CSCD 330
Network Programming

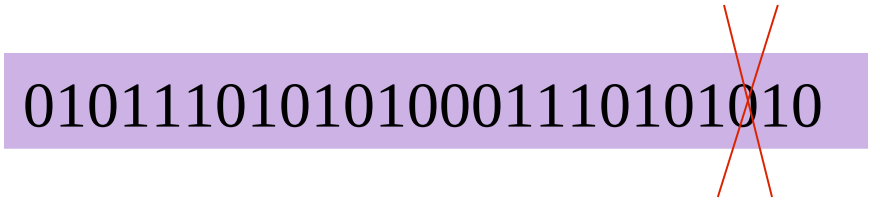
Lecture 10
Transport Layer Continued

Winter 2016

Reading: Chapter 3

Last Time ….

- Started to build a generic reliable protocol
- Covered the case where underlying medium was absolutely reliable
- Added the case where there are bit errors



Packet

Rdt3.0: Now Have Channels with Errors *and* Loss

Added this Assumption

- Underlying channel can also lose packets
- Data or ACKs
- Sender doesn't know if data packet or ACK was lost, or just delayed
  - Checksum, Sequence numbers,
  - ACKs, Retransmissions
  - Will be of help, but not enough

Rdt3.0: Now Have Channels with Errors *and* Loss

Approach

Sender waits “reasonable” amount of time for ACK

- Retransmits if no ACK received in this time
- If packet (or ACK) just delayed ... not lost
  - Retransmission will be duplicate, but use of Sequence numbers to handle this
  - Receiver must specify sequence number of packet being ACKed
- Loss, however, Requires countdown timer!!!

Rdt3.0 Stop and Wait with Timer

- Waiting for each packet to get sent or timeout via Stop and Wait is very slow …

Performance of Rdt3.0

- rdt3.0 works, but performance is bad
- **Example:** 1 Gbps link, 15 msec end-to-end propagation delay, 100,000 byte message, RTT = 30 msec, cross country

$$d_{trans} = \frac{L \text{ (message length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{800000 \text{ bits}}{10^{**}9 \text{ b/sec}} = 8 \text{ millisec}$$

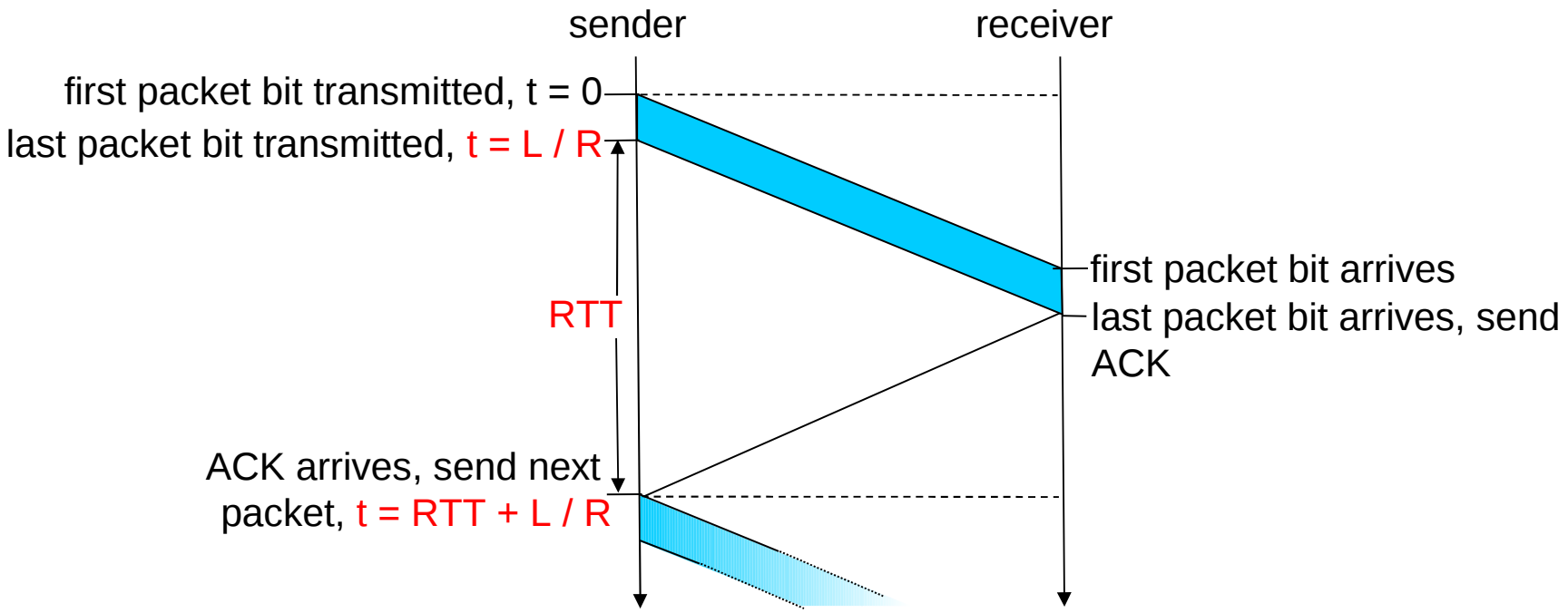
U_{sender}: Utilization is fraction of time sender busy sending

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

100 K byte mesg every 30 msec -> **276 kbs thruput of 1 Gbps link**

Example of how network protocol limits use of physical resources!

rdt3.0: Stop-and-wait Operation



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

Solution to Slowness: Pipelined protocols!!!

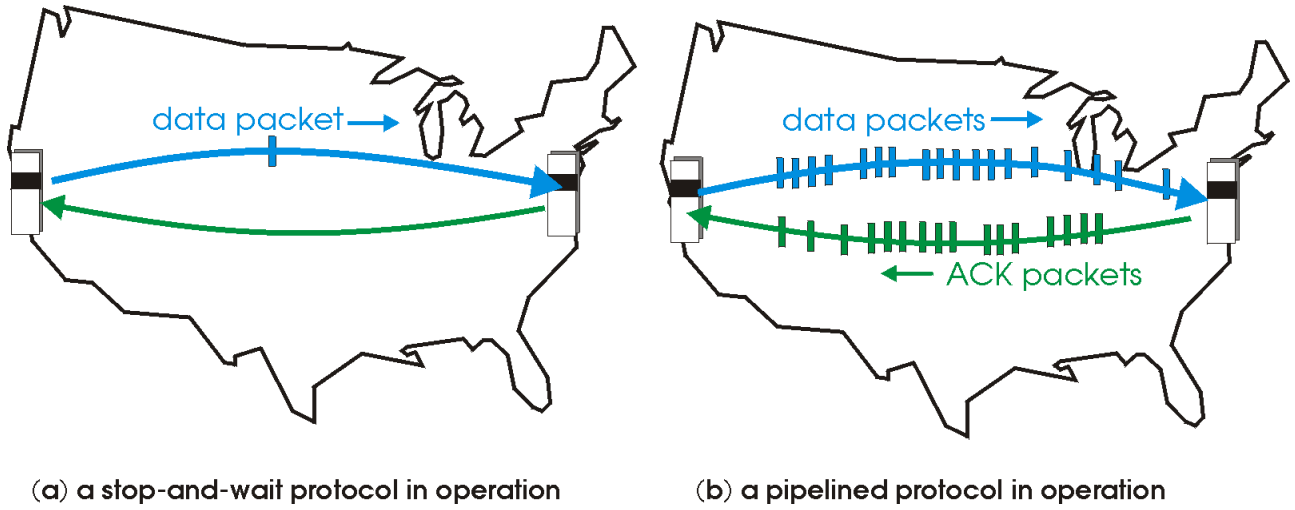
- Pipelining: Sender allows multiple, “in-flight”, yet-to-be-acknowledged packets without waiting for ACK’s

- What now needs to change from our Stop and Wait protocol?
  - Buffering of packets - establish queues
  - Increase sequence numbers

Solution to Slowness: Pipelined protocols!!!

Changes as Result of Pipelining

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

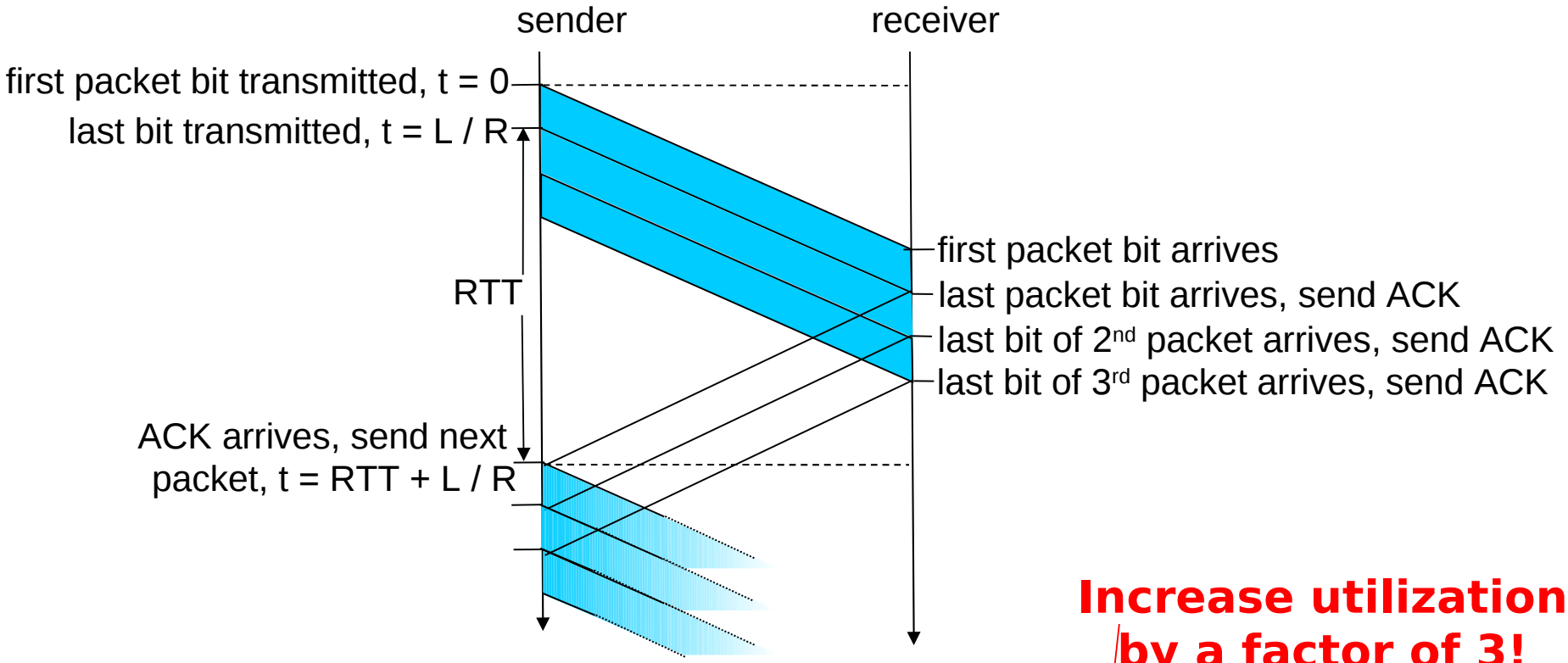


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols:
  Go-Back-N, Selective Repeat
- Elements of both these are used in TCP

Pipelining: increased utilization



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

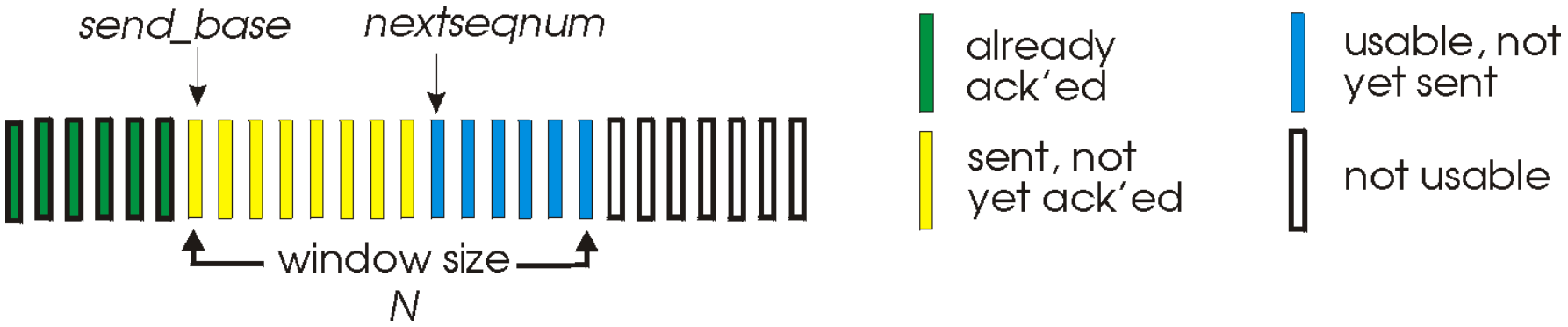
Increase utilization by a factor of 3!

Go-Back-N (sliding window protocol)

Sender

k-bit seq # in pkt header

"window" of up to N, consecutive unack'ed pkts allowed

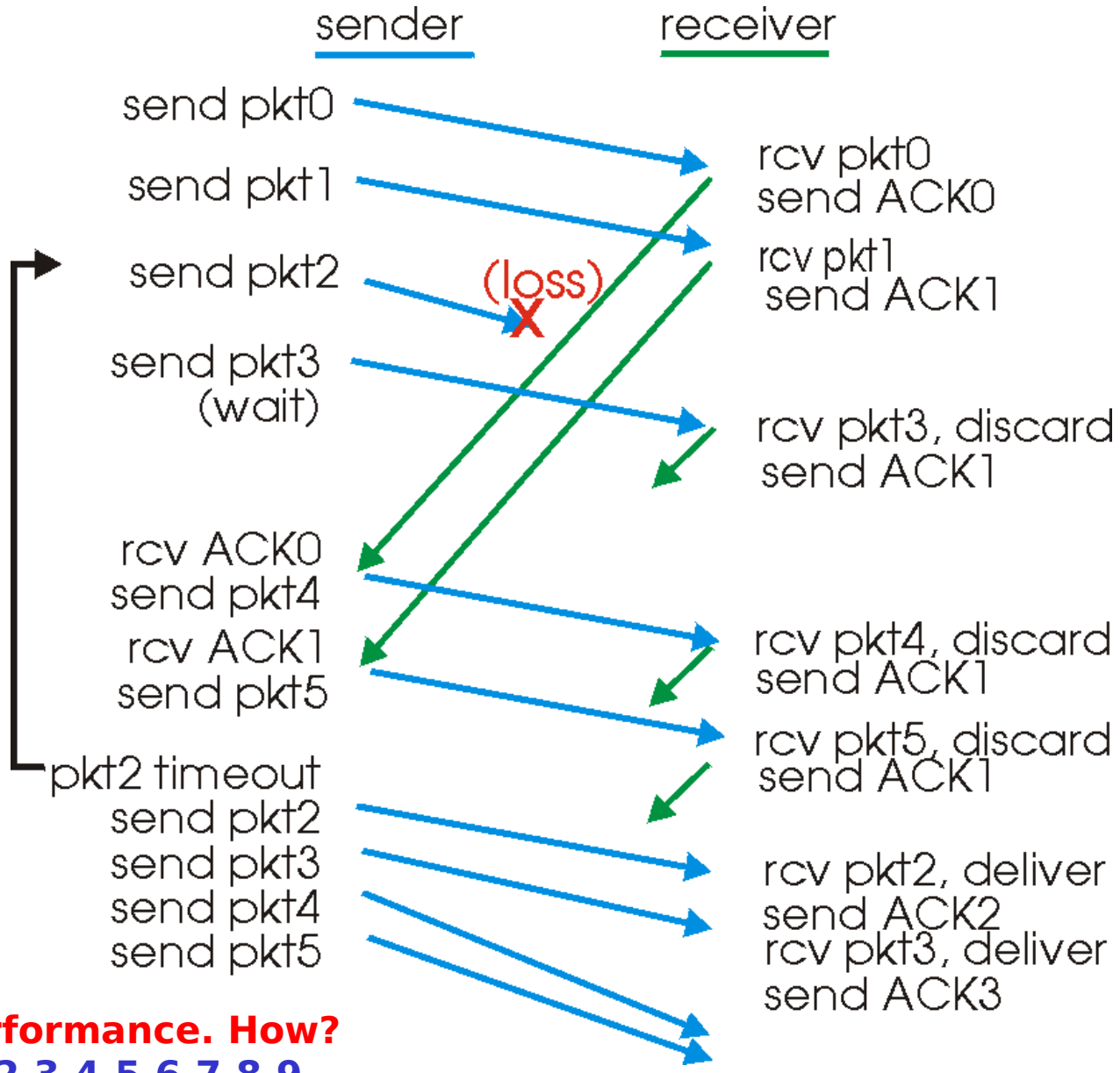


- **ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"**
  - **Sender may receive duplicate ACKs**
- **Timer for each in-flight pkt**
- ***Timeout(n):* retransmit pkt n and all higher seq # pkts in window**

Do not buffer packets after a loss!

