Transport Protocols

Reading: Sec. 2.5, 5.1 – 5.2, 6.1 – 6.4

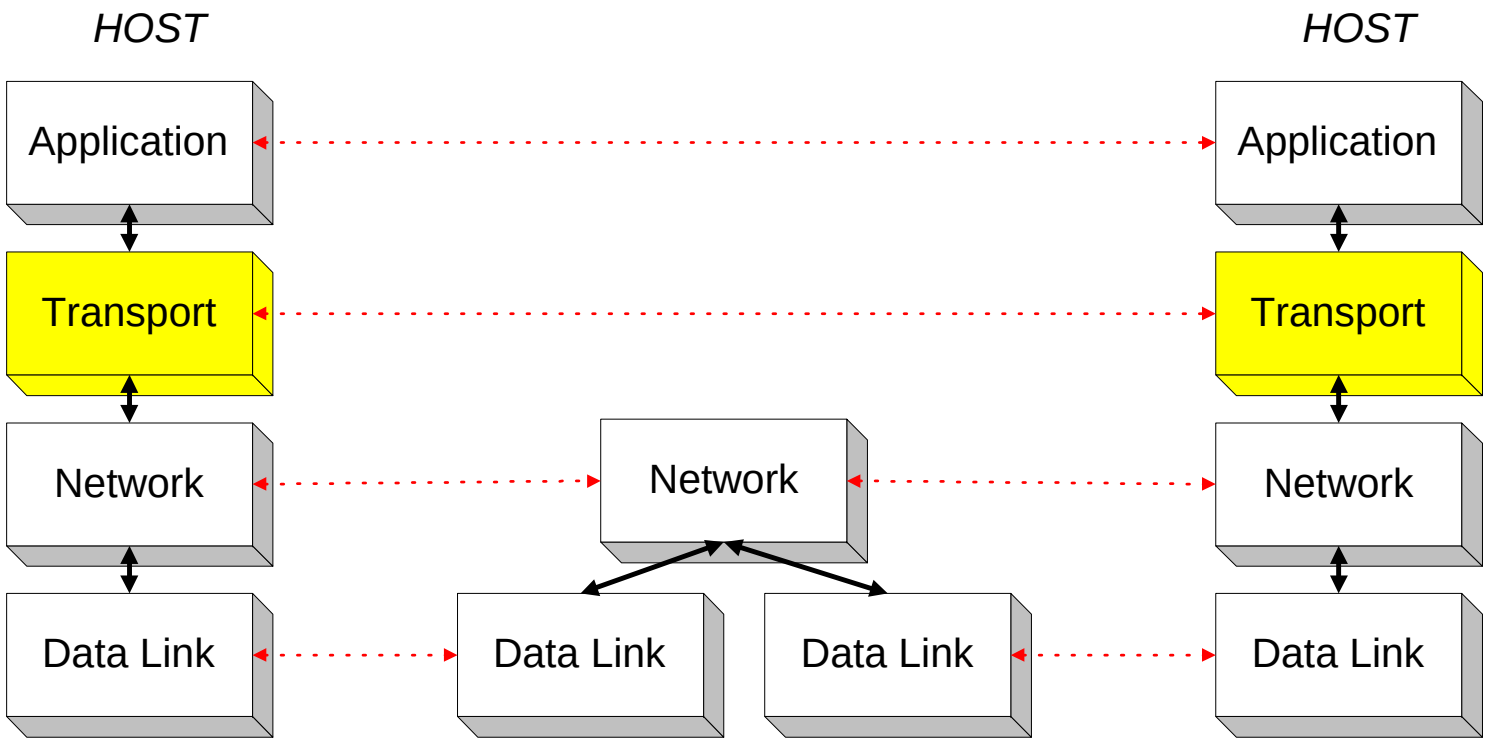
Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

Goals for Today's Lecture

- Principles underlying transport-layer services
  - (De)multiplexing
  - Reliable delivery
  - Flow control
  - Congestion control
- Transport-layer protocols in the Internet
  - User Datagram Protocol (UDP)
    - Simple (unreliable) message delivery
    - Realized by a SOCK_DGRAM socket
  - Transmission Control Protocol (TCP)
    - Reliable bidirectional stream of bytes
    - Realized by a SOCK_STREAM socket

Transport Layer

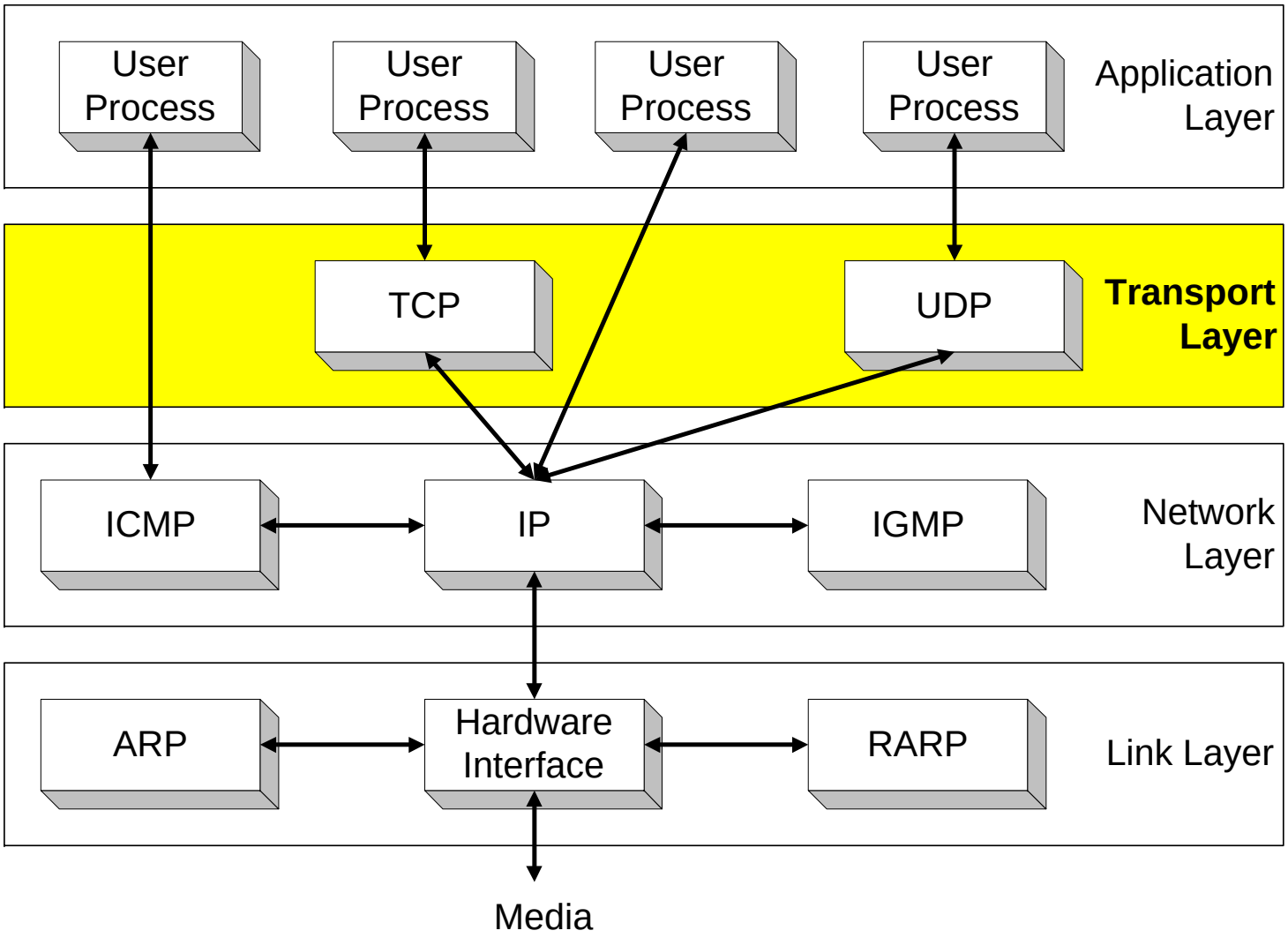
- Transport layer protocols are end-to-end protocols
- They are only implemented at the end hosts



Role of Transport Layer

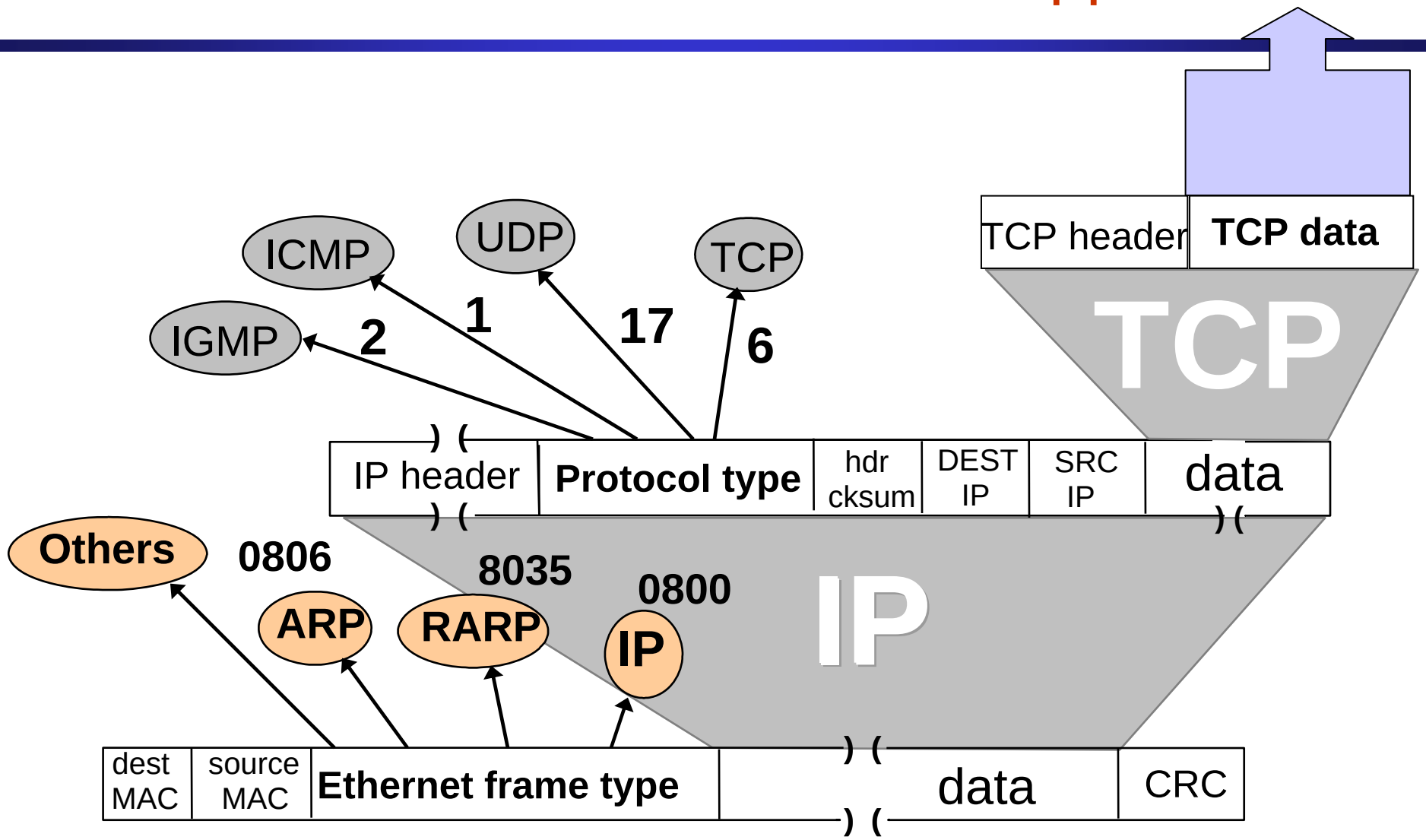
- Application layer
  - Between applications (e.g., browsers and servers)
  - E.g., HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP)
- Transport layer
  - Between processes (e.g., sockets)
  - Relies on network layer and serves the application layer
  - E.g., TCP and UDP
- Network layer
  - Between nodes (e.g., routers and hosts)
  - Hides details of the link technology (e.g., IP)

Context


User Process
User Process
User Process
User Process
Application Layer
TCP
UDP
Transport Layer
ICMP
IP
IGMP
Network Layer
ARP
Hardware Interface
RARP
Link Layer
Media

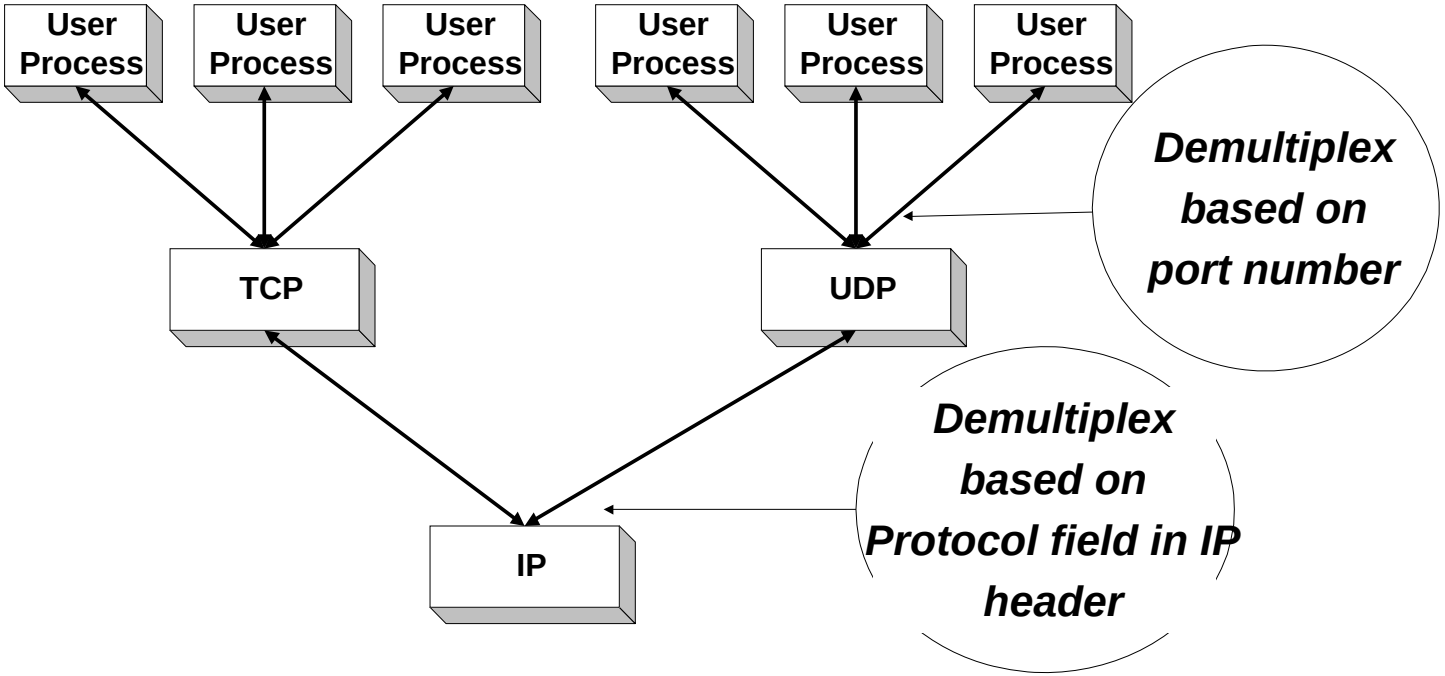
Context

To which application ?

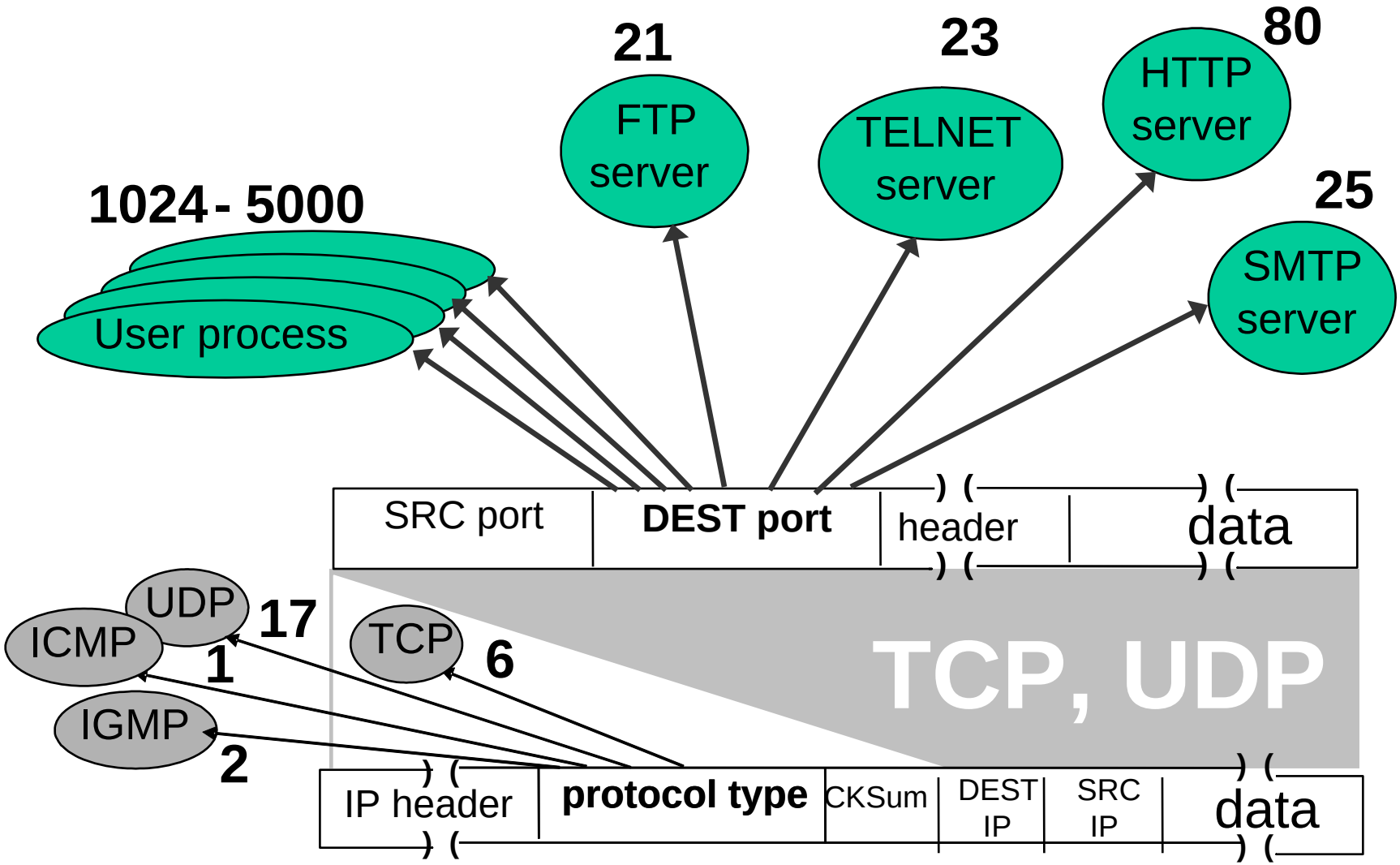


Port Numbers

- UDP (and TCP) use port numbers to identify applications
- A globally unique address at the transport layer (for both UDP and TCP) is a tuple **<IP address, port number>**
- There are 65,535 UDP ports per host.



Port Numbers

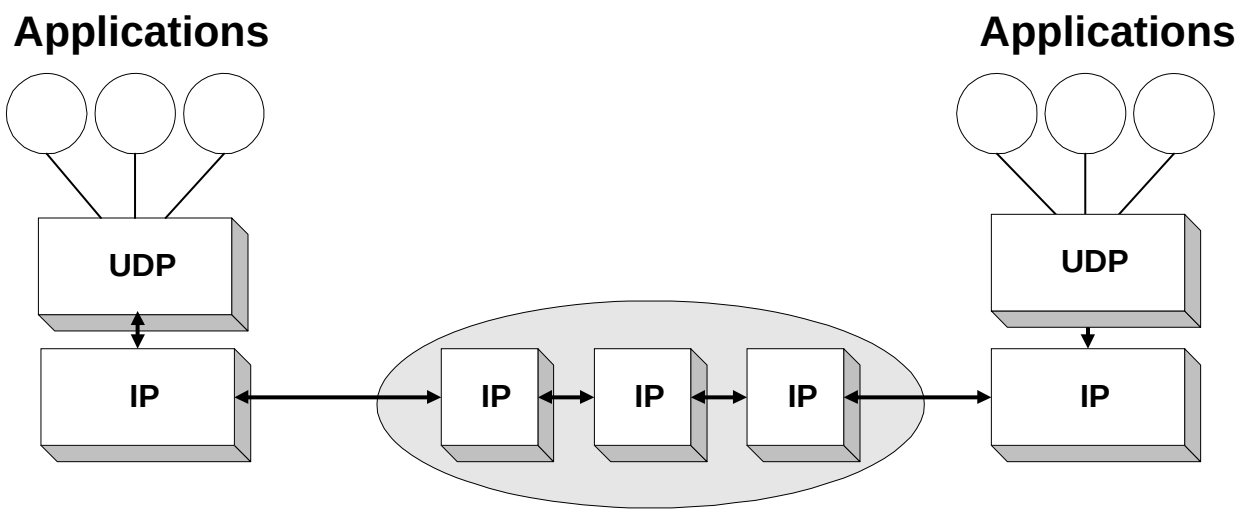


Internet Transport Protocols

- UDP → User Datagram Protocol
  - Simple, unreliable
- TCP → Transport Control Protocol
  - Complex, reliable

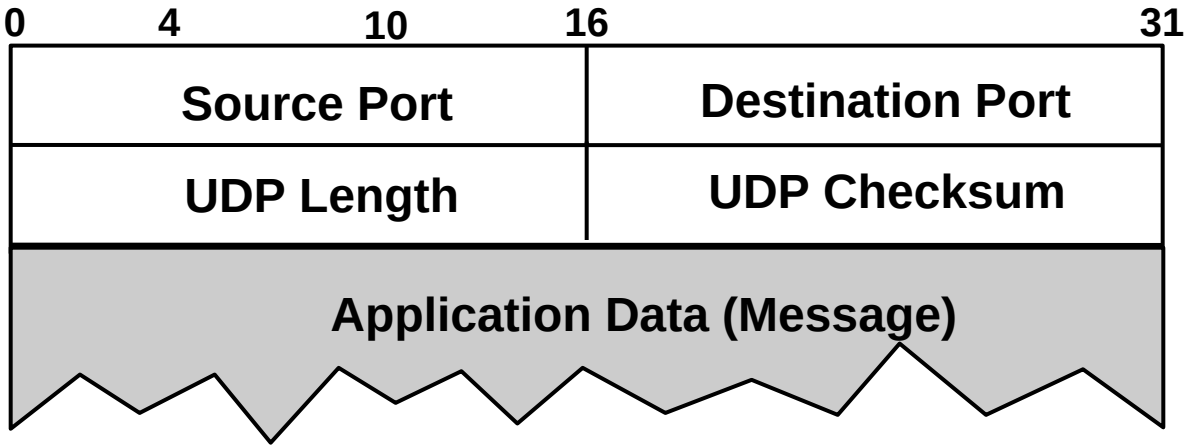
User Datagram Protocol (UDP)

- UDP supports unreliable transmissions of datagrams
- UDP merely extends the host-to-to-host delivery service of IP datagram to an application-to-application service
- The only thing that UDP adds is multiplexing and demultiplexing

