CS 330: Network Applications & Protocols

Transport Layer

Galin Zhelezov
Department of Physical Sciences
York College of Pennsylvania

Overview of Transport Layer

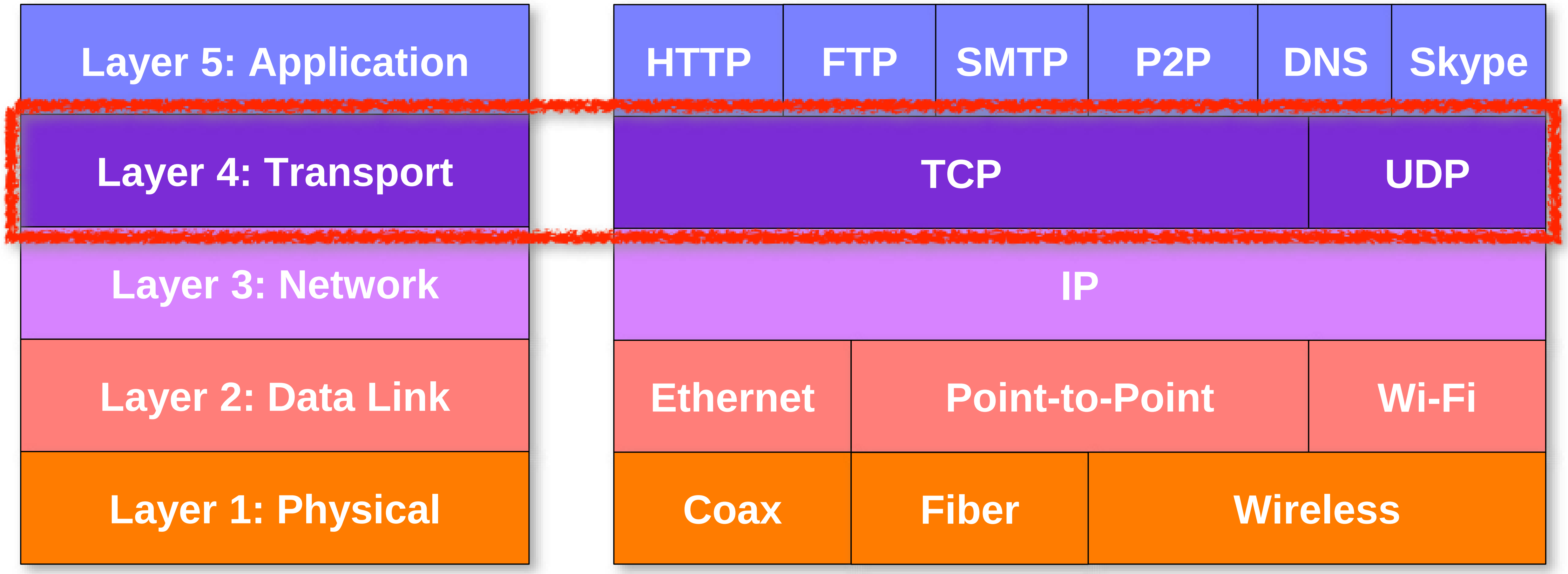
- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
- **Principles of Congestion Control**
- **TCP Congestion Control**

Overview of Transport Layer

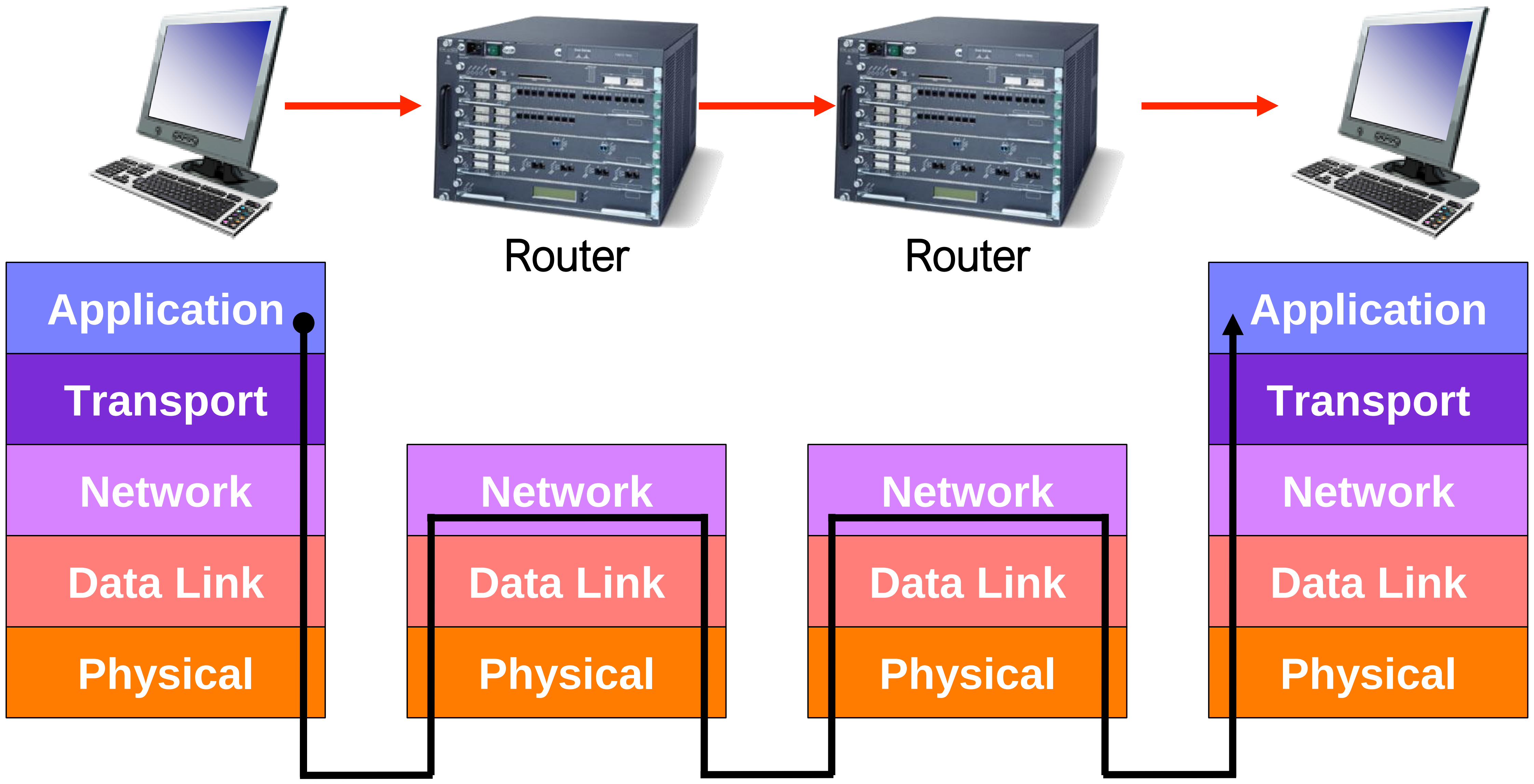
- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
- **Principles of Congestion Control**
- **TCP Congestion Control**

Protocol Layers

- **Top-Down Approach**



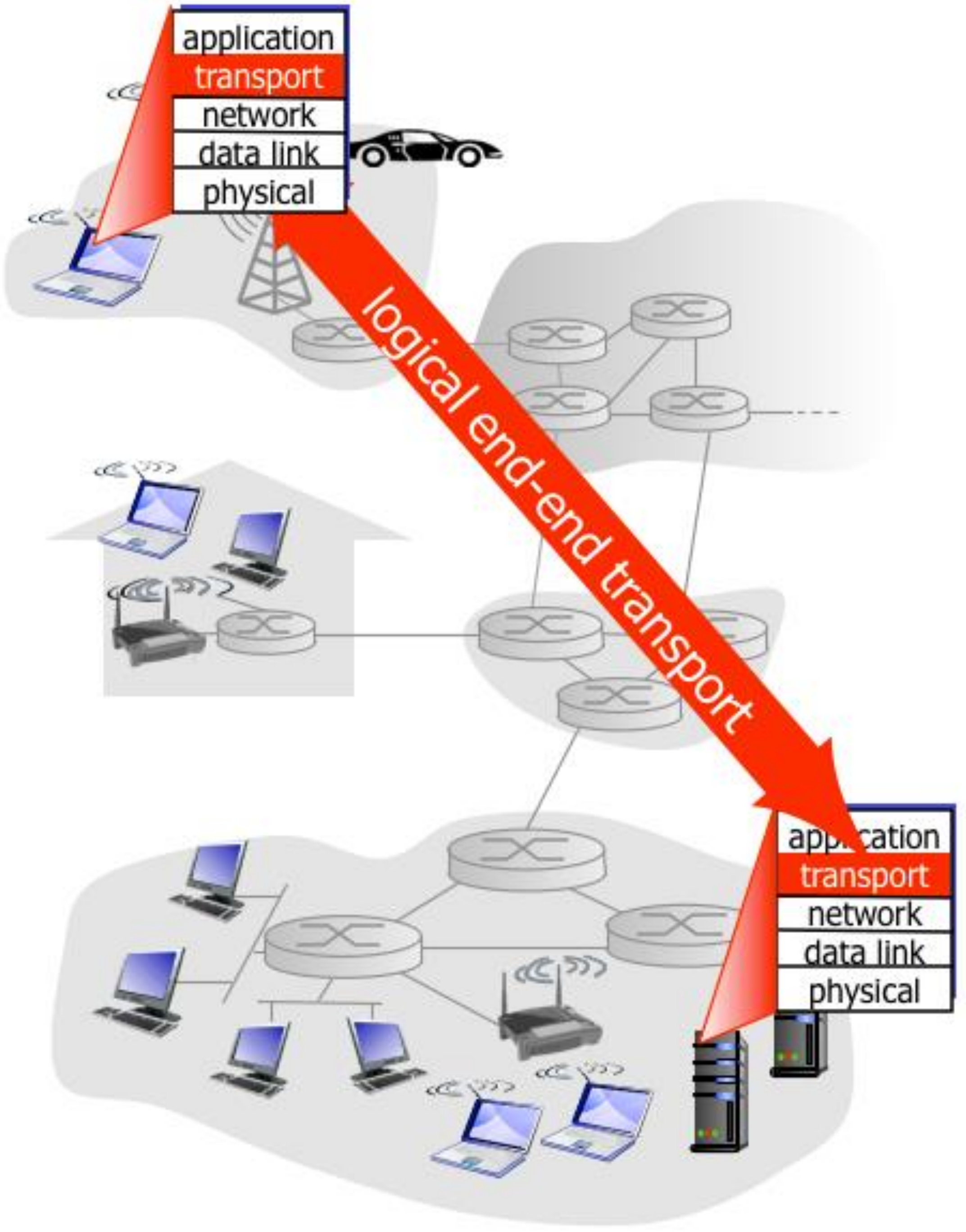
Transport Layer



- **Transport layer provides End-to-End Services**
  - Required at source and destination
  - Not required at intermediate hops

Transport Services and Protocols

- **Provide logical communication between application processes running on different hosts**
- **Transport protocols run in end systems**
  - Sending side: breaks application messages into segments, passes segments down to network layer
  - Receiving side: reassembles segments into messages, passes to application layer
- **More than one transport protocol available to apps**
  - Internet: TCP and UDP

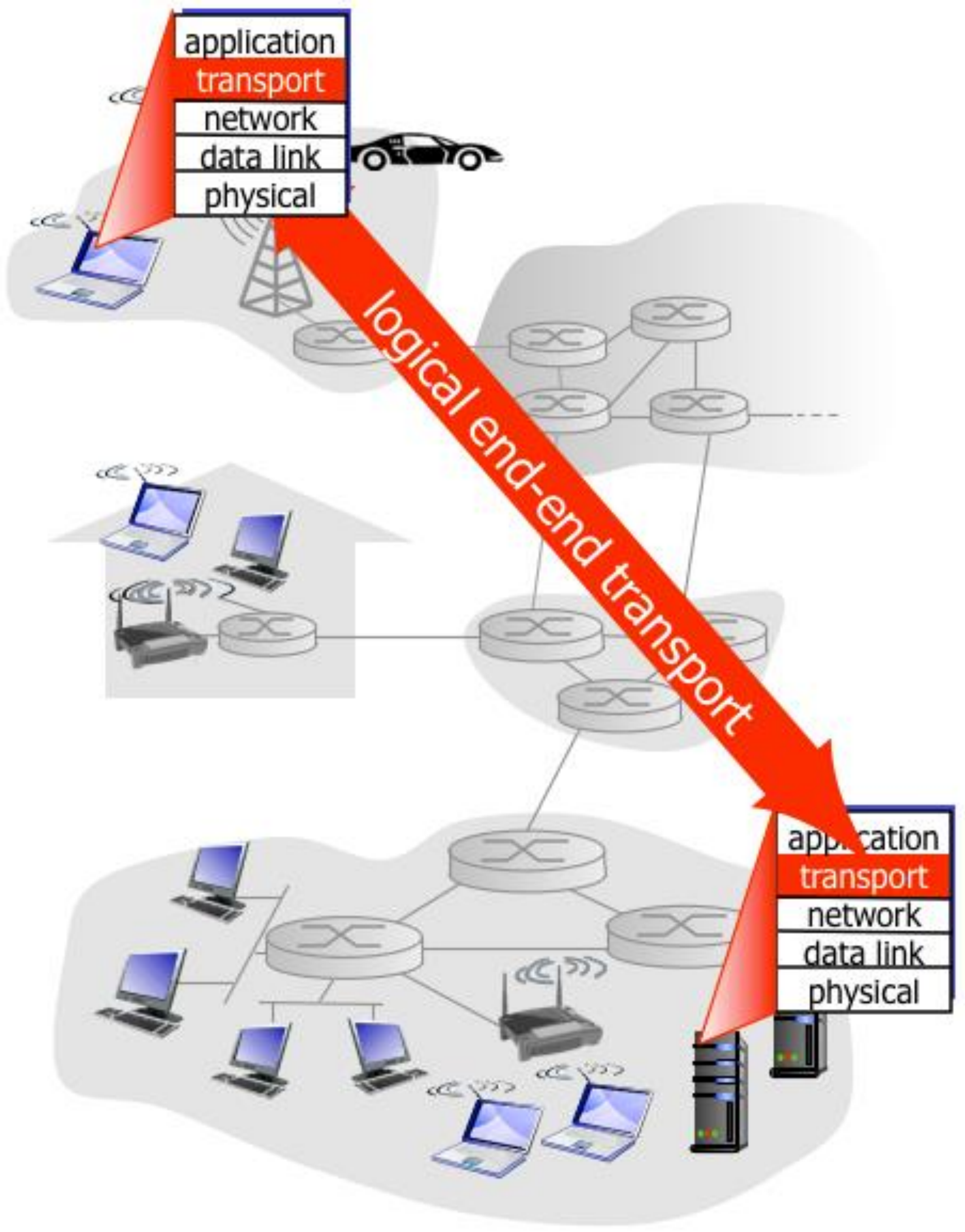


Transport vs. Network layer

- **Network layer: logical communication between hosts**
- **Transport layer: logical communication between processes**
  - Relies on, enhances, network layer services

Internet Transport-layer Protocols

- **Reliable, in-order delivery: TCP**
  - Congestion control
  - Flow control
  - Connection setup
- **Unreliable, unordered delivery: UDP**
  - No-frills extension of “best-effort” IP
- **Services not available:**
  - Delay guarantees
  - Bandwidth guarantees



Transport Layer Functions

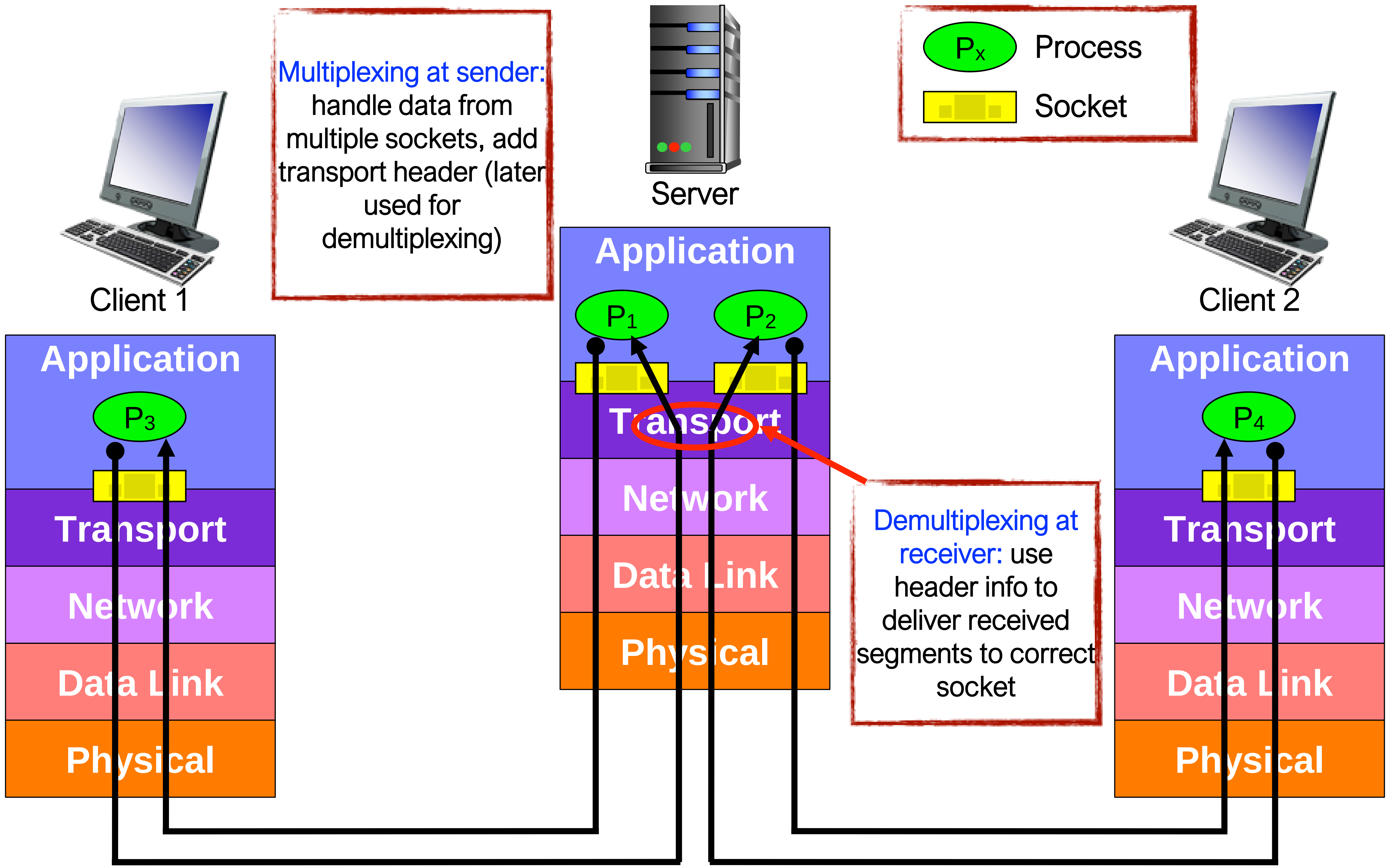
- **Multiplexing and demultiplexing among applications and processes at end systems**
- **Error detection of bit errors**
- **Loss detection - lost packets due to buffer overflow at intermediate systems**
- **Error/loss Recovery - retransmissions**
- **Flow Control - ensures receiver has buffer capacity to receive message**
- **Congestion Control - ensure the network has the capacity to transmit data**