GBN in action

Window size=N=4



What determines size of window?

1. **RTT**
2. **Buffer at the receiver(flow control)**
3. **Network congestion**

Q: GBN has poor performance. How?

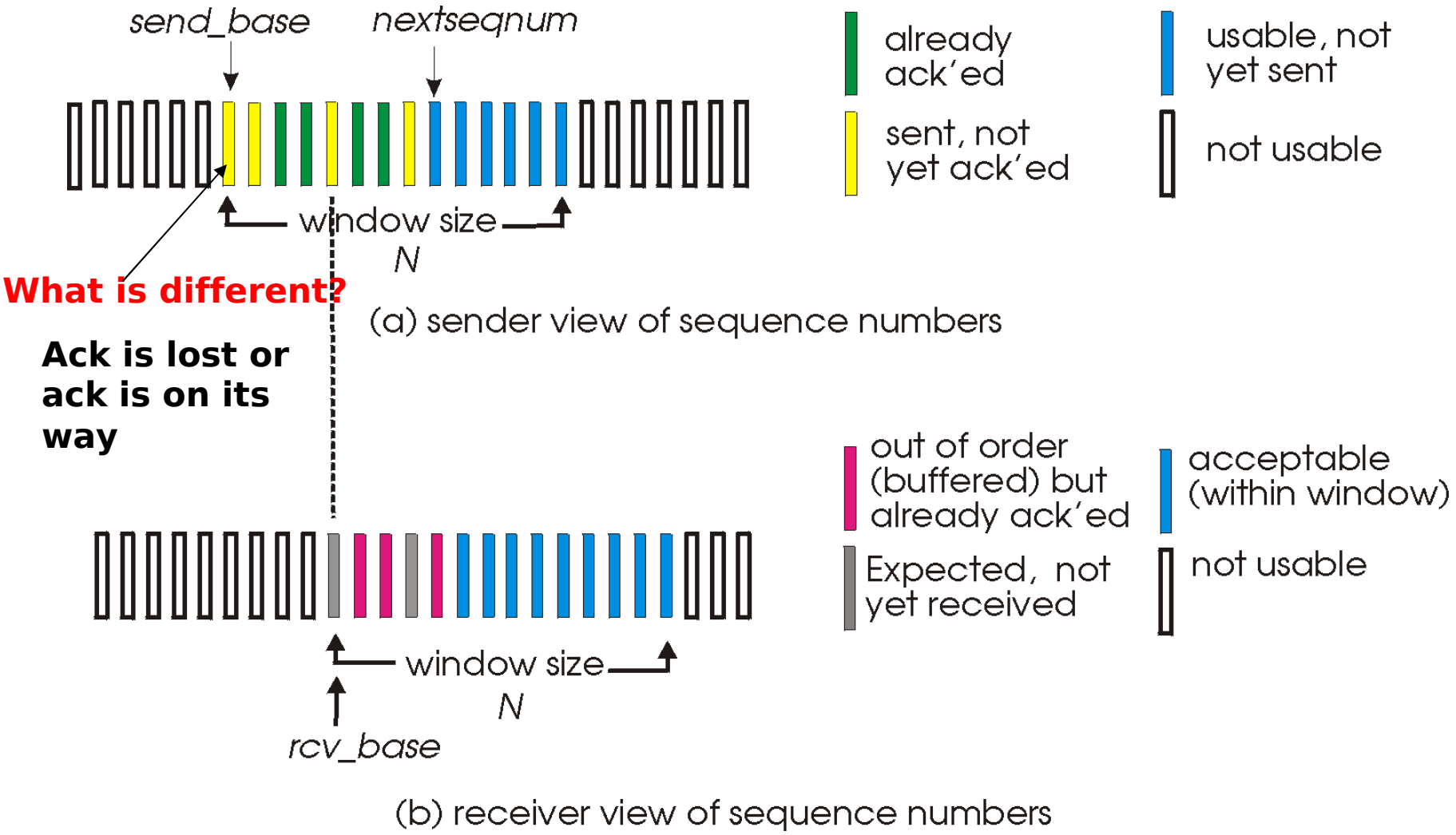
Sender sends pkt 1,2,3,4,5,6,7,8,9..

pkt 1 got lost, receiver got pkt 2,3,4,5,… but will discard them!

Selective Repeat (improvement of the GBN Protocol)

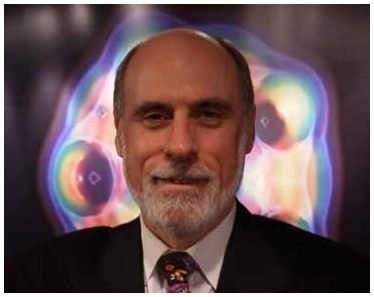
- Receiver *individually* acknowledges all correctly received pkts
  - Buffers pkts, as needed, for eventual in-order delivery to upper layer
  - E.g., sender: pkt 1,2,3,4,….,10; receiver got 2,4,6,8,10. Sender resends 1,3,5,7,9.
- Sender only resends pkts for which ACK not received
  - Sender has timer for EACH unACKed pkt
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Why bother study reliable data transfer?

- We know it is provided by TCP, so why bother to study?
- Sometimes, we may need to implement “some form” of reliable transfer without the heavy duty TCP
- A good example is multimedia streaming
  - Even though application is loss tolerant, if too many packets got lost, it affects visual quality
  - So we may want to implement some form of reliable transfer



Who is this?
Vint Cerf

+

Who is this?
Bob Kahn

=

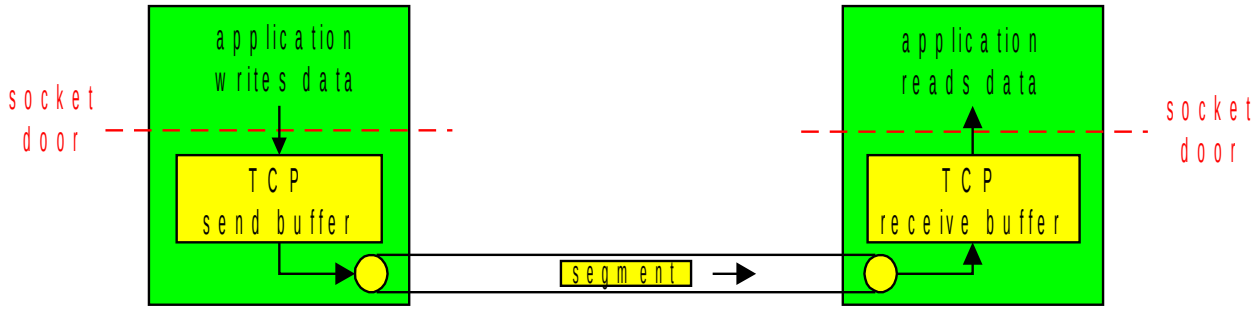
TCP

TCP Overview

RFCs: 793, 1122, 1323, 2018, 2581 and others

- Point-to-point
  - One sender, one receiver
- Reliable, in-order **byte stream**
  - Streams are reassembled at receiver
- Pipelined
  - TCP congestion and flow control set window size

Send & receive buffers

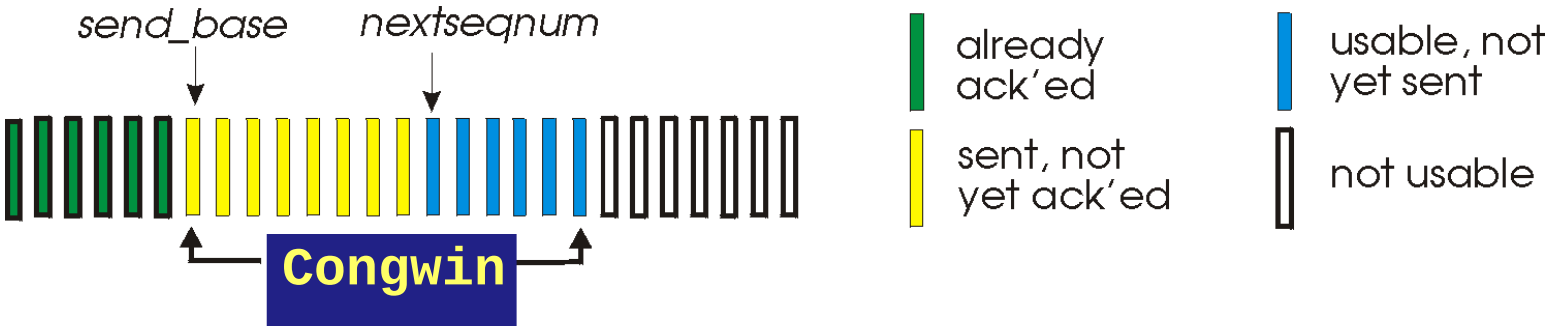


TCP Overview

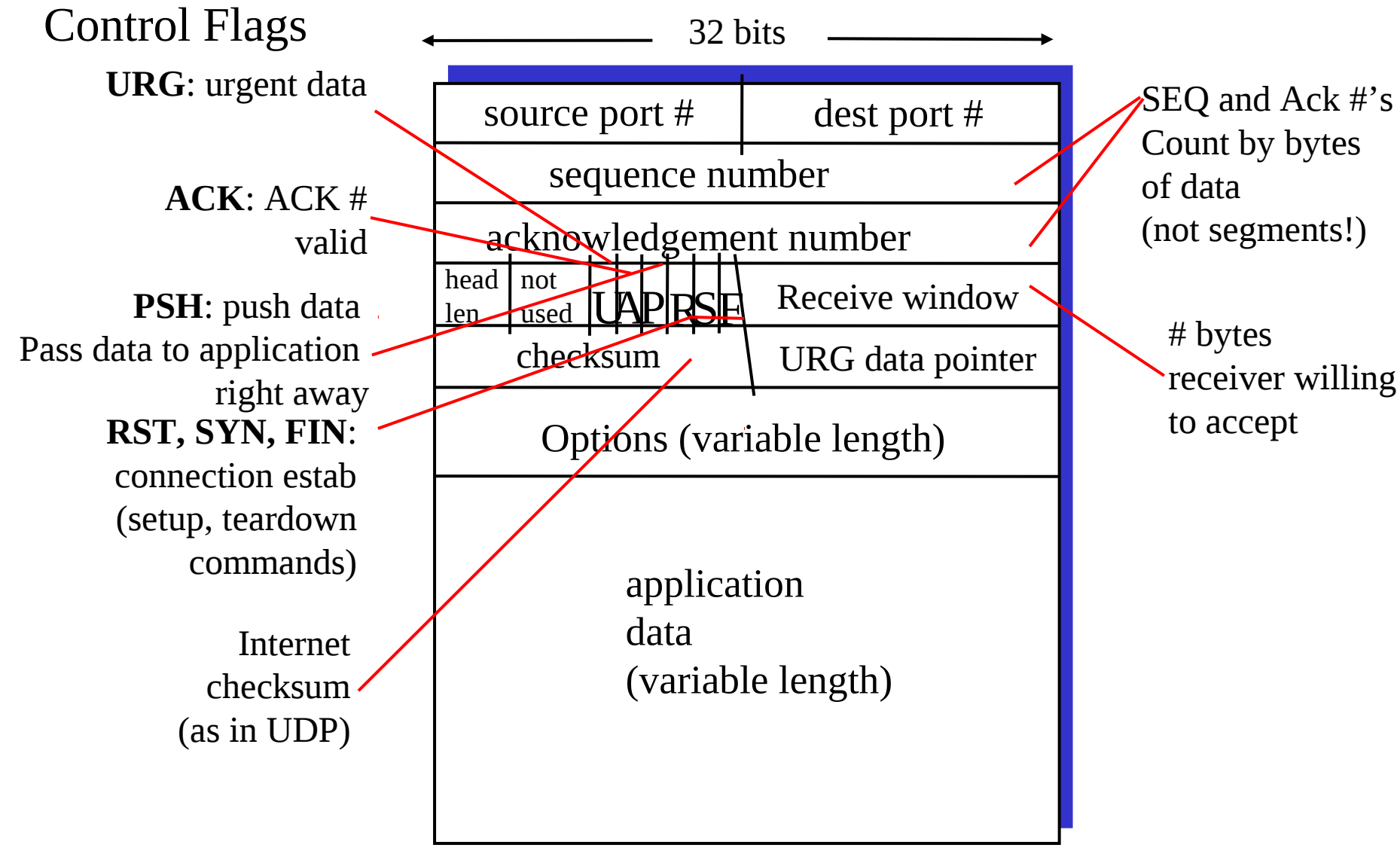
- Full Duplex Data
  - Bi-directional data flow in same connection
- Connection-oriented
  - Handshaking, exchange of control messages
  - Sender, receiver establish state before data exchange
  - One-to-one, no multicast allowed
    - Multicast recall is ...
      - One sender, many receivers
- Flow controlled
  - Sender will not overwhelm receiver