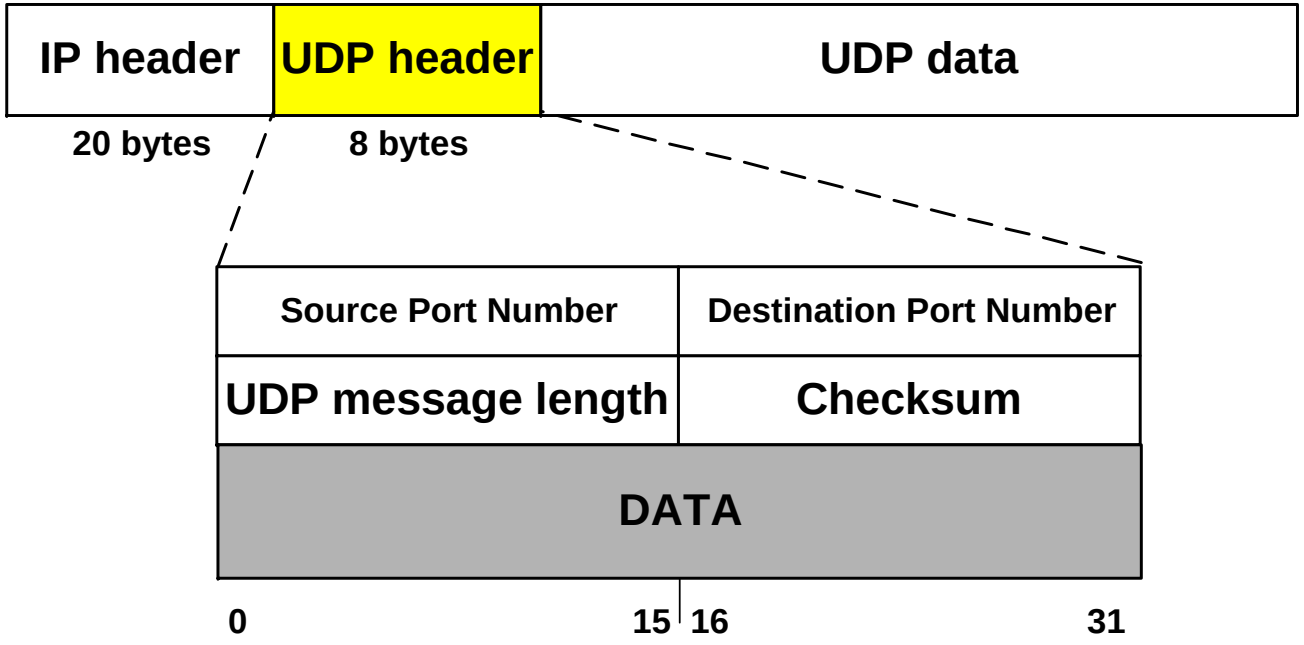


User Datagram Protocol (UDP)

- Datagram messaging service
  - Demultiplexing of messages: port numbers
  - Detecting corrupted messages: checksum
- Lightweight communication between processes
  - Send messages to and receive them from a socket
  - Avoid overhead and delays of ordered, reliable delivery



UDP Format



Port numbers identify sending and receiving applications (processes). Maximum port number is 2^{16}-1= 65,535

Message Length is at least 8 bytes (I.e., Data field can be empty) and at most 65,535

Checksum is for header (of UDP and some of the IP header fields)

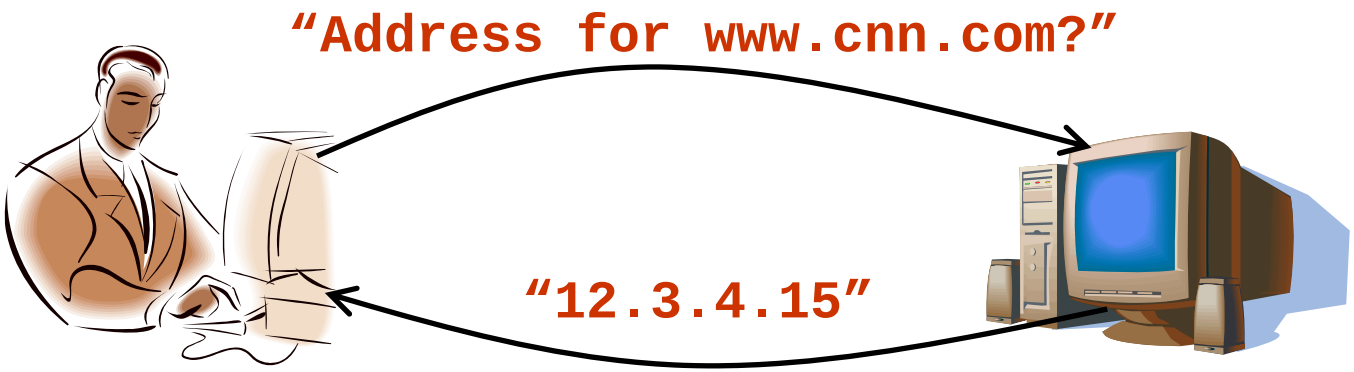
Why Would Anyone Use UDP?

- Fine control over what data is sent and when
  - As soon as an application process writes into the socket
  - … UDP will package the data and send the packet
- No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
- No connection state
  - No allocation of buffers, parameters, sequence #s, etc.
  - … making it easier to handle many active clients at once
- Small packet header overhead
  - UDP header is only eight-bytes long

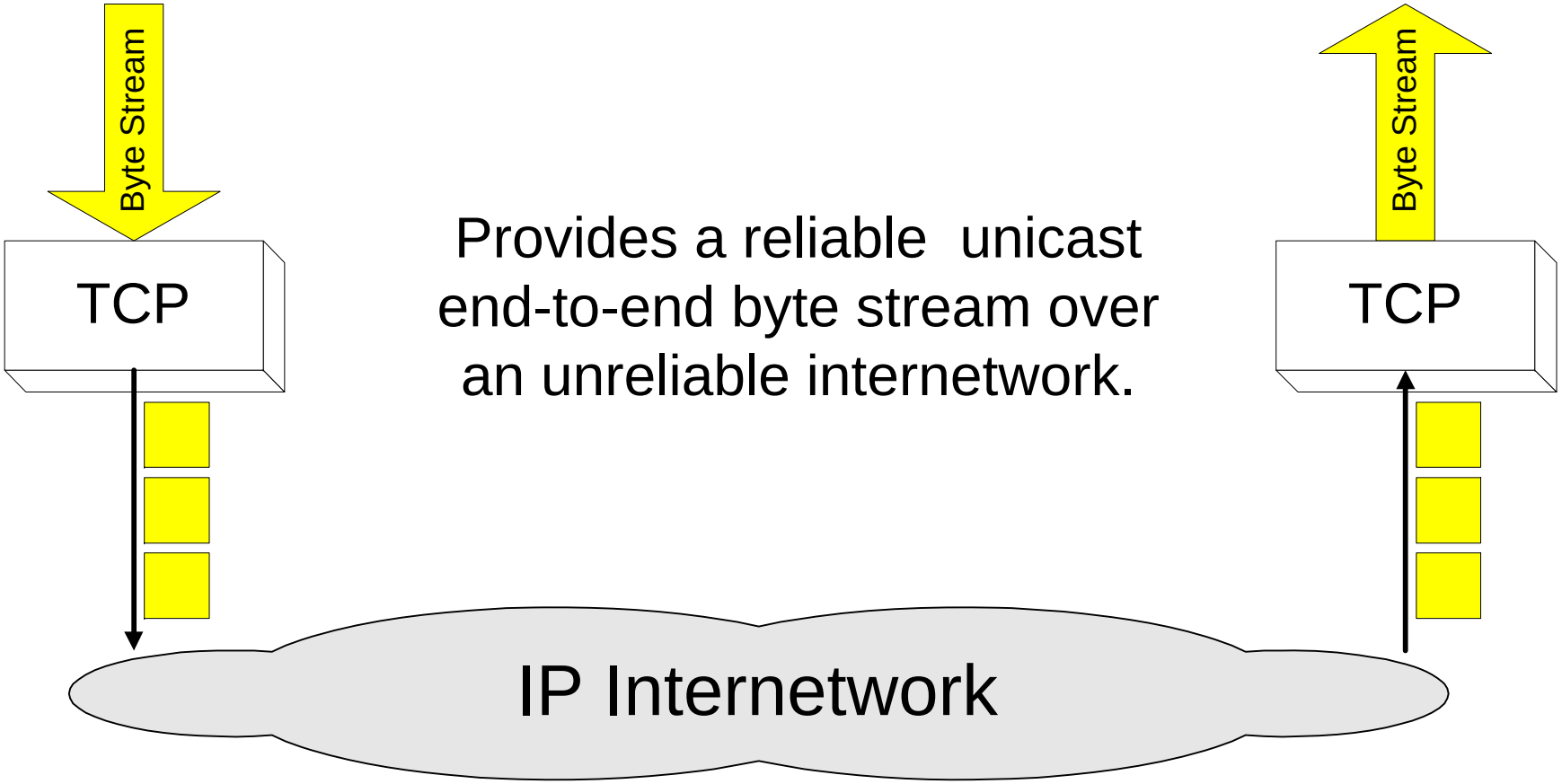
Popular Applications That Use UDP

- Multimedia streaming
  - Retransmitting lost/corrupted packets is not worthwhile
  - By the time the packet is retransmitted, it's too late
  - E.g., telephone calls, video conferencing, gaming

- Simple query protocols like Domain Name System
  - Overhead of connection establishment is overkill
  - Easier to have the application retransmit if needed



Transmission Control Protocol (TCP)



Provides a reliable unicast end-to-end byte stream over an unreliable internetwork.

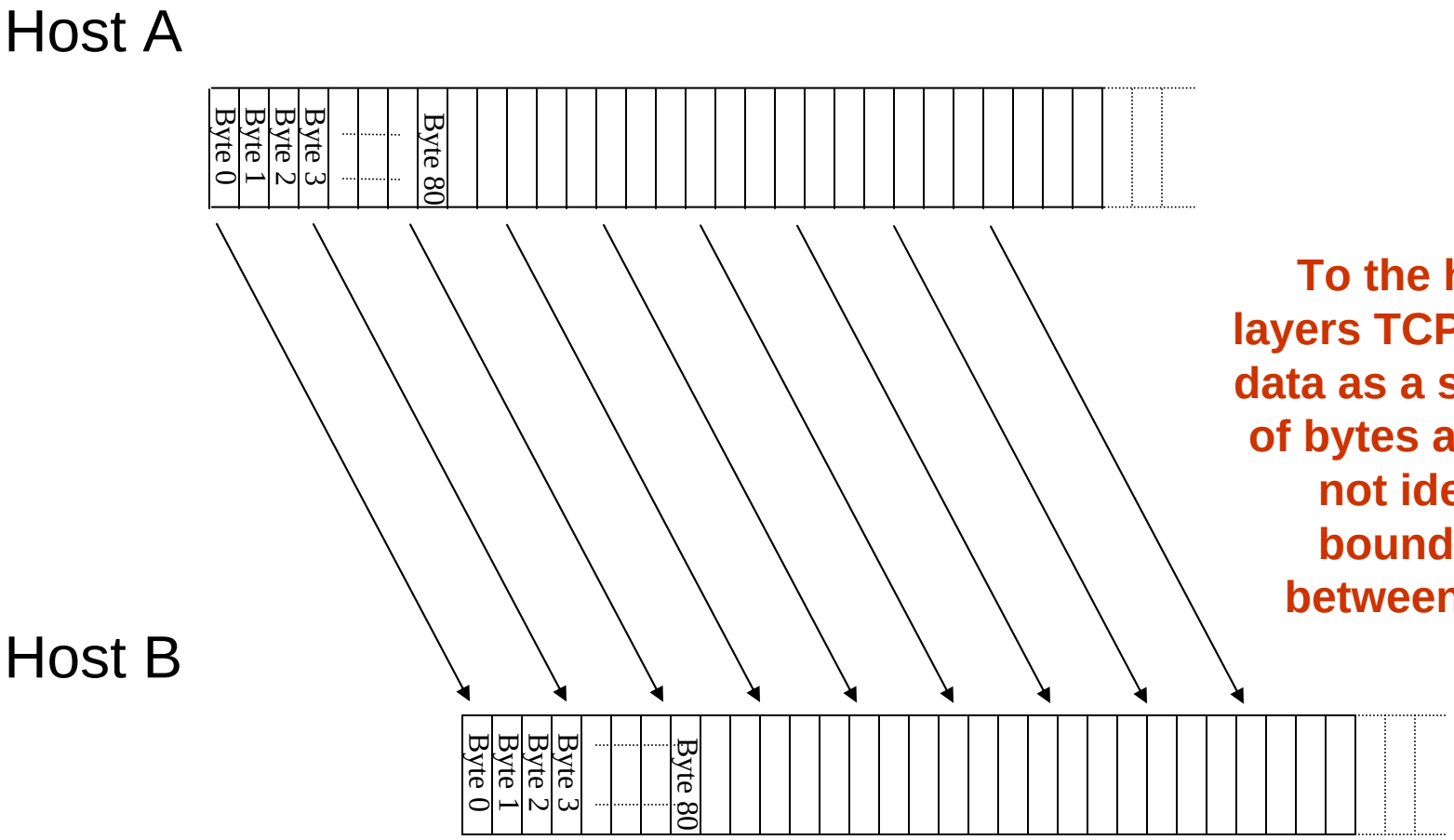
Transmission Control Protocol (TCP)

- Stream-of-bytes service
  - Sends and receives a stream of bytes, not messages
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Sequence numbers to detect losses and reorder data
  - Acknowledgments & retransmissions for reliable delivery
- Connection oriented
  - Explicit set-up and tear-down of TCP session
- Flow control
  - Prevent overflow of the receiver's buffer space
- Congestion control (next class!)
  - Adapt to network congestion for the greater good

Breaking a Stream of Bytes into TCP Segments

Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

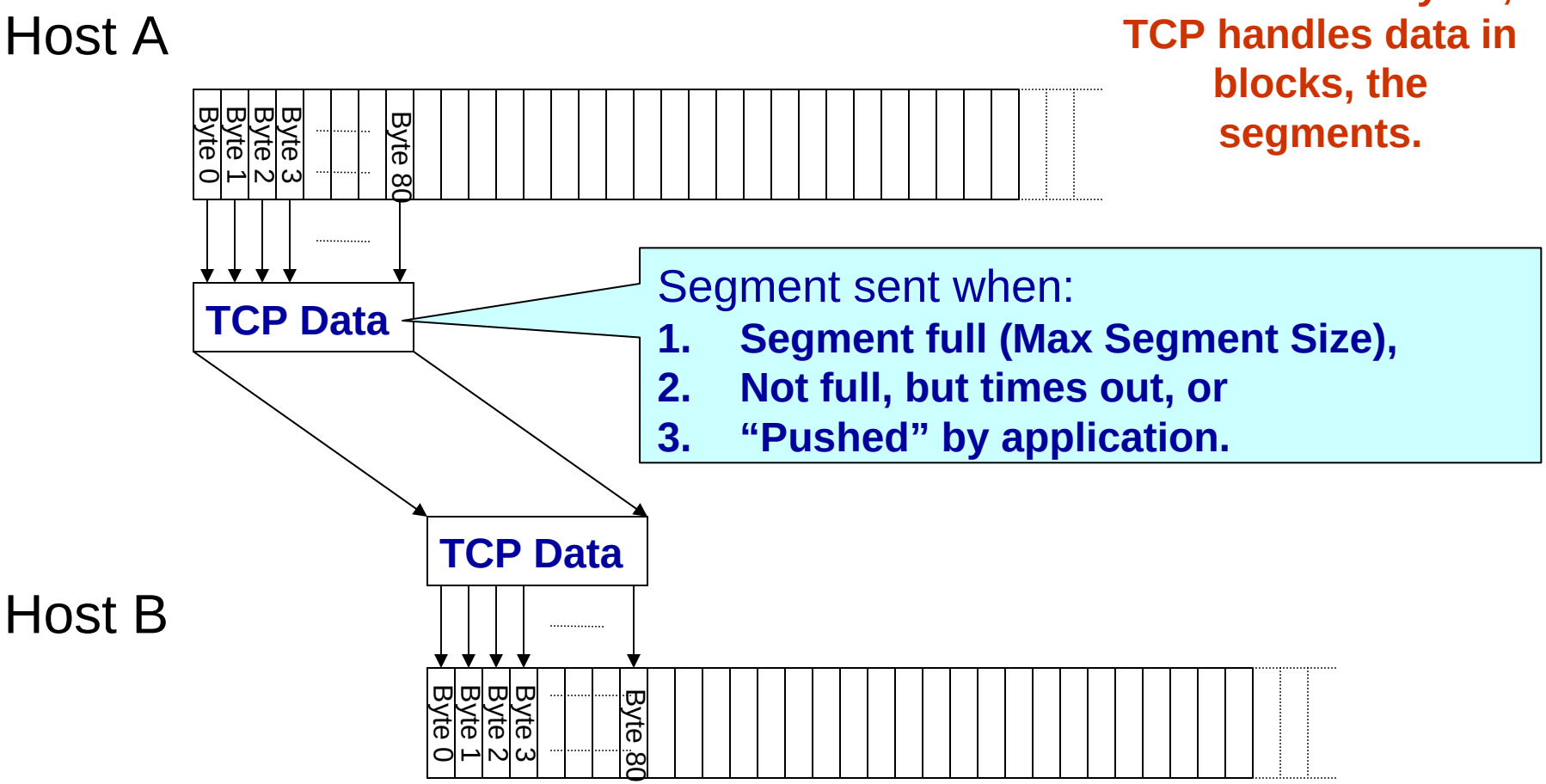
TCP “Stream of Bytes” Service



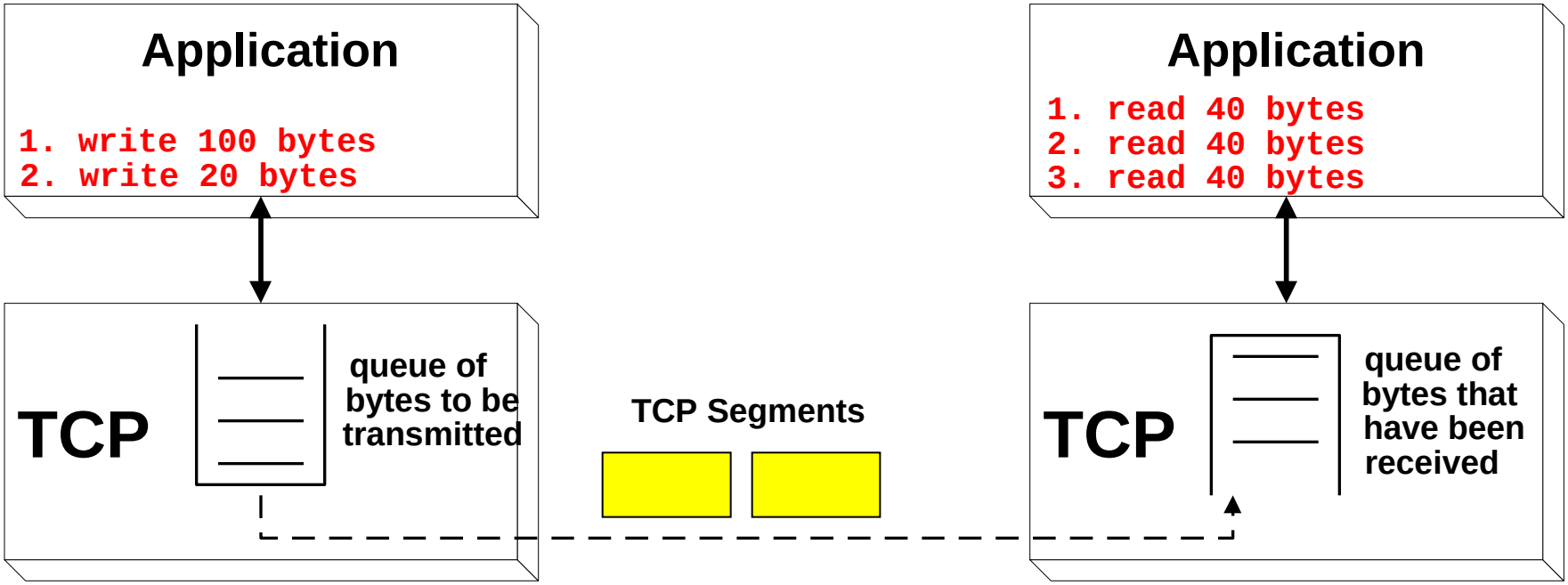
To the higher layers TCP handles data as a sequence of bytes and does not identify boundaries between bytes

…Emulated Using TCP "Segments"

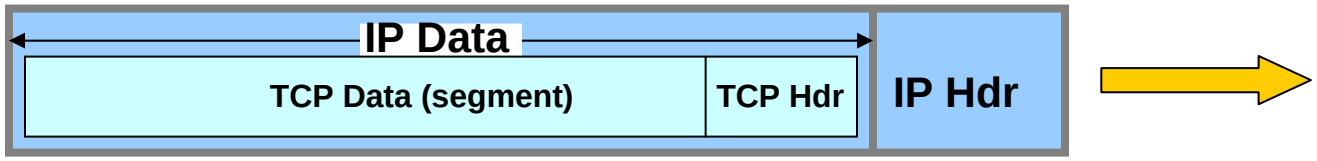
To the lower layers, TCP handles data in blocks, the segments.



