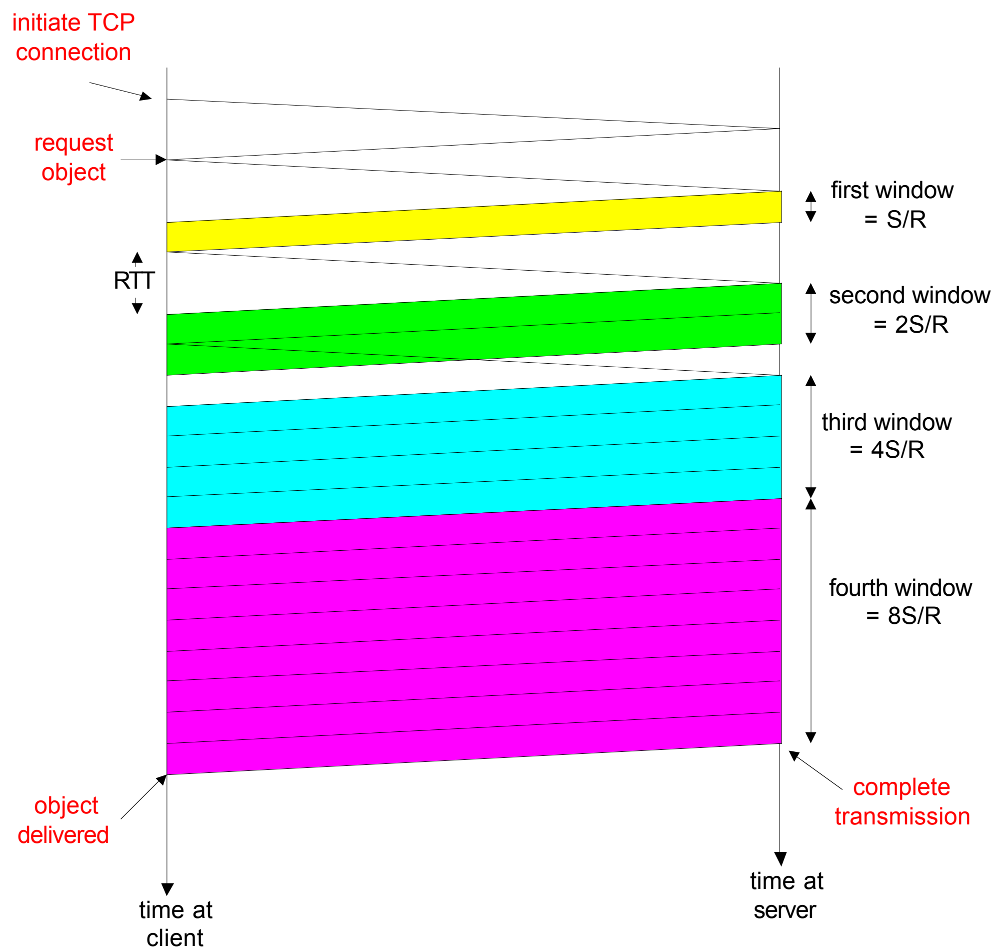
$= RTT + \frac{1}{2} RTT + \frac{1}{2} RTT + L/R + \#rounds(\text{wait time each round})$

$= 2RRT + D/R + (K-1)[\text{total\_round\_time} - \text{time\_sending}]$

$= 2RTT + D/R + (K-1)[S/R + RTT - (W*S)/R]$

IS TCP Window Big Enough or Too Small?

Both!   It starts small and grows exponentially with slow start!



Need to figure out how many idle periods.
Need to figure out how much idle time each period.

Resulting Model For Basic TCP Delay

delay = handshake + request + prop + transmt + waiting time

= RTT + $\frac{1}{2}$ RTT + $\frac{1}{2}$ RTT + L/R + wait_in_round1 + wait_in_round2 + ...

= 2RRT + D/R + [(S/R + RTT) - S/R] + [(S/R + RTT) – 2S/R] + ...

= 2RTT + D/R + K [S/R + RTT] – (S/R + 2S/R + 4S/R + 8S/R + ...)

= 2RTT + D/R + K[S/R + RTT] - (S/R)[1 + 2 + 4 + 8 + ...]

= 2RTT + D/R + K[S/R + RTT] – (S/R)[2^k -1]

TCP Throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Assume long-lived TCP flow
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: 0.75 W/RTT

Problems with Fast Links

An example to illustrate problems

- Consider the impact of high speed links:
  - 1500 byte segments,
  - 100ms RTT
  - 10 Gb/s throughput
- What is the required window size?
  - Throughput = .75 W/RTT
    - (probably a good formula to remember)
  - Requires window size W = 83,333 in-flight segments

Example (Cont.)

- 10 Gb/s throughput requires window size W = 83,333 in-flight segments
- TCP assumes every loss is due to congestion
  - Generally safe assumption for reasonable window size.
- (Magic) Formula to relate loss rate to throughput:

$$\text{Throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

Throughput of 10 Gb/s with MSS of 1500 bytes gives:

- ➜ $L = 2 \cdot 10^{-10}$

  i.e. *can only lose one in 5,000,000,000 segments!*

- We need new versions of TCP for high-speed nets (topic for later discussion)

What's Next

- Read Chapter 1, 2, 3, 4.1-4.3, and 5.1-5.2
- Next Lecture Topics from Chapter 6.4 and 6.5
  - Congestion Control
- Homework
  - Due ***Friday*** in recitation
- Project 3
  - Posted on the course webiste