TCP Overview

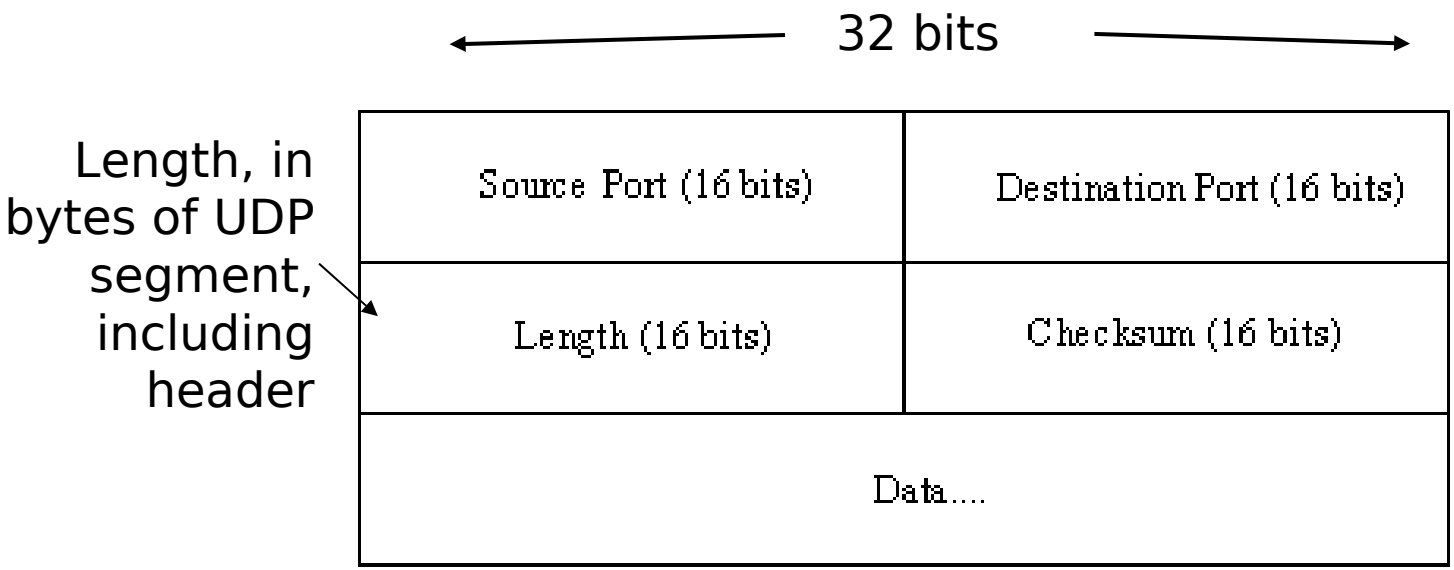
- TCP is a sliding window protocol
  - Sender has Window of bytes in flight
- Operates with cumulative ACK's
- It includes control for sending rate
  - Flow control: Receiver sets sending rate
  - Congestion control: Network-aware sending rate



TCP Header


Control Flags
32 bits
URG: urgent data
ACK: ACK # valid
PSH: push data
Pass data to application right away
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
source port #
dest port #
sequence number
acknowledgement number
head len
not used
UAPRSF
Receive window
checksum
URG data pointer
Options (variable length)
application data (variable length)
SEQ and Ack #'s Count by bytes of data (not segments!)
bytes receiver willing to accept

Recall UDP Header



TCP header fields Port Numbers

- Port Numbers - Source and Destination
  - **Port number identifies endpoint of a connection**
  - A pair **<IP address, port number>** identifies one endpoint of a connection
  - **Recall, we need two pairs of Identifiers**
  - **<client IP address, client port number>**
  - **<server IP address, server port number>**
  - Identify a TCP connection!!

TCP header fields Sequence Number

- Sequence Number - SeqNo
  - Sequence number is 32 bits long
  - So the range of SeqNo is

    $0 <= SeqNo <= 2^{32} - 1$

  - Each sequence number identifies a byte in the byte stream
  - Initial Sequence Number (ISN) of a connection is set during connection establishment
    - Used to be consecutive numbers
    - Now, it a long random number

TCP header fields ACK Number

- Acknowledgement Number (AckNo)
  - A hosts uses AckNo field to send acknowledgements of bytes received
  - If a host sends an AckNo in segment it sets

    "**ACK flag**"
  - AckNo contains next SeqNo that hosts wants to receive

TCP Header Fields Flag Bits

- Flag bits
  - URG: Urgent
    - If the bit is set, bytes contain an urgent message
  - ACK: Acknowledgment Flag
    - When set, segment contains an acknowledgment for a segment that was successfully received
  - PSH: PUSH Flag
    - From sender to receiver, that receiver should pass all data that it has to application
    - Set by sender when sender's buffer is empty

TCP Header Fields Flag Bits

- Flag bits
  - RST: Reset the connection
    - Flag causes receiver to reset connection
    - Receiver of a RST terminates connection and indicates higher layer application about reset
  - SYN: Synchronize sequence numbers
    - Sent in **first** packet when initiating a connection
  - FIN: Sender is finished with sending
    - Used for closing a connection
    - Both sides of a connection must send a **FIN**

TCP Header Fields Flag Bits

- Window Size
  - Each side of connection advertises window size
  - Window size is maximum number of bytes that receiver can accept.
  - Maximum window size is 2^{16}-1= 65535 bytes
- TCP Checksum
  - TCP checksum covers both TCP header **and** TCP data
- Urgent Pointer
  - Only valid if **URG** flag is set

TCP Sequence and ACK Numbers

- Max Segment Size – MSS
  - Maximum amount of **data** that can be grabbed and placed in a TCP segment, doesn't include headers
  - Depends on

    MTU – Maximum Transmission Unit
  - Largest Link layer frame that can be sent by the sending host
  - Common values are
    - 1460, 536 and 512 bytes

TCP Byte Stream Idea

- Review concept
  - Application pumps bytes to TCP
  - TCP responsible for packaging bytes and sending them
  - Divides byte stream into segments
  - Means
    - Each byte has an identifier
    - Keeps track of
      - What was sent,
      - What was Acked by receiver
      - What still needs to be sent
      - Space available in buffer on receiver side

TCP Sequence and ACK Numbers

- **TCP divides data up into ordered stream**
  - Sequence number is byte-stream number of first byte in segment
  - Example:
    - File of 500,000 bytes and MSS = 1000

Assume first byte = 0,

- Segment 1 Seq # = 0,
- Segment 2 Seq # = 1000
- Segment 3 Seq # = 2000, etc

Relationship Between Sequence and ACK Numbers

- ACK field in TCP header for receiver to indicate acknowledgment of data received
  - **In bytes! Not segments**
    - Acknowledges number of bytes of data received
  - What happens when TCP segments arrive out of order? (Draw this)
    - Example: Receiver got 1-535 and 900-1000, but is waiting for 536-899
      - Receiver will send ACK 536 to sender
      - Keeps a cumulative ACK count,
      - Buffers out of order bytes, waits for missing bytes

TCP Sequence Numbers/Acks

Sequence Numbers

- Byte stream “number” of first byte in segment’s data

ACKs

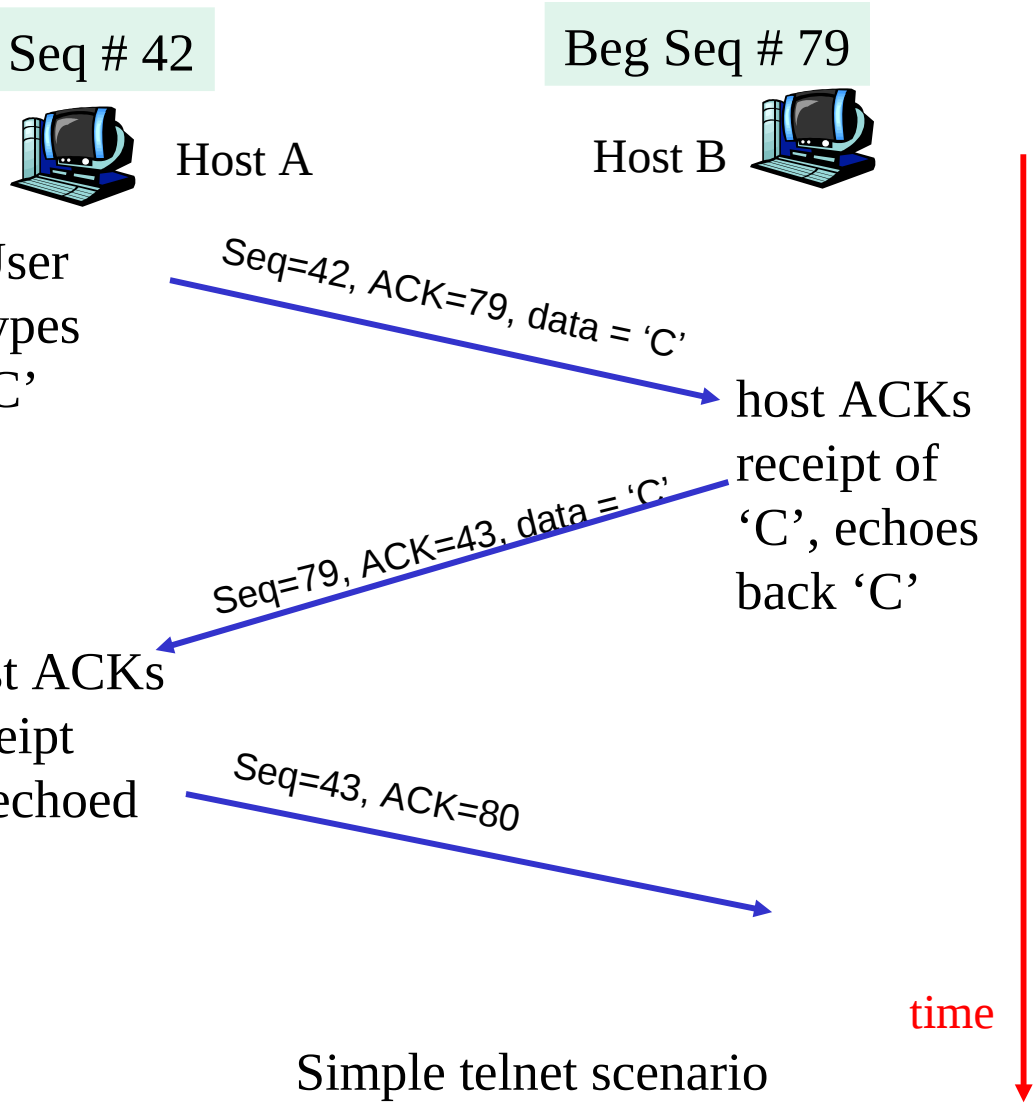
- Seq number of next byte expected from other side
- “Cumulative” ACK

Question- How receiver handles out-of-order segments

- Answer - TCP spec doesn’t say, Up to implementer

TCP Sequence and ACK Numbers

Telnet Example



Simple telnet scenario

TCP TimeOuts

How Long for TCP Time-out

- Remember, TCP has to ensure reliability
- So bytes need to be resent if there is no
  "timely" Acknowledgment
- How long should the sender wait ?
- It should be adaptive -- fluctuation in network load
- If too short, false time-outs
- If too long, then poor rate of sending
- Depends on round trip time estimation

Setting the Timeout Interval

- TCP must re-transmit segments that get lost
- Uses RTT – Round Trip Time
  - Time from when segment sent until it is ACK'd … does this for every packet sent
  - TCP updates this value continuously, calls it **SampleRTT**

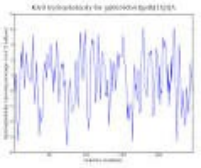
TCP Round Trip Time and Timeout

- For SampleRTT
  - Why might it Fluctuate?

TCP Round Trip Time and Timeout

- SampleRTT
  - Why does it Fluctuate?
    - Congestion in routers
    - Network outages
    - Load on hosts
    - Mis-configured devices

TCP Round Trip Time and Timeout



- What do you do to smooth out fluctuations?
  - Take an average of SampleRTT values
  - Smooths low and high values to something more reasonable for long term

TCP Round Trip Time and Timeout

When get new SampleRTT

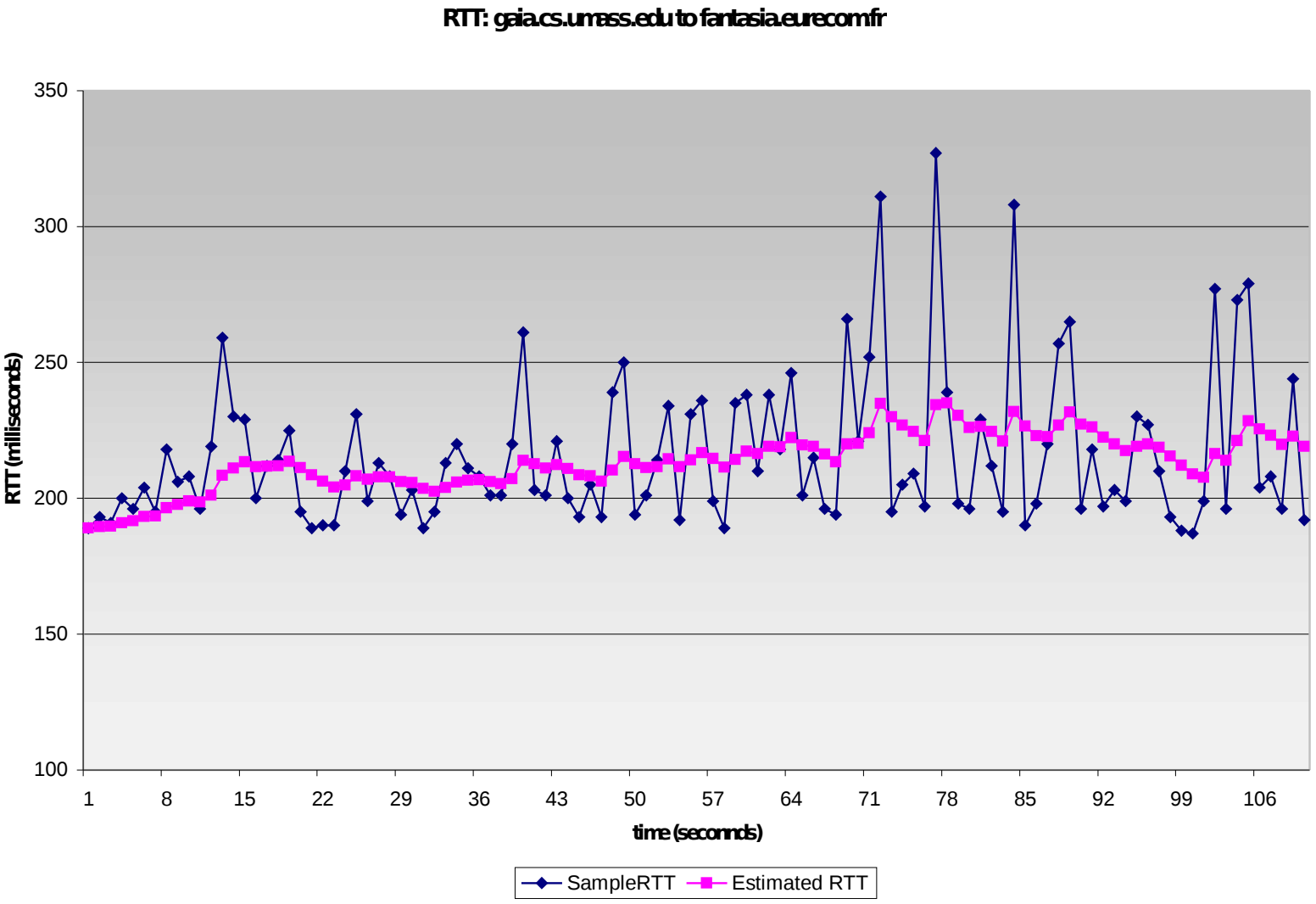
- TCP Calculates a running EstimatedRTT with new sample

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

- **Exponential weighted moving average**
- Influence of past sample decreases exponentially fast
- Typical value: **α =** 0.125

EstimatedRTT = .875*EstimatedRTT + .125*SampleRTT

Example RTT Estimation



TCP Round Trip Time and Timeout

Setting the timeout

- **EstimtedRTT** plus "safety margin"
  - Large variation in **EstimatedRTT ->** larger safety margin
  - Because larger variation means more unpredictable
- First, get estimate of how much SampleRTT deviates from EstimatedRTT,

DevRTT = (1-Δ) * DevRTT + Δ * |SampleRTT-EstimatedRTT|

(typically, Δ = 0.25)

Then set timeout interval

TimeoutInterval = EstimatedRTT + 4*DevRTT

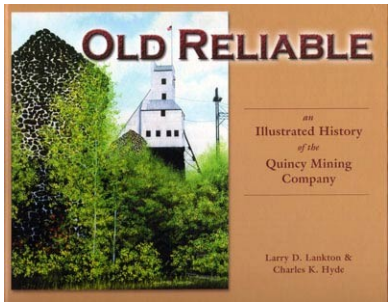
TCP Reliable Data Transfer

- TCP creates reliable service on top of IP's unreliable service

- What does that mean?
  - Pipelined segments
  - Cumulative Acks of received segments
  - TCP uses single retransmission timer

TCP Reliable Data Transfer (rdt)

- Retransmissions are triggered by
  - Timeout events
  - Duplicate Acks



First, just look at a simplified version ....