Putting It Together …



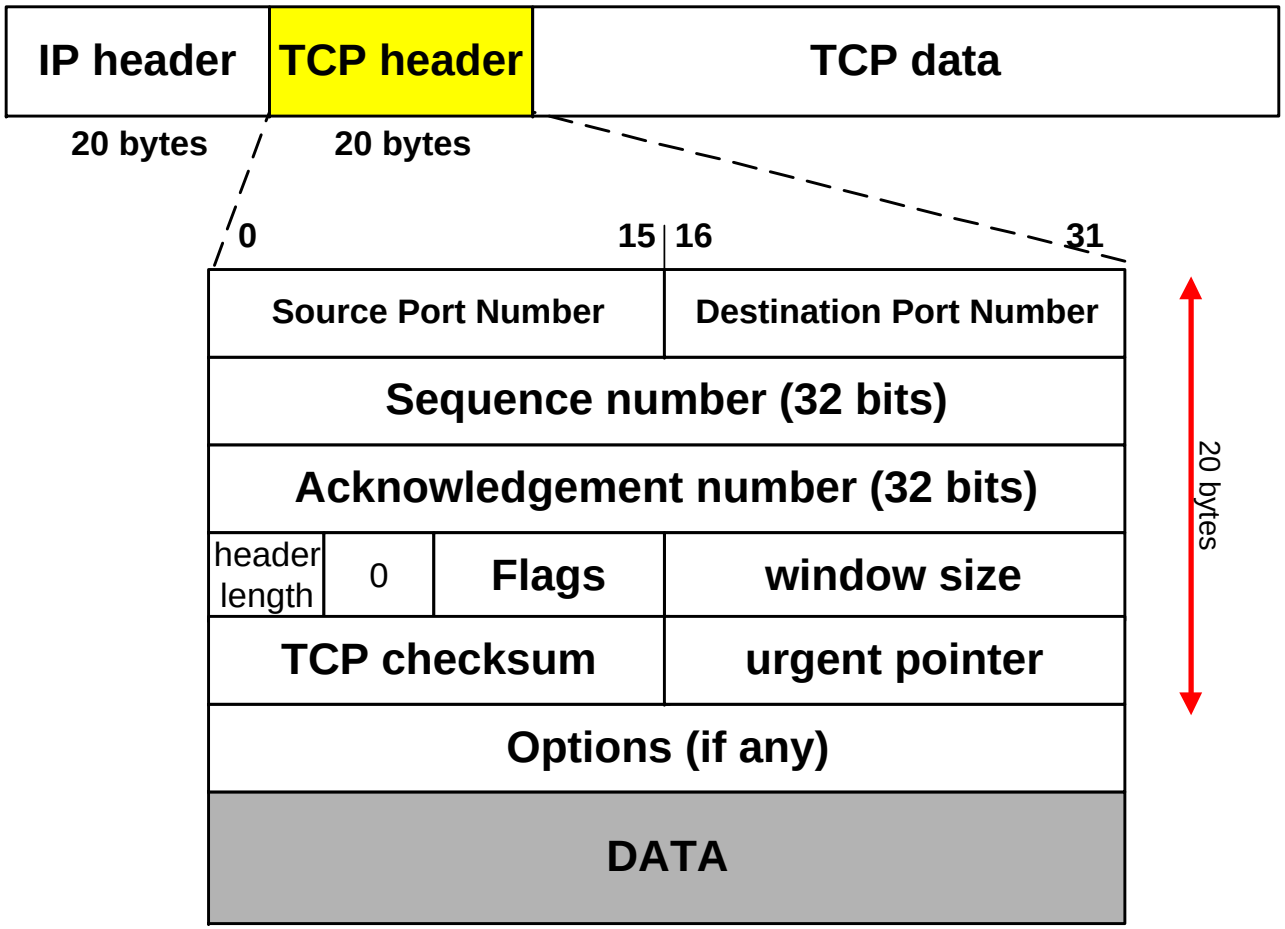
TCP Segment



- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream

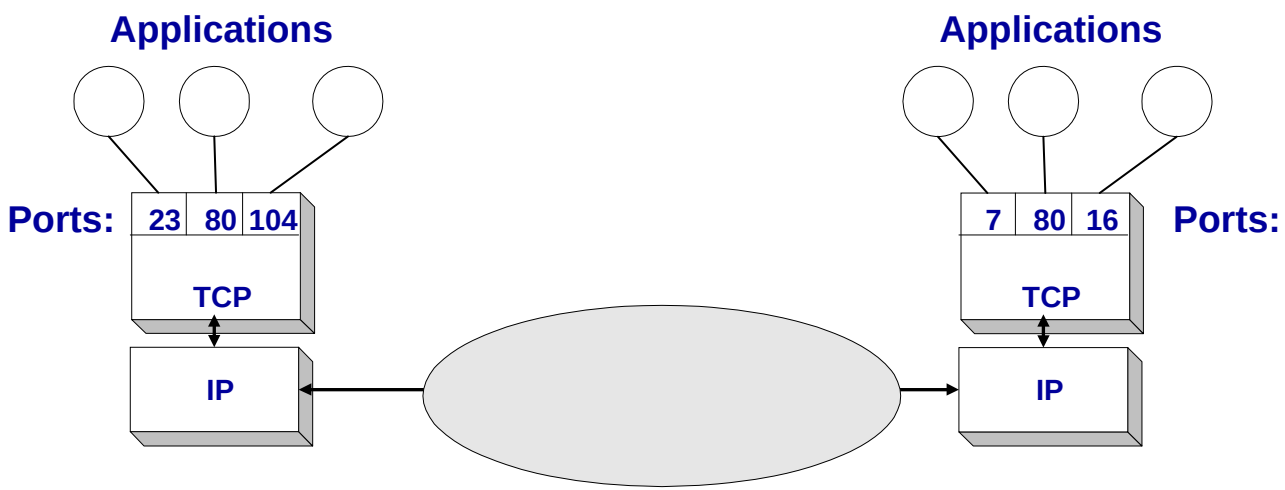
TCP Format

TCP segments have a 20 byte header with >= 0 bytes of data.

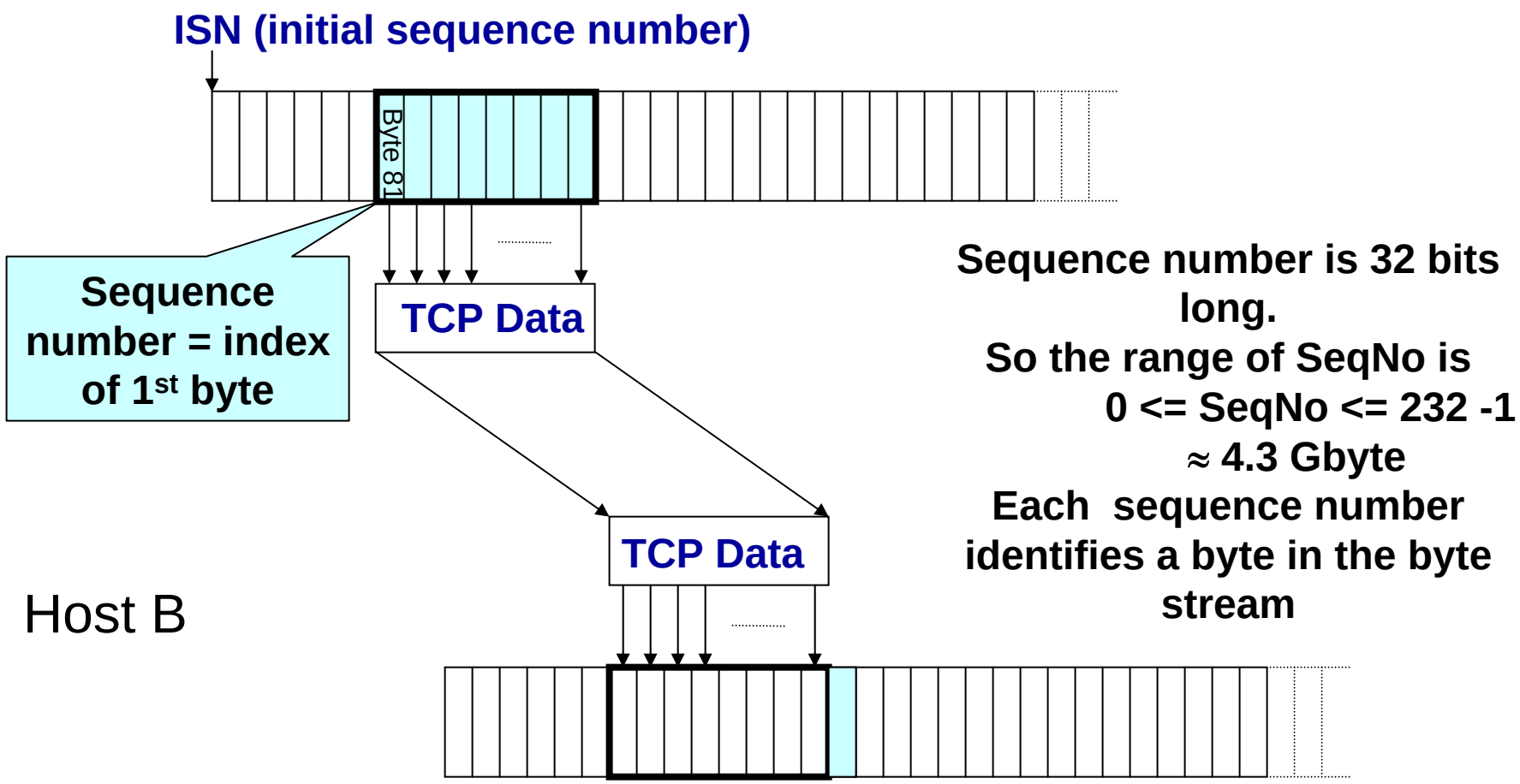


TCP Header Fields

- **Port Numbers:**
  - A port number identifies the endpoint of a connection.
  - A pair <IP address, port number> identifies one endpoint of a connection.
  - Two pairs <client IP address, server port number> and <server IP address, server port number> identify a TCP connection.



TCP Header Fields -- Sequence Number



Exercise

Suppose a TCP connection is transferring a file of 5000 bytes. The first byte is numbered 501. What are the sequence numbers for each segment if data is sent in five segments, each carrying 1000 bytes?

Segment	Sequence Number
1st	
2nd	
3rd	
4th	
5th	

Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not a de facto ISN of 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … and there is a chance an old packet is still in flight
  - … and might be associated with the new connection
- So, TCP requires changing the ISN over time
  - Set from a 32-bit clock that ticks every 4 microseconds
  - … which only wraps around once every 4.55 hours!
- But, this means the hosts need to exchange ISNs

TCP Header Fields – ACK Number

Reliable Delivery on a Lossy Channel With Bit Errors

Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

Challenges of Reliable Data Transfer

- Over a perfectly reliable channel
  - All of the data arrives in order, just as it was sent
  - Simple: sender sends data, and receiver receives data
- Over a channel with *bit errors*
  - All of the data arrives in order, but some bits corrupted
  - Receiver detects errors and says “please repeat that”
  - Sender retransmits the data that were corrupted
- Over a *lossy* channel with *bit errors*
  - Some data are missing, and some bits are corrupted
  - Receiver detects errors but cannot always detect loss
  - Sender must wait for acknowledgment (“ACK” or “OK”)
  - … and retransmit data after some time if no ACK arrives

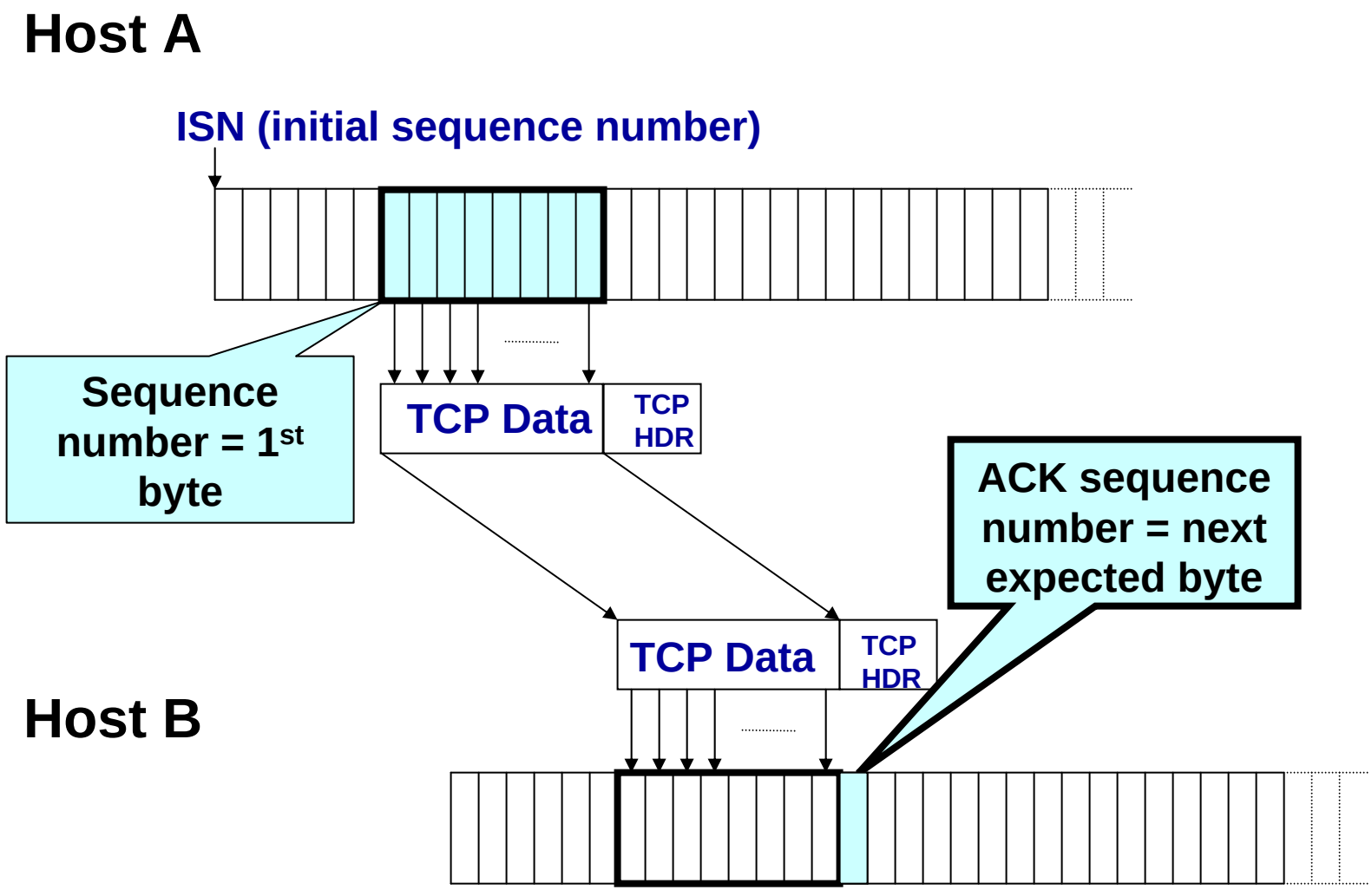
TCP Support for Reliable Delivery

- Acknowledgments from receiver
  - Positive: “okay” or “ACK”
  - Negative: “please repeat that” or “NACK”
- Timeout by the sender (“stop and wait”)
  - Don’t wait indefinitely without receiving some response
  - … whether a positive or a negative acknowledgment
- Retransmission by the sender
  - After receiving a “NACK” from the receiver
  - After receiving no feedback from the receiver

TCP Support for Reliable Delivery

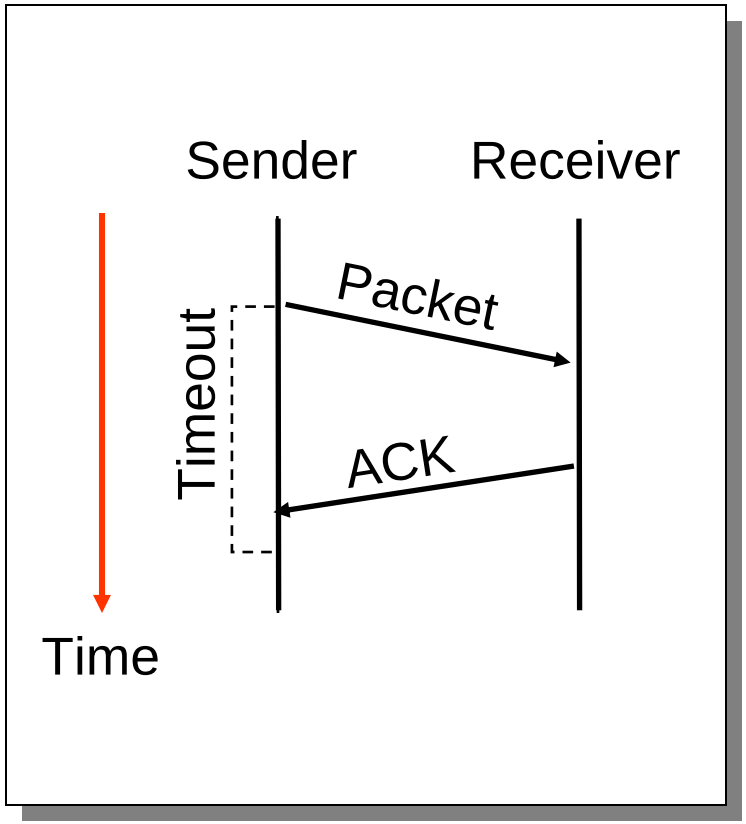
- **Detect bit errors:** checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- **Detect missing data:** sequence number
  - Used to detect a gap in the stream of bytes
  - ... and for putting the data back in order
- **Recover from lost data:** retransmission
  - Sender retransmits lost or corrupted data
  - Two main ways to detect lost packets

TCP Acknowledgments

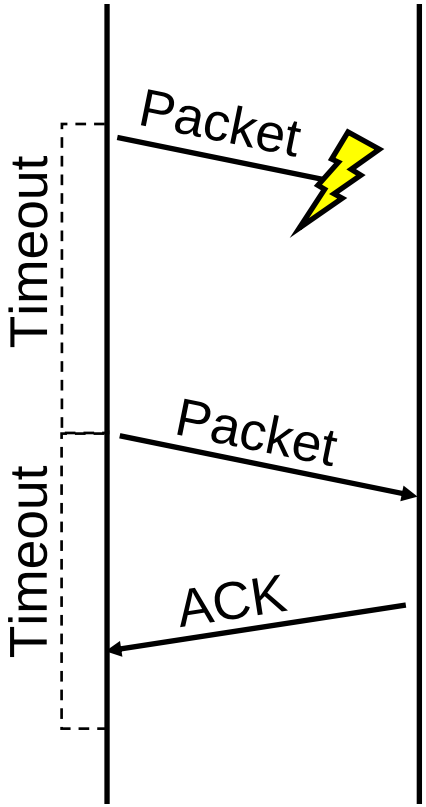


Automatic Repeat reQuest (ARQ)

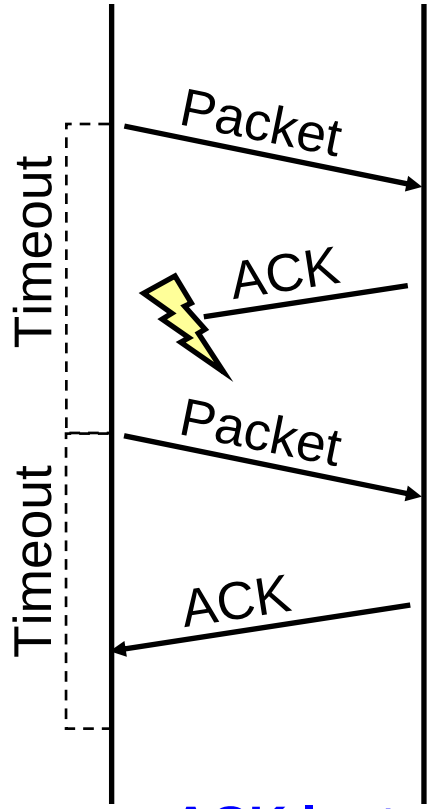
- Automatic Repeat reQuest
  - Receiver sends acknowledgment (ACK) when it receives packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period
- Simplest ARQ protocol
  - Stop and wait
  - Send a packet, stop and wait until ACK arrives



Reasons for Retransmission

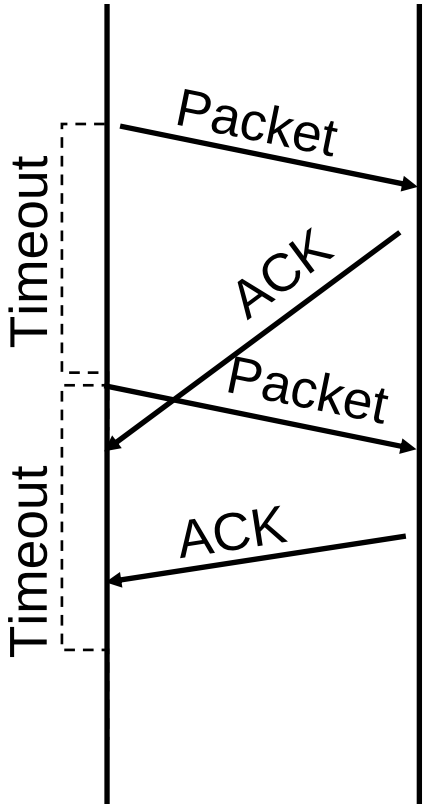


Packet lost



ACK lost

DUPLICATE PACKET



Early timeout

DUPLICATE PACKETS

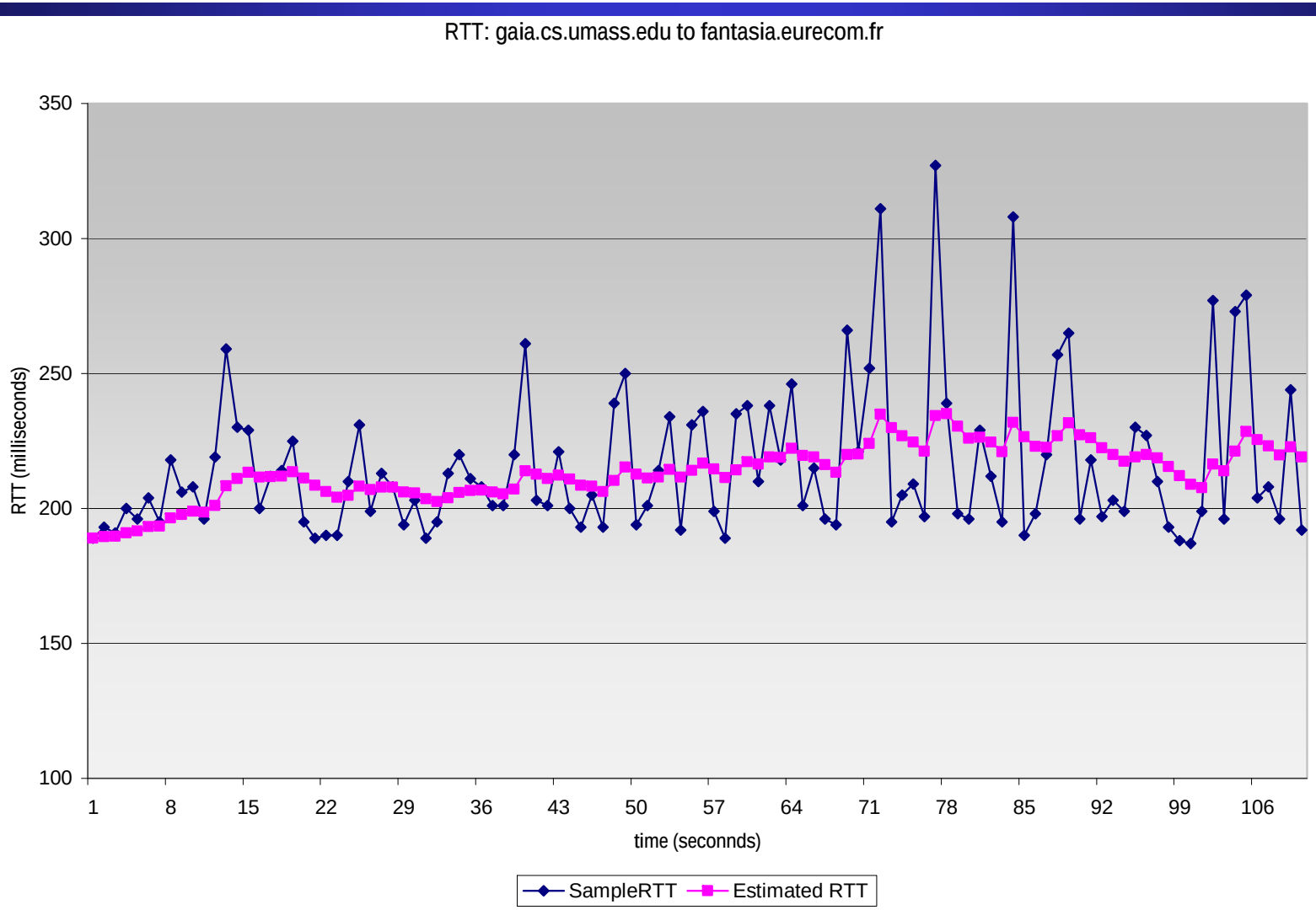
How Long Should Sender Wait?