Not all transport protocols provide all of the above functionality

Overview of Transport Layer

- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
  - Connectionless Demultiplexing
  - Connection-Oriented Demultiplexing
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
- **Principles of Congestion Control**
- **TCP Congestion Control**

Multiplexing/Demultiplexing

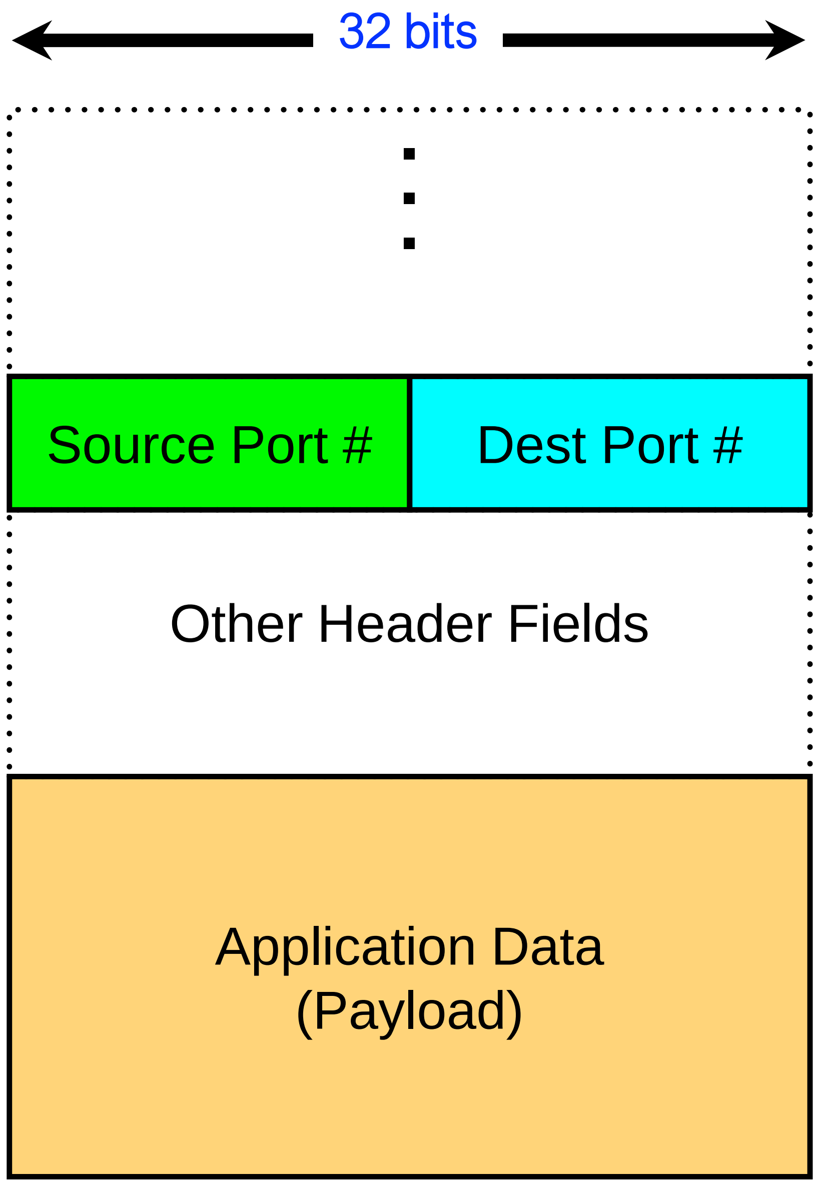


Multiplexing at sender: handle data from multiple sockets, add transport header (later used for demultiplexing)

Demultiplexing at receiver: use header info to deliver received segments to correct socket

How Demultiplexing Works

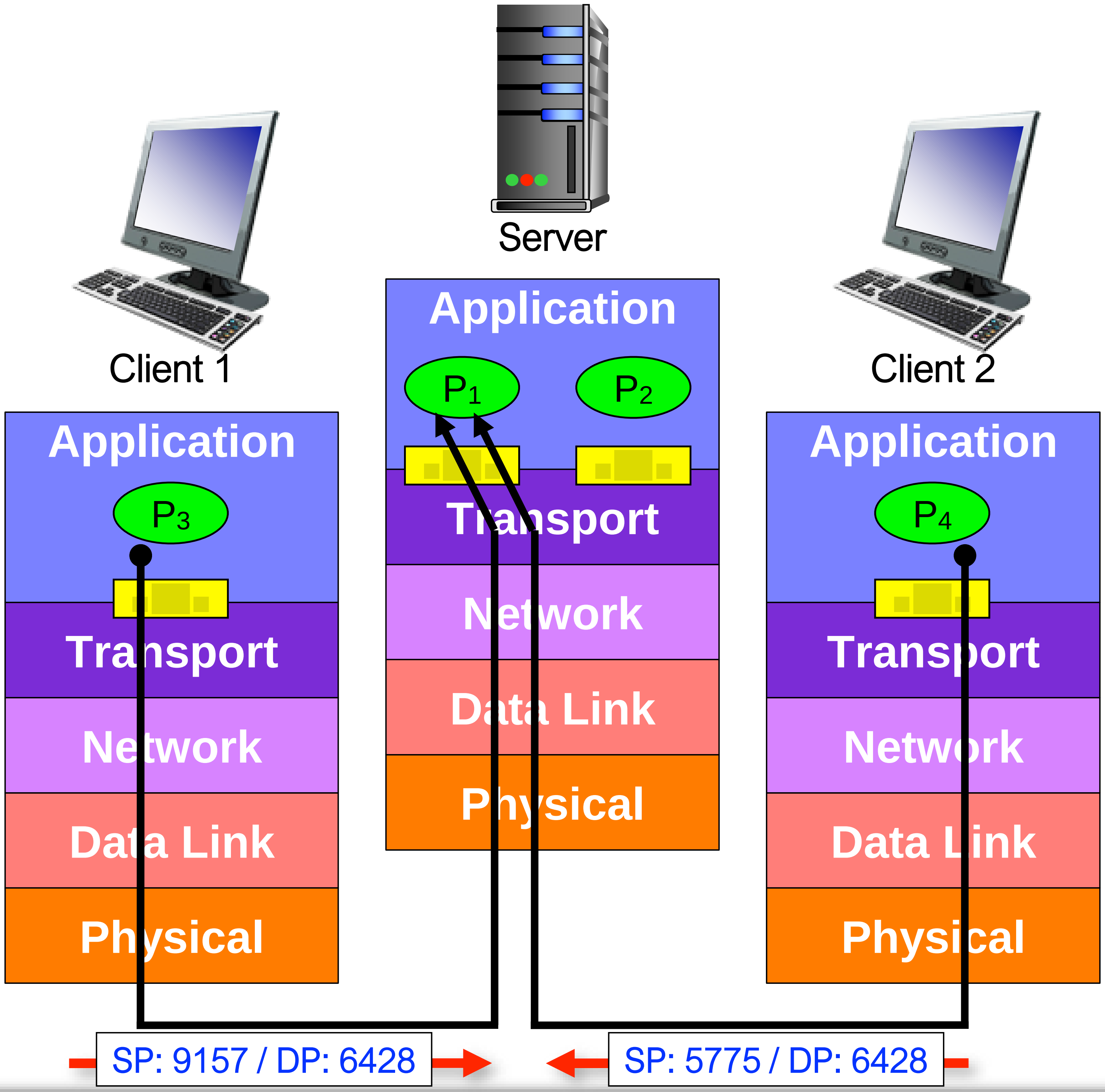
- **Host receives IP datagrams**
  - Each datagram has source IP address and destination IP address
  - Each datagram carries one transport-layer segment
  - Each segment has source and destination port number
- **Host uses IP addresses & port numbers to direct segment to appropriate socket**



TCP/UDP Segment Format

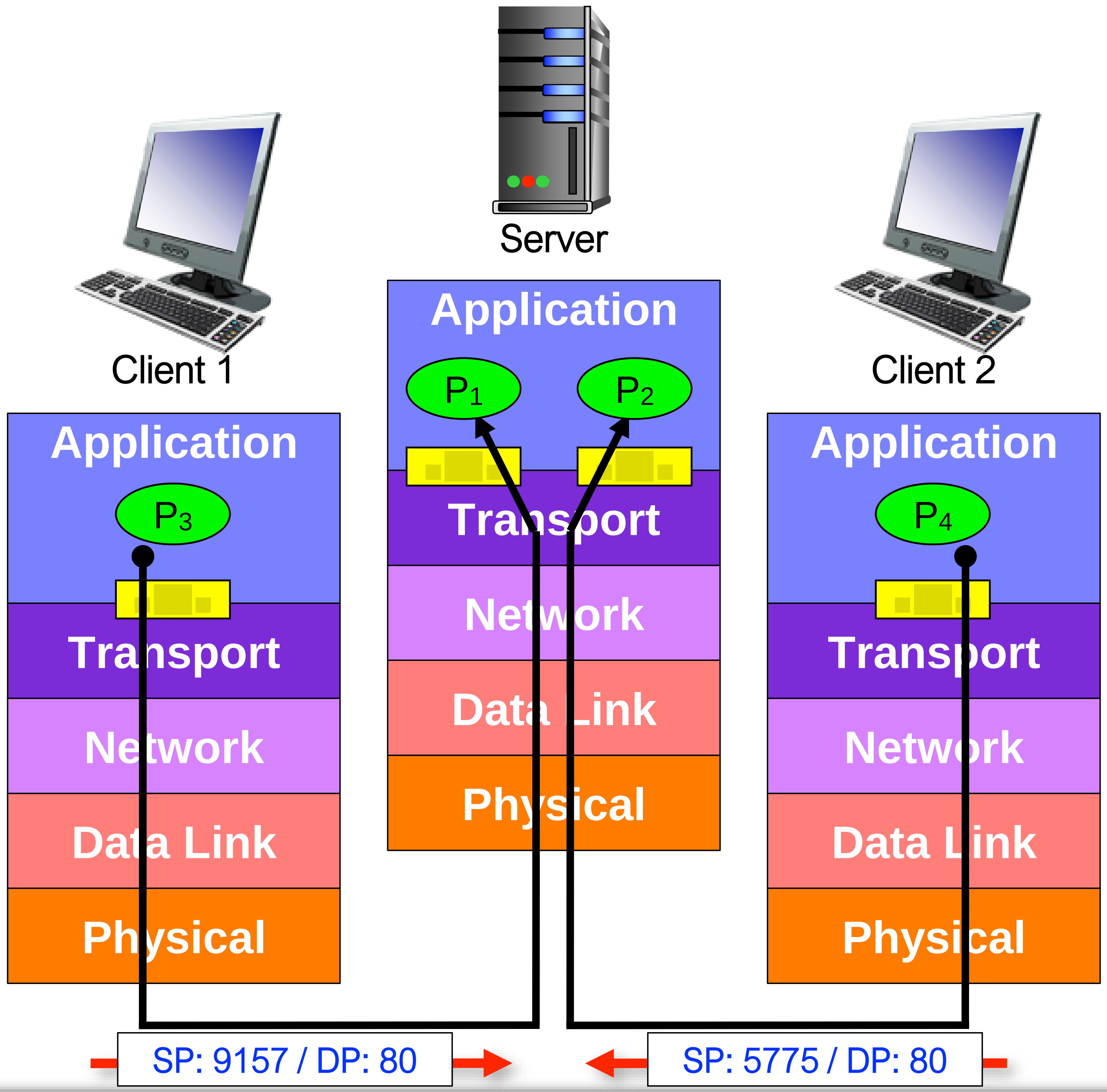
Connectionless Demux: Example (UDP)

- **Server UDP socket has a local port #**
  - Same socket is shared for incoming connections destined for that port #
- **Each client:**
  - Creates own local socket with own local port #
  - When sending UDP datagram to server, client must specify IP address & port # of server's UDP socket



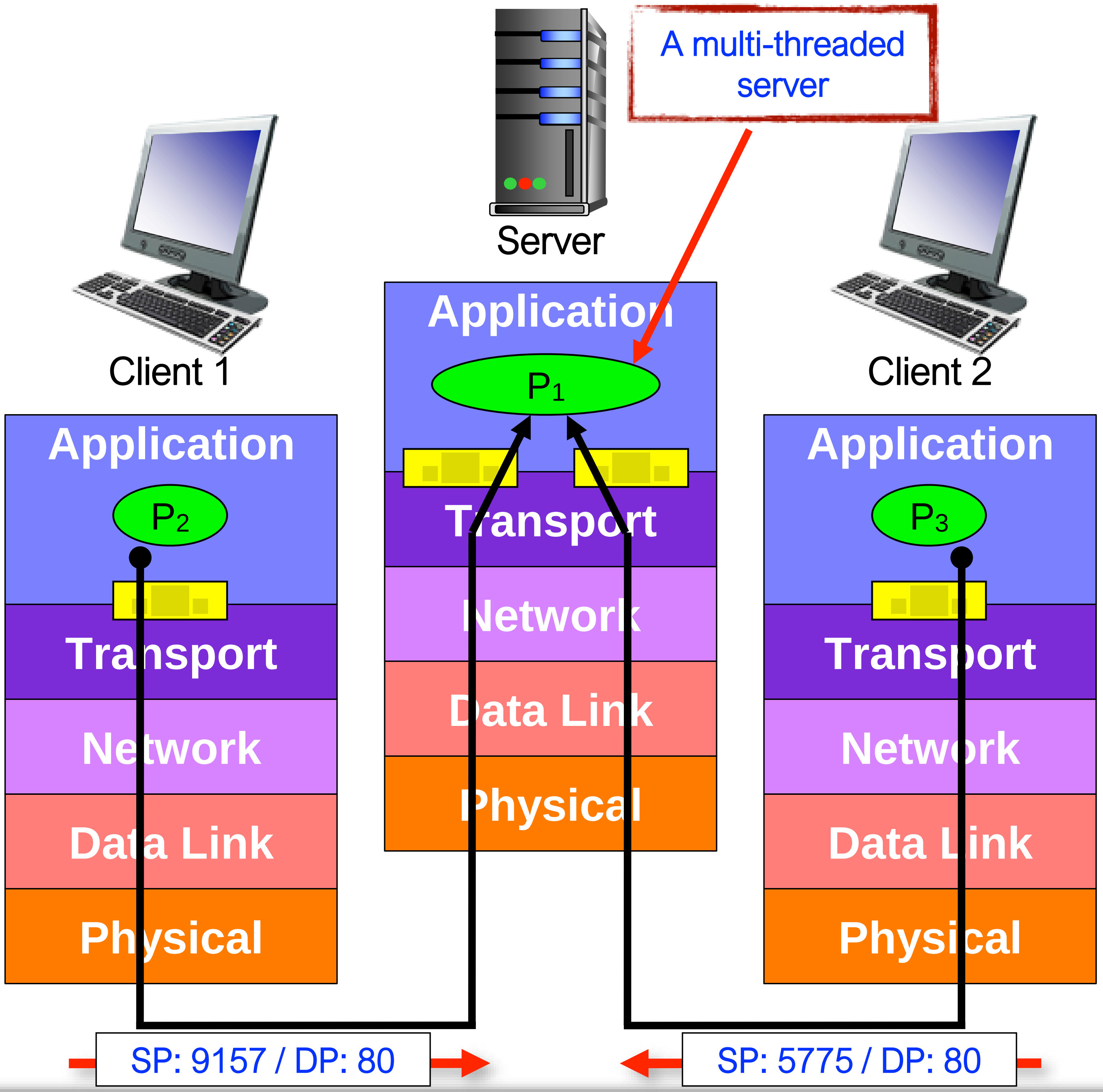
Connection-Oriented Demux: Example (TCP)

- **TCP socket is identified by a 4-tuple**
  - Source IP address, Source port number, Dest IP address, Dest port number
- **A new TCP socket is created for each unique 4-tuple**
  - Server host may support many simultaneous TCP sockets (even multiple from same client)



Connection-Oriented Demux: Example (TCP)

- **TCP socket is identified by a 4-tuple**
  - Source IP address, Source port number, Dest IP address, Dest port number
- **A new TCP socket is created for each unique 4-tuple**
  - Server host may support many simultaneous TCP sockets (even multiple from same client)



Overview of Transport Layer

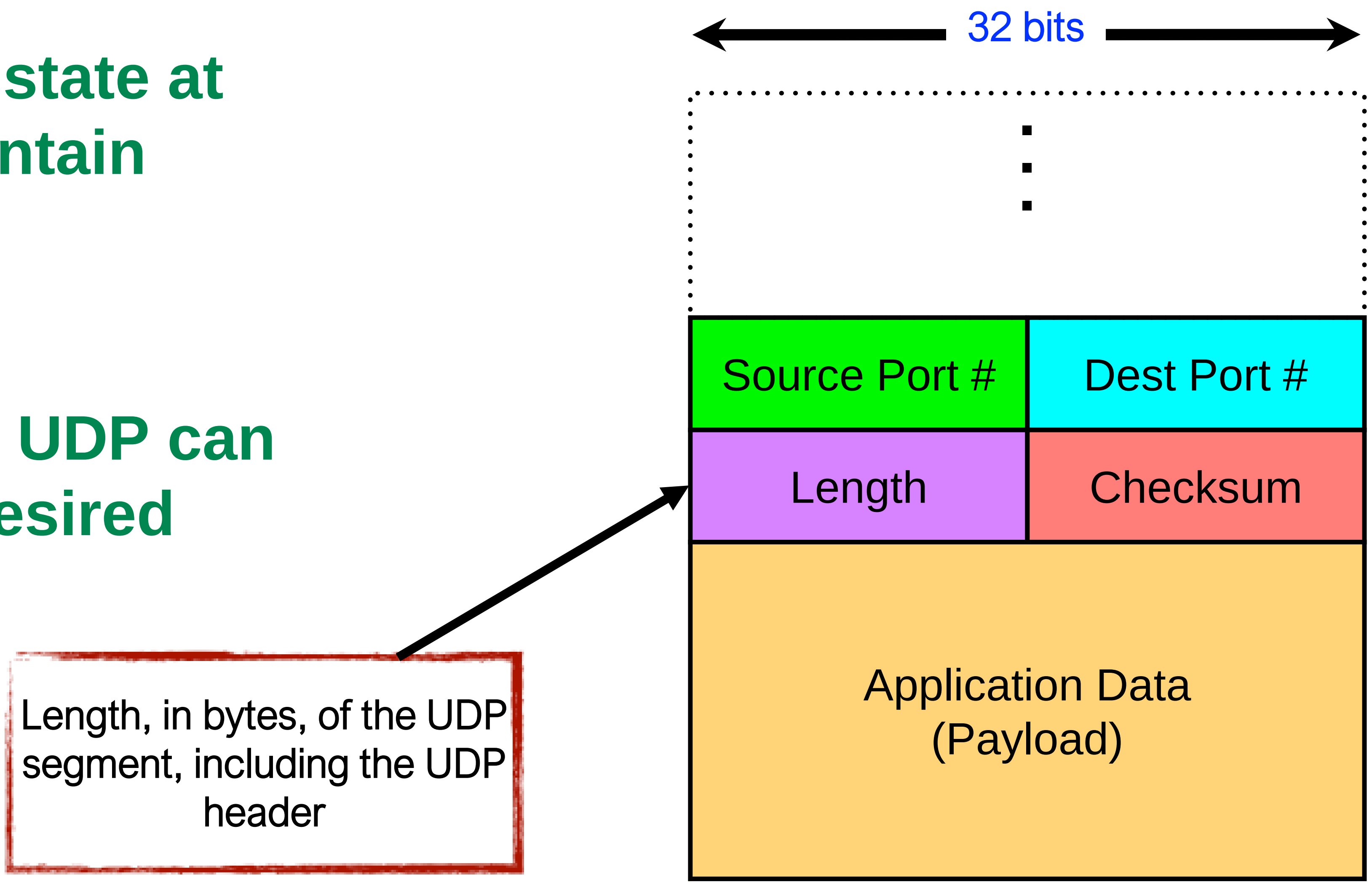
- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
  - Overview
  - Checksum
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
- **Principles of Congestion Control**
- **TCP Congestion Control**

UDP: User Datagram Protocol [RFC 768]

- **"No frills" / "bare bones" Internet transport protocol**
- **Best effort service, UDP segments may be:**
  - Lost
  - Delivered out-of-order to application
- **Connectionless:**
  - No handshaking between UDP sender and receiver
  - Each UDP segment is handled independently of others
- **UDP use:**
  - Streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS (Domain Name System)
  - SNMP (Simple Network Management Protocol)
- **Reliable transfer over UDP:**
  - Must add reliability at application layer
  - Application-specific error recovery!