- Initially consider simplified TCP sender
  - Ignore duplicate Acks
  - Ignore flow control
  - Ignore congestion control

TCP Sender Events

Data received from Application

- Create segment with seq number
- Seq number is byte-stream number of first data byte in segment
- Start timer if not already running
  - Timer is for oldest un-Acked segment

- Expiration interval: TimeOutInterval

Timeout Modifications

- TCP uses other mechanisms than simple timeout to manage retransmissions
  - Each time TCP retransmits segment, doubles the time-out interval instead of deriving it, we assume congestion
  - Why?

  - Say timeout is .75 secs, sends segment
    - First time, timeout doubled to 1.5 secs
    - Second time, timeout doubled to 3.0 secs

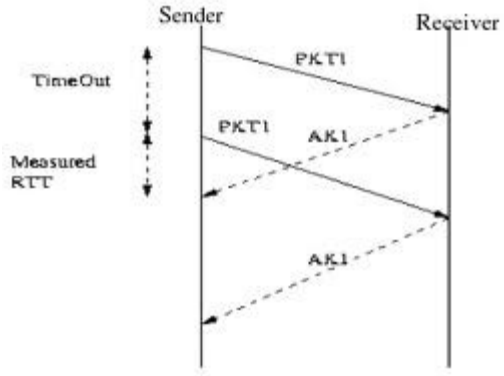
  - Intervals grow exponentially, provides limited form of congestion control

Fast Retransmit Another TCP Modification

- One problem with time-out period, can be long ...
  - Long delay before resending lost packet!

- How could you detect lost packets?

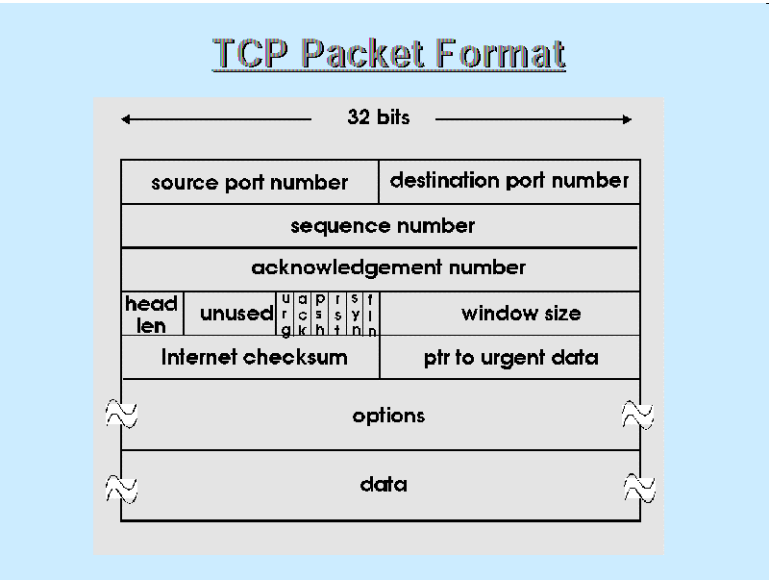
Fast Retransmit



- Detect lost segments before timeout happens
  - Duplicate ACKs

- Sender often sends many segments back-to-back
- If segment is lost, there will likely be many duplicate ACKs
- If sender receives **3 ACKs** for the same data, it supposes that segment after ACKd data was lost

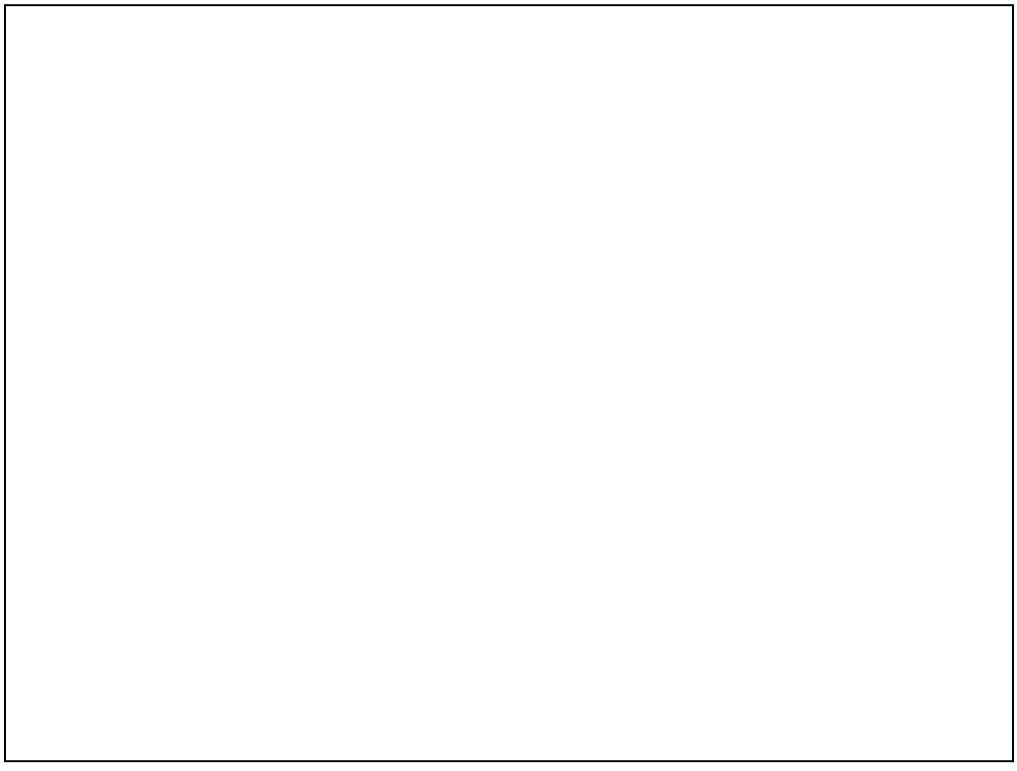
- Fast Retransmit resend segment before timer expires