- Sender sets a timeout to wait for an ACK
  - Too short: wasted retransmissions
  - Too long: excessive delays when packet lost
- TCP sets timeout as a function of the RTT
  - Expect ACK to arrive after an "round-trip time"
  - … plus a fudge factor to account for queuing
- But, how does the sender know the RTT?
  - Can estimate the RTT by watching the ACKs
  - Smooth estimate: keep a running average of the RTT

    $$\text{EstimatedRTT} = \alpha * \text{EstimatedRTT} + (1 - \alpha) * \text{SampleRTT}$$

  - Compute timeout: TimeOut = 2 * EstimatedRTT

Example RTT Estimation

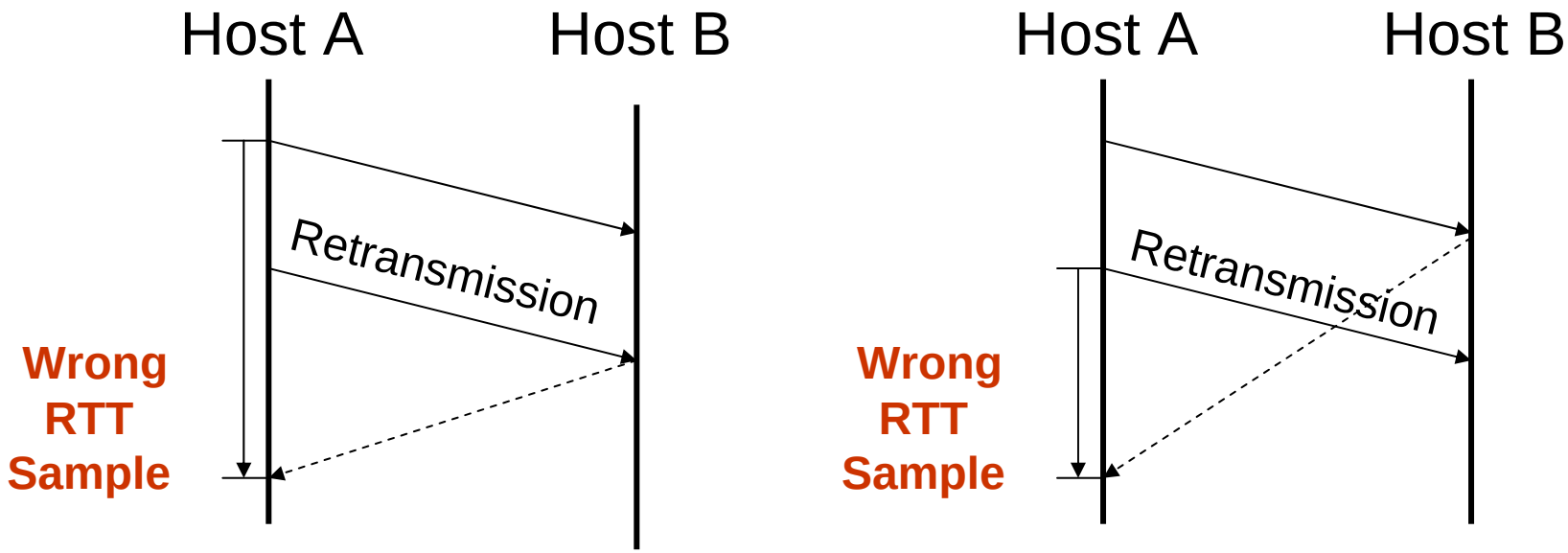


A Flaw in This Approach

- An ACK doesn't really acknowledge a transmission
  - Rather, it acknowledges receipt of the data
- Consider a retransmission of a lost packet
  - If you assume the ACK goes with the 1st transmission
  - … the SampleRTT comes out way too large
- Consider a duplicate packet
  - If you assume the ACK goes with the 2nd transmission
  - … the Sample RTT comes out way too small
- Simple solution in the Karn/Partridge algorithm
  - Only collect samples for segments sent one single time

Sample RTT (Karn's Algorithm)

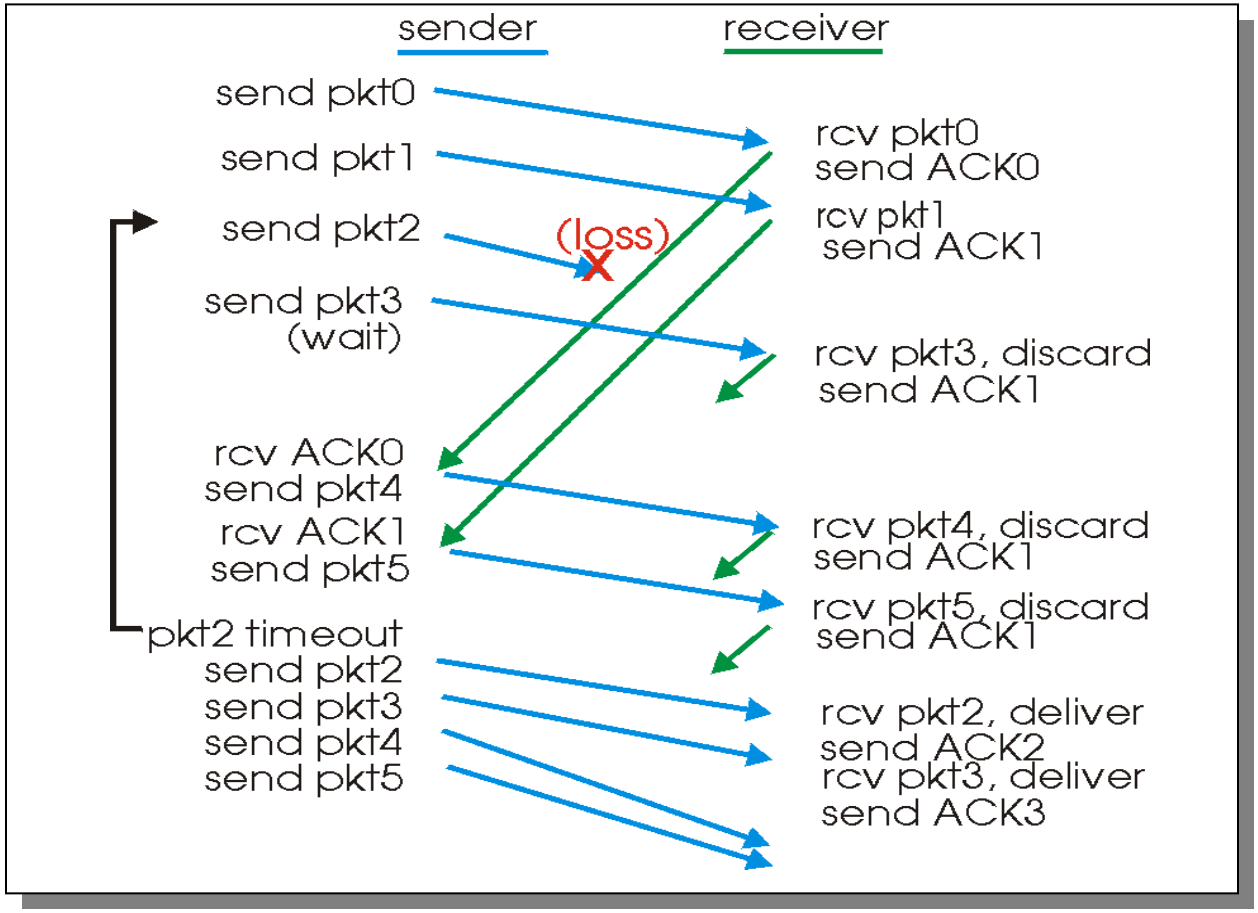
- Record time when TCP segment sent
- Record time when ACK arrives and compute the difference



- On retransmissions
  - do not take samples, double the timeout value

Still, Timeouts are Inefficient

- Timeout-based retransmission
  - Sender transmits a packet and waits until timer expires
  - … and then retransmits from the lost packet onward



Fast Retransmission

- Better solution possible under sliding window
  - Although packet n might have been lost
  - … packets n+1, n+2, and so on might get through
- Idea: have the receiver send ACK packets
  - ACK says that receiver is still awaiting n^{th} packet
    - And *repeated* ACKs suggest later packets have arrived
  - Sender can view the "duplicate ACKs" as an early hint
    - … that the nth packet must have been lost
    - … and perform the retransmission early
- Fast retransmission
  - Sender retransmits data after the triple duplicate ACK

Effectiveness of Fast Retransmit

- When does Fast Retransmit work best?
  - Long data transfers
    - High likelihood of many packets in flight
  - High window size
    - High likelihood of many packets in flight
  - Low burstiness in packet losses
    - Higher likelihood that later packets arrive successfully
- Implications for Web traffic
  - Most Web transfers are short (e.g., 10 packets)
  - So, often there aren't many packets in flight
  - … making fast retransmit less likely to "kick in"
  - Forcing users to like "reload" more often… ☺

Starting and Ending a Connection: TCP Handshakes

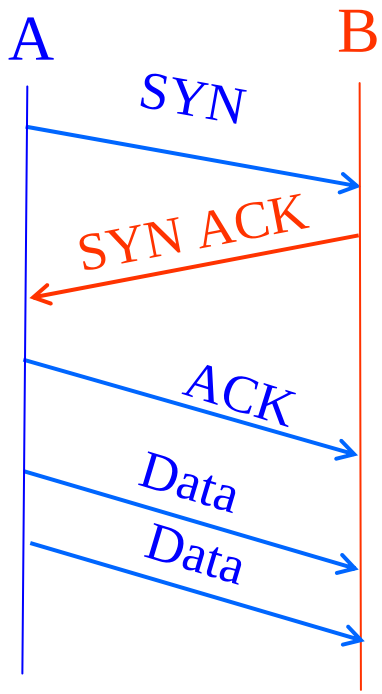
Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

TCP Header

Flags: SYN
FIN
RST
PSH
URG
ACK

Source port			Destination port
Sequence number			
Acknowledgment			
HdrLen	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			
Data			

Establishing a TCP Connection



SYN: Synchronize sequence numbers

Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open) to the host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

Step 1: A's Initial SYN Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

A's port			B's port
A's Initial Sequence Number			
Acknowledgment			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

A tells B it wants to open a connection…

Step 2: B's SYN-ACK Packet

Flags: SYN
FIN
RST
PSH
URG
ACK

B's port			A's port
B's Initial Sequence Number			
A's ISN plus 1			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

Step 3: A's ACK of the SYN-ACK

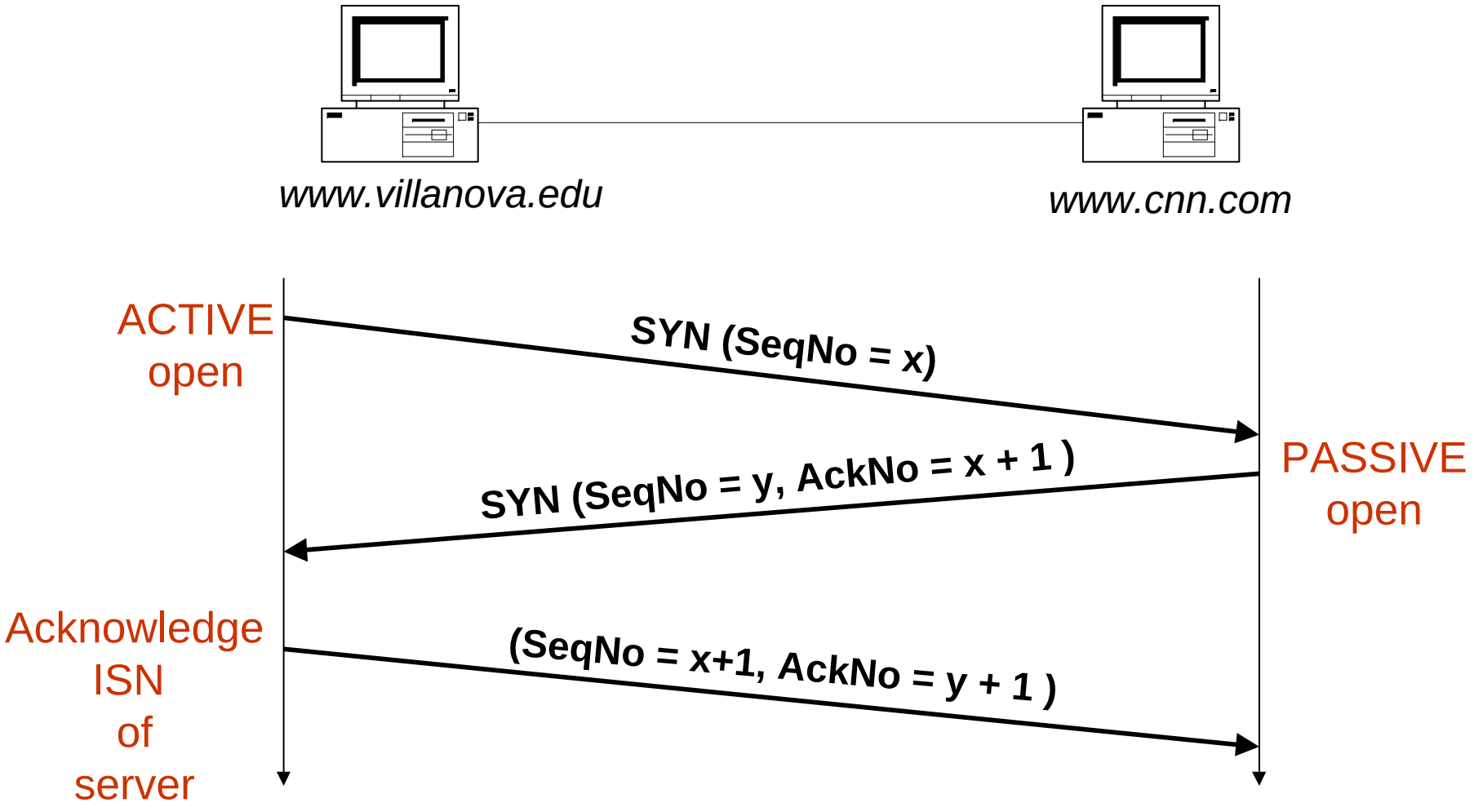
Flags: SYN
FIN
RST
PSH
URG
ACK

A's port			B's port
Sequence number			
B's ISN plus 1			
20	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			

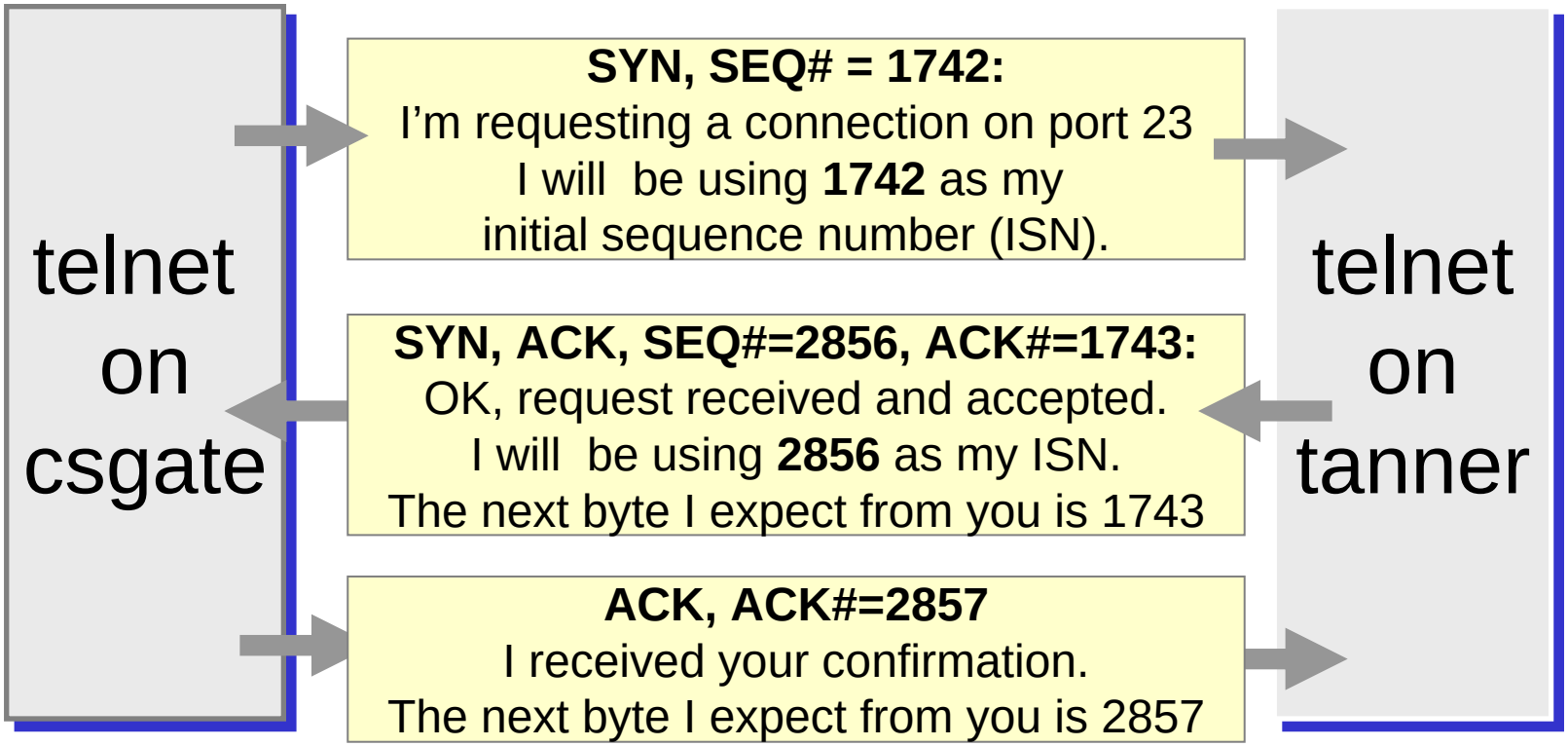
A tells B it is okay to start sending

… upon receiving this packet, B can start sending data

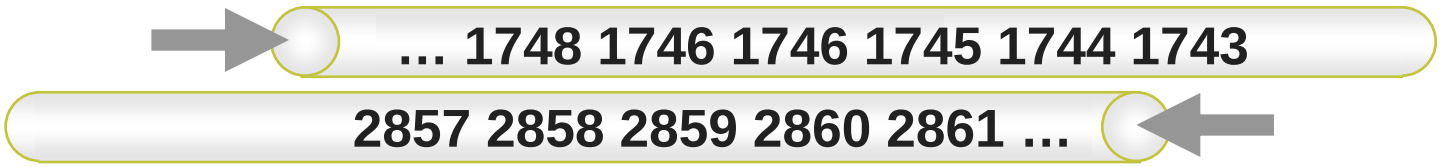
Putting it Together …



Example



Subsequent Data Exchange Phase



Exercise

Suppose Process X sends to Process Y a TCP segment with flags SYN and ACK set to 1 and Acknowledgement number equal to 100. What is A telling B?

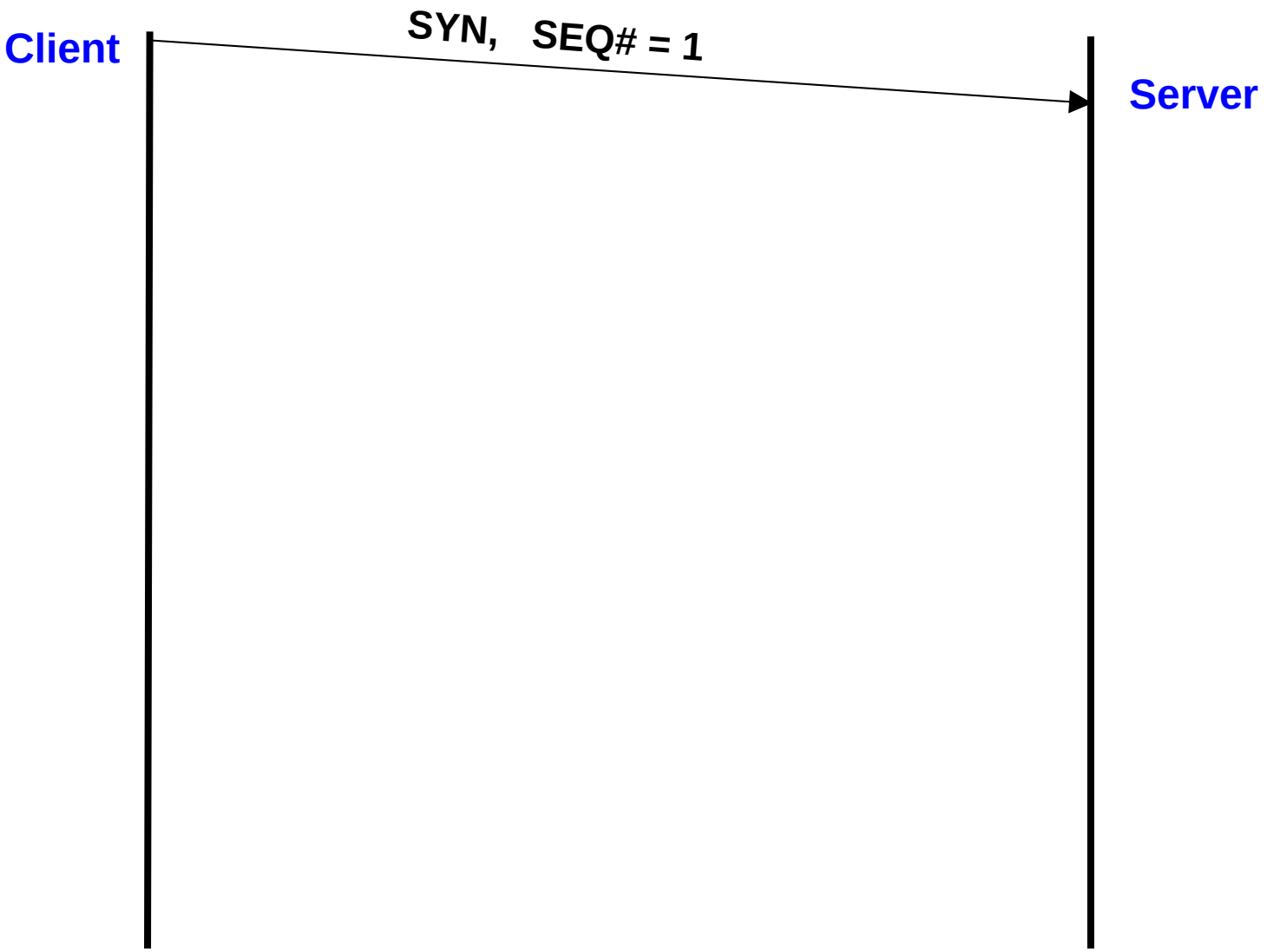
Exercise

Assume the following parameters associated with a TCP connection between a Client and a Server:

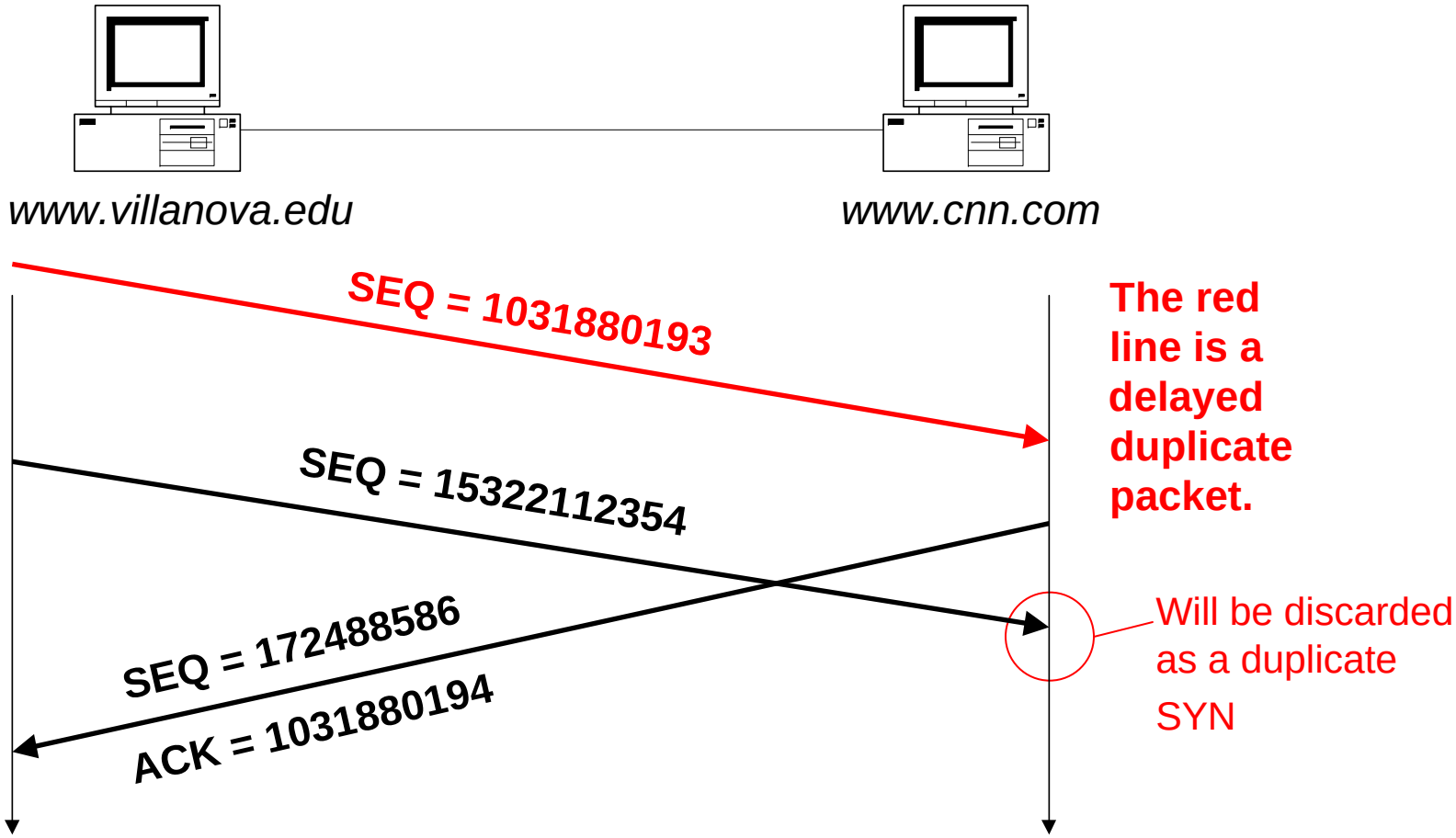
- The MSS (Maximum Segment Size) in both directions is 1000 bytes.
- The ISN (Initial Sequence Number) for Client is 50.
- The ISN for Server is 81.