UDP: User Datagram Protocol (Cont.)

- **No connection establishment**
  - Eliminates a source of delay
- **Simple: no connection state at sender, receiver to maintain**
- **Small header size**
- **No congestion control: UDP can blast away as fast as desired**



32 bits
Source Port #
Dest Port #
Length
Checksum
Application Data (Payload)
Length, in bytes, of the UDP segment, including the UDP header

UDP Segment Format

UDP Checksum

- **Used to detect errors (e.g. flipped bits) in transmitted data segment**
- **Sender:**
  - Treat segment contents, including the header fields, as sequence of 16-bit integers
  - Perform one's complement sum of segment contents, then take one's complement of that sum
  - Insert checksum value into UDP checksum field
- **Receiver:**
  - Compute one's complement sum of received segment (including checksum field)
  - Check if computed sum equals **`0xFFFF`**
    - YES - no error detected. But may have errors nonetheless? More later ….
    - NO - error detected

Checksum: Example

Example: add two 16-bit integers (a UDP checksum would add many more 16-bit integers)

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Partial Sum	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
																	1
Sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
Checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

- **In one's complement addition, add overflow back into the partial sum to get the sum**
- **Take one's complement (invert) of sum to get the checksum**

Overview of Transport Layer

- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
  - Overview
  - Pipelined Protocols
    - Go-Back-N
    - Selective Repeat
- **Connection-oriented Transport: TCP**
- **Principles of Congestion Control**
- **TCP Congestion Control**

Principles of Reliable Data Transfer

- **Reliable data transfer is very important application, transport, and link layers**

- **The characteristics of unreliable channel will determine the complexity of a Reliable Data Transfer (RDT) protocol**

- **To explore reliable data transfer, examine different types of loss and how to address them**

RDT 1.0: Reliable Transfer Over a Reliable Channel

- **Underlying data transmission channel is perfectly reliable**
  - No bit errors
  - No loss of packets

- **We don’t have networks like this, but it’s a good place to start**

RDT 2.0: Data Channel With Bit Errors

- **Underlying data channel may flip bits in packet**
  - Use a checksum to detect bit errors
- **Question: How should system recover from these errors?**
  - Acknowledgements (ACKs): receiver explicitly tells sender that a packet is received OK
  - Negative acknowledgements (NAKs): receiver explicitly tells sender that a packet had errors when it was received
    - Sender retransmits packet on receipt of a NAK
- **New mechanisms in RDT 2.0 (beyond RDT 1.0):**
  - Error detection
  - Feedback: send control messages (ACK, NAK) from receiver to sender

A Fatal Flaw in RDT 2.0, on to RDT 2.1

- **What happens if ACK/NAK messages are corrupted?**
  - Sender doesn't know if receiver correctly received the data
    - Data may have been corrupt on the way to receiver
    - ACK/NAK may have been corrupt on the way back to sender
  - Can't just retransmit since receiver may receive duplicate data!
- **Handling duplicates:**
  - Sender retransmits current packet if ACK/NAK is corrupted
  - Sender adds a sequence number to each packet
  - Receiver discards duplicate packets at Transport Layer
    - Those packets are not delivered up to the Application Layer

RDT 2.1: Discussion

- **Sender:**
  - Sequence number added to packets
  - Two sequence numbers (0,1) will suffice
  - Must check if received ACK/NAK corrupted
  - Sender must “remember” if it should be expecting a sequence number of 0 or 1

- **Receiver:**
  - Must check if received packet is duplicate
    - Receiver must “remember” if it should be expecting a sequence number of 0 or 1
  - Note: the receiver does not know if its last ACK/NAK message was received OK at sender

RDT 2.2: Eliminating the NAK Messages

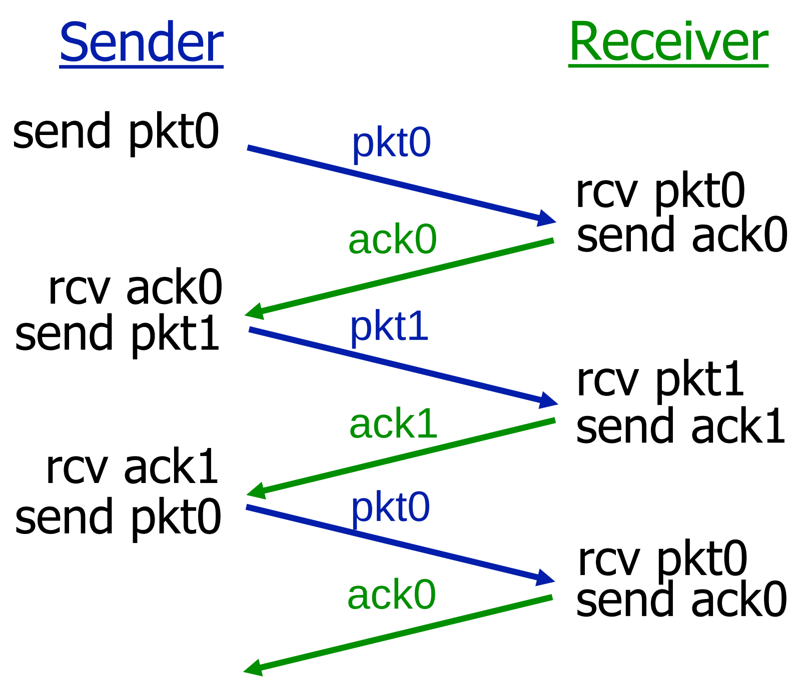
- **Possible to achieve same functionality as RDT 2.1 using ACKs messages only**
- **Receiver sends ACK messages for last packet that was received correctly**
  - No message is sent for a packet this is received with errors
- **Duplicate ACK messages received at sender results in same action as NAK from RDT 2.1 -- retransmit packet**
  - Duplicate ACK message would be detected at sender by receiving two consecutive ACK_0 messages or two consecutive ACK_0 messages

RDT 3.0: Channels with Bit Errors & Packet Loss

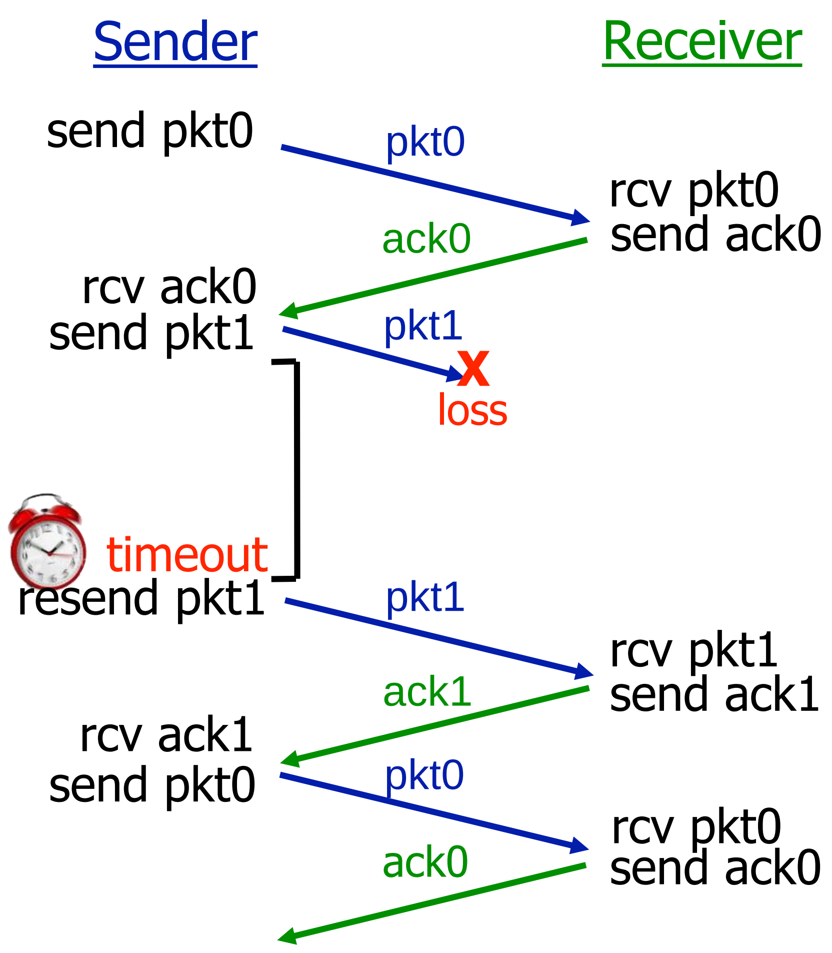
- **New assumption: underlying channel can also lose packets**
  - Can lose data packets or ACK messages
  - Checksum, sequence number, ACKs, retransmissions will be of help … but not enough

- **Approach: sender waits a “reasonable” amount of time for an ACK**
  - Retransmits packet if no ACK is received in this time
  - If packet (or ACK) is just delayed and not lost:
    - Retransmission will result in duplicate data, but sequence numbers from RDT 2.2. already handle this issue
    - Receiver must specify sequence number of packet being ACKed
  - Requires countdown timer

RDT 3.0 in Action with Packet Loss

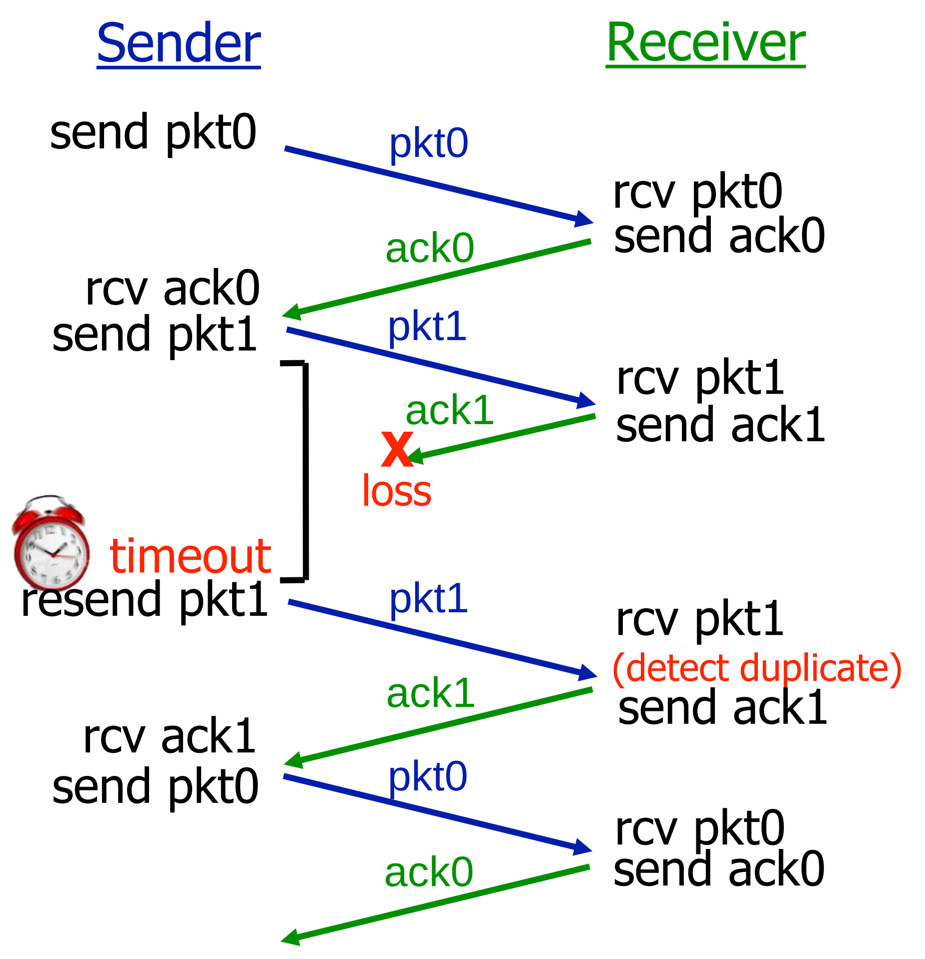


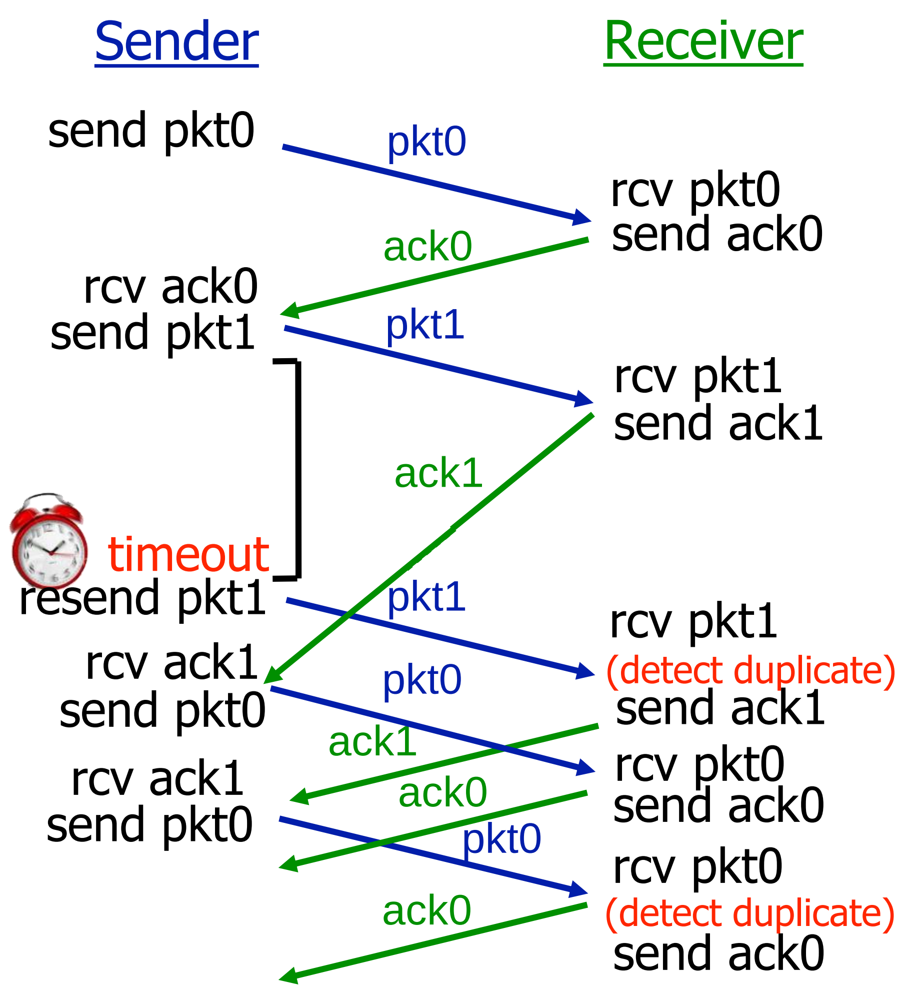
No Packet Loss



With Packet Loss

RDT 3.0 in Action with Lost/Delayed ACK





ACK Loss

Premature Timeout/Delayed ACK

Performance of RDT 3.0

- **RDT 3.0 will work reliably, but would have terrible performance**
  - RDT 3.0 utilizes a Stop-and-Wait protocol
    - Sender sends one packet, then waits for receiver response before sending the next packet

- **Example: Assume a system sending 8000 bit packets on a 1 Gbps link, where there is a 15 ms propagation delay between the sender and the receiver**

Time for sender to transmit data

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

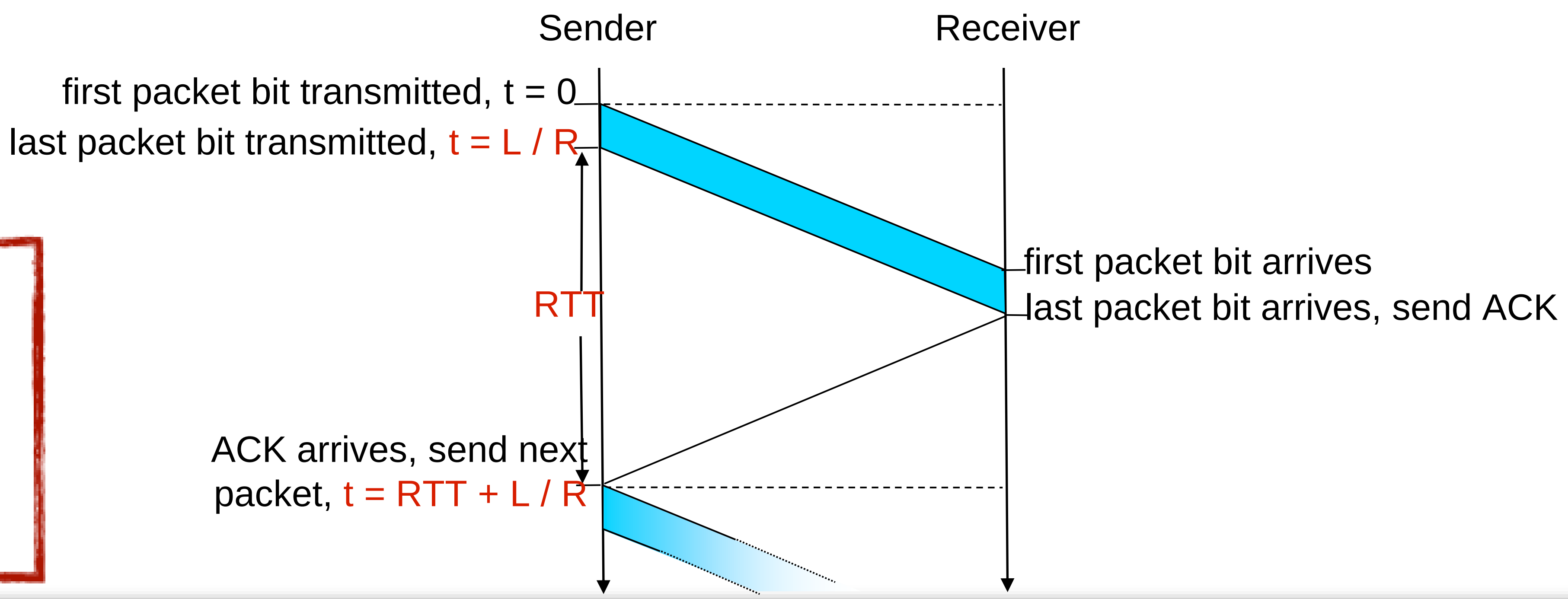
Performance of RDT 3.0 (Cont.)