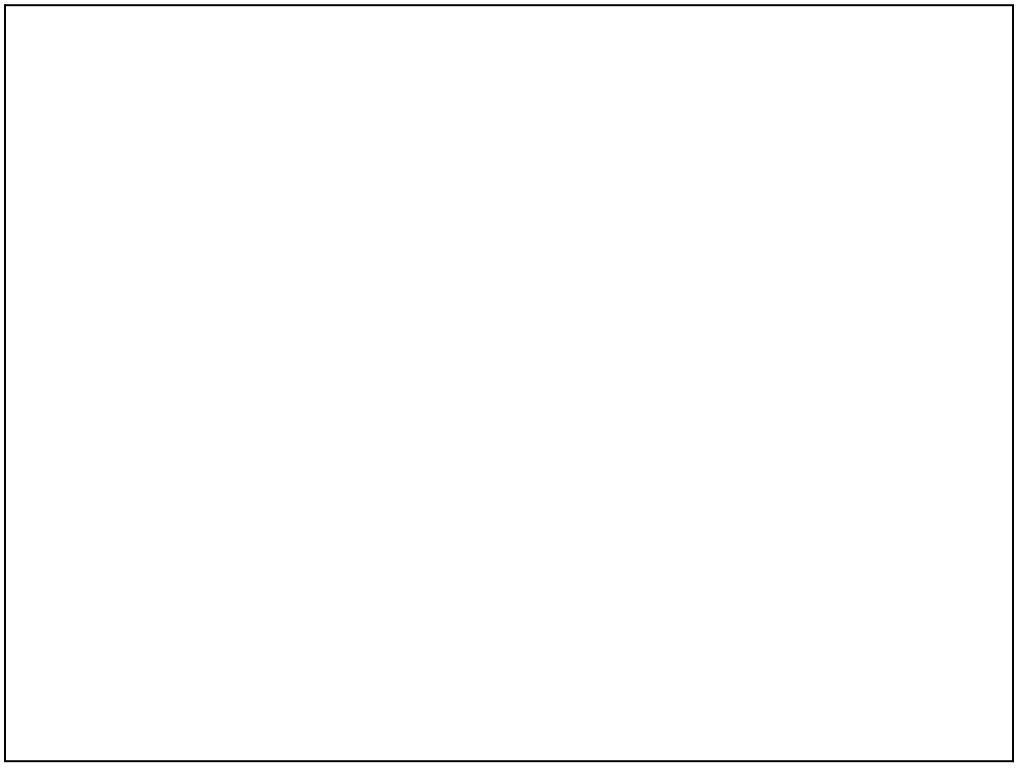


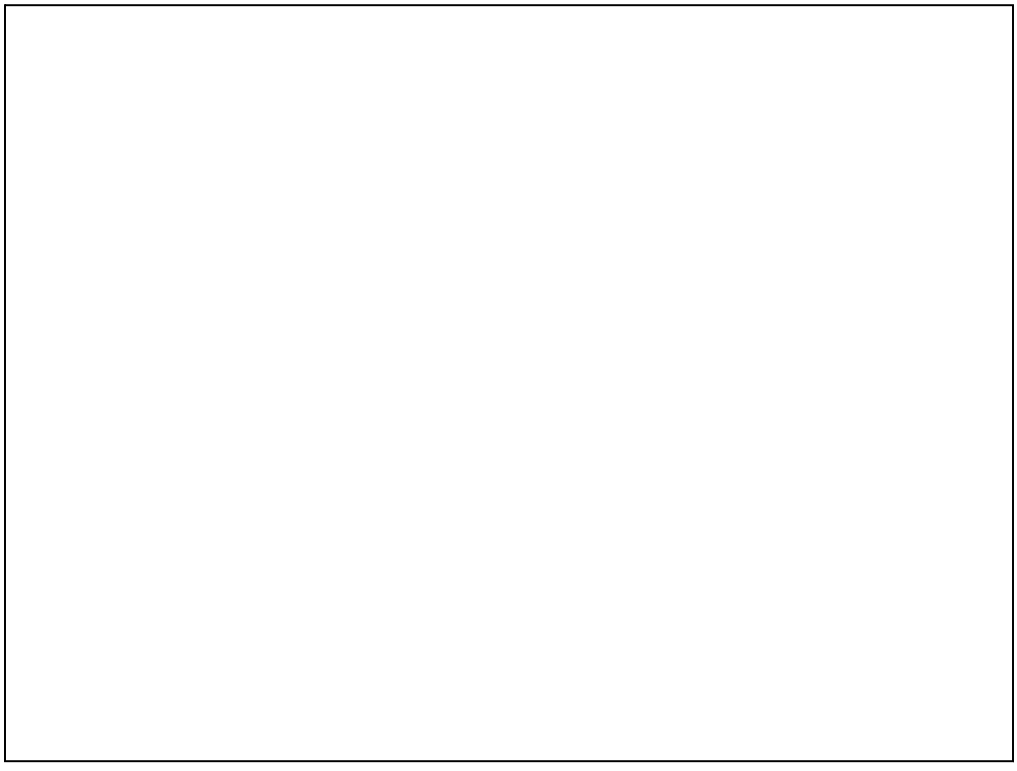
Transport Protocol

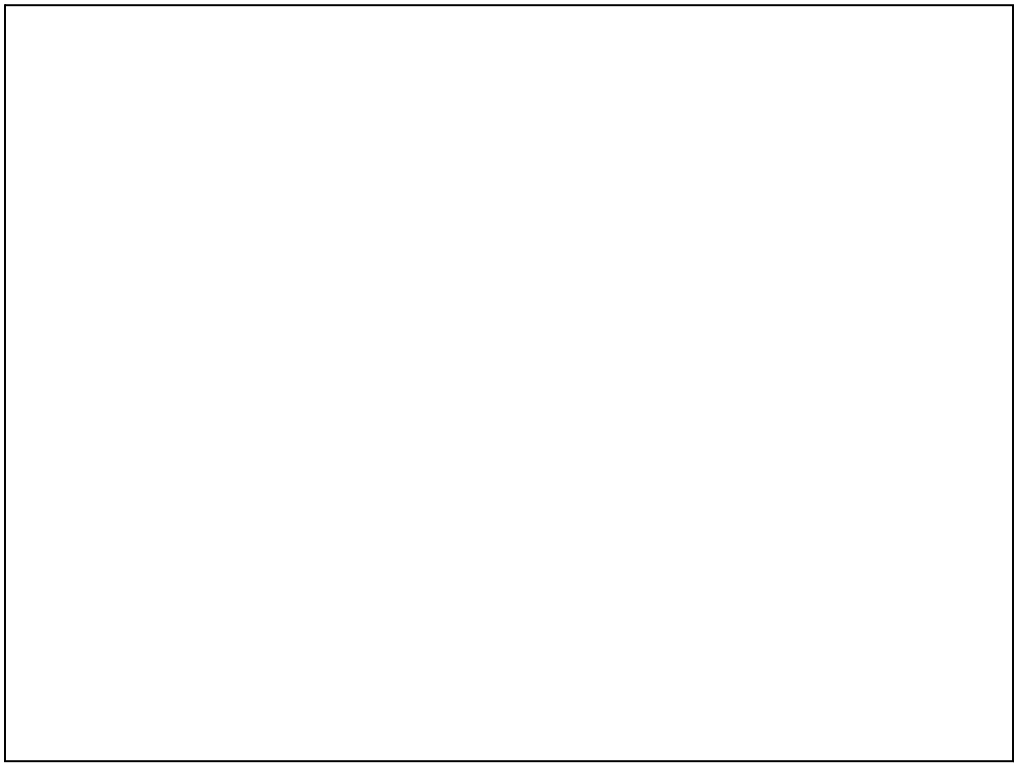
Take-home Midterm will be up on Friday

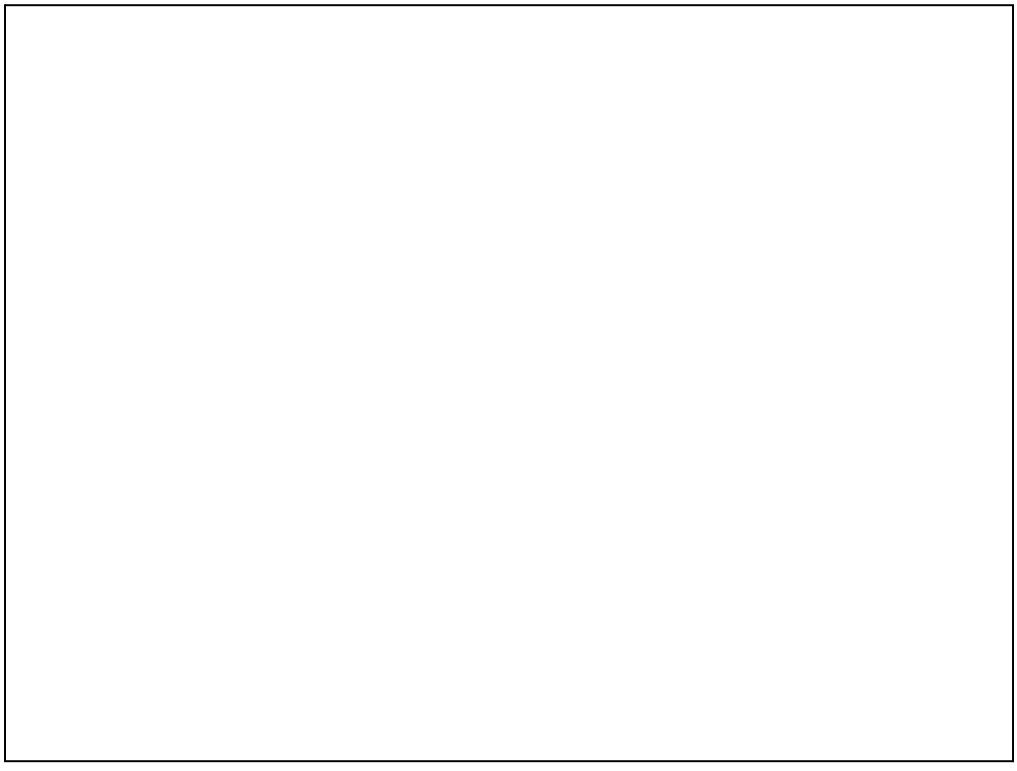
Discuss it on Friday

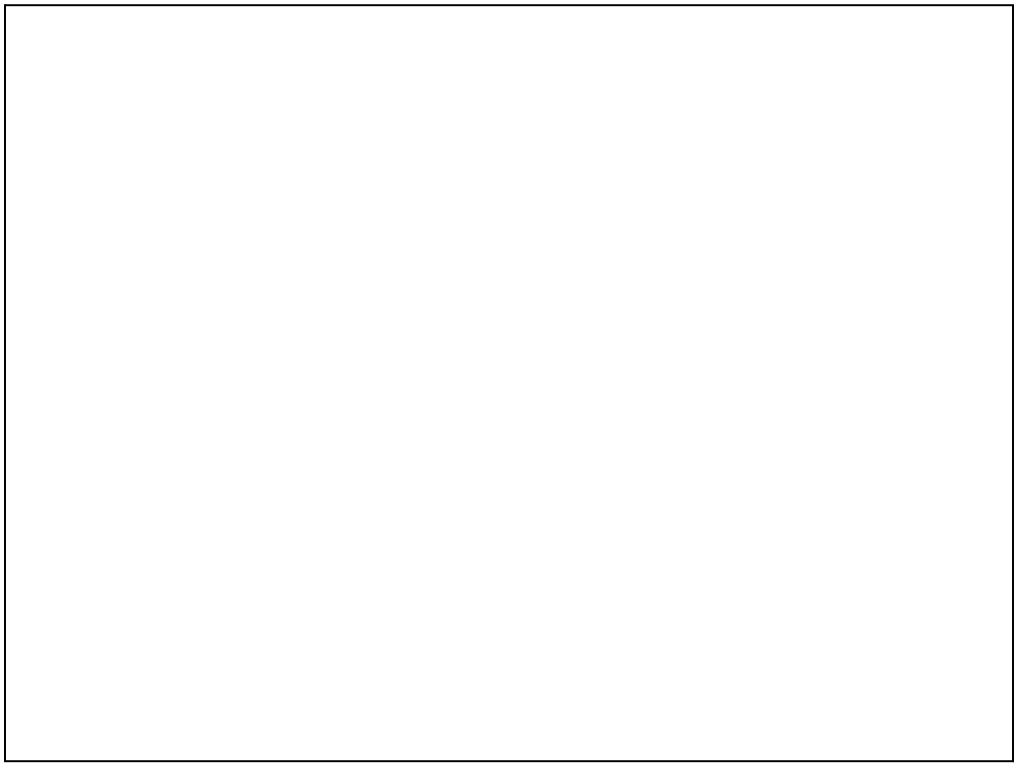




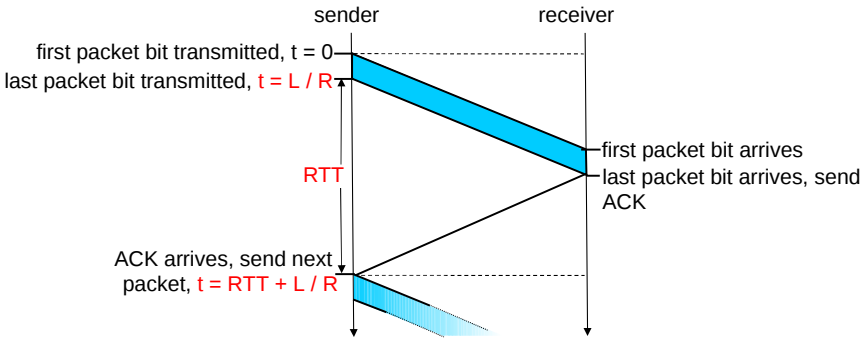




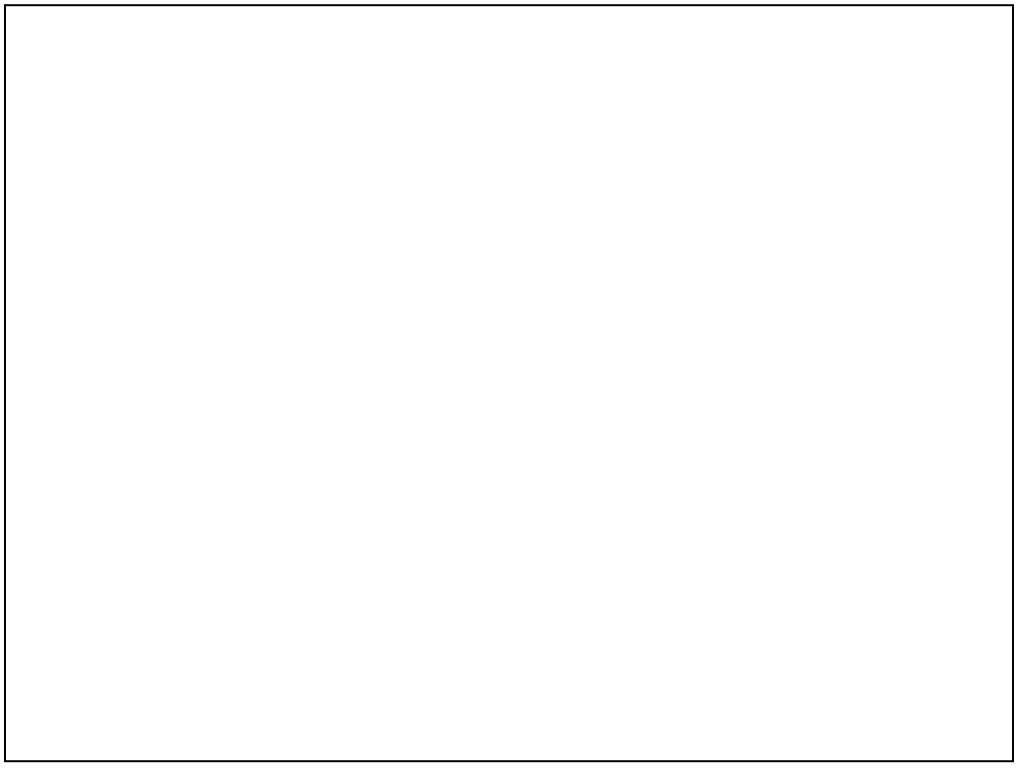


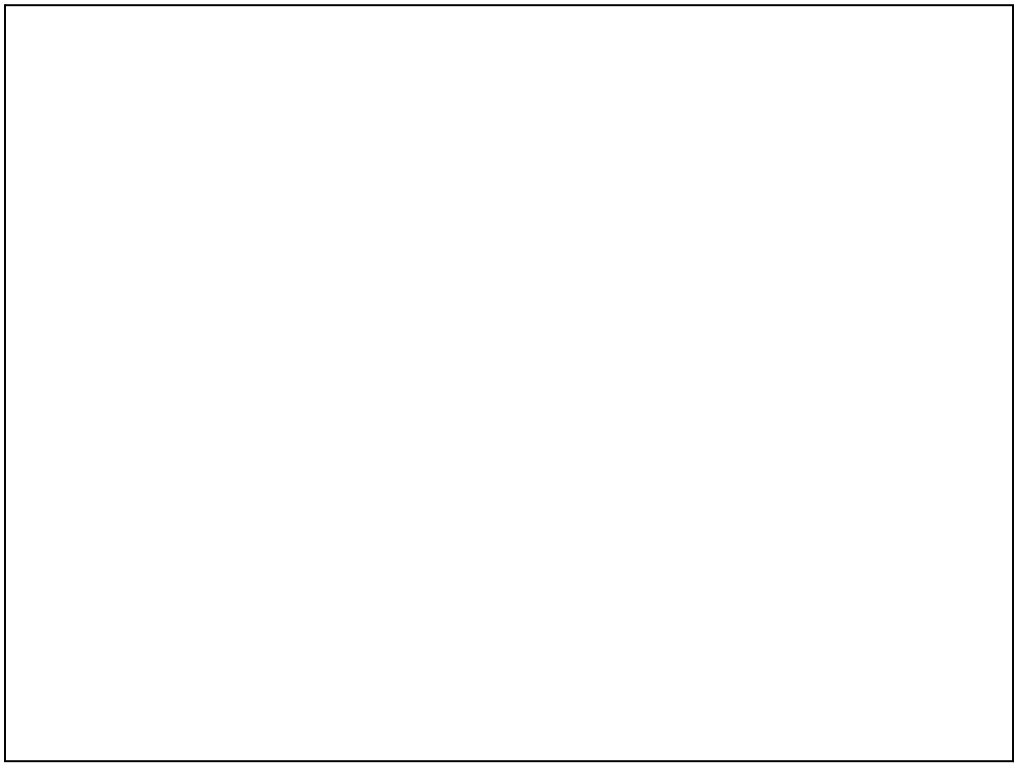


rdt3.0: Stop-and-wait Operation

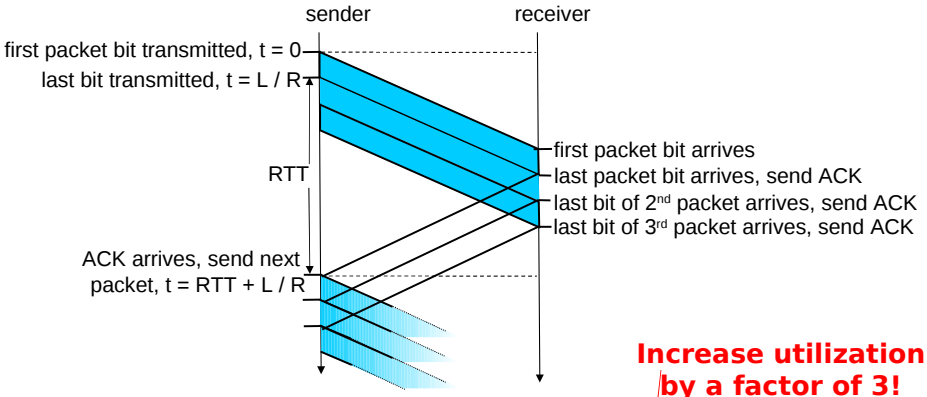


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$





Pipelining: increased utilization



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

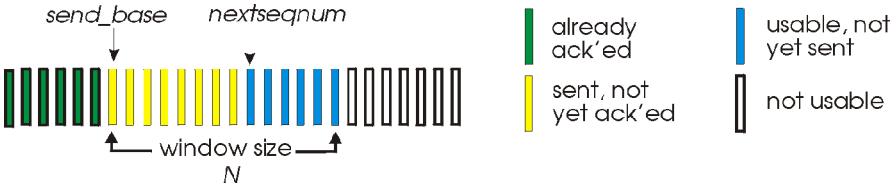
Increase utilization by a factor of 3!

Go-Back-N (sliding window protocol)

Sender

k-bit seq # in pkt header

"window" of up to N, consecutive unack'ed pkts allowed



- **ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"**
  - **Sender may receive duplicate ACKs**
- **Timer for each in-flight pkt**
- ***Timeout(n):* retransmit pkt n and all higher seq # pkts in window**

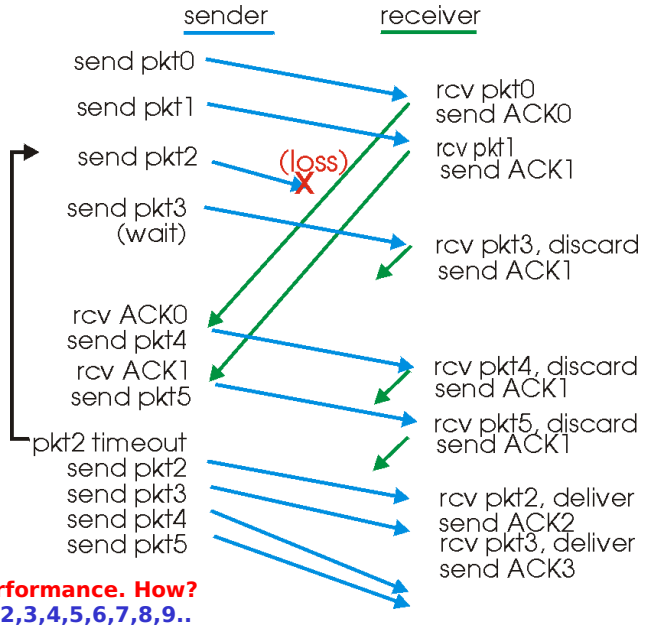
Do not buffer packets after a loss!

GBN in action

Window size=N=4

What determines size of window?

1. **RTT**
2. **Buffer at the receiver(flow control)**
3. **Network congestion**

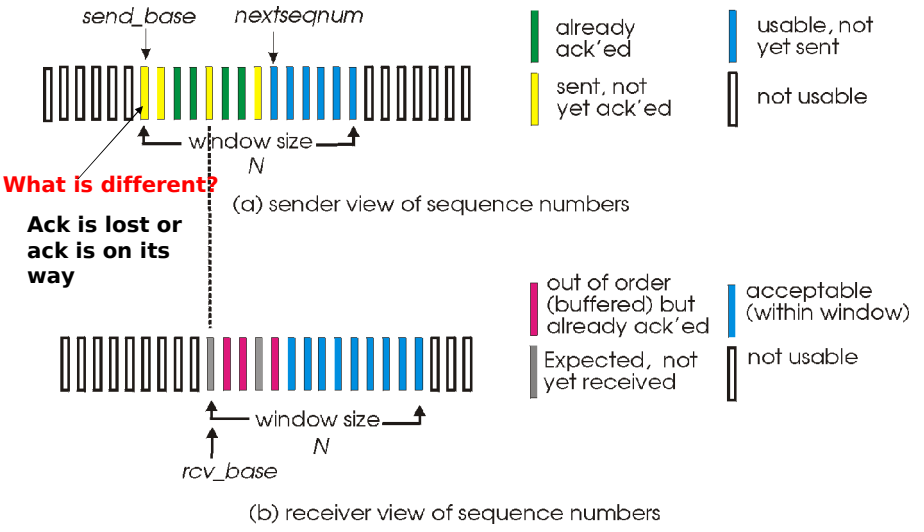


Q: GBN has poor performance. How?
Sender sends pkt 1,2,3,4,5,6,7,8,9..
pkt 1 got lost, receiver got pkt 2,3,4,5,... but will discard them!