The Client sends 2000 bytes to the Server and the Server sends 3000 bytes to the client. Give the complete TCP message exchange between client and server. For each segment draw a vector showing the value of the SYN, ACK and FIN bits, with the value of the SEQ (Sequence Number) and the ACK (Acknowledgment Number). Assume no packets are lost and the application consumes the data as soon as it is received.

Exercise(contd.)



Why is a Two-Way Handshake not Enough?



When villanova initiates the data transfer (starting with SeqNo=15322112355), cnn will reject all data.

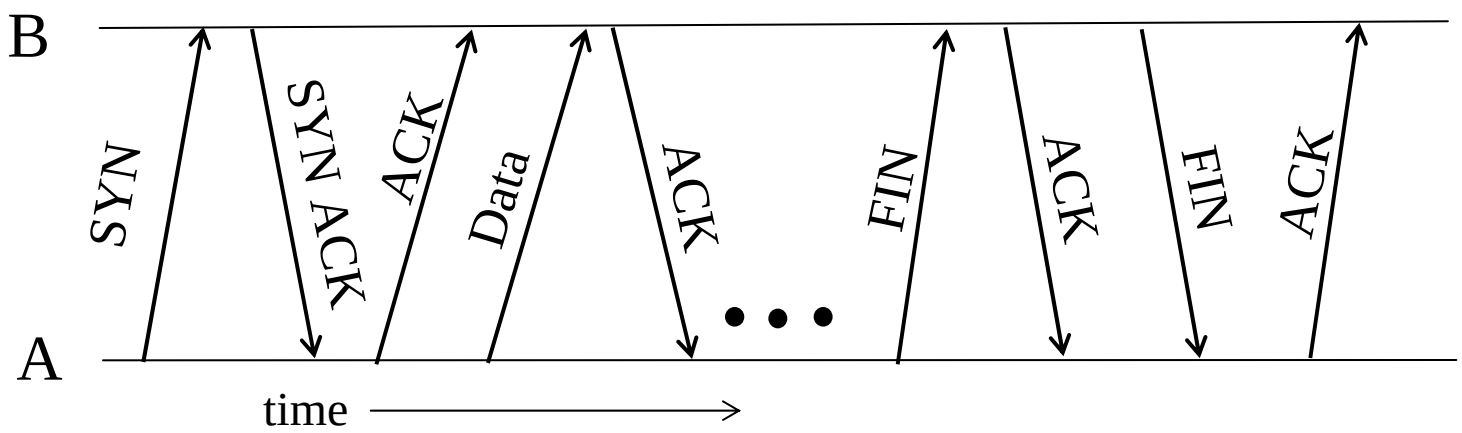
What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or
  - Server rejects the packet (e.g., listen queue is full)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and wait for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - Some TCPs use a default of 3 or 6 seconds

SYN Loss and Web Downloads

- User clicks on a hypertext link
  - Browser creates a socket and does a “connect”
  - The “connect” triggers the OS to transmit a SYN
- If the SYN is lost…
  - The 3-6 seconds of delay may be very long
  - The user may get impatient
  - … and click the hyperlink again, or click “reload”
- User triggers an “abort” of the “connect”
  - Browser creates a new socket and does a “connect”
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes fast

Tearing Down the Connection



- Closing (each end of) the connection
  - Finish (FIN) to close and receive remaining bytes
  - And other host sends a FIN ACK to acknowledge
  - Reset (RST) to close and not receive remaining bytes

Sending/Receiving the FIN Packet

- Each end of the data flow must be shut down independently **("half-close")**
- If one end is done it sends a FIN segment. This means that no more data will be sent

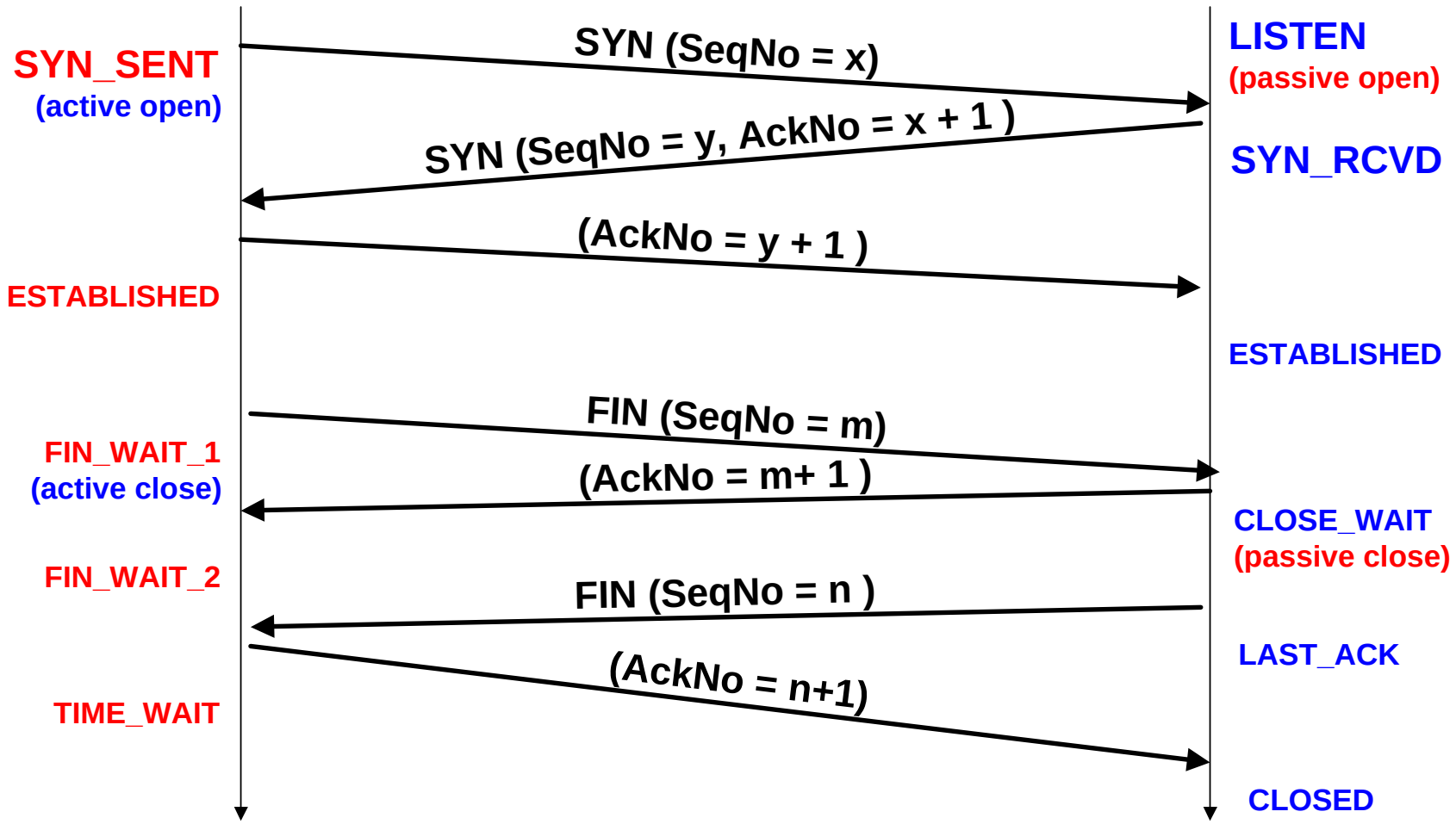
• Sending a FIN: close() – Process is done sending data via the socket – Process invokes "close()" to close the socket – Once TCP has sent all of the outstanding bytes… – … then TCP sends a FIN	• Receiving a FIN: EOF – Process is reading data from the socket – Eventually, the attempt to read returns an EOF

TCP States

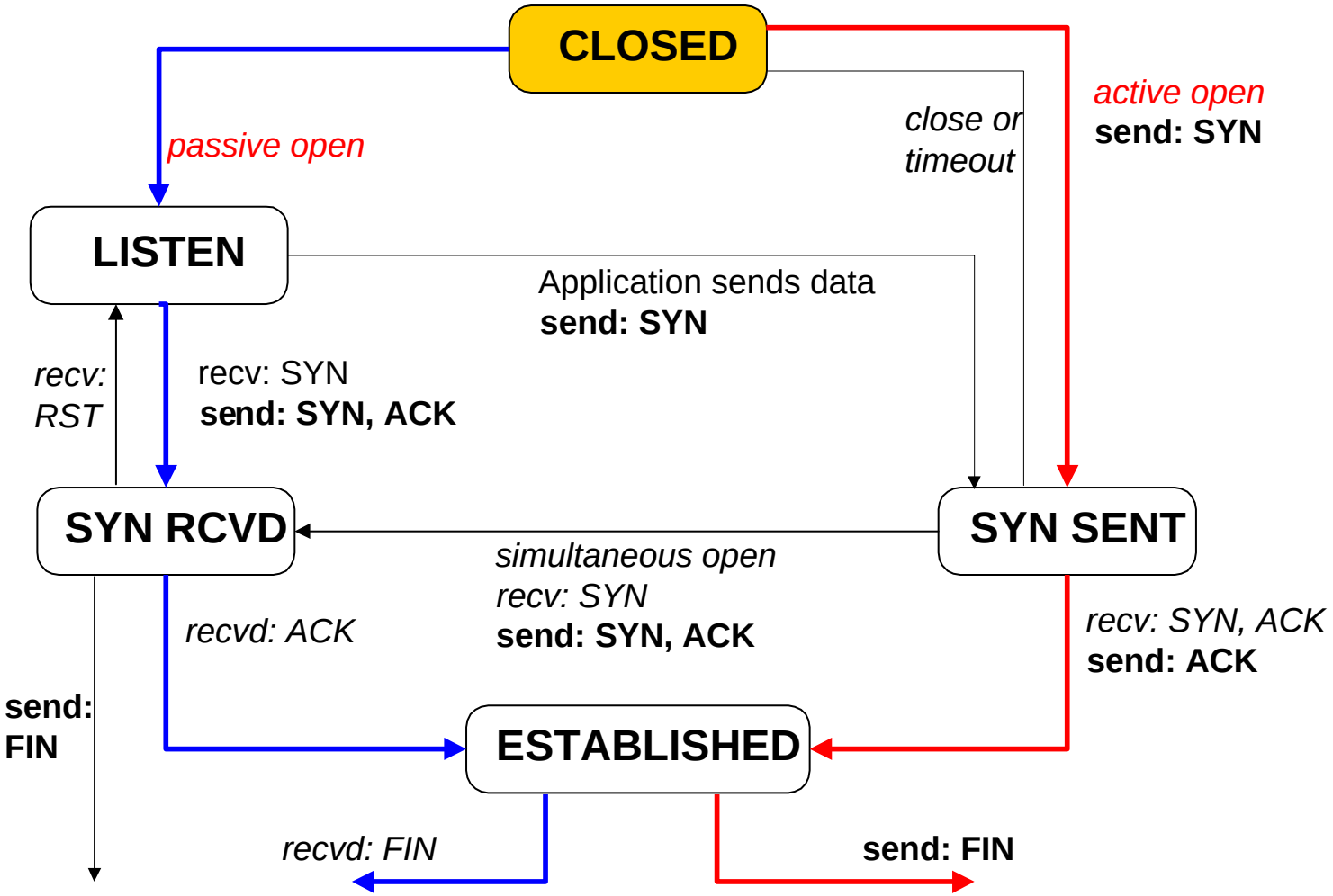
State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The client has started to open a connection
ESTABLISHED	Normal data transfer state
FIN WAIT 1	Client has said it is finished
FIN WAIT 2	Server has agreed to release
TIMED WAIT	Wait for pending packets (2MSL wait state)
CLOSING	Both Sides have tried to close simultaneously
CLOSE WAIT	Server has initiated a release
LAST ACK	Wait for pending packets

MSL = Maximum Segment Lifetime
Maximum time segment can exist in the network before being discarded (TCP estimates it as 2 minutes)

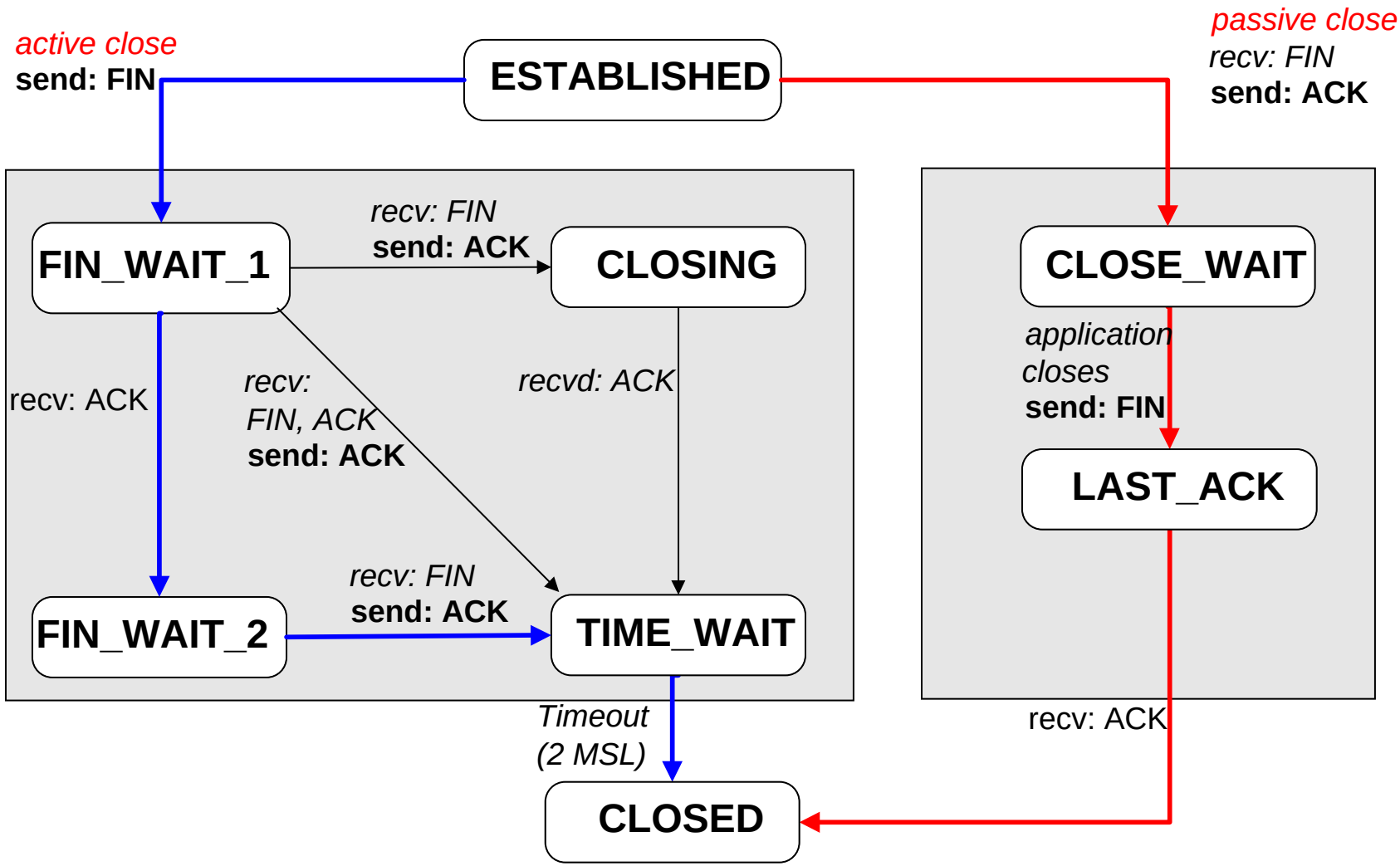
TCP States in “Normal” Connection Lifetime



TCP State Transition Diagram
Opening A Connection



TCP State Transition Diagram
Closing A Connection



2MSL Wait State

2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection **must stay in in the TIME_WAIT state for twice the maximum segment lifetime**.

2MSL= 2 * Maximum Segment Lifetime

- Why?
  TCP is given a chance to resend the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)
- The MSL is set to 2 minutes or 1 minute or 30 seconds.

Try it out

- netstat –a –n
  - shows open connections in various states
- Example:

Protocol	LocalAddr ForeignAddr	State
TCP	**0.0.0.0:23** **0.0.0.0:0**	
LISTENING		
TCP	**192.168.0.1:139** **207.200.89.225:80**	

Resetting Connections

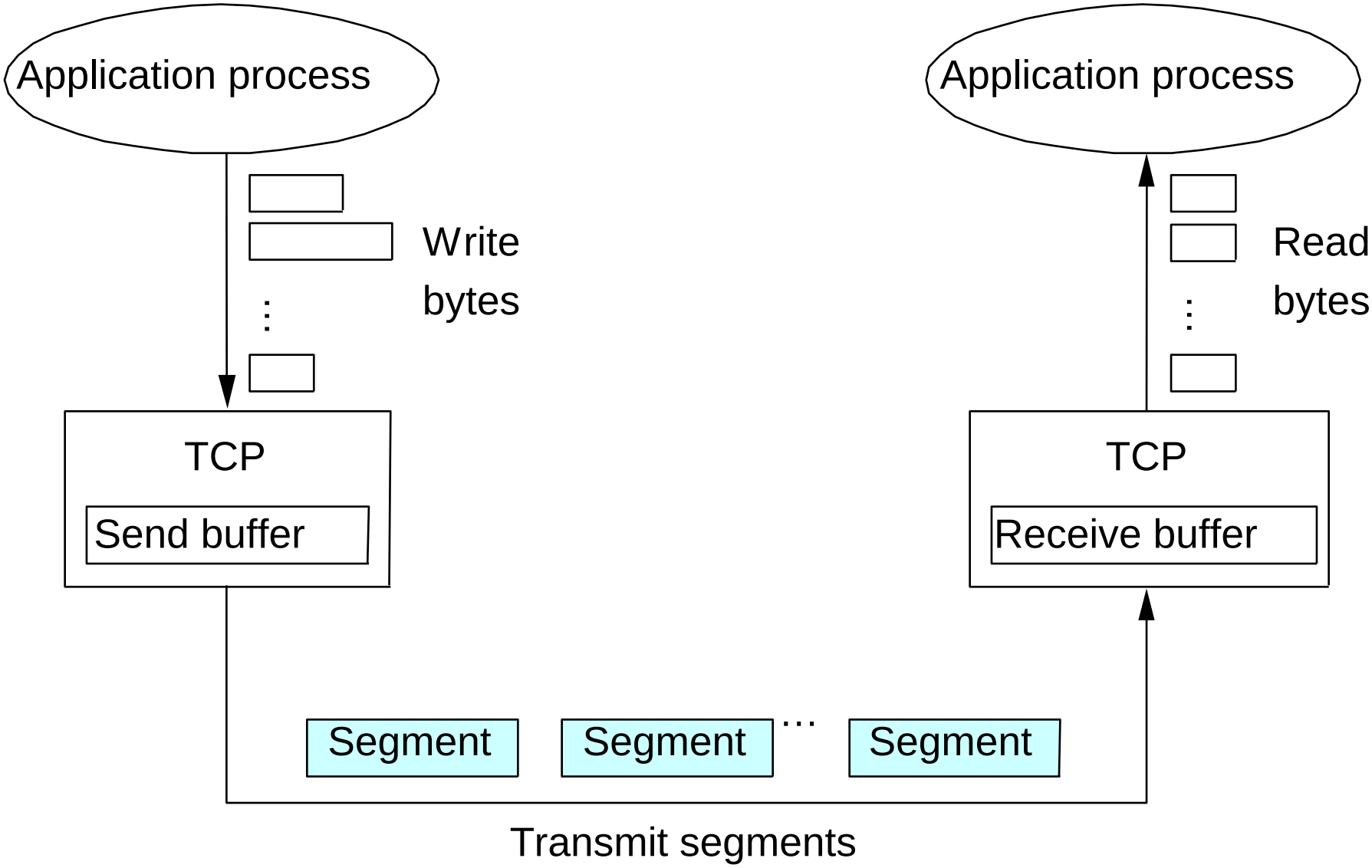
- Resetting connections is done by setting the RST flag

- **When is the RST flag set?**
  - Connection request arrives and no server process is waiting on the destination port
  - Abort (Terminate) a connection
    Causes the receiver to throw away buffered data. Receiver does not acknowledge the RST segment