- **If Round-Trip Time (RTT) is 30 ms:**
  - Sender is only sending 8 microseconds
  - Can only send a new packet every 30.008 microseconds
  - Effectively makes 1 Gbps link run at ~270 Kbps!!!

Utilization: Fraction of time the sender is busy sending

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

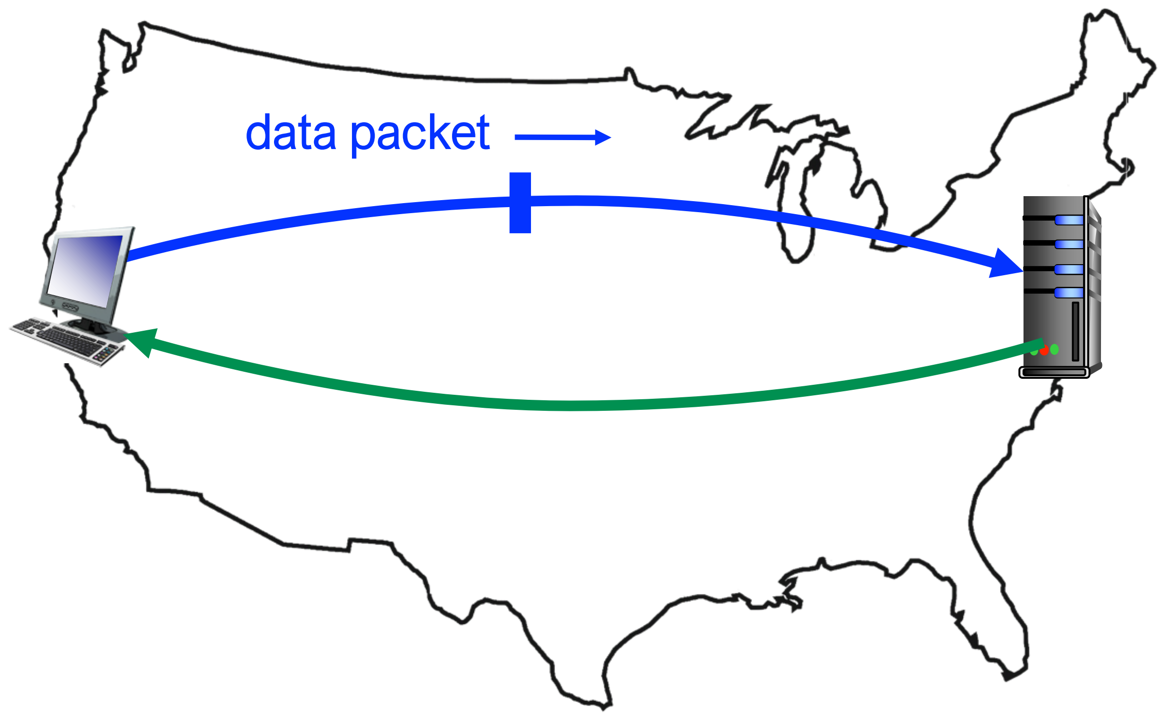
.027%


Sender
Receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

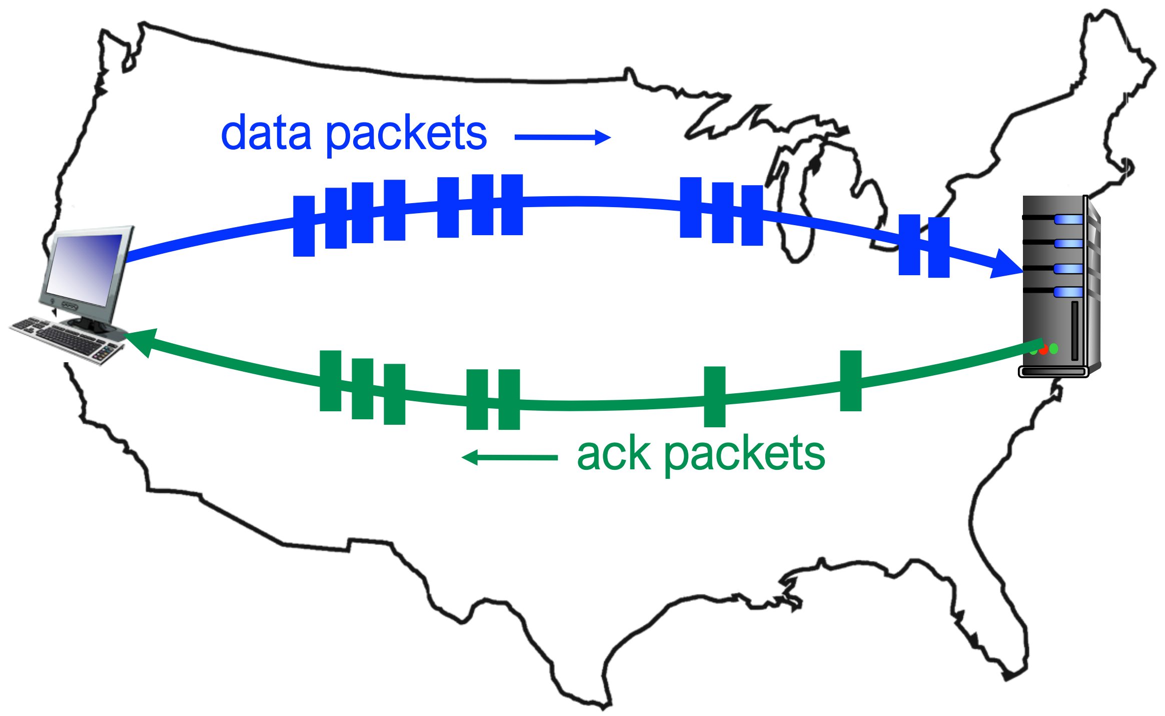
The network protocol limits use of physical resources!

Pipelined Protocols

- **Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets**
  - Range of sequence numbers must be increased (0 and 1 will no longer suffice)
  - Buffering at sender and/or receiver is required
- **Two generic forms of pipelined protocols: Go-Back-N, and Selective Repeat**

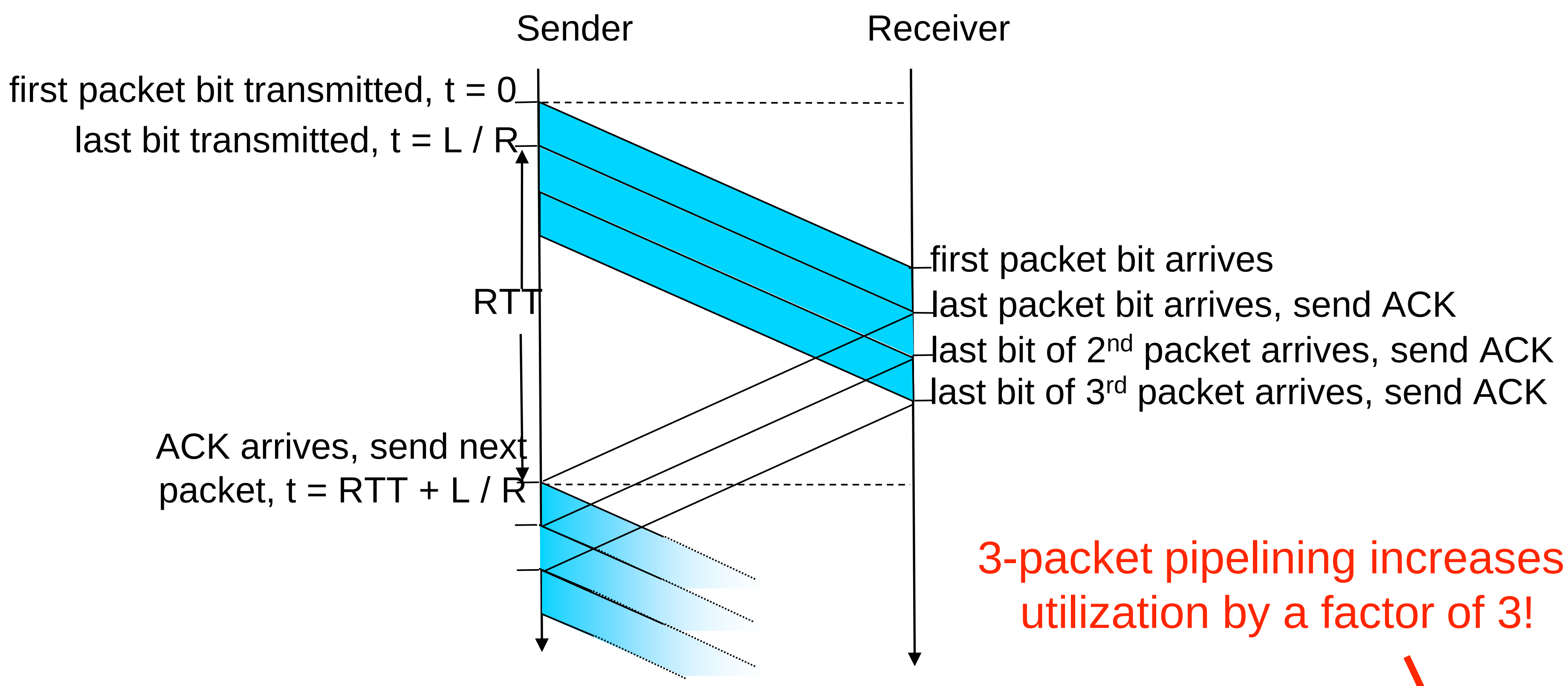


Stop-and-Wait Protocol



Go-Back-N Protocol

Pipelining: Increased Utilization


Sender
Receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

3-packet pipelining increases utilization by a factor of 3!

Utilization: Fraction of time the sender is busy sending

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.024}{30.008} = 0.0008$$

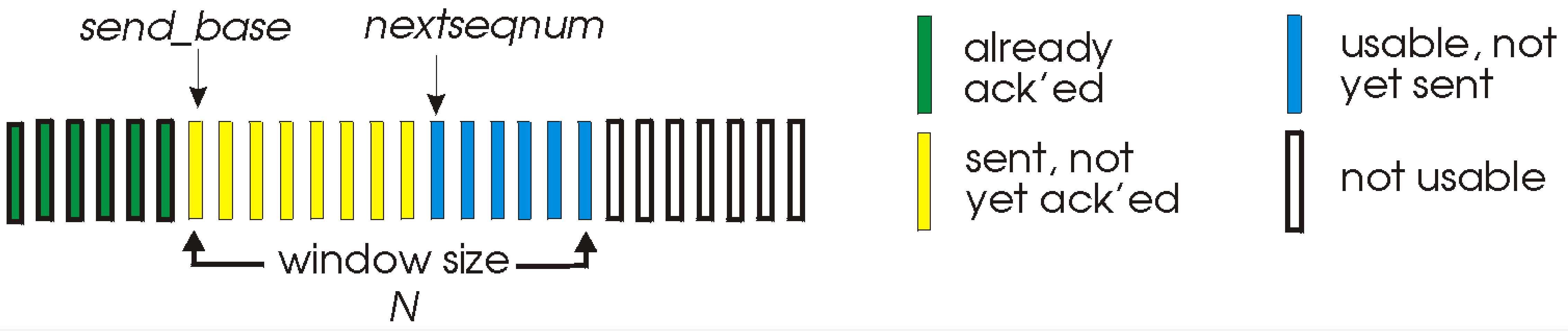
Pipelined Protocols: Overview

- **Go-back-N:**
  - Sender can have up to N unacked packets in pipeline
  - Receiver only sends cumulative acks
    - Doesn't ack a packet if there is a gap
  - Sender has a timer for the oldest unacked packet
    - When timer expires, retransmit ***all*** unacked packets

- **Selective Repeat:**
  - Sender can have up to N unacked packets in pipeline
  - Receiver sends individual acks for each packet
  - Sender maintains timer for each unacked packet
    - When timer expires, retransmit only that unacked packet

Go-Back-N: Sender Side

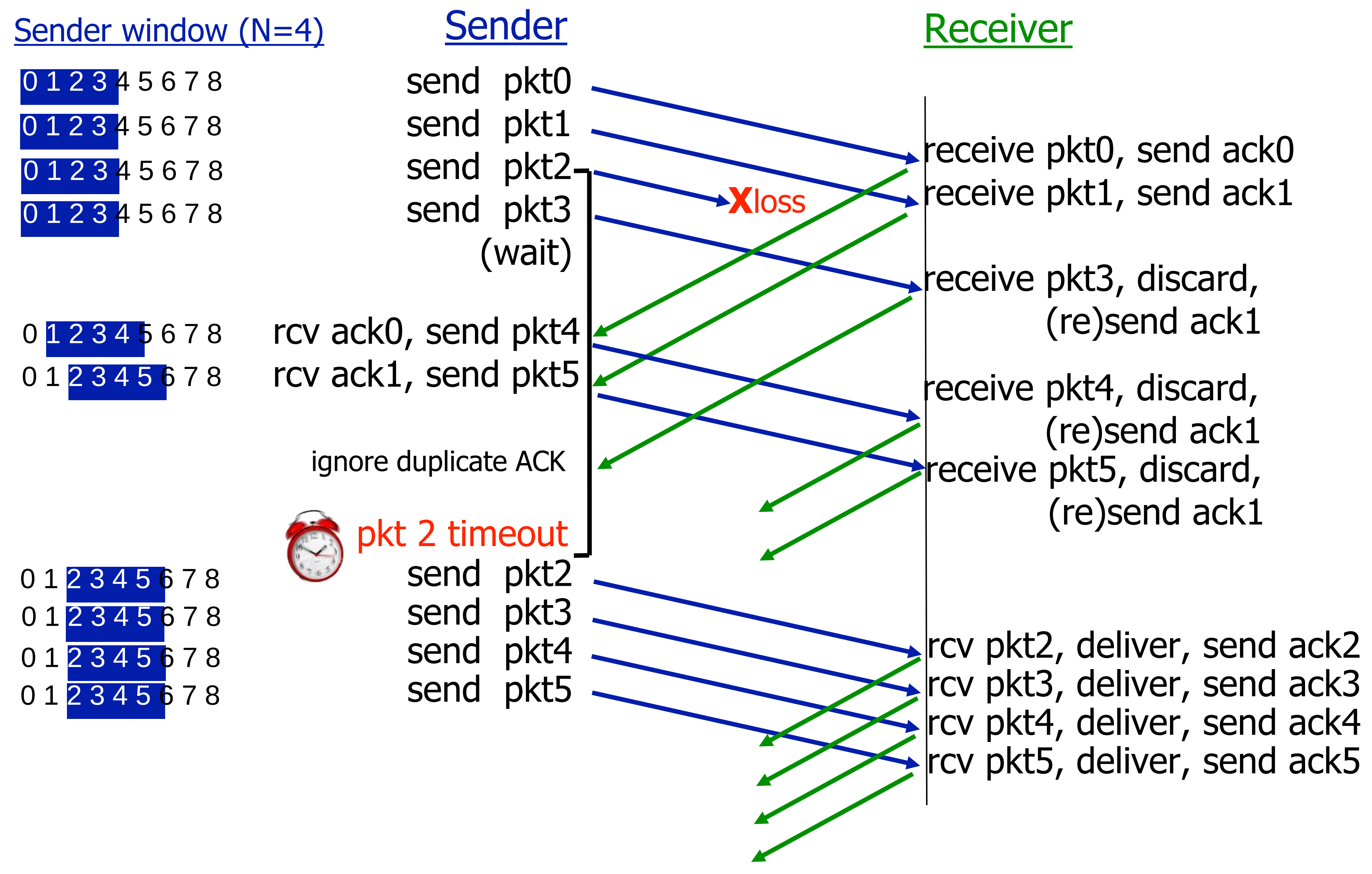
- **Sender can transmit multiple packets without waiting for an ack**
  - A “sliding window” of up to N consecutive unacked packets allowed
- **Include a k-bit sequence number in the packet header**
- **ACK(n): ACKs all packets up to and including sequence number n**
  - Cumulative ACK
  - May receive duplicate ACKs
- **Maintain timer for oldest in-flight packet**
- **Timeout(n): retransmit packet n and all higher sequence number packets in window**



Go-Back-N: Receiver

- **Send ACK message for correctly-received packet with highest in-order sequence number**
  - May generate duplicate ACKs
  - Need only remember expected sequence number
  - Must receive packets in-order to send ACK
- **Out-of-order packets:**
  - Discard (don't buffer) out-of-order packets (no receiver buffering)
    - Yes, even if they are correctly formatted and error-free
    - Will be retransmitted anyway based on sender's rules
  - Re-ACK packet with highest in-order sequence number

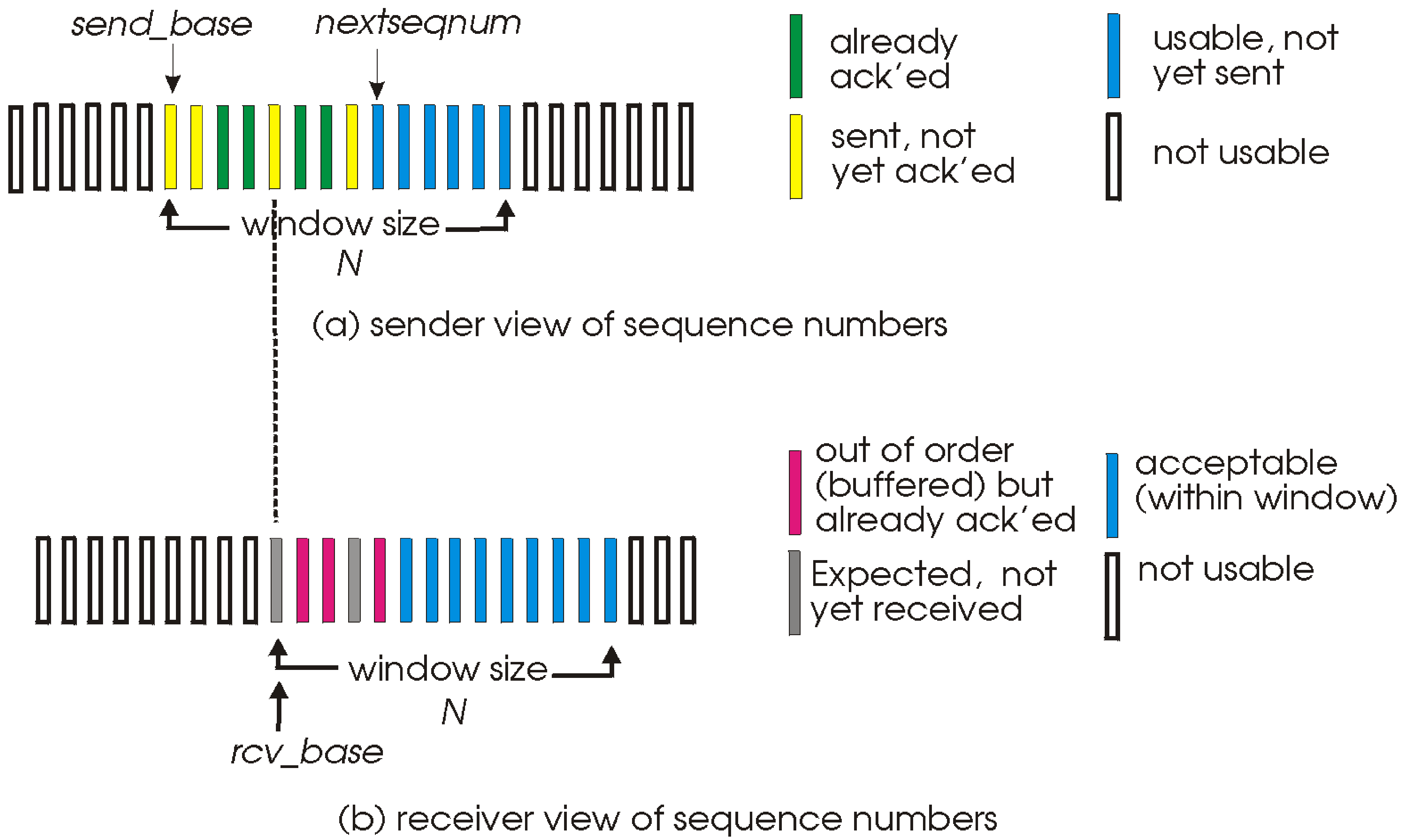
Go-Back-N in Action



Selective Repeat

- **Receiver individually acknowledges all correctly received packets**
  - Out-of-order packets are buffered at the receiver
  - Buffered packets are eventual delivered in-order to upper layer
- **Avoids unnecessary retransmission -- sender only resends packets for which an ACK was not received**
  - Sender has separate timer for each unACKed packet
- **Sender window**
  - N consecutive sequence numbers
  - Limits sequences numbers of sent, unACKed packets