Selective Repeat (improvement of the GBN Protocol)

- Receiver *individually* acknowledges all correctly received pkts
  - Buffers pkts, as needed, for eventual in-order delivery to upper layer
  - E.g., sender: pkt 1,2,3,4,….,10; receiver got 2,4,6,8,10. Sender resends 1,3,5,7,9.
- Sender only resends pkts for which ACK not received
  - Sender has timer for EACH unACKed pkt
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed pkts

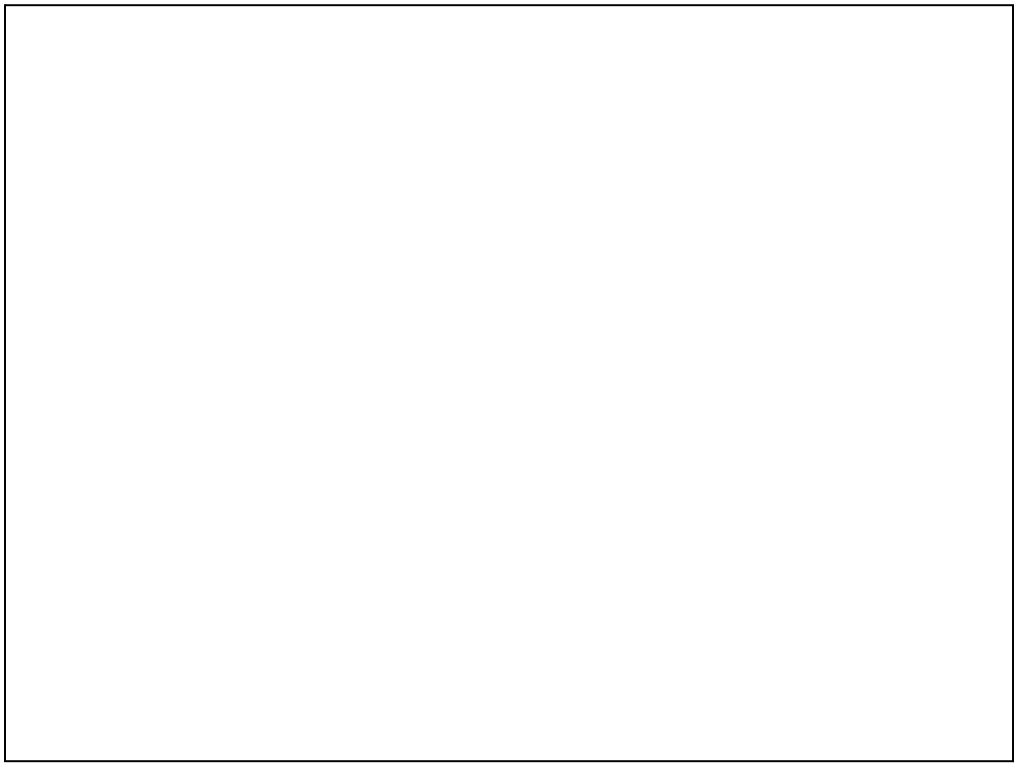
Selective repeat: sender, receiver windows

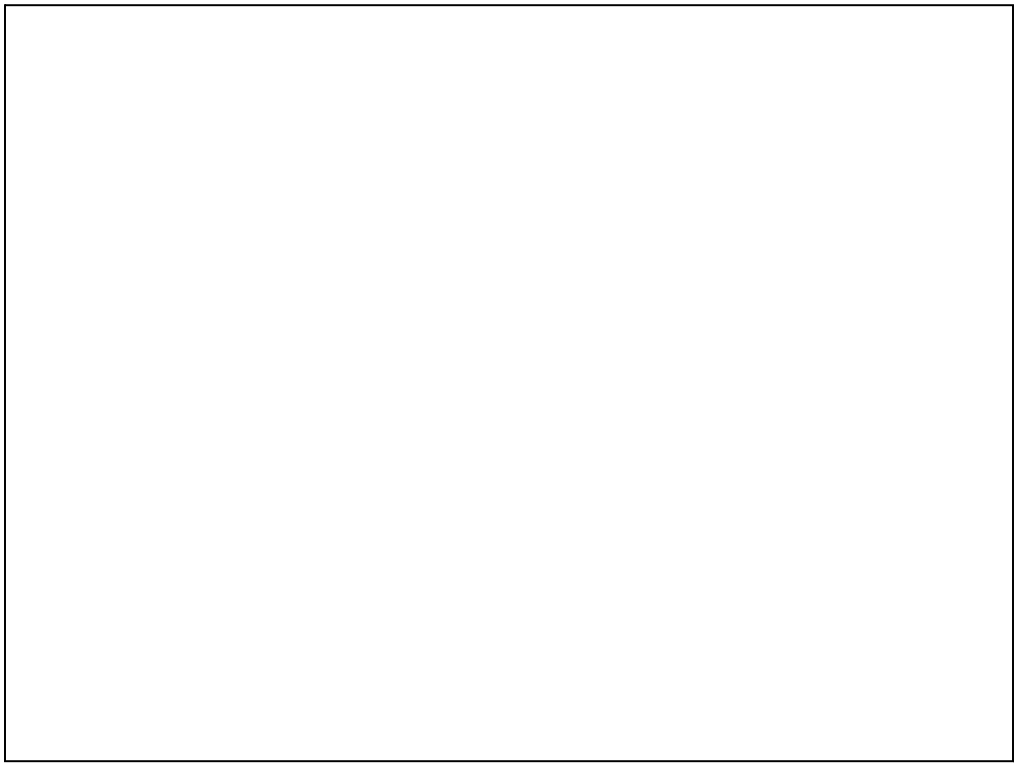


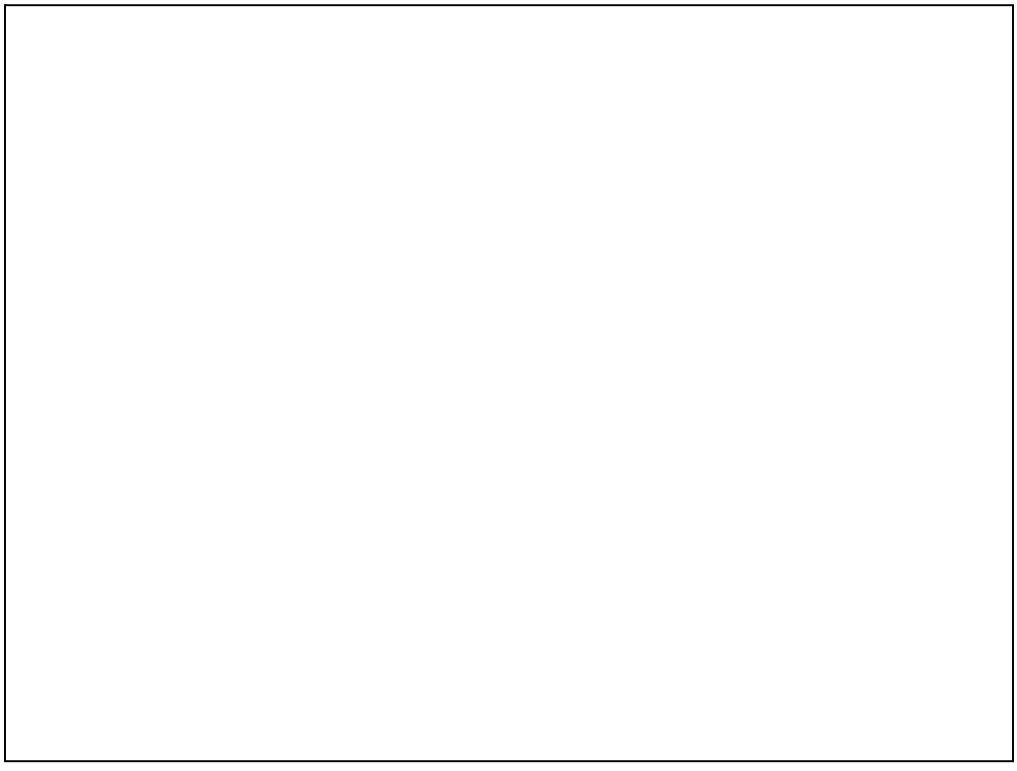
14

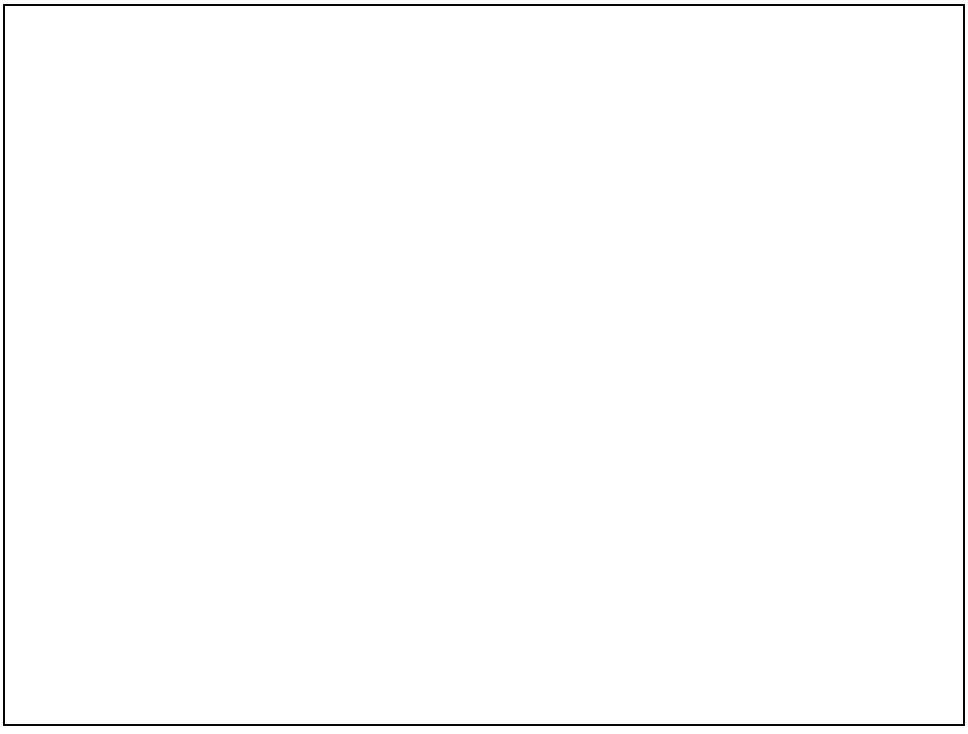
Why bother study reliable data transfer?

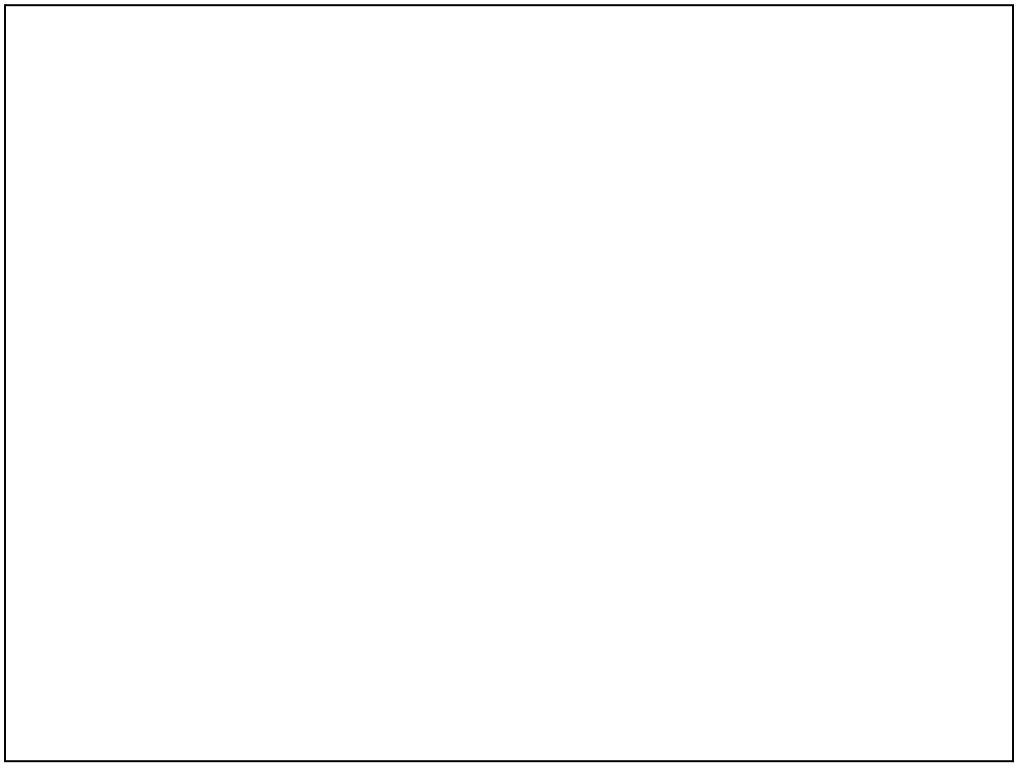
- We know it is provided by TCP, so why bother to study?
- Sometimes, we may need to implement “some form” of reliable transfer without the heavy duty TCP
- A good example is multimedia streaming
  - Even though application is loss tolerant, if too many packets got lost, it affects visual quality
  - So we may want to implement some form of reliable transfer