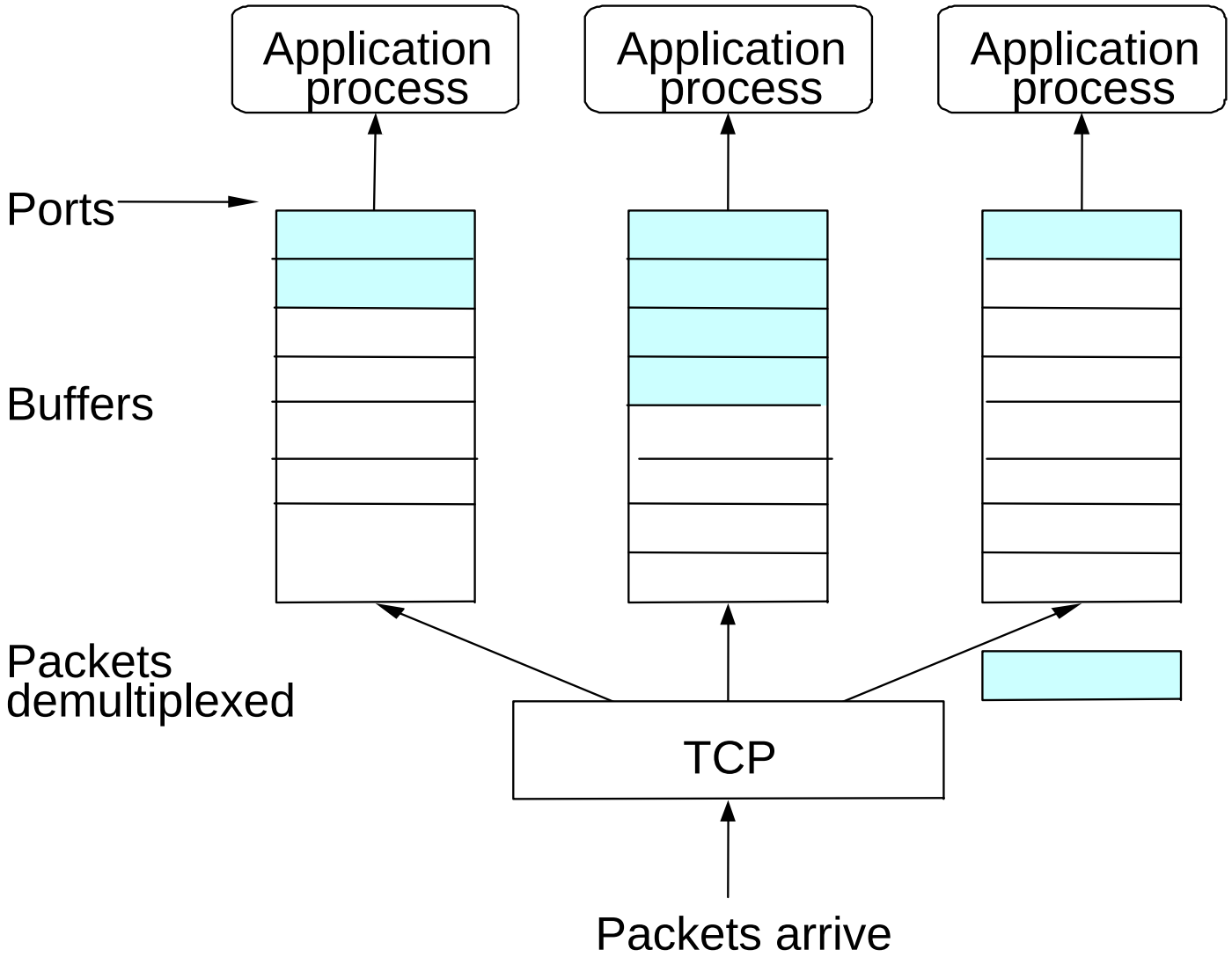
Flow Control:
TCP Sliding Window

Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

Review: How TCP Manages a Byte Stream

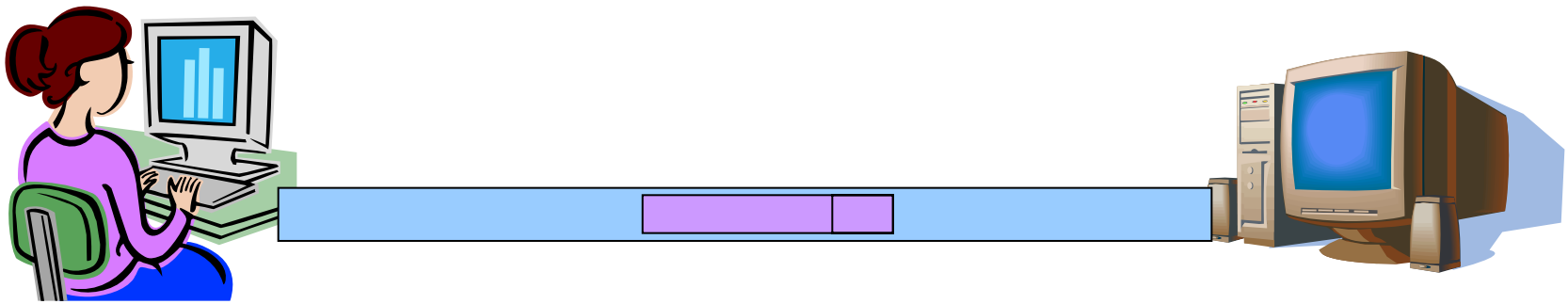


TCP Receive Buffers



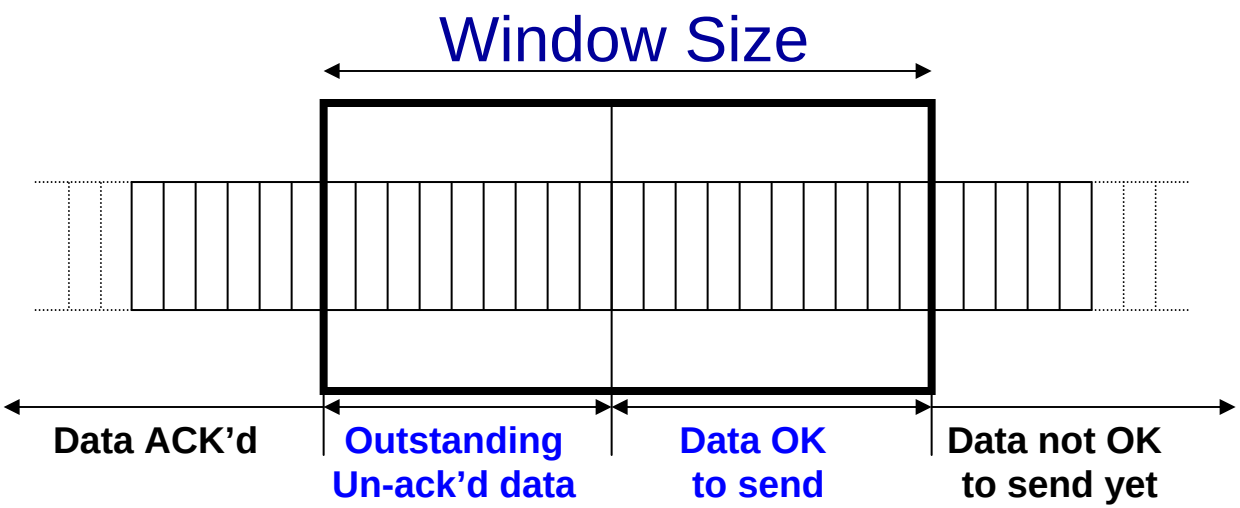
Motivation for Sliding Window

- Stop-and-wait is inefficient
  - Only one TCP segment is “in flight” at a time
  - Especially bad when delay-bandwidth product is high
- Numerical example
  - 1.5 Mbps link with a 45 msec round-trip time (RTT)
    - Delay-bandwidth product is 67.5 Kbits (or 8 KBytes)
  - But, sender can send at most one packet per RTT
    - Assuming a segment size of 1 KB (8 Kbits)
    - … leads to 8 Kbits/segment / 45 msec/segment ➔ 182 Kbps
    - That’s just one-eighth of the 1.5 Mbps link capacity



Receiver Buffering

- Window size
  - Amount that can be sent without acknowledgment
  - Receiver needs to be able to store this amount of data
- Receiver advertises the window to the receiver
  - Tells the receiver the amount of free space left
  - … and the sender agrees not to exceed this amount



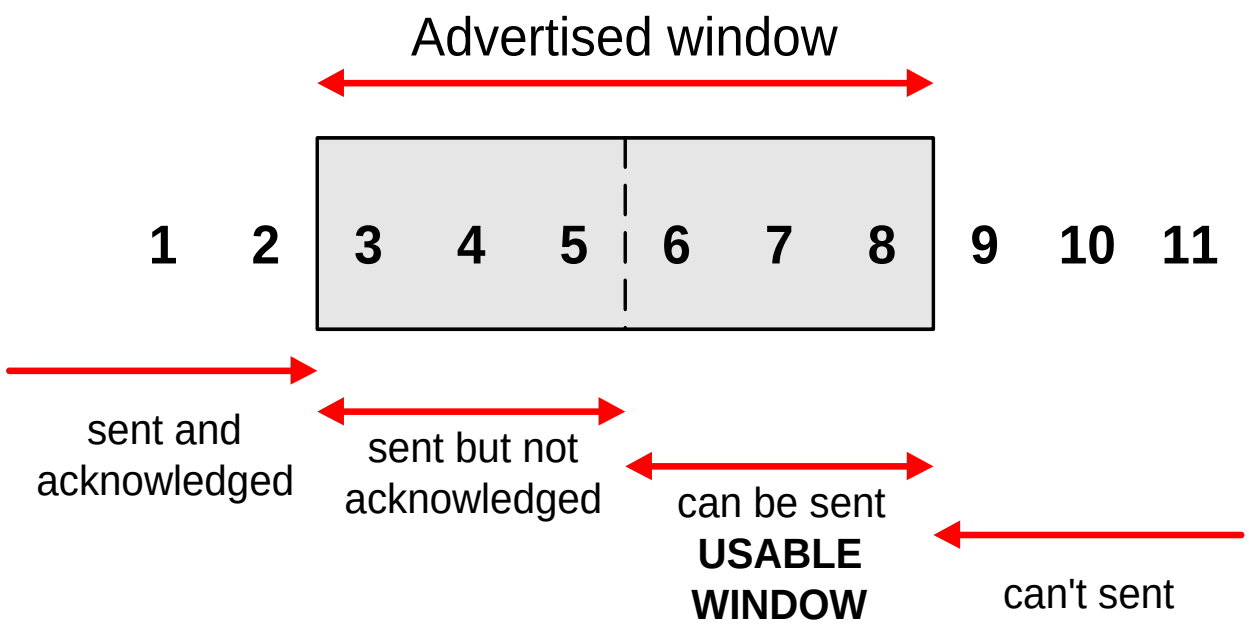
TCP Header for Receiver Buffering

Flags: SYN
FIN
RST
PSH
URG
ACK

Source port			Destination port
Sequence number			
Acknowledgment			
HdrLen	0	Flags	**Advertised window**
Checksum			Urgent pointer
Options (variable)			
Data			

Sliding Window Flow Control

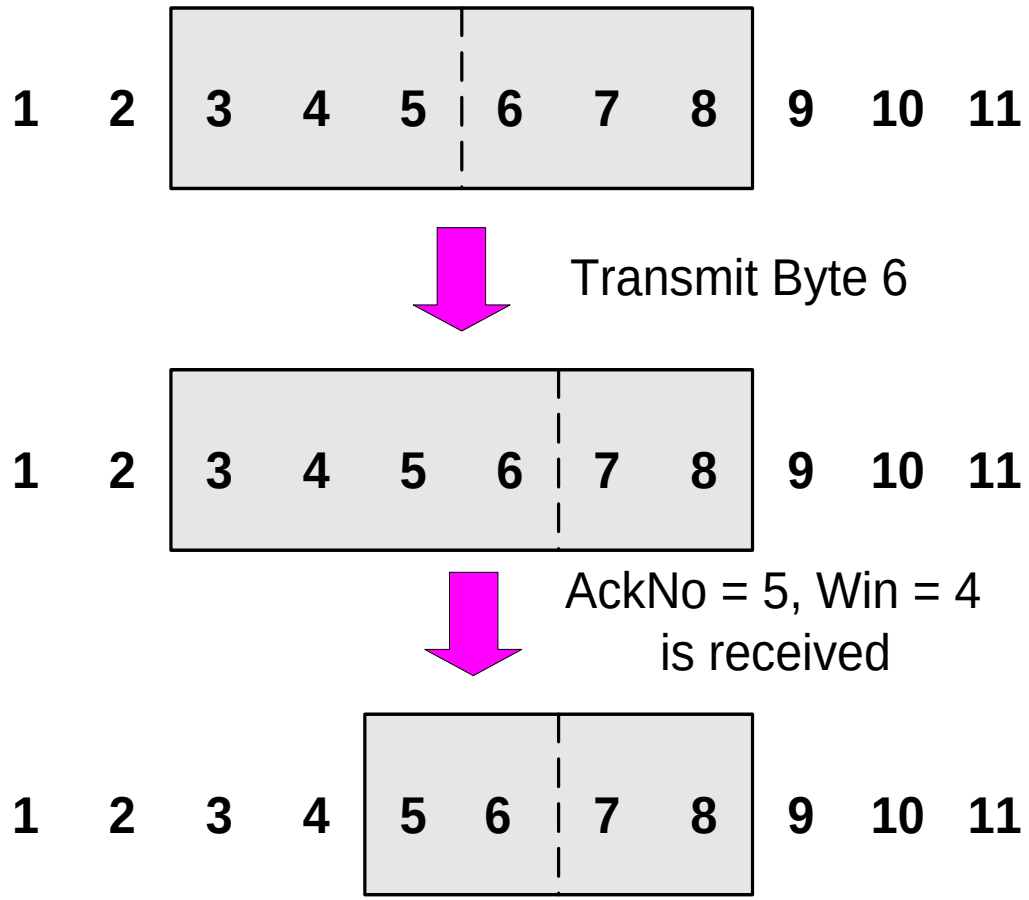
- Sliding Window Protocol is performed at the byte level:



- Here: Sender can transmit sequence numbers 6,7,8.

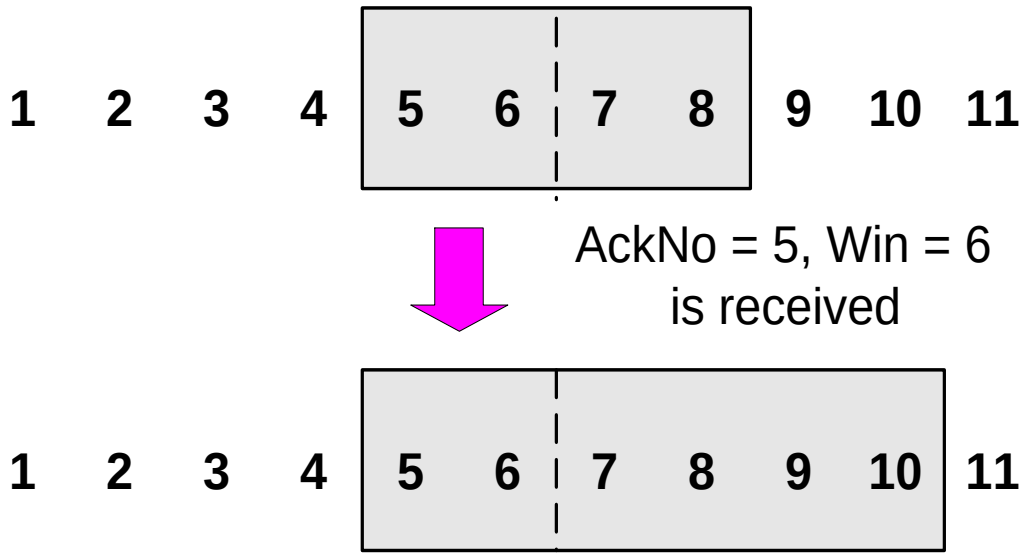
Sliding Window: "Window Closes"

Transmission of a single byte (with SeqNo = 6) and acknowledgement is received (AckNo = 5, Win=4):



Sliding Window: "Window Opens"

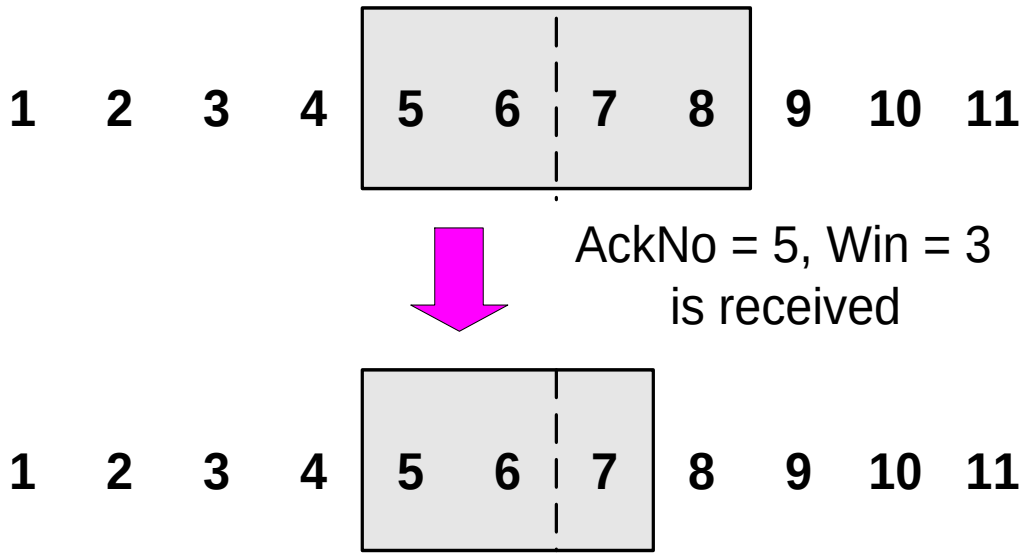
Acknowledgement is received that enlarges the window to the right (AckNo = 5, Win=6):



A receiver opens a window when TCP buffer empties (meaning that data is delivered to the application).

Sliding Window: "Window Shrinks"

Acknowledgement is received that reduces the window from the right (AckNo = 5, Win=3):



Shrinking a window should not be used.

Window Management in TCP

- The receiver is returning two parameters to the sender

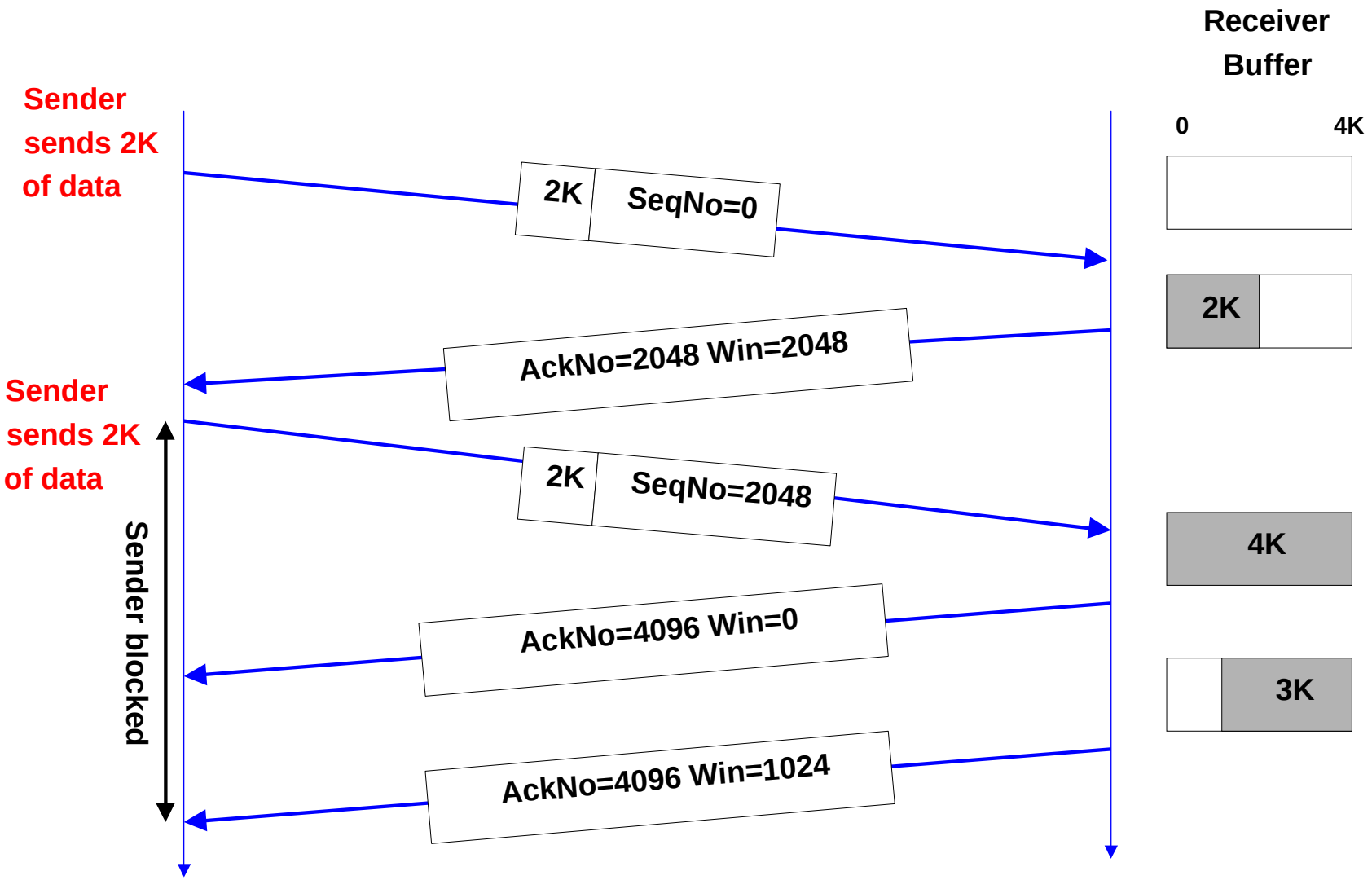
AckNo	window size (win)
32 bits	16 bits

- The interpretation is:
  - I am ready to receive new data with

    SeqNo= AckNo, AckNo+1, …., AckNo+Win-1

- Receiver can acknowledge data without opening the window
- Receiver can change the window size without acknowledging data

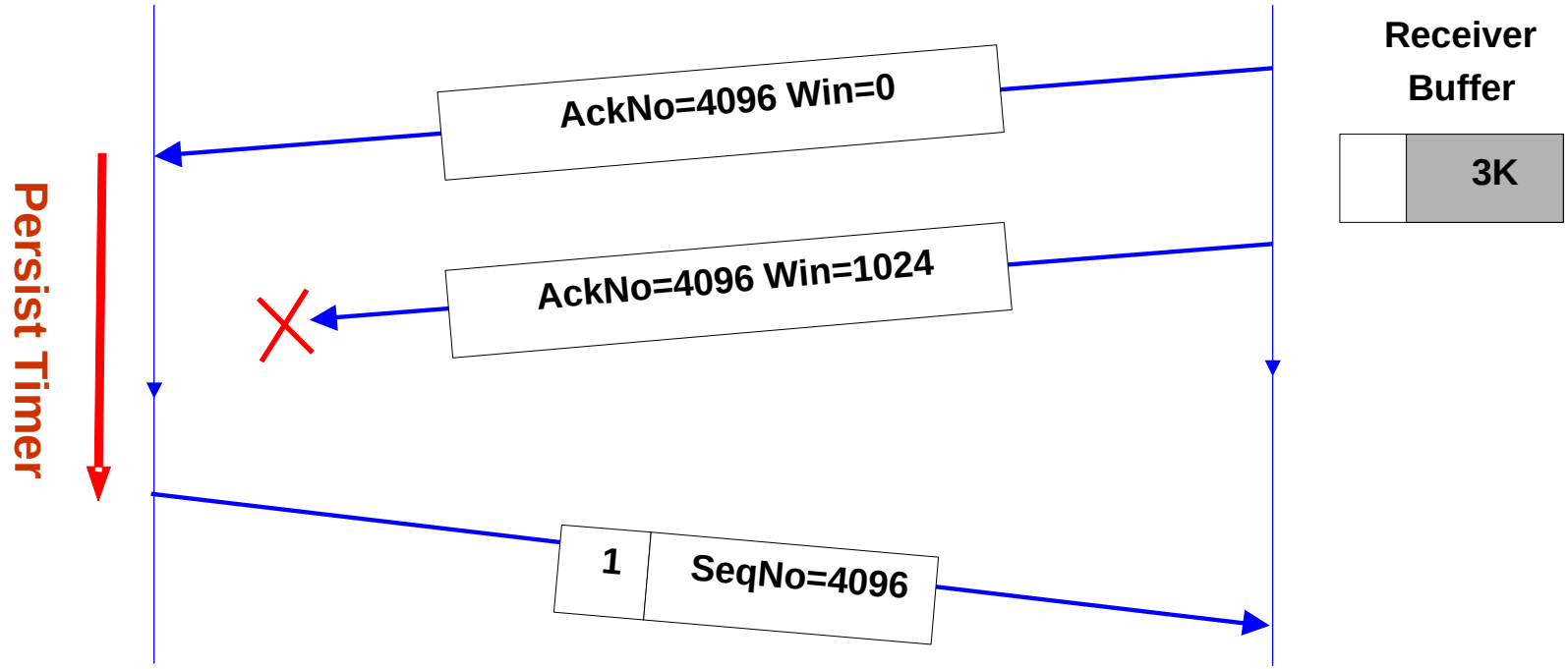
Sliding Window: Example



Lost Window Update?

- What if the last window update message is lost?
  - Receiver waiting for data
  - Sender not allowed to send anything

TCP Persist Timer



- Sender set persist timer when windows size goes to 0
- When timer expires, sender sends a “window probe” message (TCP packet with 1 byte of data)
- If receiver still has window 0, it will send an ACK, but that ACK does not cover the “illegal” 1 byte just sent.