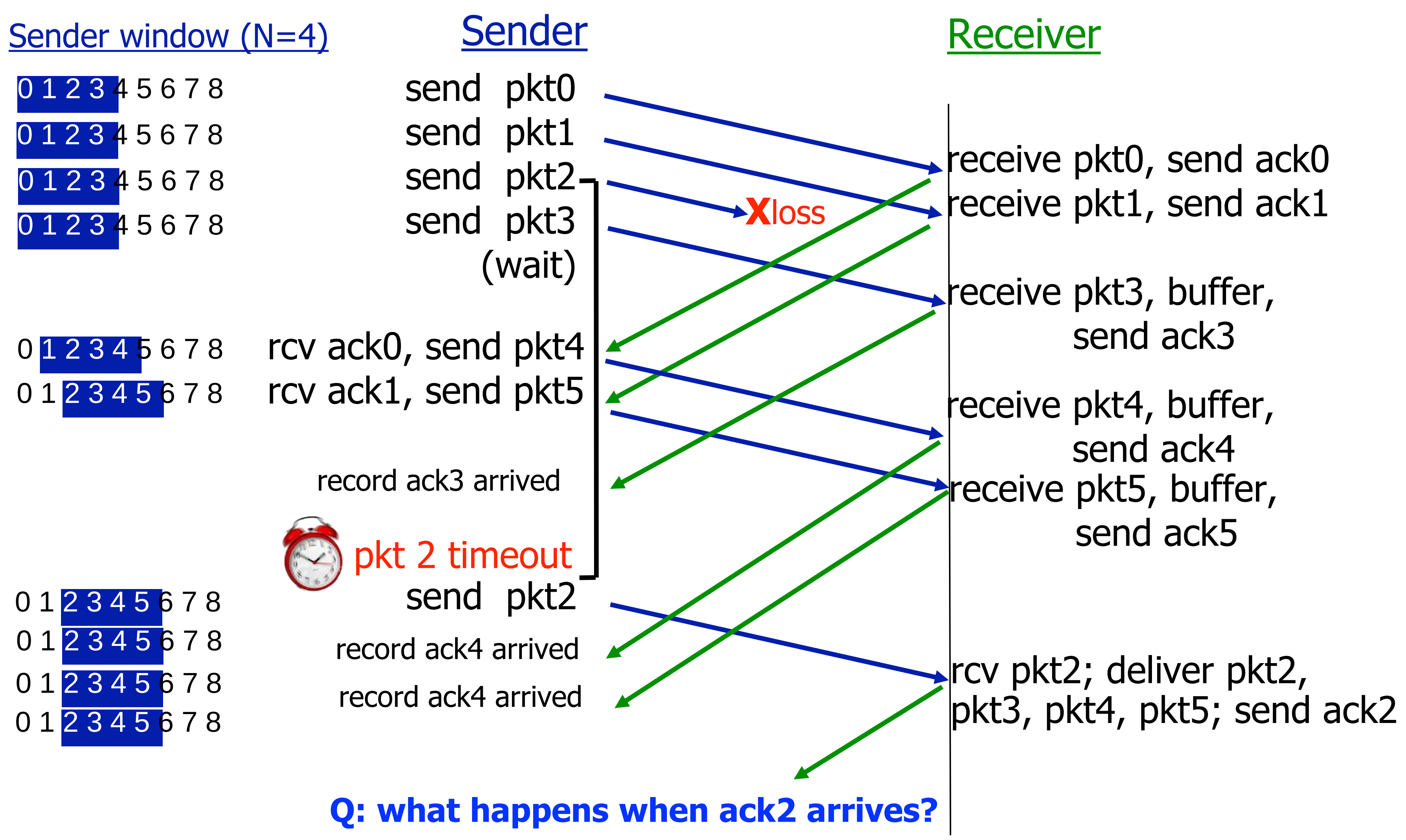
Selective Repeat: Sender/Receiver Windows


send_base
nextseqnum
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable
window size N
(a) sender view of sequence numbers
out of order (buffered) but already ack'ed
Expected, not yet received
acceptable (within window)
not usable
window size N
rcv_base
(b) receiver view of sequence numbers

Selective Repeat

- **Sender**
  - Receives data from upper layer:
    - If next available sequence number is in the sliding window, send the packet
  - A timeout occurs for packet n:
    - Resend packet n, restart timer
  - ACK(n) received for packet in current window:
    - Mark packet n as received
    - If n is smallest unACKed packet in window, advance the window base to next unACKed sequence number
- **Receiver**
  - Receives packet n in receivers window
    - Send ACK(n)
    - If out-of-order, buffer
    - If in-order, deliver with other buffered, in-order packets to the upper layer. Also, advance window to next not-yet-received packet
  - Receives packet n that has already been seen and ACKed by the receiver
    - Send ACK(n) again
  - Otherwise:
    - Ignore the packet

Selective Repeat in Action



Q: what happens when ack2 arrives?

Overview of Transport Layer

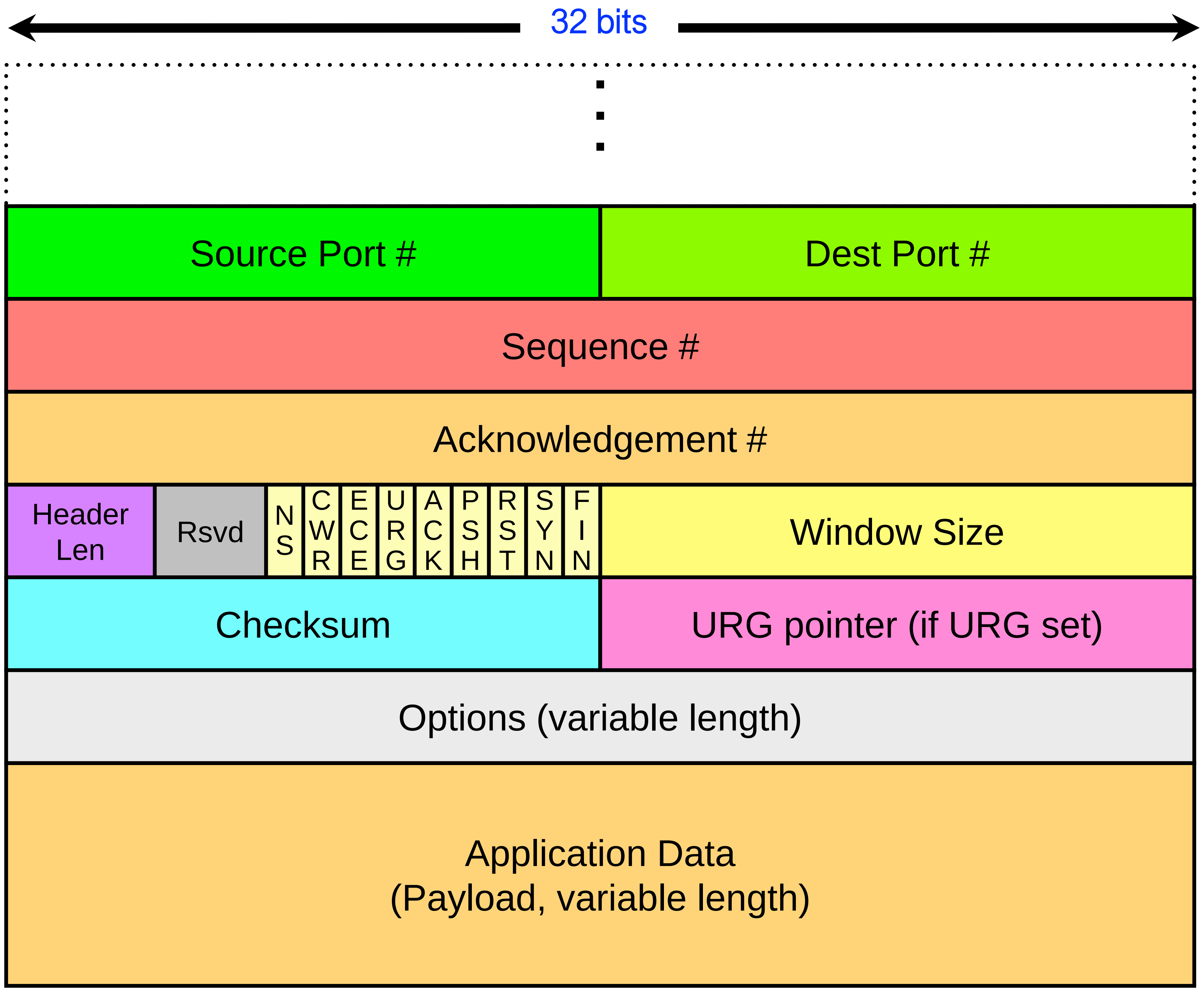
- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- **Principles of Congestion Control**
- **TCP Congestion Control**

TCP: Overview

- **A point-to-point protocol - one sender, one receiver**
- **Provides a reliable, in-order byte stream**
- **A pipelined protocol -- allows multiple in-flight packets**
  - TCP congestion and flow control set window size
- **Full duplex data - bi-directional data flow on same connection**
- **Connection-oriented:**
  - Handshaking (the exchange of control messages) initializes sender and receiver state before data exchange starts
- **Flow controlled:**
  - Sender will not overwhelm receiver

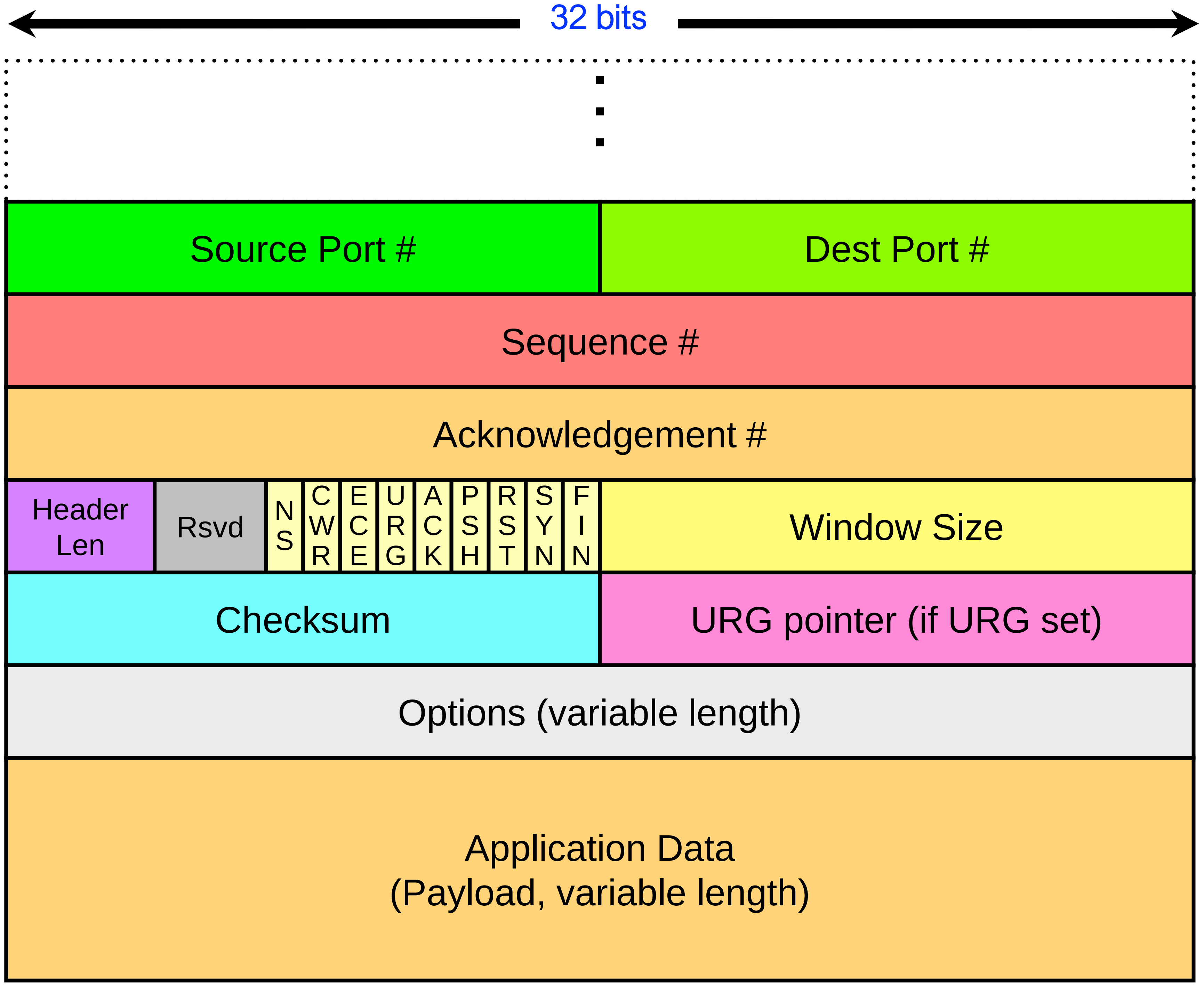
TCP Segment Structure

- **Source Port # - the port on the sender**
- **Dest Port # - the port on the receiver**
- **Sequence # - 32-bit number that represents the byte stream ‘number’ of the first byte in this segment’s data**
  - e.g. if Sequence # is 10 and there are 7 bytes of data in this packet, then this segment contains bytes 10-16 of the data stream
- **Acknowledgement # - 32-bit number that represents the next sequence number that the receiver is expecting**



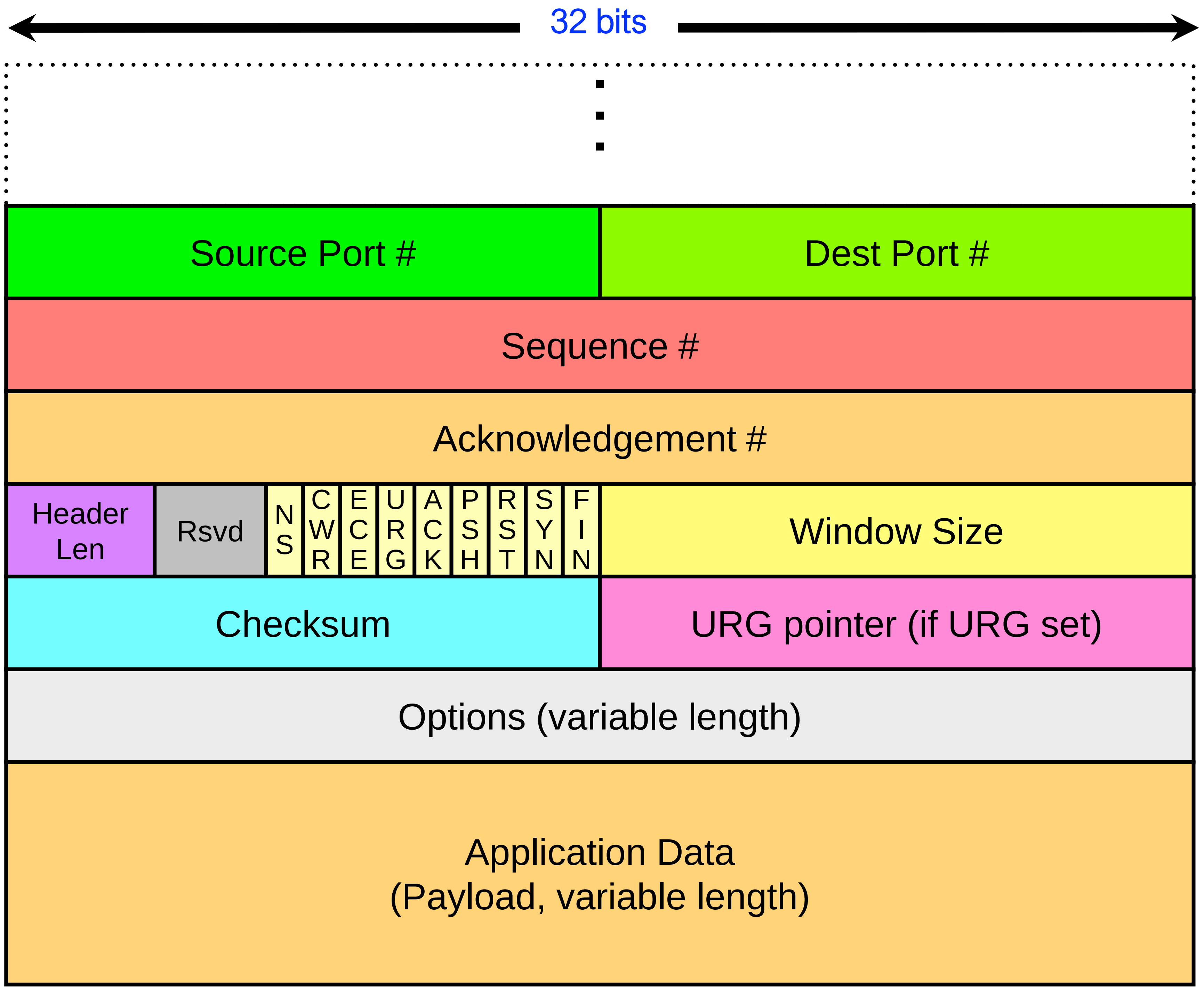
TCP Segment Structure (Cont.)