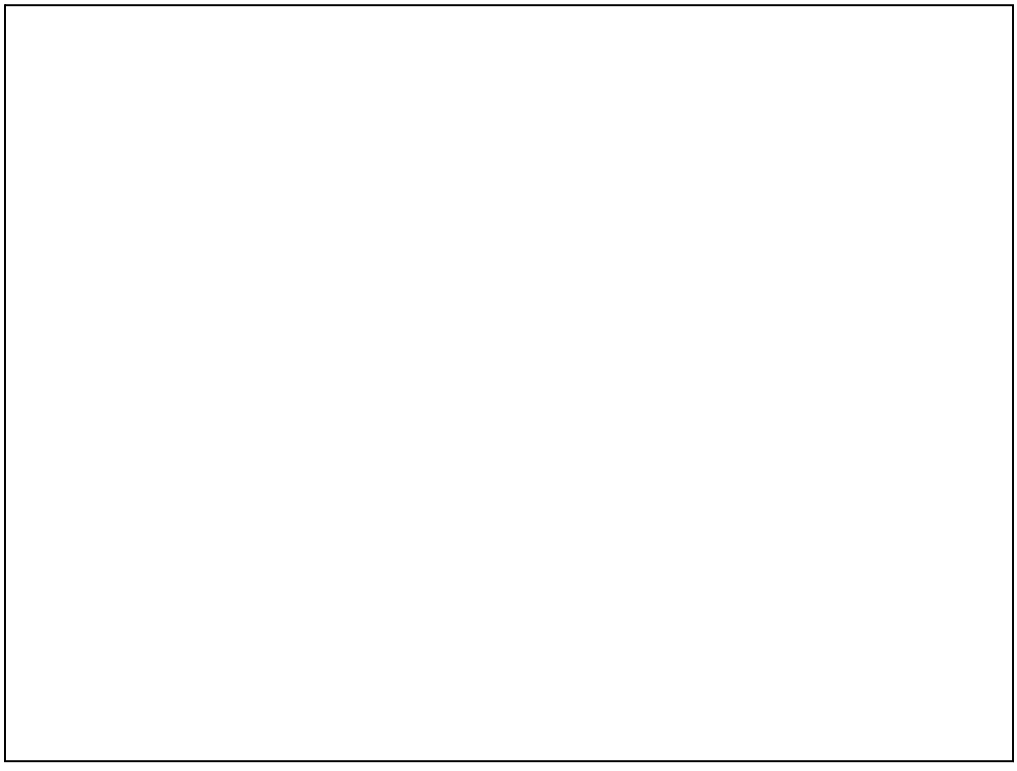


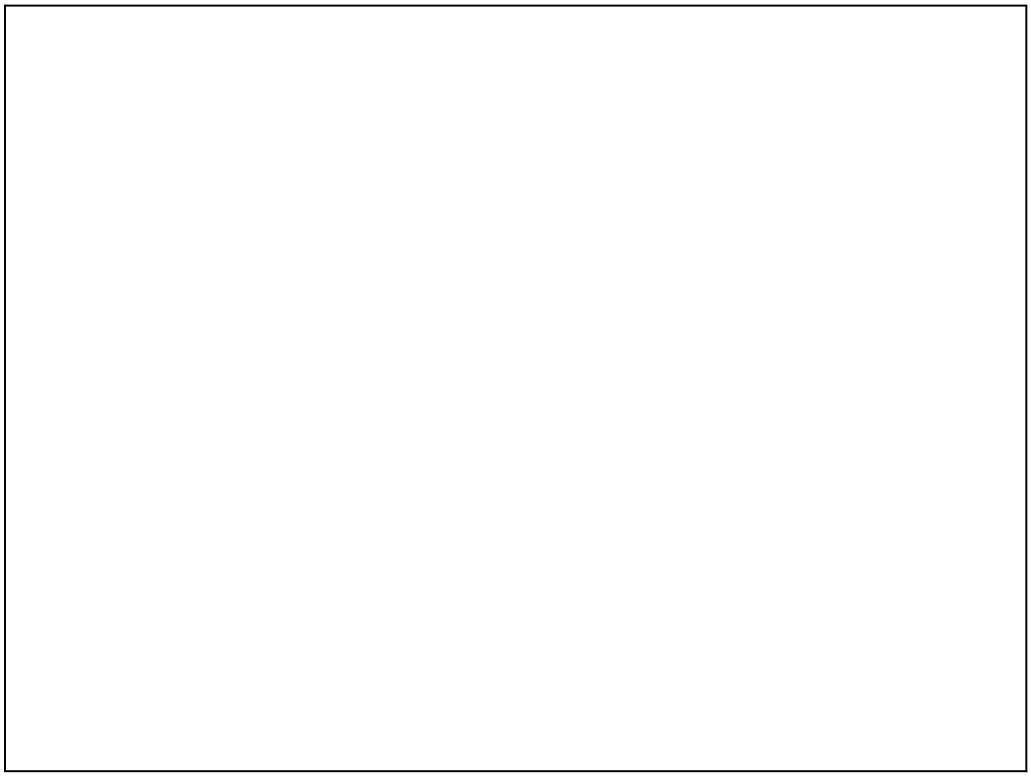


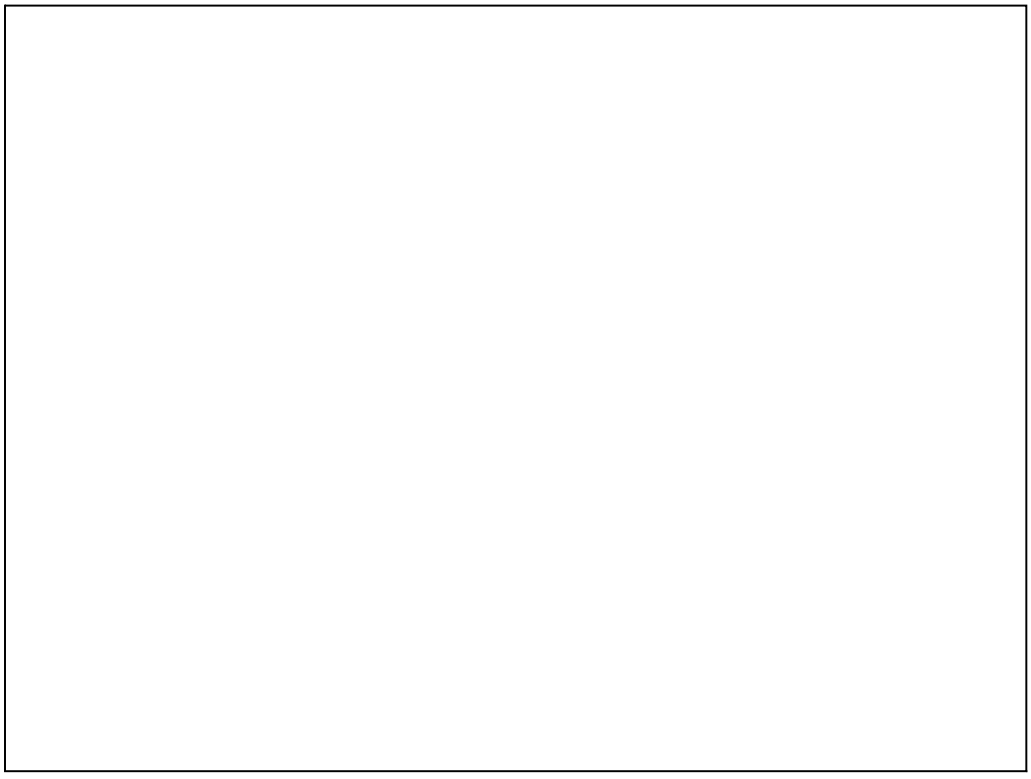


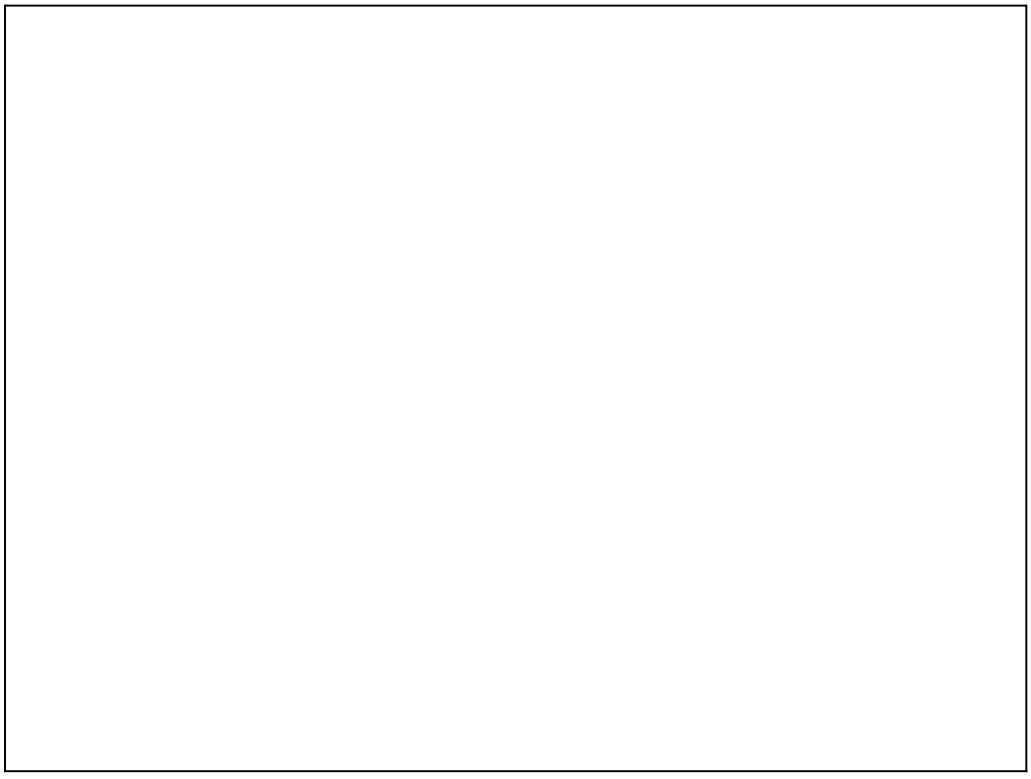


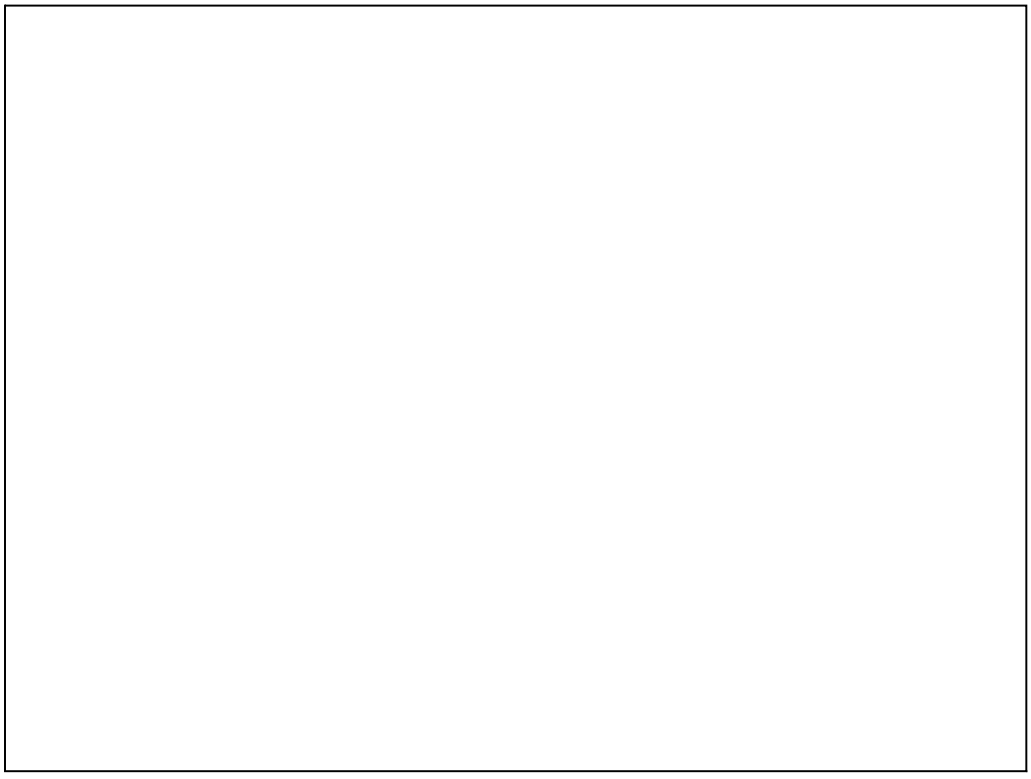


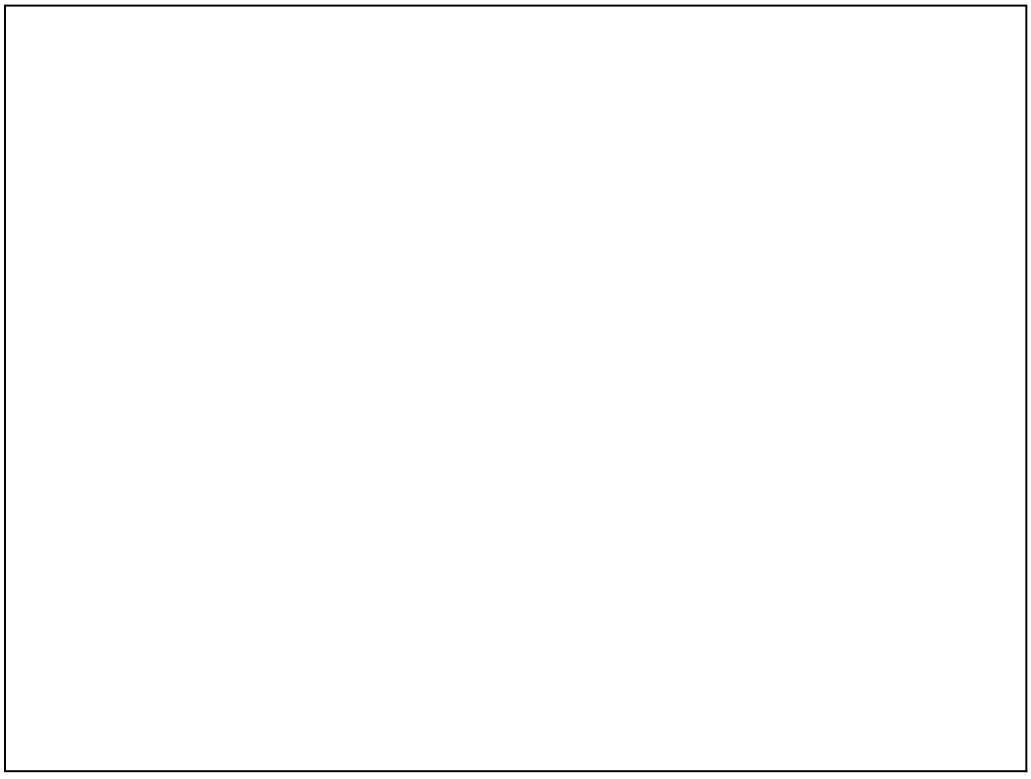


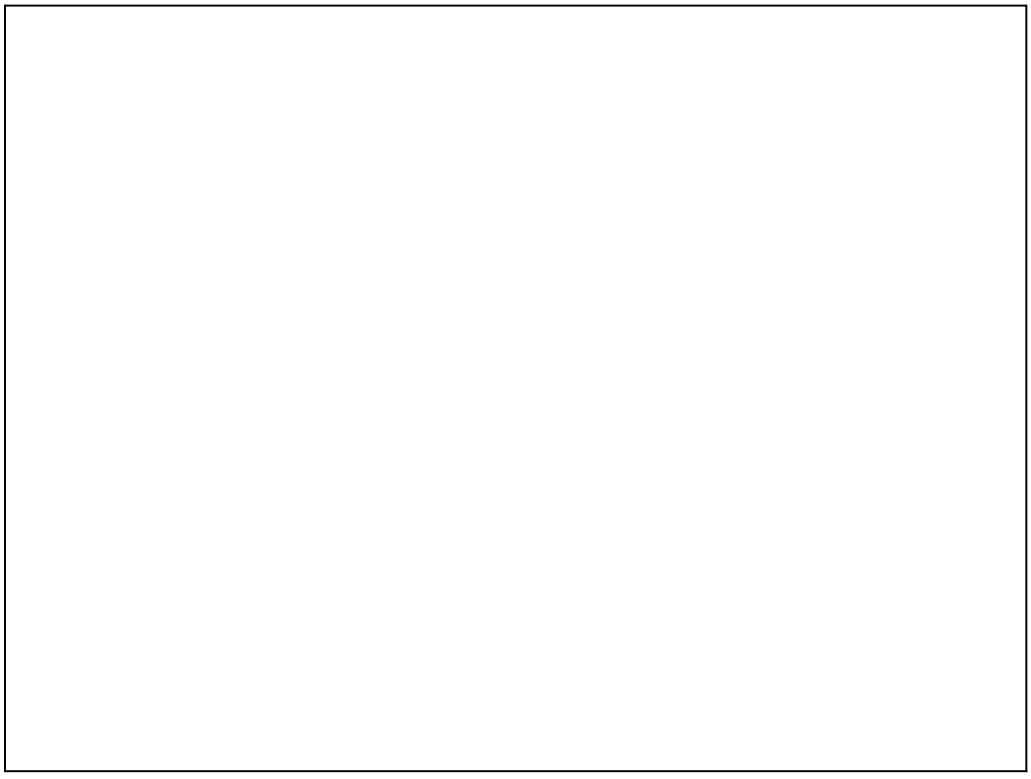


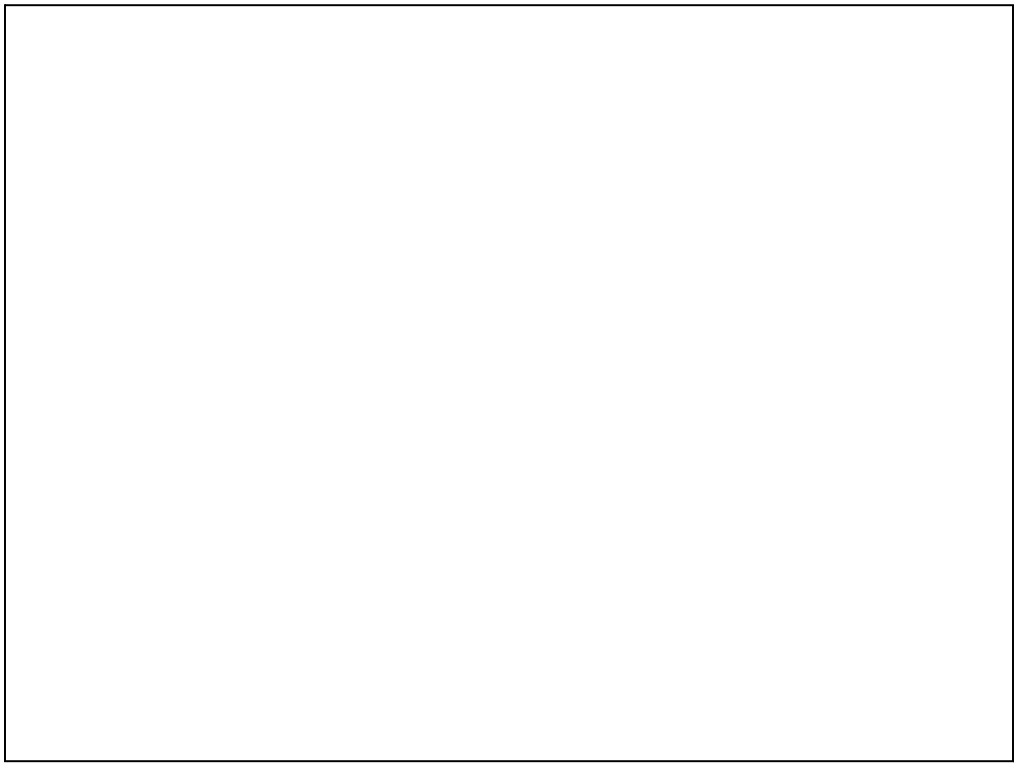


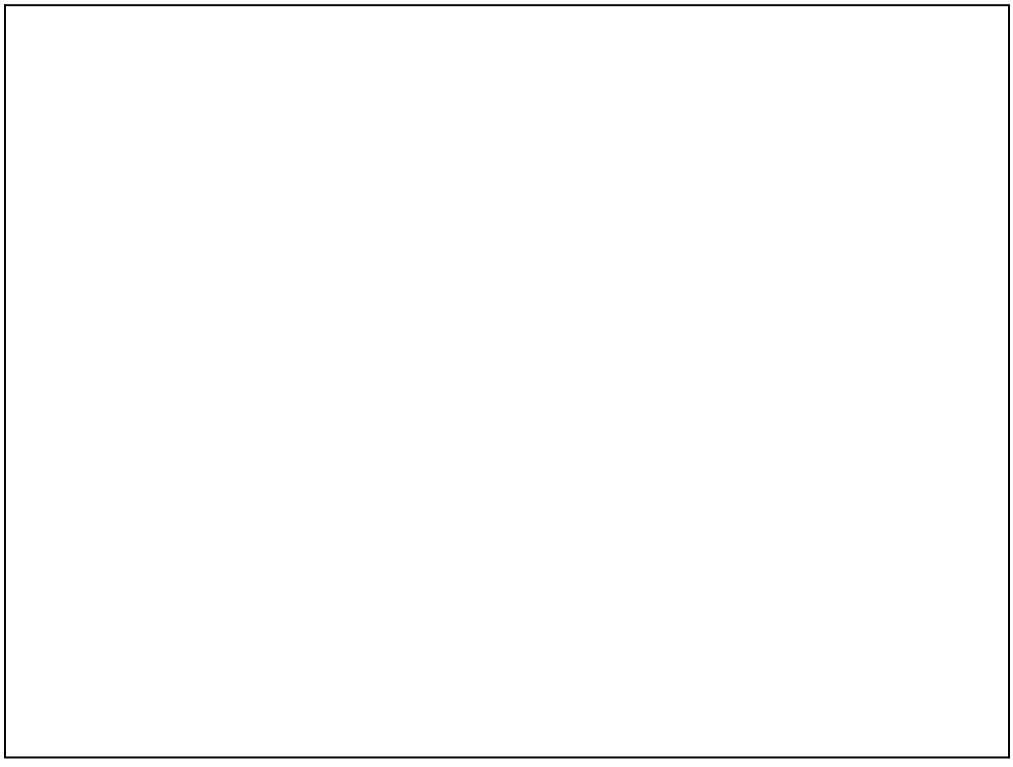


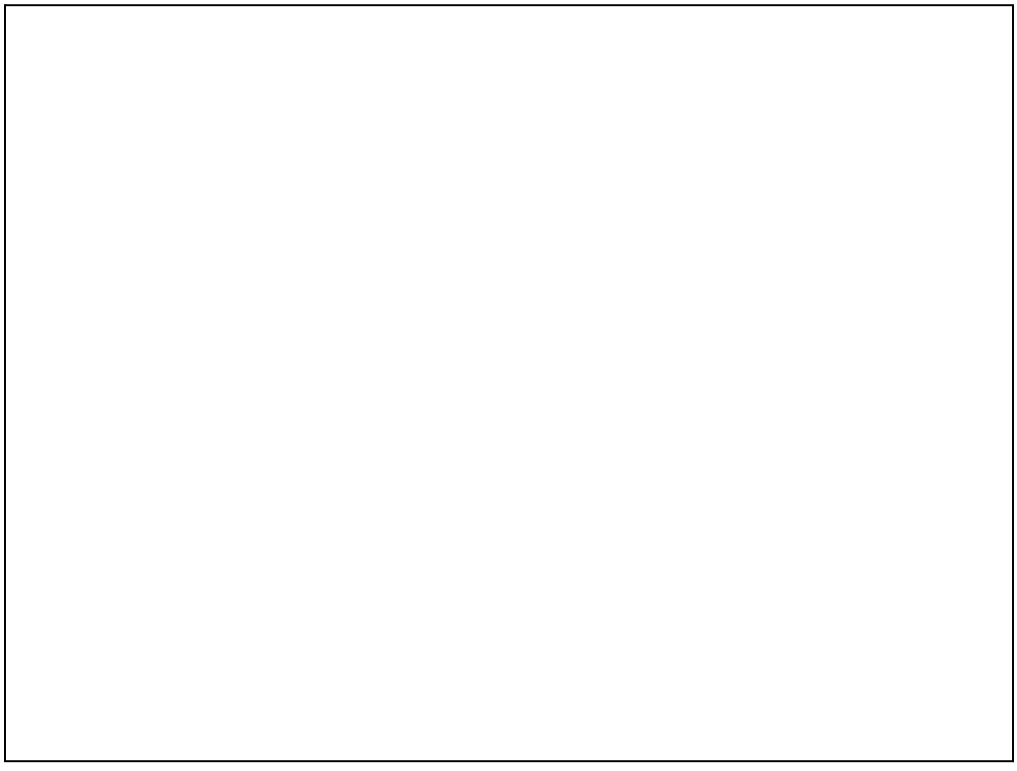


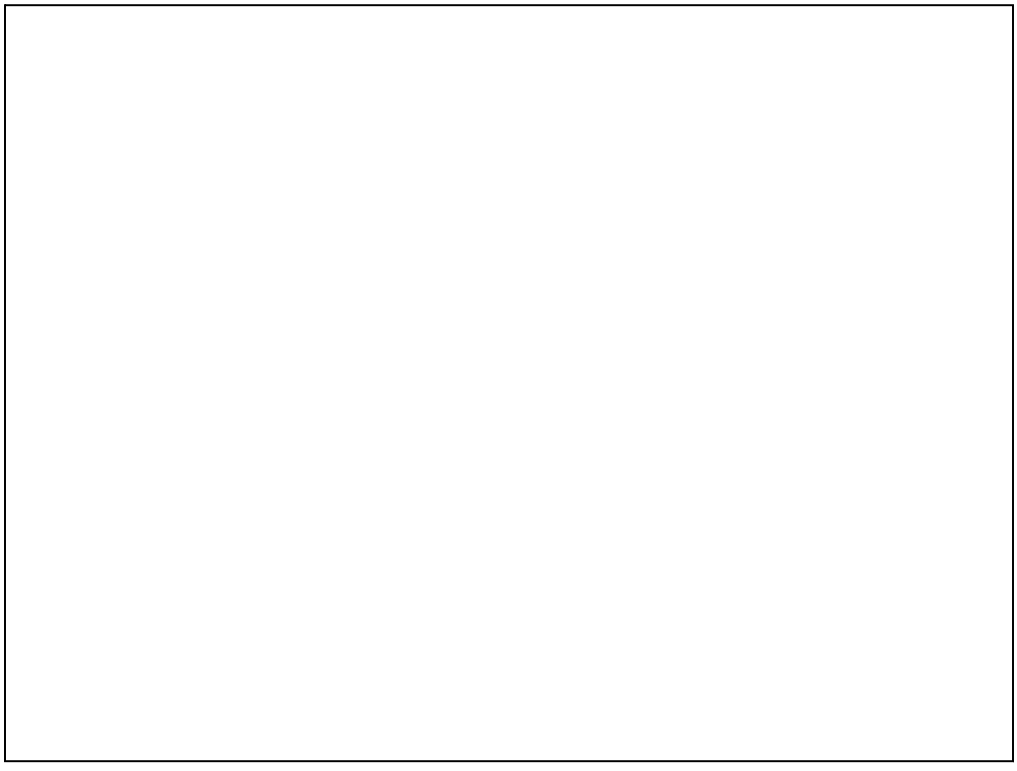