TCP Congestion Control

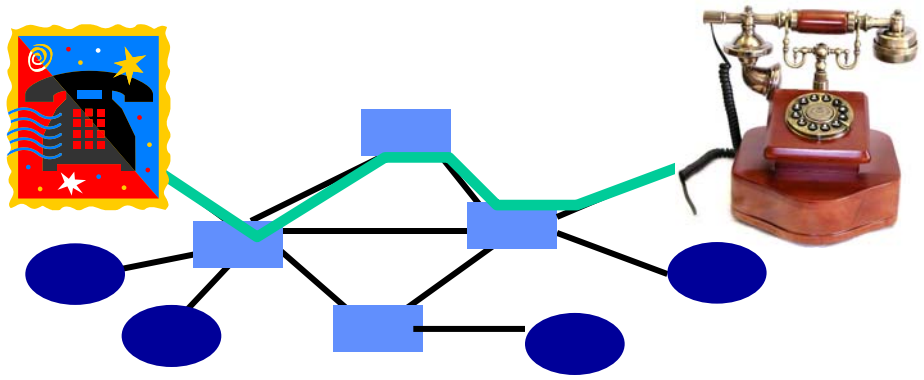
Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

Congestion Topics

- Congestion in IP networks
  - Unavoidable due to best-effort service model
  - IP philosophy: decentralized control at end hosts
- Congestion control by the TCP senders
  - Infers congestion is occurring (e.g., from packet losses)
  - Slows down to alleviate congestion, for the greater good
- TCP congestion-control algorithm
  - Additive-increase, multiplicative-decrease
  - Slow start and slow-start restart
- Active Queue Management (AQM)
  - Random Early Detection (RED)
  - Explicit Congestion Notification (ECN)

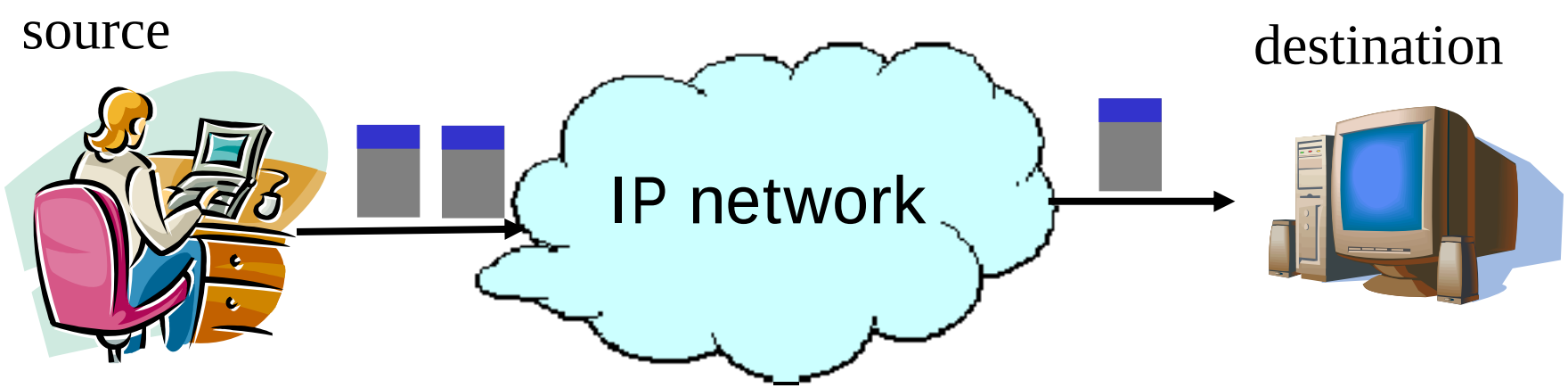
No Problem Under Circuit Switching

- Source establishes connection to destination
  - Nodes reserve resources for the connection
  - Circuit rejected if the resources aren't available
  - Cannot have more than the network can handle



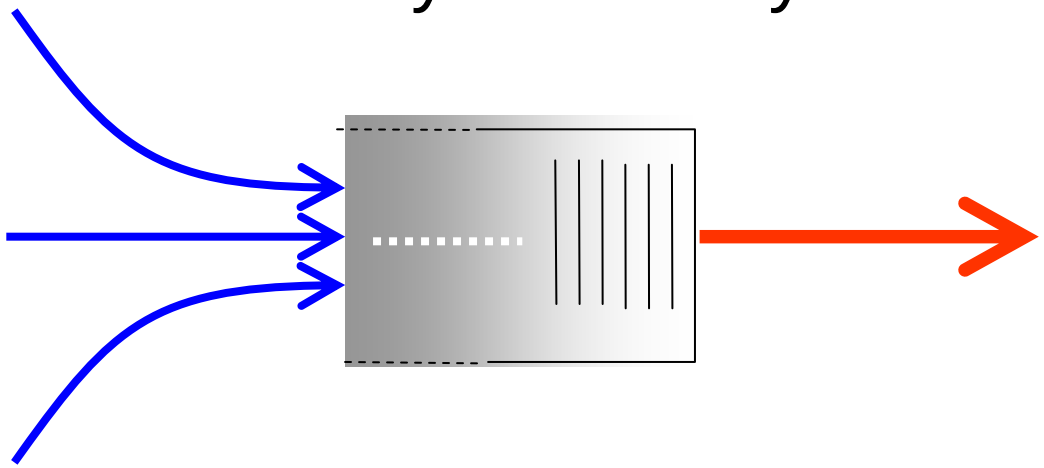
IP Best-Effort Design Philosophy

- Best-effort delivery
  - Let everybody send
  - Try to deliver what you can
  - … and just drop the rest



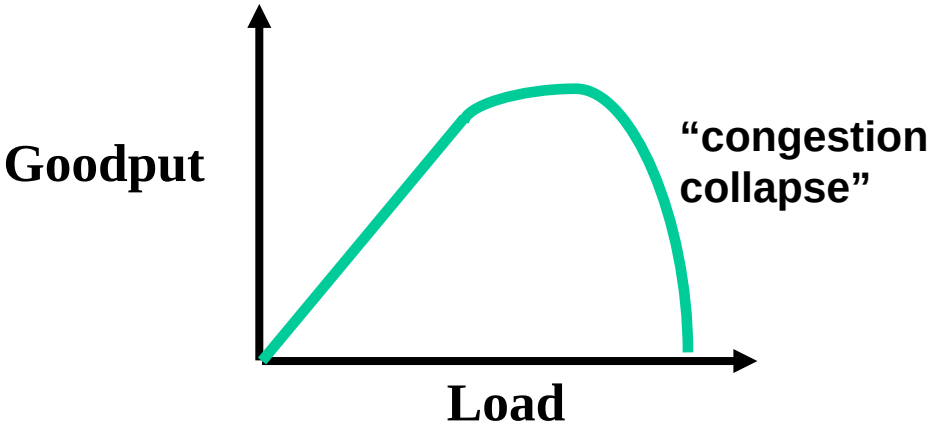
Congestion is Unavoidable

- Two packets arrive at the same time
  - The node can only transmit one
  - … and either buffer or drop the other
- If many packets arrive in short period of time
  - The node cannot keep up with the arriving traffic
  - … and the buffer may eventually overflow



The Problem of Congestion

- What is congestion?
  - Load is higher than capacity
- What do IP routers do?
  - Drop the excess packets
- Why is this bad?
  - Wasted bandwidth for retransmissions



Increase in load that results in a *decrease* in useful work done.

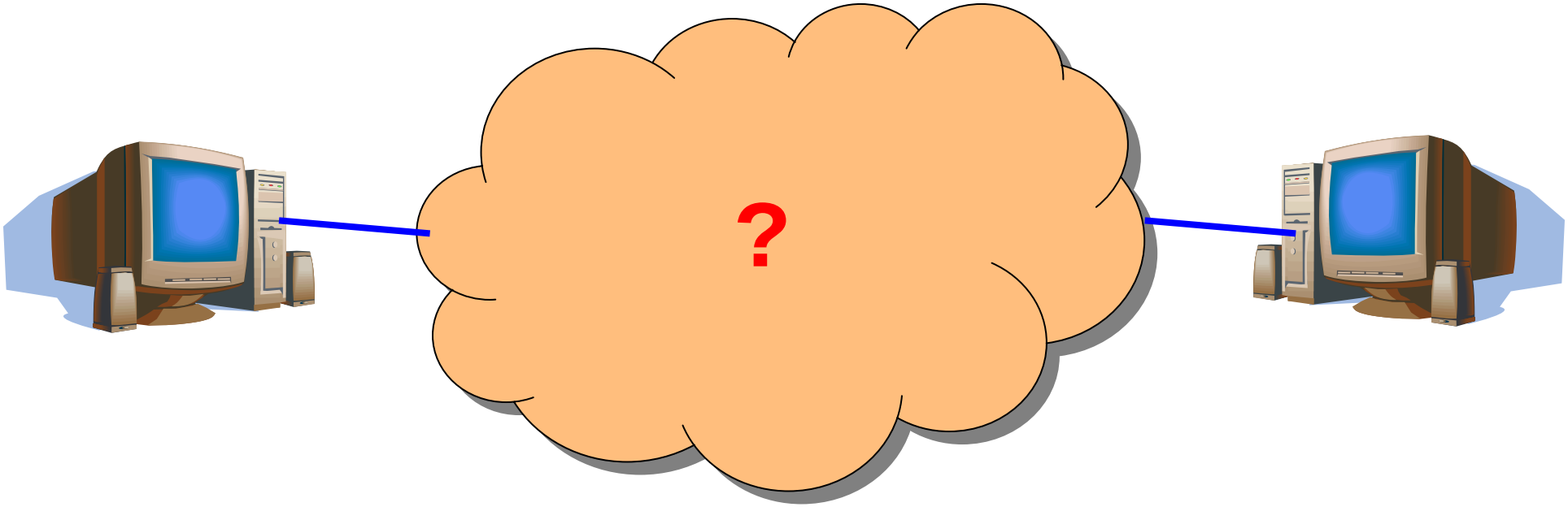
Ways to Deal With Congestion

- Ignore the problem
  - Many dropped (and retransmitted) packets
  - Can cause congestion collapse
- Pricing
  - Don’t drop packets for the high-bidders
  - Requires a payment model
- Dynamic adjustment (TCP)
  - Every sender infers the level of congestion
  - And adapts its sending rate, for the greater good

Many Important Questions

- How does the sender know there is congestion?
  - Explicit feedback from the network?
  - Inference based on network performance?
- How should the sender adapt?
  - Explicit sending rate computed by the network?
  - End host coordinates with other hosts?
  - End host thinks globally but acts locally?
- What is the performance objective?
  - Maximizing goodput, even if some users suffer more?
  - Fairness? (Whatever the heck *that* means!)
- How fast should new TCP senders send?

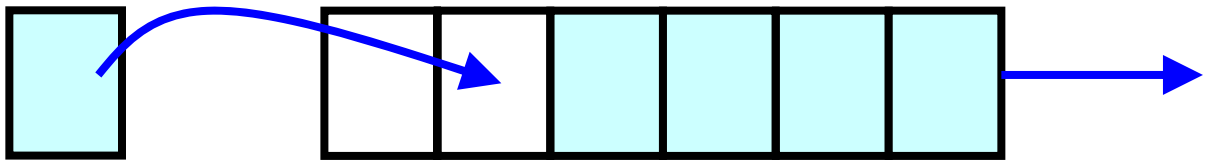
Inferring From Implicit Feedback



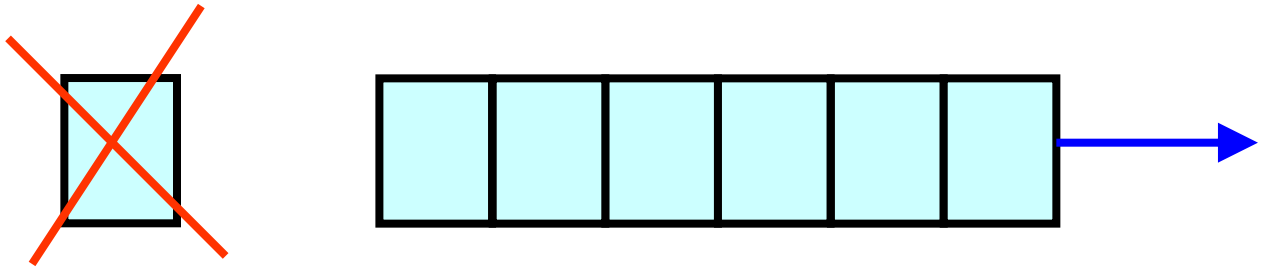
- What does the end host see?
- What can the end host change?

Where Congestion Happens: Links

- Simple resource allocation: FIFO queue & drop-tail
- Access to the bandwidth: first-in first-out queue
  - Packets transmitted in the order they arrive



- Access to the buffer space: drop-tail queuing
  - If the queue is full, drop the incoming packet



How it Looks to the End Host

- Packet delay
  - Packet experiences high delay
- Packet loss
  - Packet gets dropped along the way
- How does TCP sender learn this?
  - Delay
    - Round-trip time estimate
  - Loss
    - Timeout
    - Duplicate acknowledgments

What Can the End Host Do?

- Upon detecting congestion
  - Decrease the sending rate (e.g., divide in half)
  - End host does its part to alleviate the congestion
- But, what if conditions change?
  - Suppose there is more bandwidth available
  - Would be a shame to stay at a low sending rate
- Upon *not* detecting congestion
  - Increase the sending rate, a little at a time
  - And see if the packets are successfully delivered

TCP Congestion Window

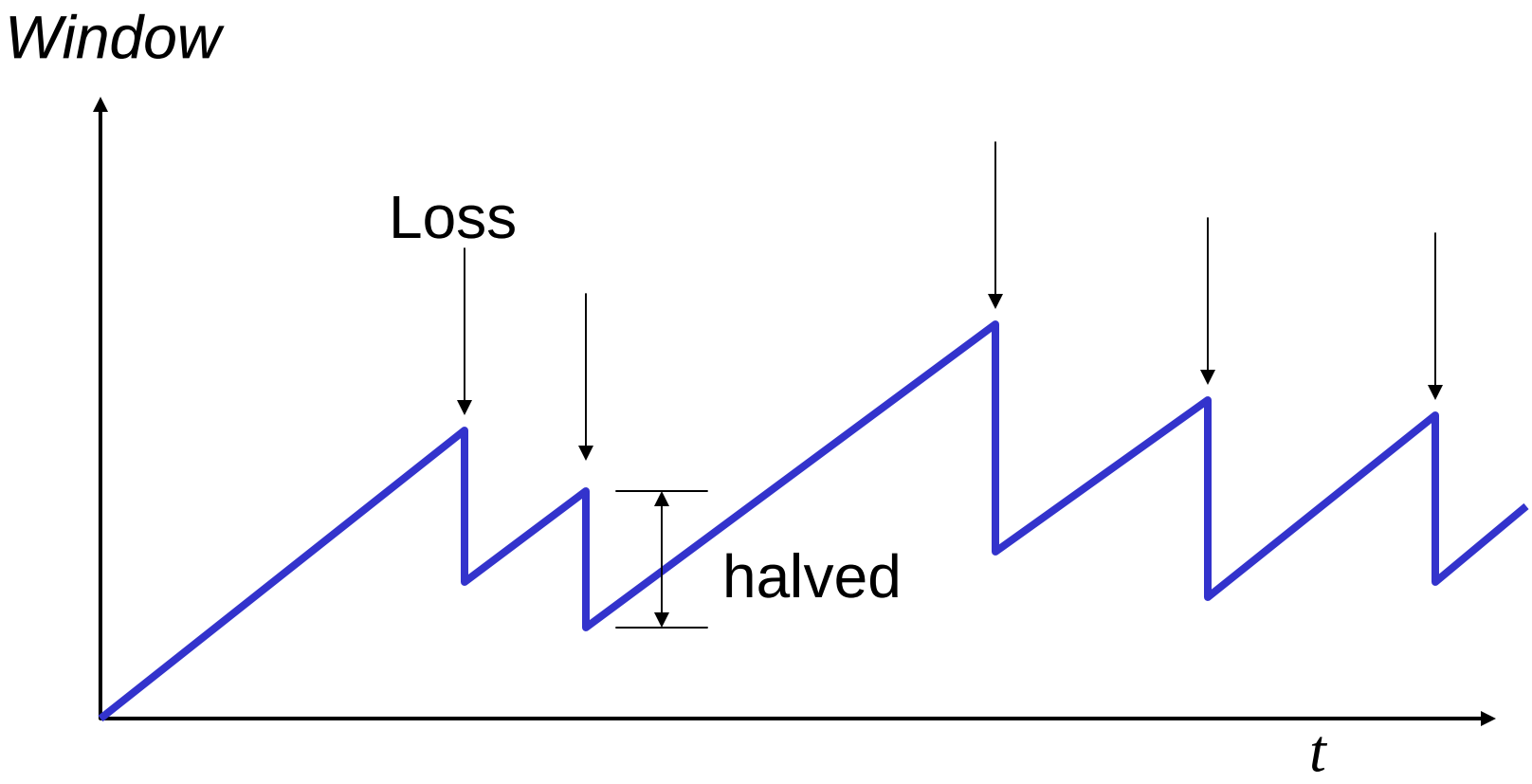
- Each TCP sender maintains a congestion window
  - Maximum number of bytes to have in transit
  - I.e., number of bytes still awaiting acknowledgments
- Adapting the congestion window
  - Decrease upon losing a packet: backing off
  - Increase upon success: optimistically exploring
  - Always struggling to find the right transfer rate

- Both good and bad
  - Pro: avoids having explicit feedback from network
  - Con: under-shooting and over-shooting the rate

Additive Increase, Multiplicative Decrease

- How much to increase and decrease?
  - Increase linearly, decrease multiplicatively
  - A necessary condition for stability of TCP
  - Consequences of over-sized window are much worse than having an under-sized window
    - Over-sized window: packets dropped and retransmitted
    - Under-sized window: somewhat lower throughput
- Multiplicative decrease
  - On loss of packet, divide congestion window in half
- Additive increase
  - On success for last window of data, increase linearly

Leads to the TCP "Sawtooth"



Practical Details

- Congestion window
  - Represented in bytes, not in packets (Why?)
  - Packets have MSS (Maximum Segment Size) bytes
- Increasing the congestion window
  - Increase by MSS on success for last window of data
- Decreasing the congestion window
  - Never drop congestion window below 1 MSS

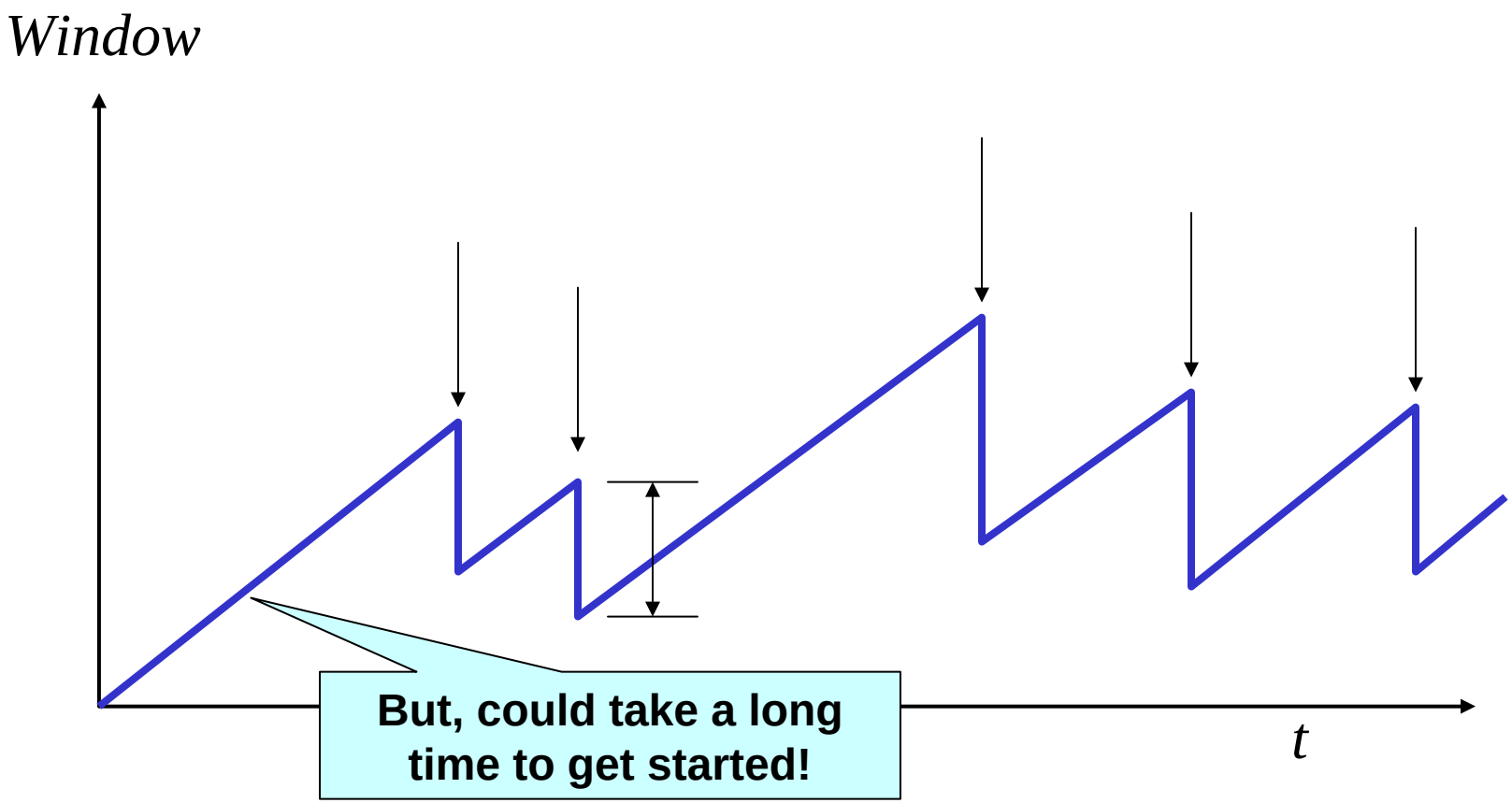
Receiver Window vs. Congestion Window

- Flow control
  - Keep a *fast sender* from overwhelming *a slow receiver*
- Congestion control
  - Keep a *set of senders* from overloading the *network*

- Different concepts, but similar mechanisms
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - TCP window: min{congestion window, receiver window}

How Should a New Flow Start

Need to start with a small CWND to avoid overloading the network.