- **Header Length - 4-bit field that specifies the size of the TCP header in 32-bit words**
  - min = 5 ; max = 16
- **Reserved - 3-bits, not currently used**
- **Window Size - the size of the receive window that specifies the number of bytes that the receiver of this packet is currently willing to receive**
- **URG Pointer - specifies an offset of urgent data**
- **Checksum - 16-bit checksum computed over header and data (similar to UDP)**
- **Options - optional header fields**



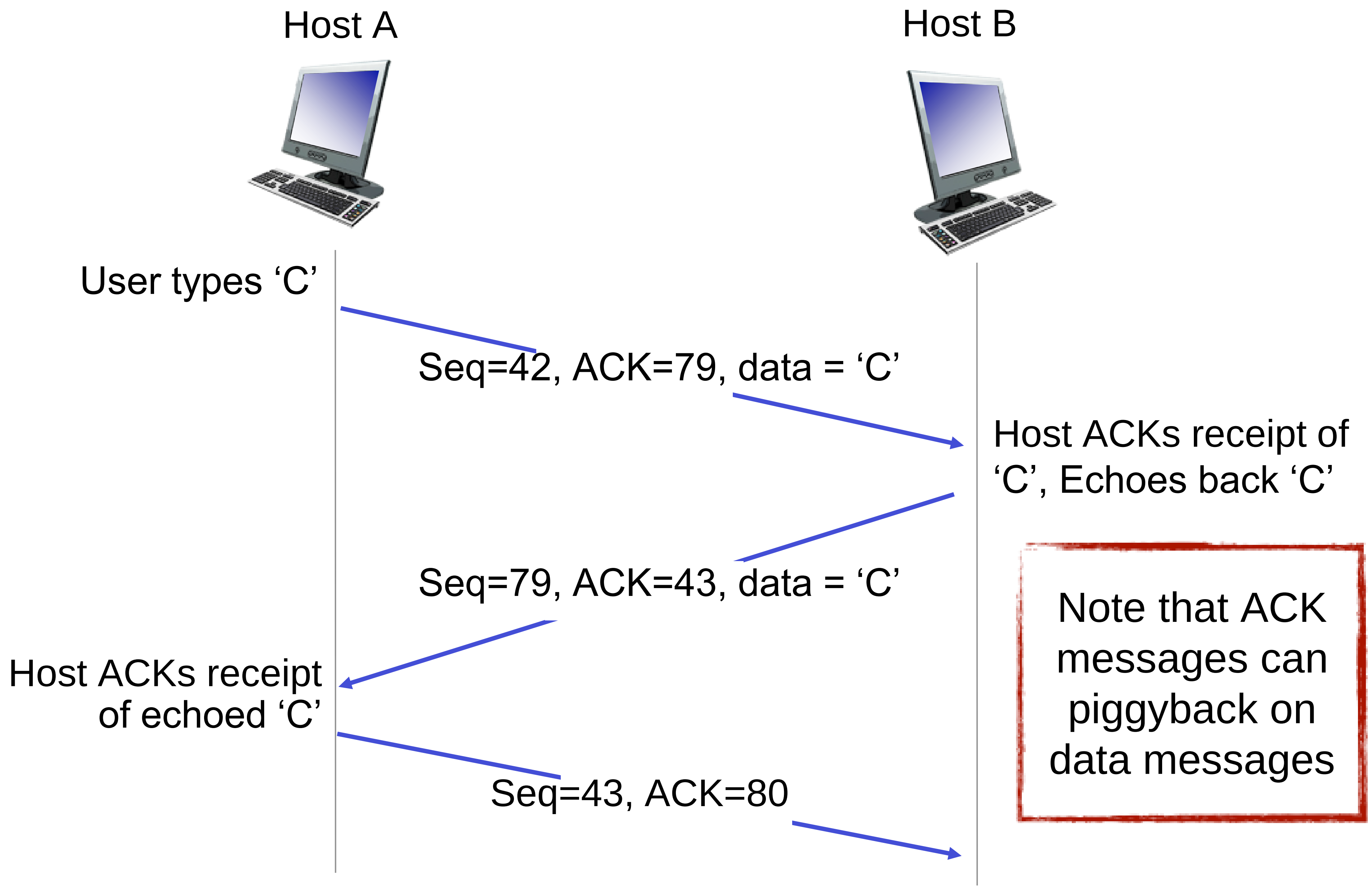
TCP Segment Structure: Flags (1-bit each)

- **NS, CWR, & ECE - used in congestion mechanism**
- **URG - indicates that an URG pointer is present (not used much)**
- **ACK - indicates that the ACK # is significant**
- **PSH - request to push data to receiving application (not used much)**
- **RST - reset the connection**
- **SYN - synchronize sequence numbers. First packet from each end of connection should set this.**
- **FIN - indicates that there is no more data from the sender**



TCP Sequence Numbers/ACKs

Simple Telnet Scenario


Host A
Host B
User types ‘C’
Seq=42, ACK=79, data = ‘C’
Host ACKs receipt of ‘C’, Echoes back ‘C’
Seq=79, ACK=43, data = ‘C’
Host ACKs receipt of echoed ‘C’
Seq=43, ACK=80

Note that ACK messages can piggyback on data messages

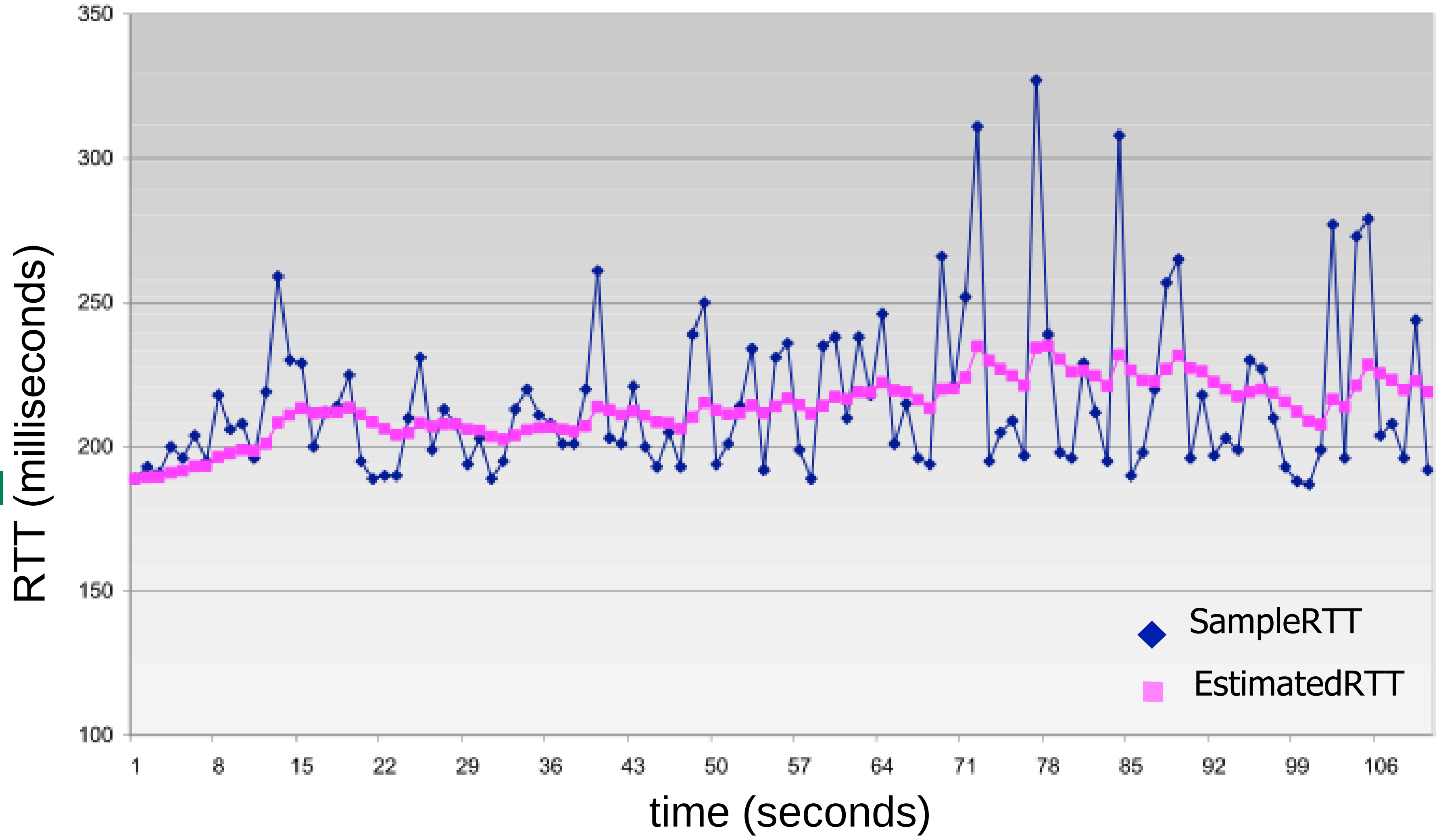
TCP Round Trip Time & Timeout

- **How should the TCP timeout value be set?**
  - Must be longer than RTT -- but RTT varies
  - Too short and timer will timeout prematurely causing unnecessary retransmissions of data
  - Too long and sender will have a slow reaction to segment loss
- **How can the RTT be estimated?**
  - Sample the RTT -- measure the time from segment transmission until ACK receipt (`SampleRTT`)
    - Ignore retransmissions
    - `SampleRTT` will vary, need to compute average over time to 'smooth' value

TCP Round Trip Time & Timeout

- **Compute `EstimatedRTT` from `SampleRTT`**
- **Exponentially weighted moving average**
- **Influence of past sample decreases exponentially fast**
- **Typical value: `α = 0.125`**
- **Use `EstimatedRTT` plus some safety-margin to determine timeout interval**

```
EstimatedRTT =
   (1-α)*EstimatedRTT + α*SampleRTT
```



Overview of Transport Layer

- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- **Principles of Congestion Control**
- **TCP Congestion Control**

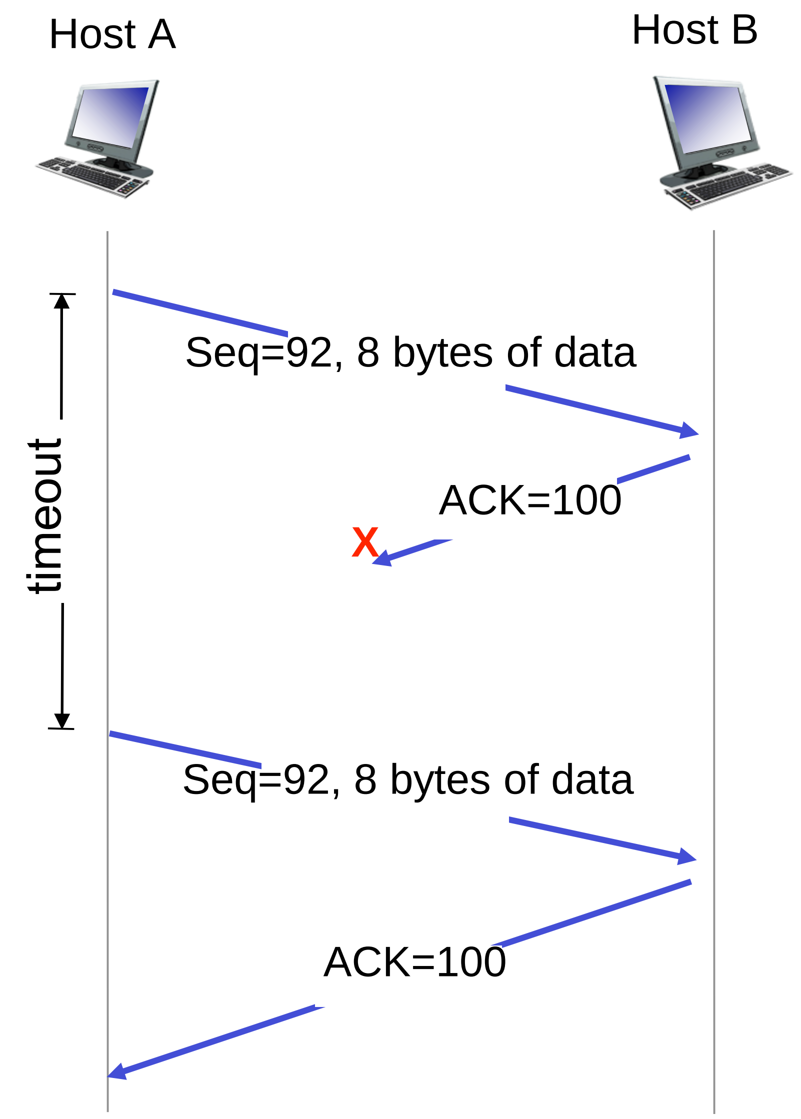
TCP Reliable Data Transfer

- **TCP creates reliable data transfer service on top of the IP protocol's unreliable service**
  - Send data as pipelined segments
  - Uses cumulative ACKs
  - Uses a single retransmission timer
- **Retransmissions are triggered by:**
  - Timeout events
  - Duplicate ACK messages

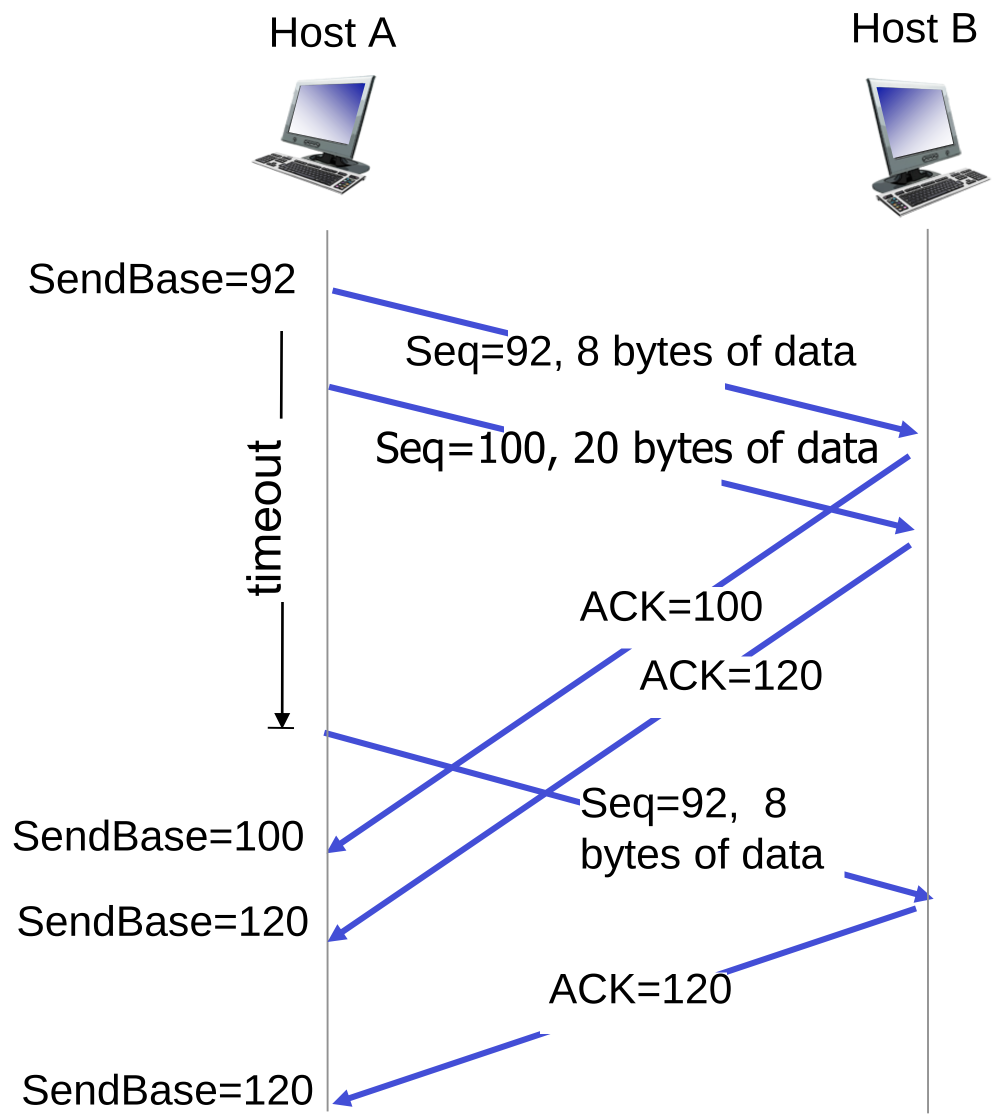
TCP Sender Events

- **Data received from application layer**
  - Create segment with sequence number
  - Sequence number is byte-stream number of the first data byte in segment
  - Start a timer if not already running
    - Timer is for oldest unacked segment
    - Use timeout interval computed from sampling RTT
- **If timeout occurs:**
  - Retransmit the segment that caused timeout
  - Restart the timer
- **When ACK is received:**
  - If ACK is for previously unacked segments
    - Update what is known to be ACKed
    - Restart timer if there are still unacked segments

TCP: Retransmission Scenarios

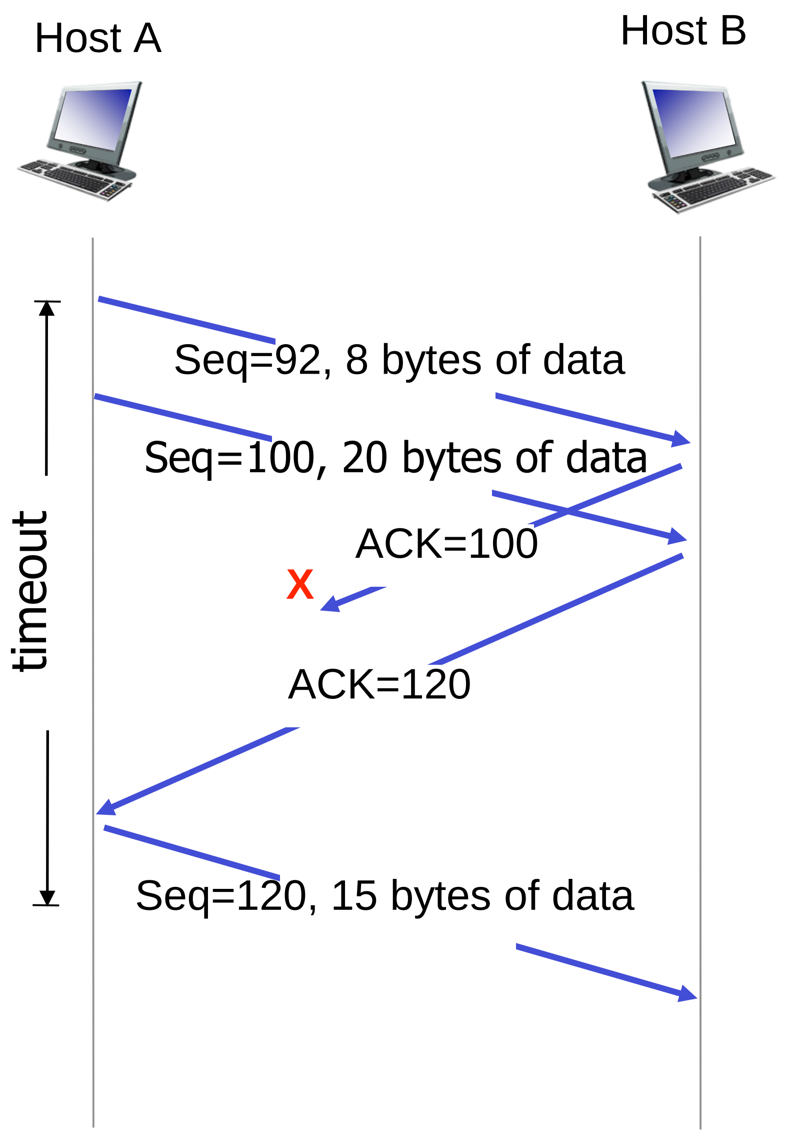


Lost ACK



Premature Timeout

TCP: Retransmission Scenarios (Cont.)