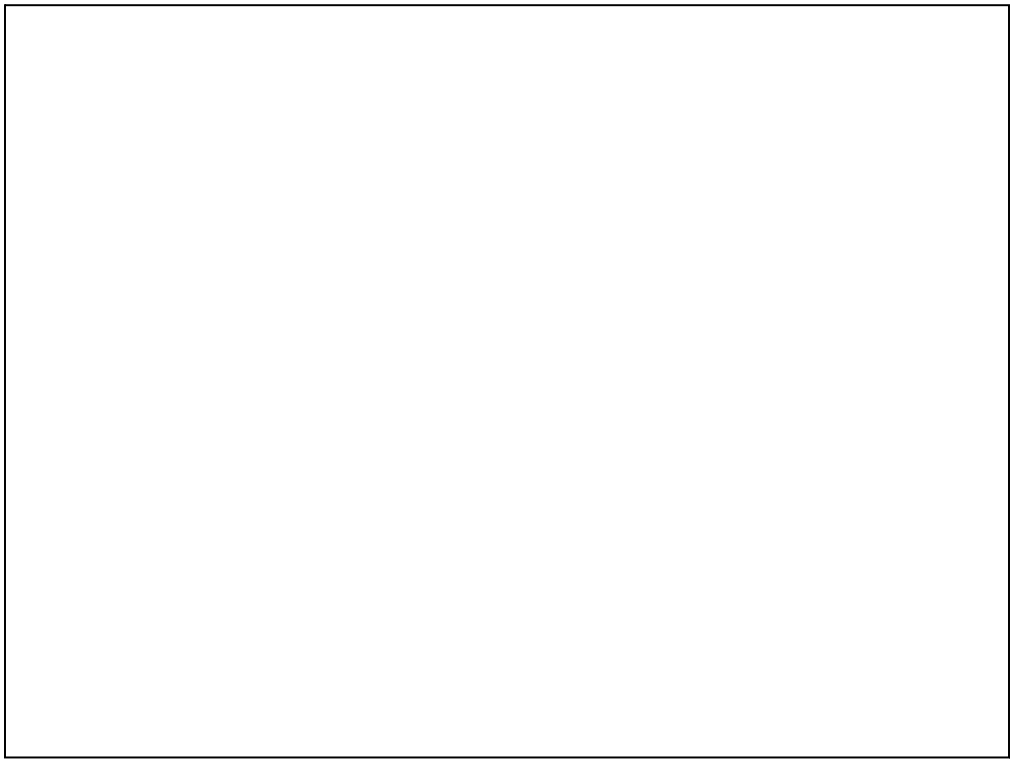


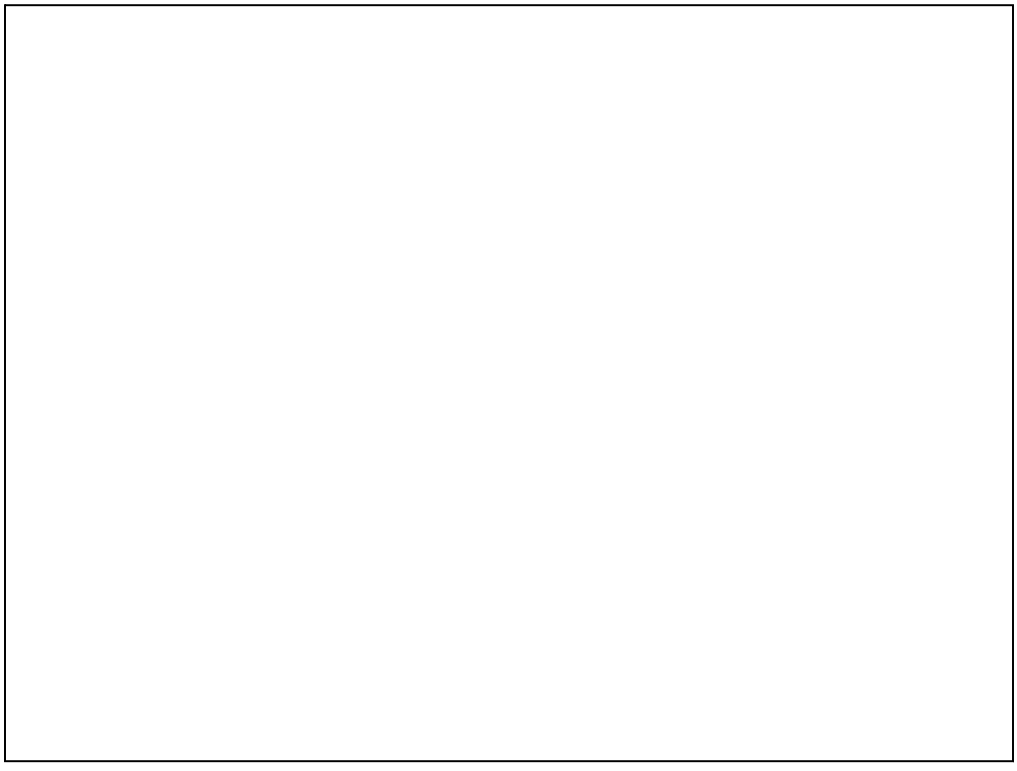












TCP TimeOuts

How Long for TCP Time-out

- Remember, TCP has to ensure reliability
- So bytes need to be resent if there is no "timely" Acknowledgment
- How long should the sender wait ?
- It should be adaptive -- fluctuation in network load
- If too short, false time-outs
- If too long, then poor rate of sending
- Depends on round trip time estimation