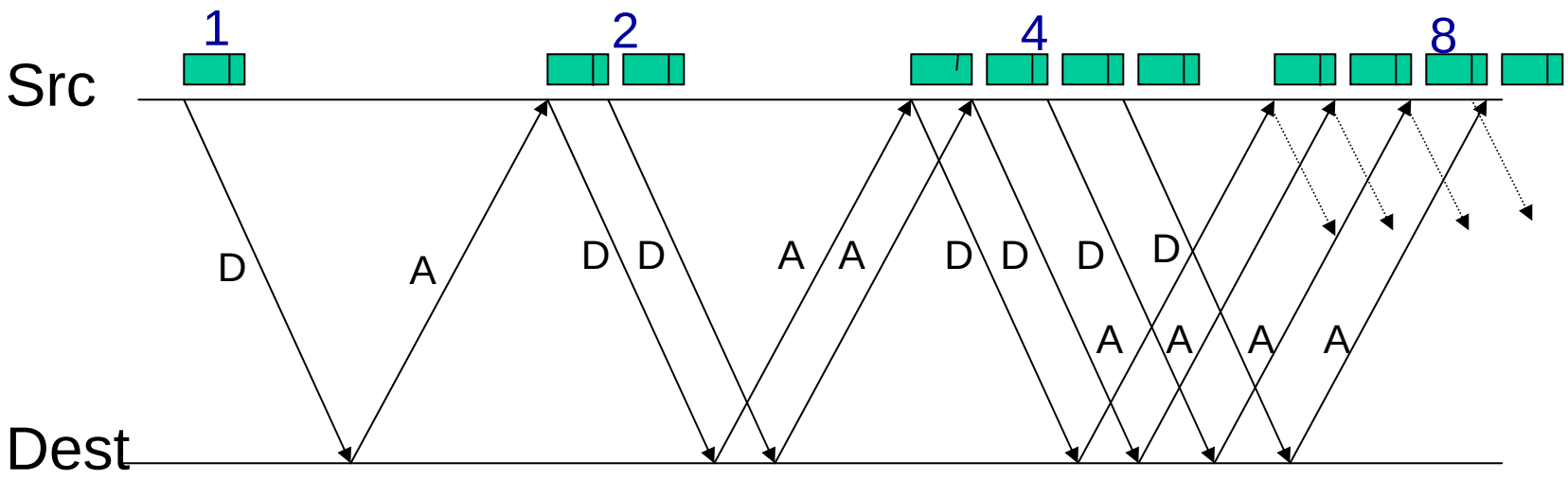


“Slow Start” Phase

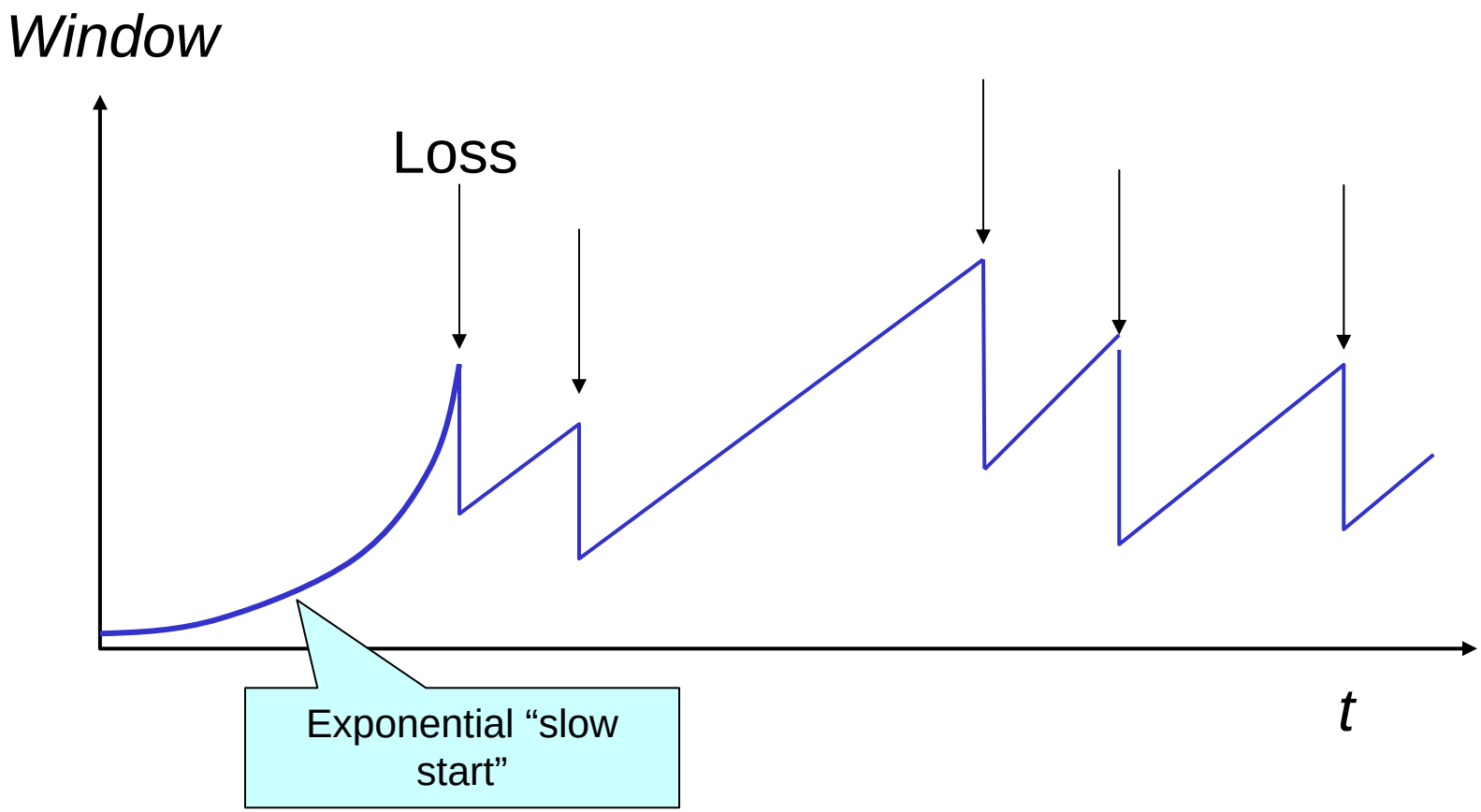
- Start with a small congestion window
  - Initially, CWND is 1 Max Segment Size (MSS)
  - So, initial sending rate is MSS/RTT
- That could be pretty wasteful
  - Might be much less than the actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (really “fast start”)
  - Sender starts at a slow rate (hence the name)
  - … but increases the rate exponentially
  - … until the first loss event

Slow Start in Action

Double CWND per round-trip time



Slow Start and the TCP Sawtooth

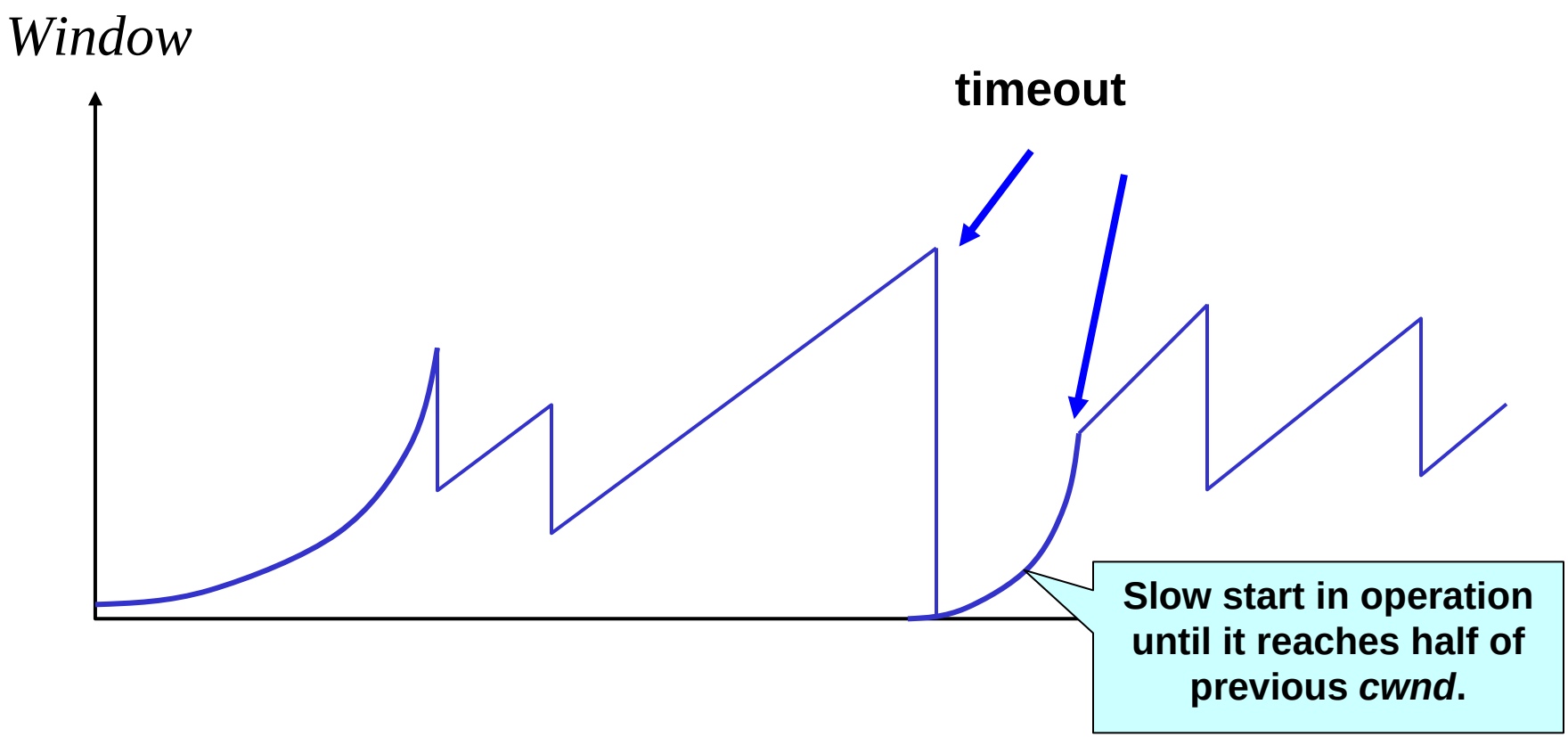


Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole receiver window’s worth of data.

Two Kinds of Loss in TCP

- Timeout
  - Packet n is lost and detected via a timeout
  - E.g., 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 low CWND
- 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

Repeating Slow Start After Timeout



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

Exercise

A TCP sender is using a MSS of 1000 bytes and the receiver's Advertised Window is 8000 bytes. Suppose that the TCP Congestion Window is equal to 6000 bytes just before a timeout occurs.

After the timeout, six full segments of data – S1, S2, S3, S4, S5, S6 - are successfully sent and acknowledged. The seventh segment is lost and must be retransmitted.

Show the values for Congestion Window (CW) and Threshold (TH) as they change with each arriving ACK and then after the retransmission of the seventh segment.

Exercise (contd.)

Event	Congestion W.	Threshold	Justification
Timeout			
ACK for S1 arrives			
ACK for S2 arrives			
ACK for S3 arrives			
ACK for S4 arrives			
ACK for S5 arrives			
ACK for S6 arrives			
Timeout for S7			

Exercise

Consider a TCP connection in which the TCP Advertised Window is equal to 4 Kbytes, the TCP Congestion Window is equal to 6Kbytes, the Maximum Segment Size (MSS) is 1Kbyte and all transmitted packets are 1Kbyte in length.

1. How many packets could be in traffic unacknowledged yet?

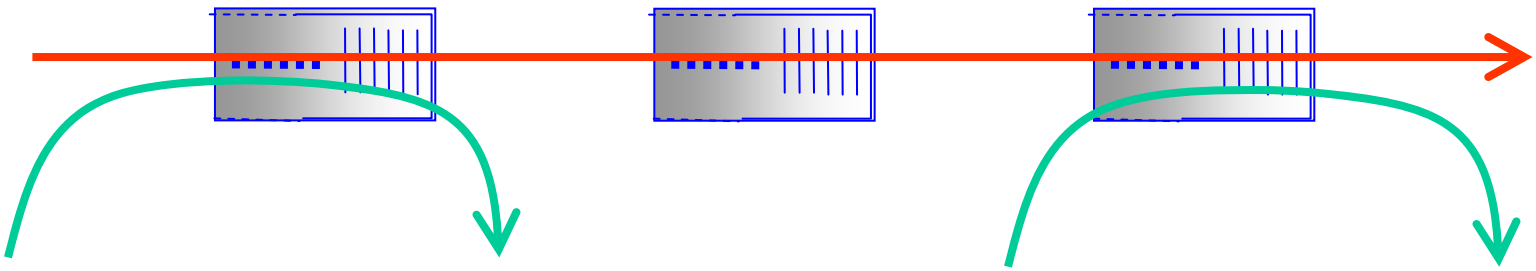
2. If a timeout occurs, how many packets it takes to send and get acknowledged to get the Congestion Window up to 6Kbytes again? Assume no transmission errors.

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

TCP Achieves Some Notion of Fairness

- Effective utilization is not the only goal
  - We also want to be *fair* to the various flows
  - … but what the heck does *that* mean?
- Simple definition: equal shares of the bandwidth
  - N flows that each get 1/N of the bandwidth?
  - E.g., bandwidth shared in proportion to the RTT
  - But, what if the flows traverse different paths?



What About Cheating?

- Some folks are more fair than others
  - Running multiple TCP connections in parallel
  - Modifying the TCP implementation in the OS
  - Use the User Datagram Protocol
- What is the impact
  - Good guys slow down to make room for you
  - You get an unfair share of the bandwidth
- Possible solutions?
  - Routers detect cheating and drop excess packets?
  - Peer pressure?
  - ???

Queuing Mechanisms

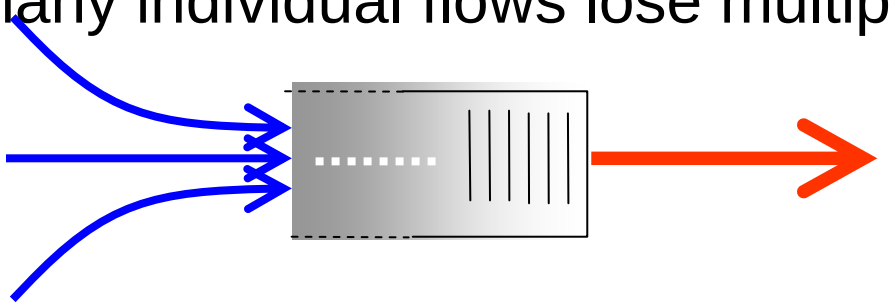
Random Early Detection (RED)

Explicit Congestion Notification (ECN)

Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

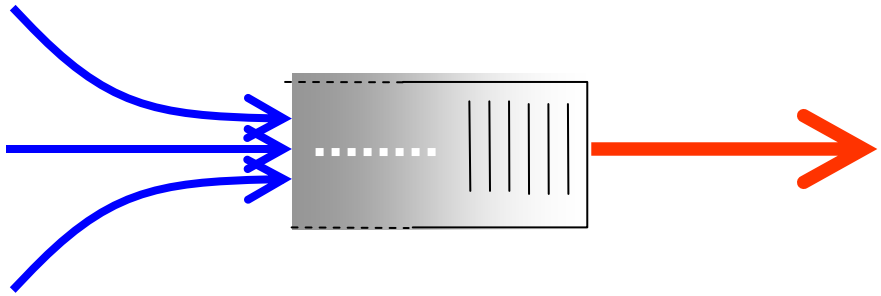
Bursty Loss From Drop-Tail Queuing

- TCP depends on packet loss
  - Packet loss is the indication of congestion
  - In fact, TCP *drives* the network into packet loss
  - … by continuing to increase the sending rate
- Drop-tail queuing leads to *bursty* loss
  - When a link becomes congested…
  - … many arriving packets encounter a full queue
  - And, as a result, many flows divide sending rate in half
  - … and, many individual flows lose multiple packets



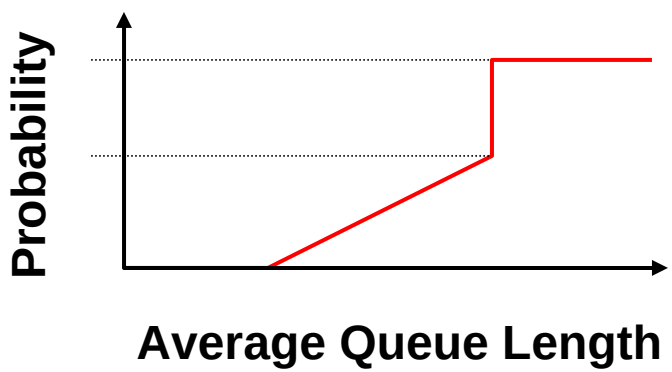
Slow Feedback from Drop Tail

- Feedback comes when buffer is completely full
  - … even though the buffer has been filling for a while
- Plus, the filling buffer is increasing RTT
  - … and the variance in the RTT
- Might be better to give early feedback
  - Get one or two connections to slow down, not all of them
  - Get these connections to slow down before it is too late



Random Early Detection (RED)

- Basic idea of RED
  - Router notices that the queue is getting backlogged
  - … and randomly drops packets to signal congestion
- Packet drop probability
  - Drop probability increases as queue length increases
  - If buffer is below some level, don't drop anything
  - … otherwise, set drop probability as function of queue



Properties of RED

- Drops packets before queue is full
  - In the hope of reducing the rates of some flows
- Drops packet in proportion to each flow's rate
  - High-rate flows have more packets
  - … and, hence, a higher chance of being selected
- Drops are spaced out in time
  - Which should help desynchronize the TCP senders
- Tolerant of burstiness in the traffic
  - By basing the decisions on *average* queue length

Problems With RED

- Hard to get the tunable parameters just right
  - How early to start dropping packets?
  - What slope for the increase in drop probability?
  - What time scale for averaging the queue length?
- Sometimes RED helps but sometimes not
  - If the parameters aren't set right, RED doesn't help
  - And it is hard to know how to set the parameters
- RED is implemented in practice
  - But, often not used due to the challenges of tuning right
- Many variations in the research community
  - With cute names like "Blue" and "FRED"… ☺

Explicit Congestion Notification

- Early dropping of packets
  - Good: gives early feedback
  - Bad: has to drop the packet to give the feedback
- Explicit Congestion Notification
  - Router marks the packet with an ECN bit
  - … and sending host interprets as a sign of congestion
- Surmounting the challenges
  - Must be supported by the end hosts and the routers
  - Requires two bits in the IP header (one for the ECN mark, and one to indicate the ECN capability)
  - Solution: borrow two of the Type-Of-Service bits in the IPv4 packet header

Other TCP Mechanisms

Nagle's Algorithm and Delayed ACK

Slides by Rexford @ Princeton & Slides accompanying the Internet Lab Manual, slightly altered by M.D.

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
  - … but don't introduce much delay by waiting

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: transmits in MSS-sized packets anyway

Motivation for Delayed ACK

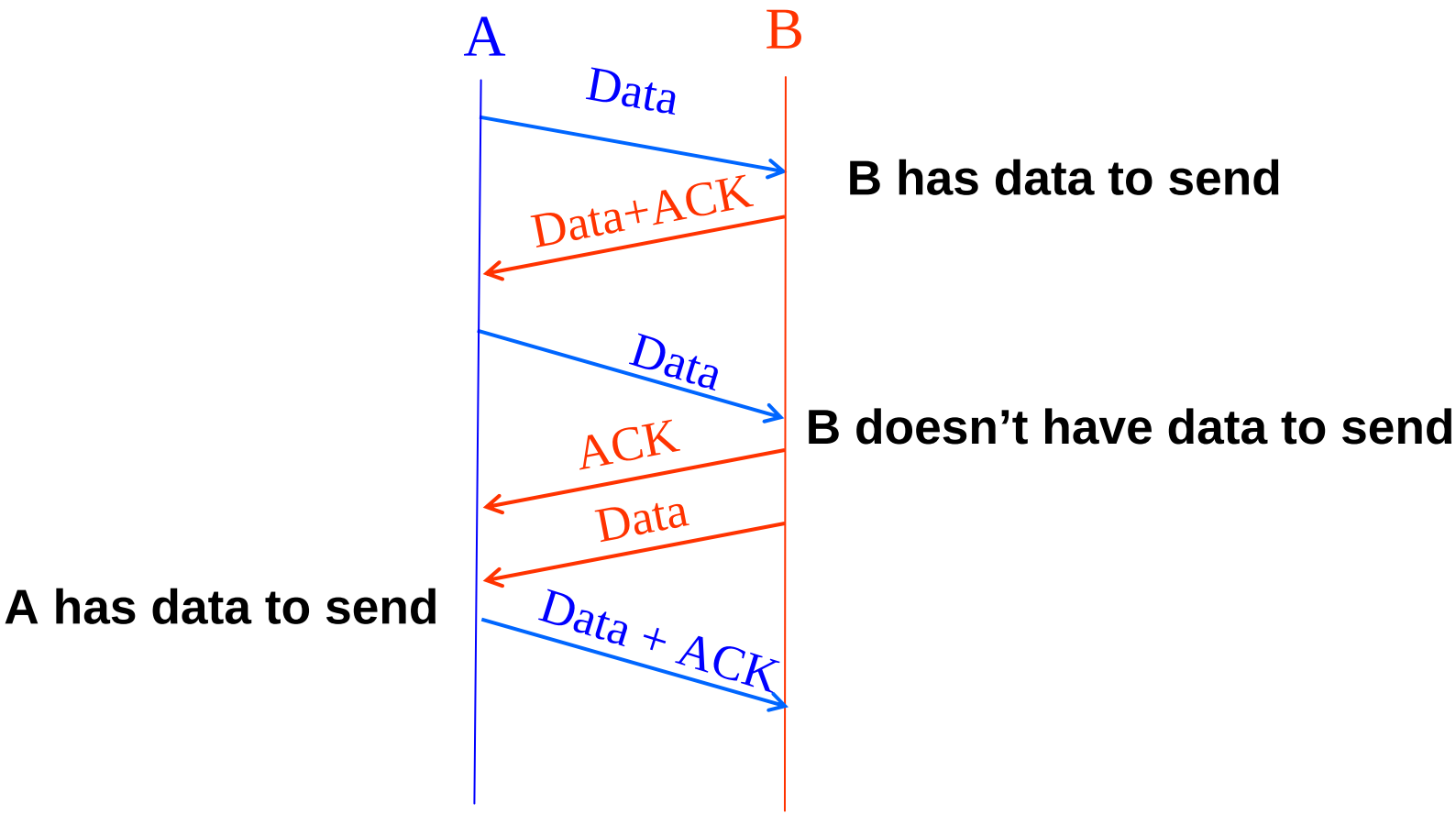
- TCP traffic is often bidirectional
  - Data traveling in both directions
  - ACKs traveling in both directions
- ACK packets have high overhead
  - 40 bytes for the IP header and TCP header
  - … and zero data traffic
- Piggybacking is appealing
  - Host B can send an ACK to host A
  - … as part of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

TCP Header Allows Piggybacking

Flags: SYN
FIN
RST
PSH
URG
ACK

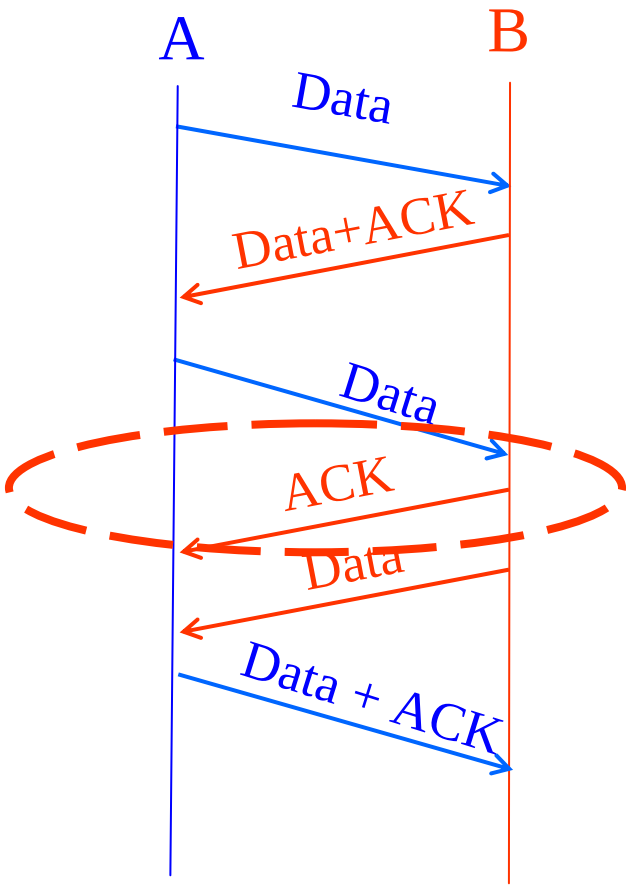
<table>
<tr><td colspan="3">Source port</td><td>Destination port</td></tr>
<tr><td colspan="4">Sequence number</td></tr>
<tr><td colspan="4">Acknowledgment</td></tr>
<tr><td>HdrLen</td><td>0</td><td>Flags</td><td>Advertised window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options (variable)</td></tr>
<tr><td colspan="4">Data</td></tr>
</table>

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
    - Typically, 200 msec or 500 msec
  - 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

Reading

- Read sections 2.5, 5.1 – 5.2, 6.1 – 6.4 from the textbook.