Cumulative ACK

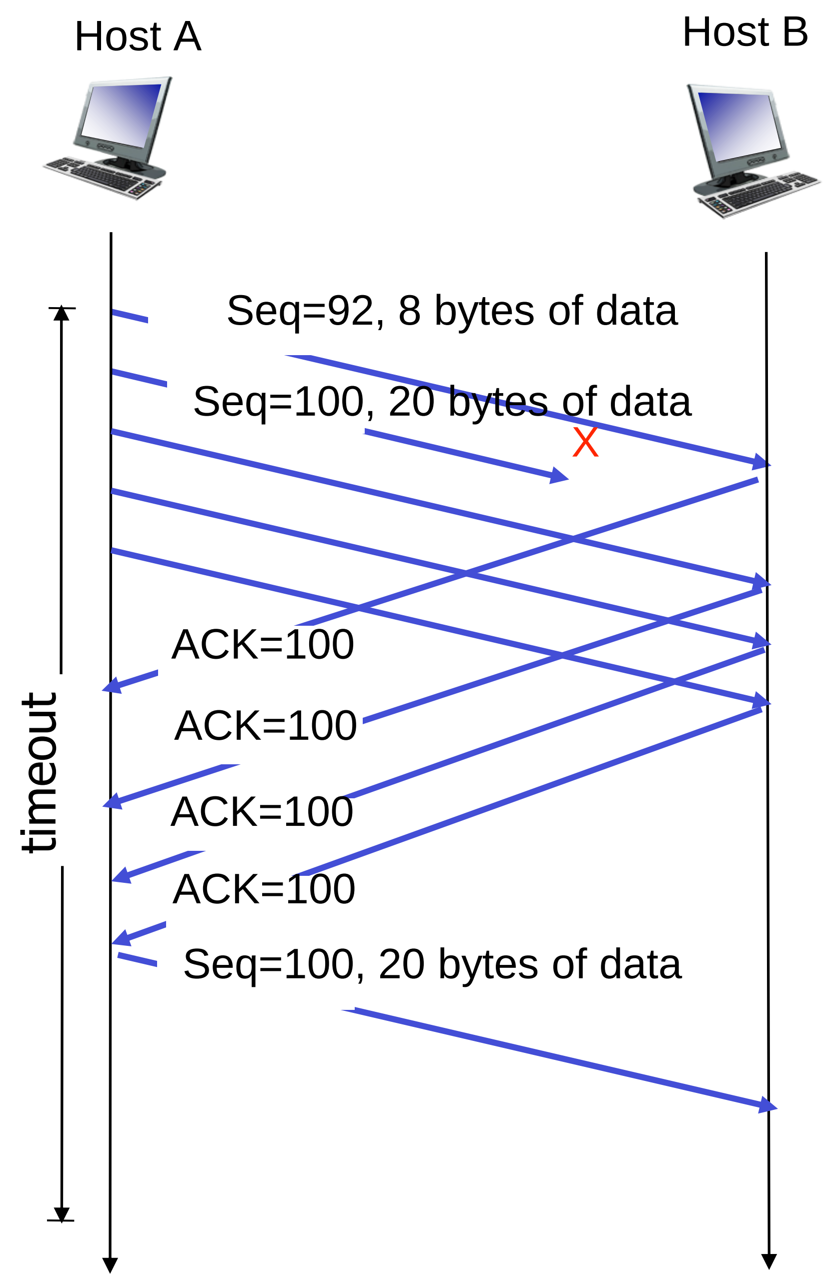
TCP ACK Generation

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 500 ms for next segment. If no segment arrives, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send a single cumulative ACK, ACKing both 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 a segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

TCP Fast Retransmit

- **Time-out period for a segment is often relatively long**
  - Means there is a long delay before sender resends lost packet
- **Lost segments can usually be detected via duplicate ACKs before timeout occurs on that segment**
  - Sender often sends many segments back-to-back
  - If a segment is lost, there will likely be many duplicate ACKs indicating that the lost segment should be retransmitted
- **TCP fast retransmit protocol**
  - If sender receives 3 ACKs for same data (triple duplicate ACKs), resend unacked segment with the smallest sequence number
    - Very likely that the unacked is segment lost, so don't wait for timeout

TCP Fast Retransmit (Cont.)



Fast retransmit after sender receipt of triple duplicate ACKs

Overview of Transport Layer

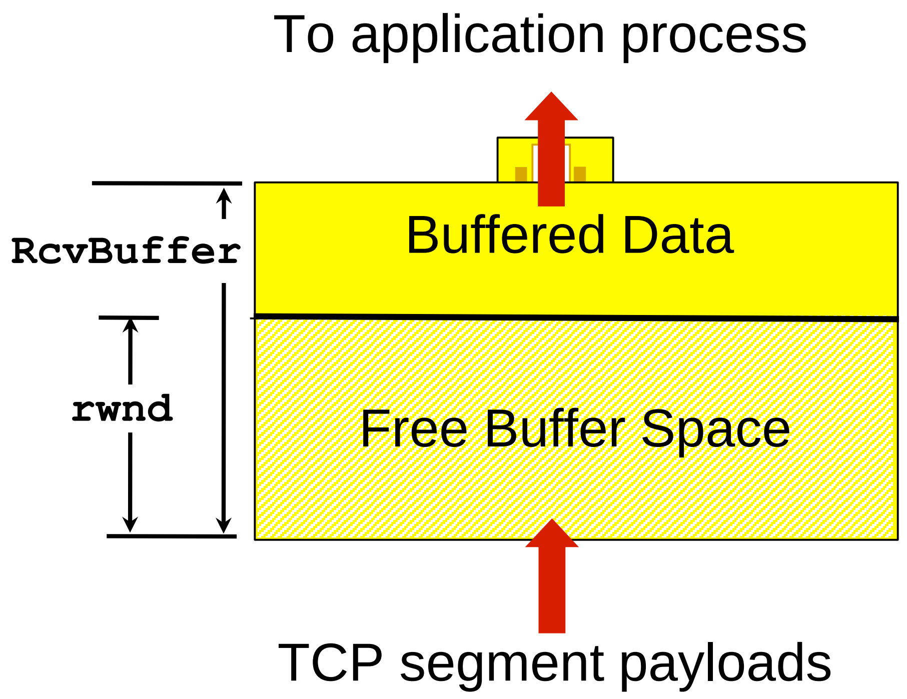
- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- **Principles of Congestion Control**
- **TCP Congestion Control**

TCP Flow Control

- **The receiver can control the sender, so the sender won't overflow the receiver's buffer by transmitting too much, too fast**
  - TCP socket between Transport and Application layer contains buffers to accumulate received data
  - Data buffer may fill faster than receiver's Application Layer app can empty the buffer
- **Receiver needs way to indicate to sender that its data buffers are too full and the sender should send data at a slower rate**

TCP Flow Control (Cont.)

- **Receiver "advertises" how much buffer space it has available by including a value `rwnd`, in the Window Size field of the TCP header**
  - The `rwnd` value indicates how much free space is available in the buffer
  - Receive buffer size can typically be set via socket options (default is usually 4096 bytes)
  - Many operating systems will automatically adjust the size of the receive buffer
- **The sender limits amount of unacked ("in-flight") data to receiver based on the Window Size field**
  - Guarantees receive buffer will not overflow



Receiver-side Buffering

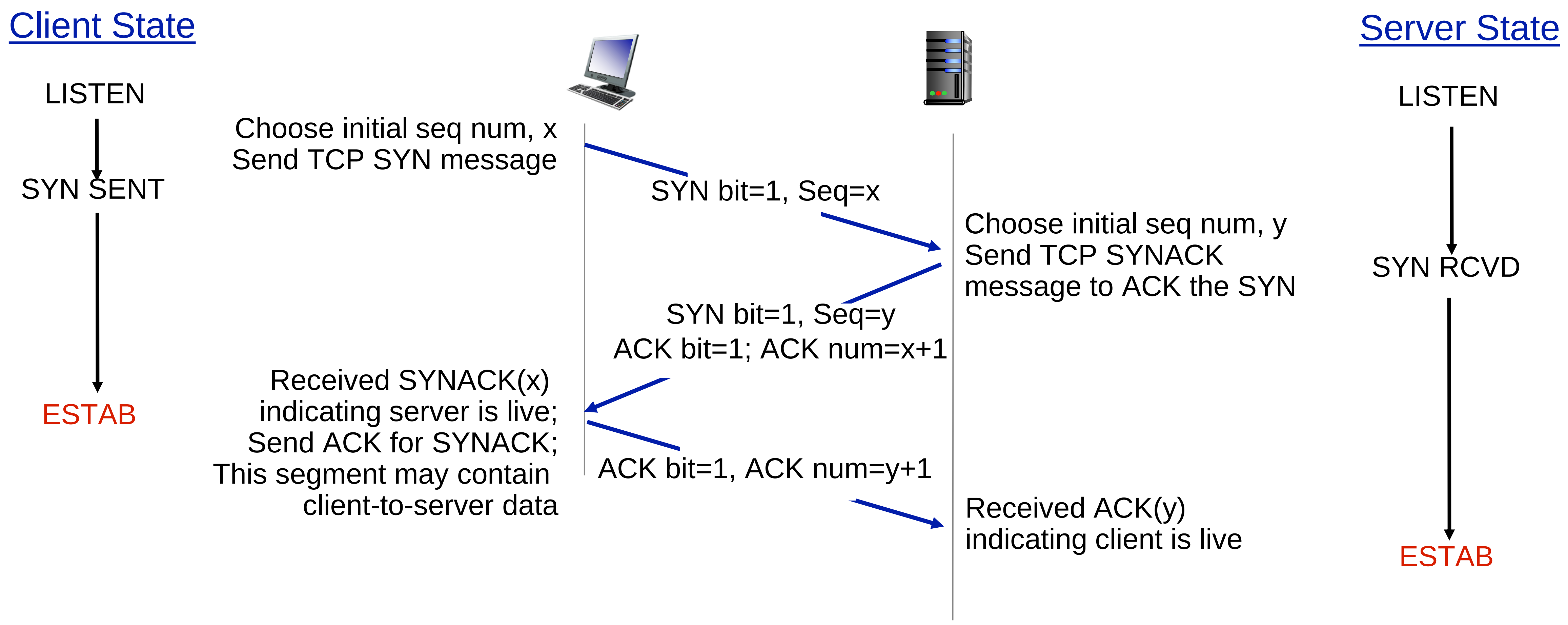
Overview of Transport Layer

- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- **Principles of Congestion Control**
- **TCP Congestion Control**

Connection Management

- **Before exchanging data, the sender and receiver perform a 3-way-handshake:**
  - Agree to establish connection (each knowing the other is willing to establish connection)
  - Agree on connection parameters

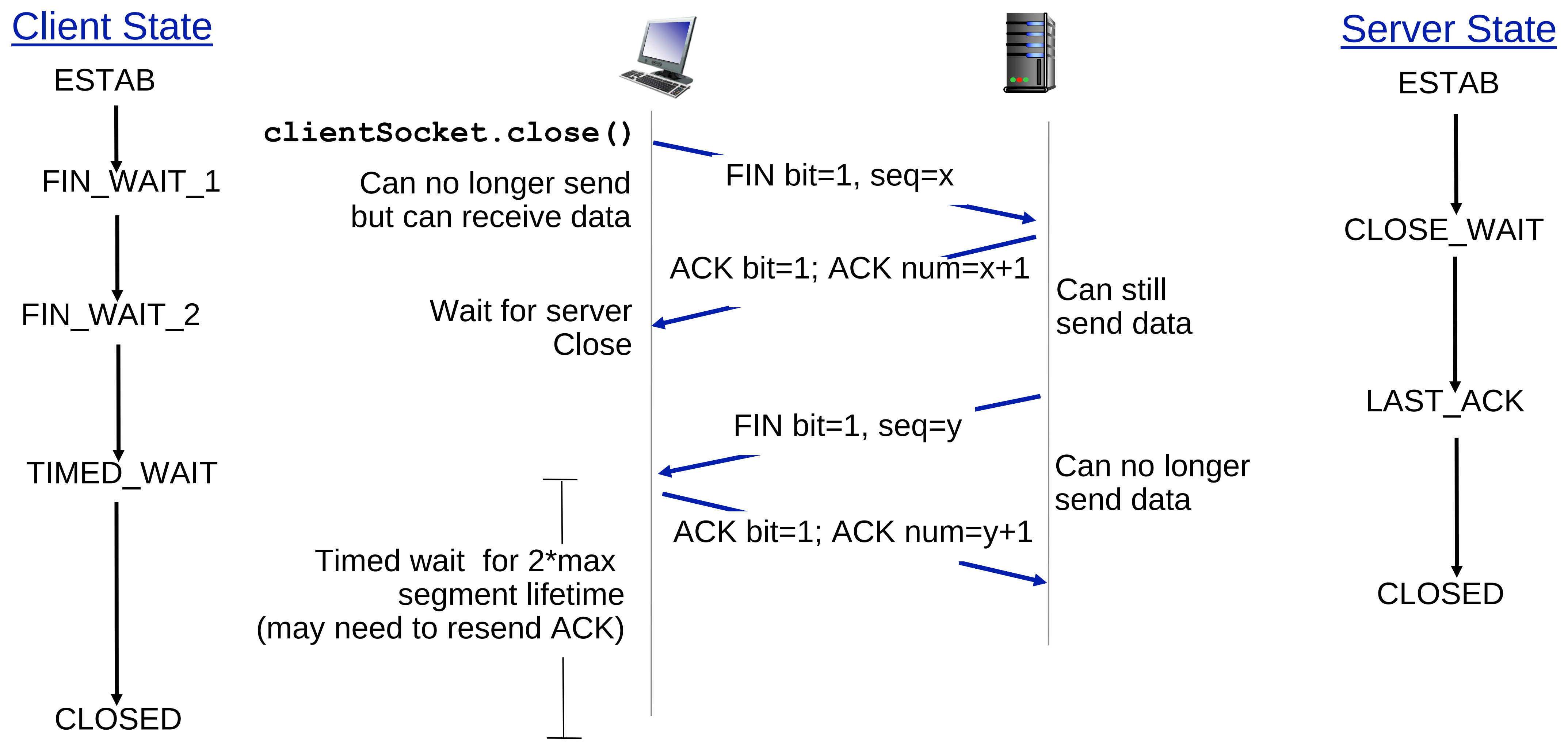
TCP 3-way Handshake



TCP: Closing a Connection

- **Client and server each close their side of the connection**
  - Send TCP segment with FIN bit = 1
- **Respond to received FIN with ACK**
  - On receiving FIN, ACK can be combined with own FIN
- **Simultaneous FIN exchanges can be handled**

TCP: Closing a Connection (Cont.)



Overview of Transport Layer

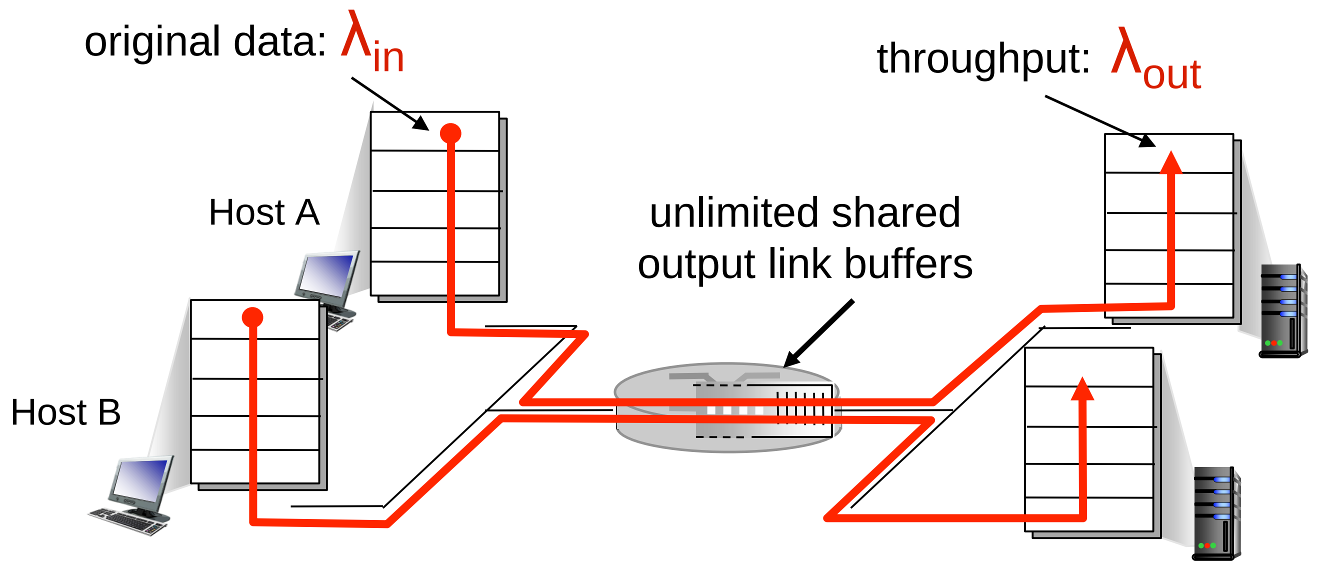
- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- **Principles of Congestion Control**
- **TCP Congestion Control**

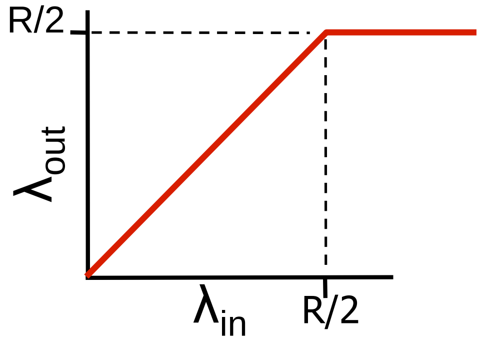
Principles of Congestion Control

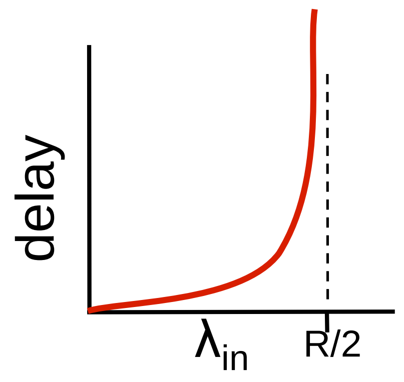
- **What is congestion?**
  - Informally: "too many sources sending too much data too fast for the ***network*** to handle"
  - Different from flow control
    - Flow control used to ensure buffers at receiver do not overflow
    - Flow control does nothing to prevent router buffers from overflowing
- **What problems can congestion cause?**
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)

Causes/Costs of Congestion: Scenario #1

- **Consider the following scenario**
  - Two senders, two receivers
  - One router with infinite buffers
  - Single output link capacity shared by senders with capacity R
  - Assume no retransmission necessary
- **Senders cannot send at a rate higher than R/2 since they are sharing single link**
- **As senders max out the output link, the delay between source and destination increases**


original data: λ_{in}
throughput: λ_{out}
Host A
Host B
unlimited shared output link buffers



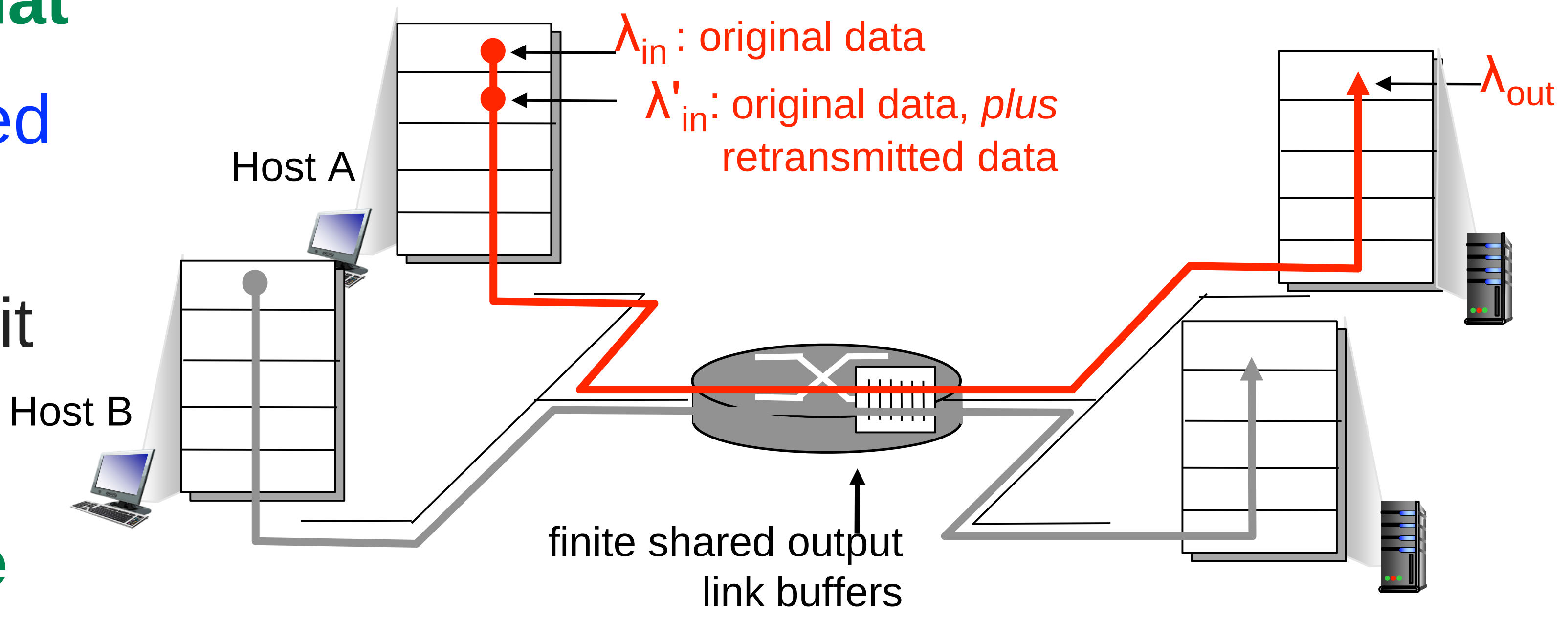


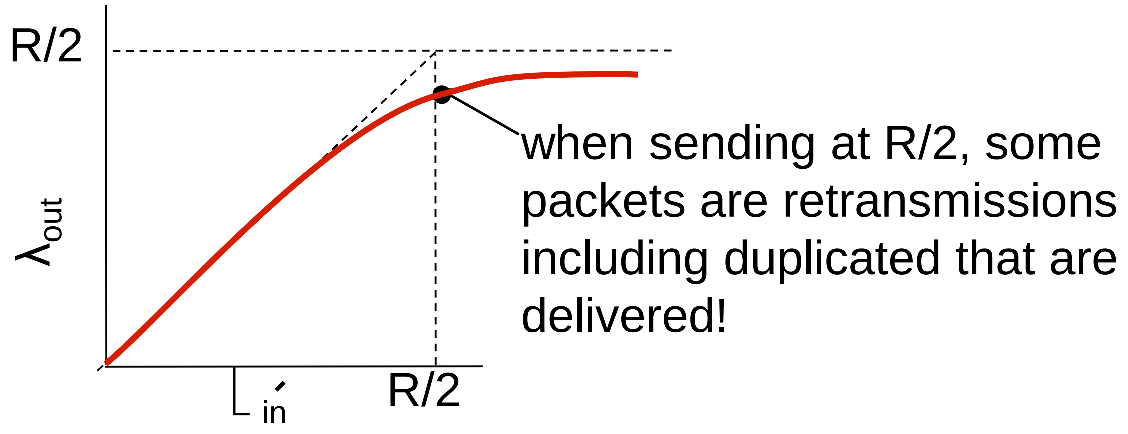
maximum per-connection throughput is R/2

large delays as arrival rate, λ_{in}, approaches capacity

Causes/Costs of Congestion: Scenario #2

- **Modify scenario #1 such that**
  - The router has finite shared buffers
  - The sender may retransmit packets
- **Retransmission reduce the throughput**
  - Packets can be dropped at router if buffers are full
  - Sender may timeout prematurely due to delay in router; send multiple copies of same data


λ_{in} : original data
λ'_{in}: original data, *plus* retransmitted data
λ_{out}
Host A
Host B
finite shared output link buffers



Two Approaches Towards Congestion Control

- **End-to-end congestion control**
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP

- **Network-assisted congestion control**
  - Routers provide feedback to end systems
  - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Explicit rate for sender to send at

Overview of Transport Layer

- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
- **Connection-oriented Transport: TCP**
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  - Connection Management
- **Principles of Congestion Control**
- **TCP Congestion Control**

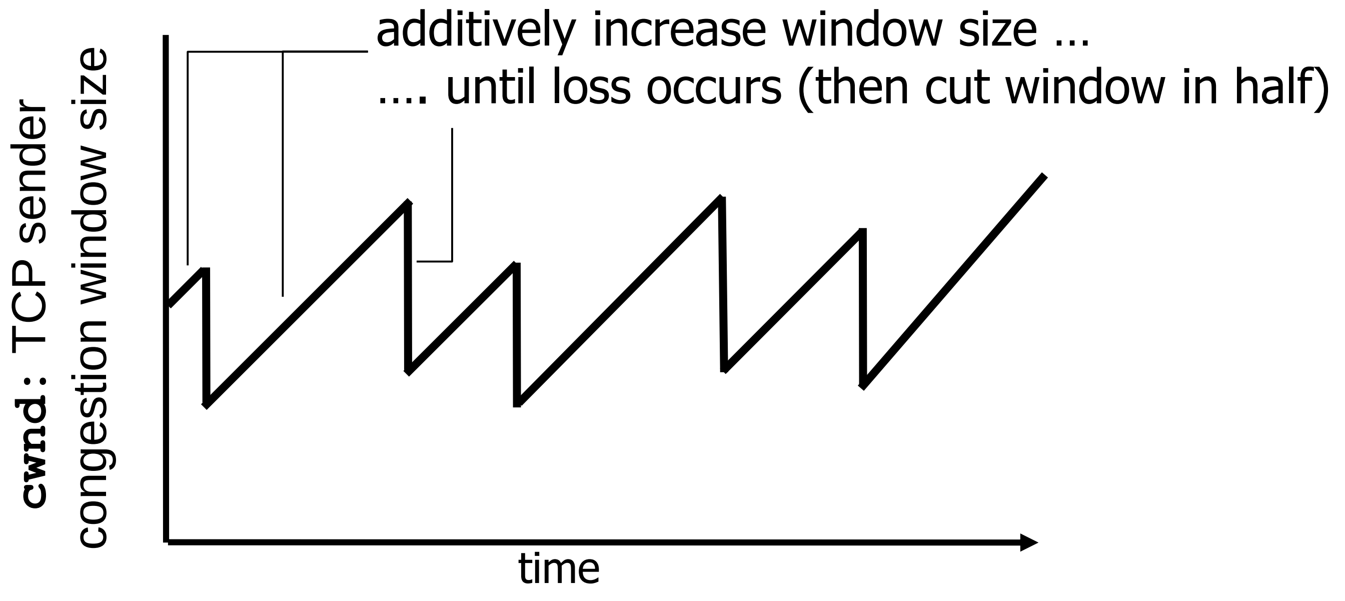
TCP Congestion Control

- **Must have each sender limit the rate at which it sends traffic as a function of the network congestion**
  - Too much congestion? Send less data.
  - Not much congestion? Full speed ahead!

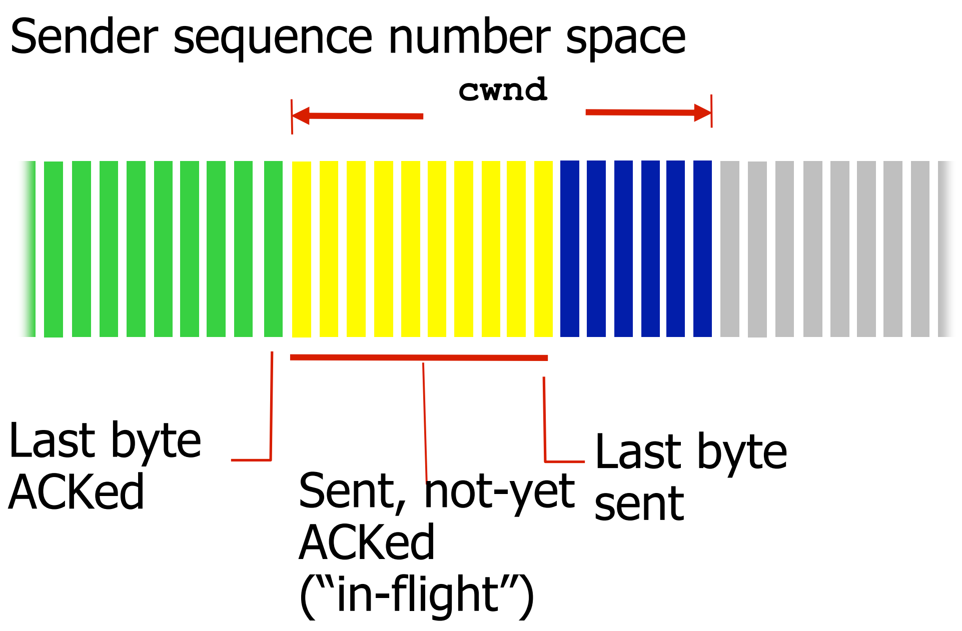
- **How does a TCP sender limit the rate at which it sends data?**
- **How does a TCP sender detect congestion between itself and the destination?**
- **How should the sender change the rate at which it sends data based on the network congestion?**

TCP Congestion Control: AIMD

- **Additive Increase Multiplicative Decrease (AIMD)**
  - **Approach**: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
    - Additive increase: increase congestion window (**`cwnd`**) by 1 MSS (Maximum Segment Size) every RTT until loss detected
      - ACKs arriving at sender signal the sender to increase its window
    - Multiplicative decrease: cut congestion window in half after a packet loss occurs



TCP Congestion Control (Cont.)


Sender sequence number space
cwnd
Last byte ACKed
Sent, not-yet ACKed ("in-flight")
Last byte sent

- **TCP sending rate (assuming no limit on receiver's buffer (`rwnd`))**
  - Send **`cwnd`** bytes, wait RTT for ACKS, then send more bytes

$$\boxed{\texttt{rate} \approx \frac{\texttt{cwnd}}{\texttt{RTT}}\ \texttt{bytes/sec}}$$

- **Congestion window (`cwnd`) is a dynamic, function of perceived network congestion**
  - Sender is limited by both **`cwnd`** and **`rwnd`**
- **Amount of unacked data at sender may not exceed the minimum of `cwnd` and `rwnd`**

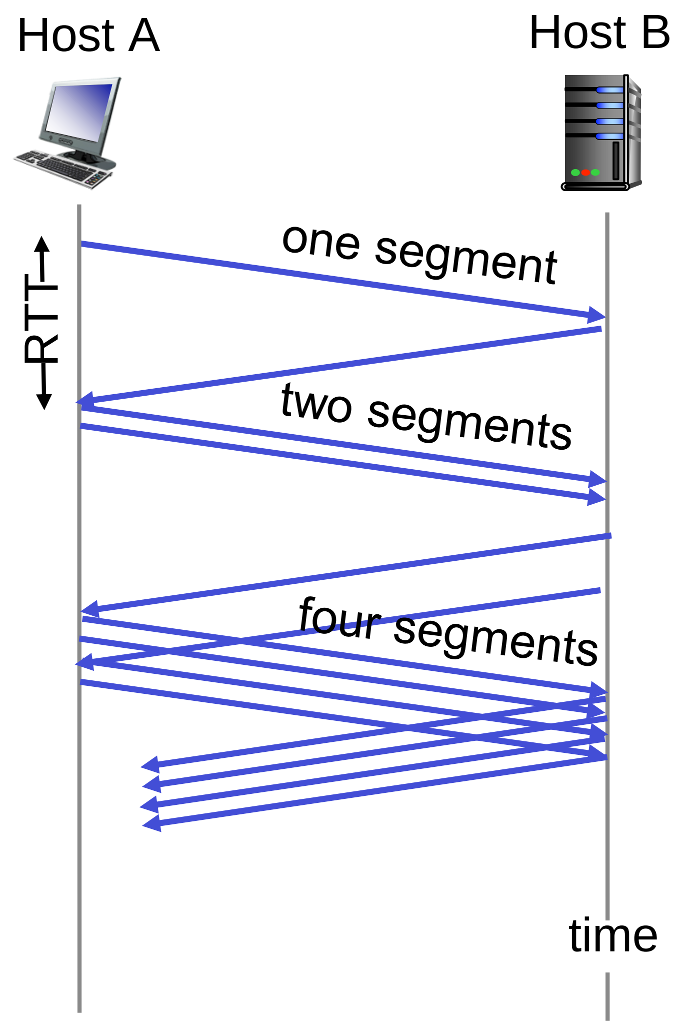
$$\boxed{\texttt{LastByteSent - LastByteAcked} \leq \texttt{min\{cwnd, rwnd\}}}$$

TCP Congestion-Control Algorithm

- **TCPs congestion control algorithm contain three main components**
  - Slow-start
  - Congestion Avoidance
  - Fast Recovery

TCP Slow-Start

- **When connection begins, increase rate exponentially until first loss event:**
  - Initially, **`cwnd`** = 1 MSS
  - Double **`cwnd`** every RTT
    - Done by incrementing **`cwnd`** for every ACK received
  - Initial rate is slow but ramps up exponentially fast
- **When first loss occurs, store (`.5*cwnd`) as `SSThresh` and restart slow-start**
- **When `cwnd` reaches `SSThresh`, switch from slow-start mode to congestion avoidance mode**

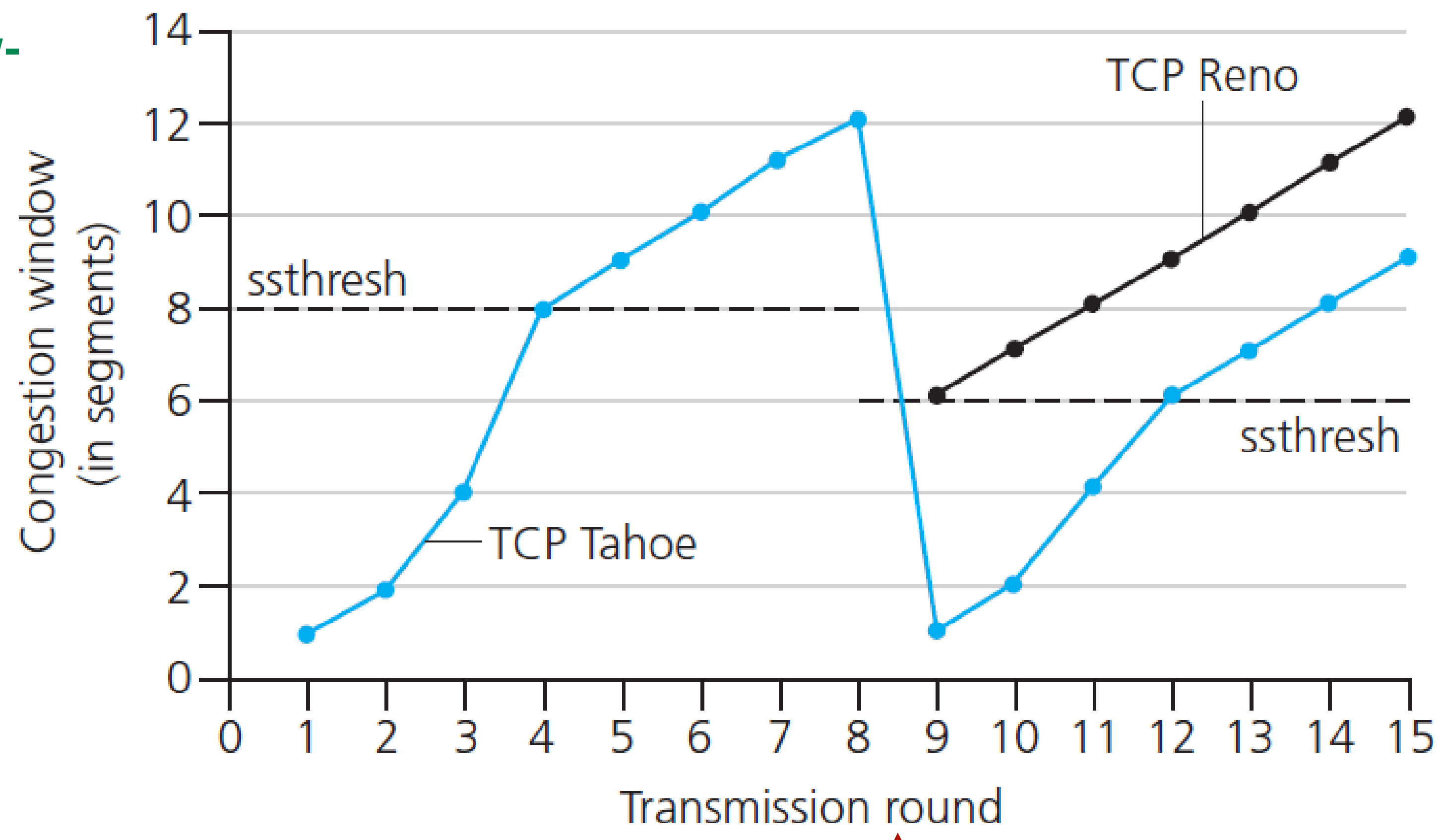


Loss During Slow-Start

- **If loss is indicated by a timeout**
  - Store (**`.5*cwnd`**) as **`SSThresh`** and restart slow-start
  - Set **`cwnd`** set to 1 MSS
  - **`cwnd`** then grows exponentially (as in slow start) to threshold value **`SSThresh`**, then grows linearly in congestion avoidance phase
- **If loss is indicated by 3 duplicate ACKs (only in TCP Reno)**
  - Duplicate ACKs indicate network capable of delivering some segments, so don't drop **`cwnd`** all the way down to 1 MSS
  - Store **`(.5*cwnd)`** as **`SSThresh`**
  - **`cwnd`** is also set to **`(.5*cwnd)`**, but will be increment for each duplicate ACK
- **TCP Tahoe always sets `cwnd` to 1 (for either timeout or 3 duplicate acks)**

Switch from Slow Start to Congestion Avoidance

- **Exponential growth phase shows TCP slow-start**
- **Linear phase after crossing over `SSThresh` shows the congestion avoidance phase**
- **TCP Tahoe**
  - Set **`cwnd = 1`** for both a timeout and for triple duplicate ACKs
  - Set **`SSThresh = cwnd/2`**
  - Re-enters slow-start phase
- **TCP Reno**
  - Implements Fast Recovery
  - Retransmits missing segment
  - Set **`SSThresh = cwnd/2`**
  - Set **`cwnd = SSThresh + 3`**
  - In congestion avoidance phase



Triple-duplicate ACKs