Network Transport Layer: TCP

Qiao Xiang

https://qiaoxiang.me/courses/cnns-xmuf21/index.shtml

11/16/2021

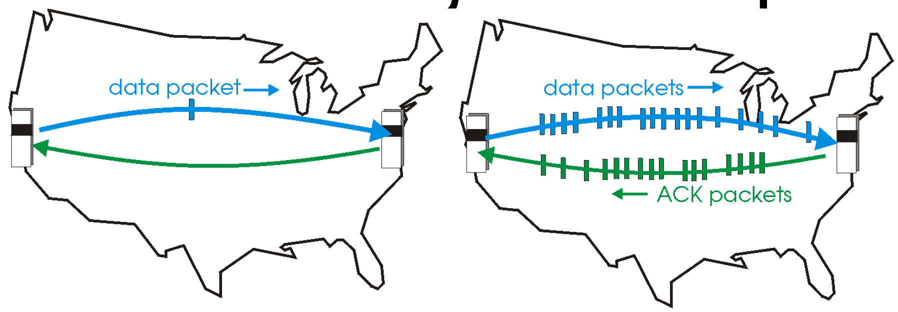
This deck of slides are heavily based on CPSC 433/533 at Yale University, by courtesy of Dr. Y. Richard Yang.

Admin

- Lab assignment 2 to be returned on Nov. 18
- Class Project
  - 15% of your final score
  - Please start ASAP
  - Talk to the instructor or TA to get feedback

Recap: Reliable Transport

- Basic structure: sliding window protocols



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

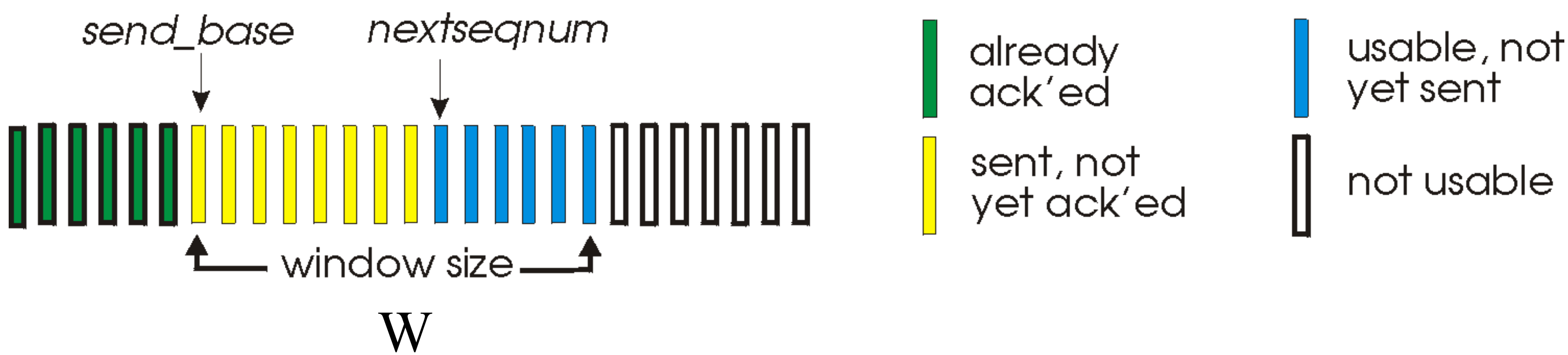
General technique: pipelining.

- Realization: GBN or SR

Recap: Go-Back-N

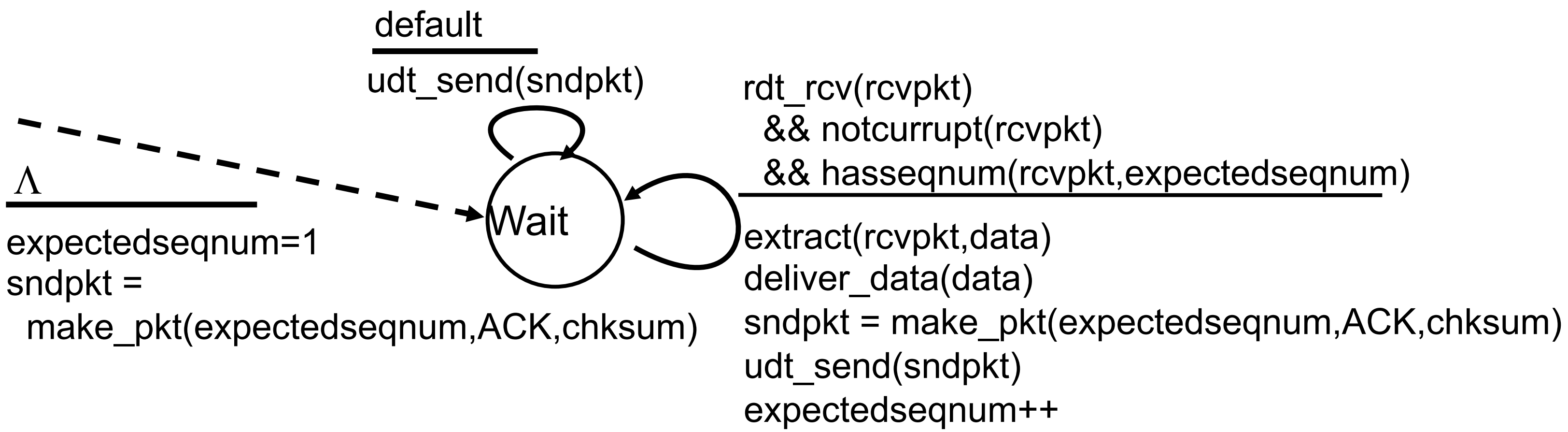
Sender:

- k-bit seq # in pkt header
- "window" of up to W, consecutive unack'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - note: ACK(n) could mean two things: I have received upto and include n, or I am waiting for n
- timer for the packet at base
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

Recap: Go-Back-N



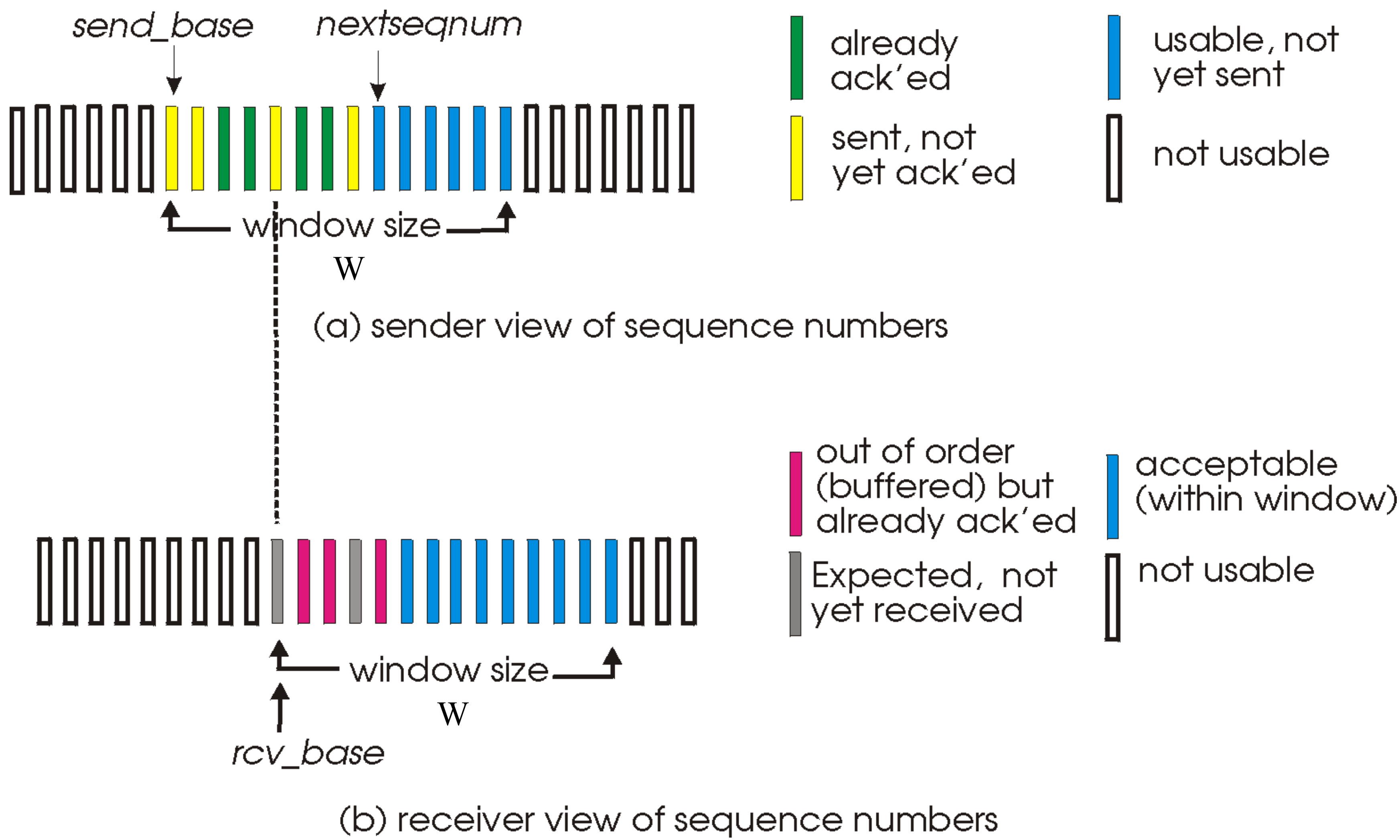
Only state: **`expectedseqnum`**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - re-ACK pkt with highest in-order seq #
  - may generate duplicate ACKs

Selective Repeat

- Sender window
  - Window size W: W consecutive unACKed seq #'s
- Receiver *individually* acknowledges correctly received pkts
  - buffers out-of-order pkts, for eventual in-order delivery to upper layer
  - ACK(n) means received packet with seq# n only
  - buffer size at receiver: window size
- Sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

Selective Repeat: Sender, Receiver Windows



Selective Repeat

sender

data from above :

- unACKed packets is less than window size W, send; otherwise block app.

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+W-1]:

- mark pkt n as received
- update sendbase to the first packet unACKed

receiver

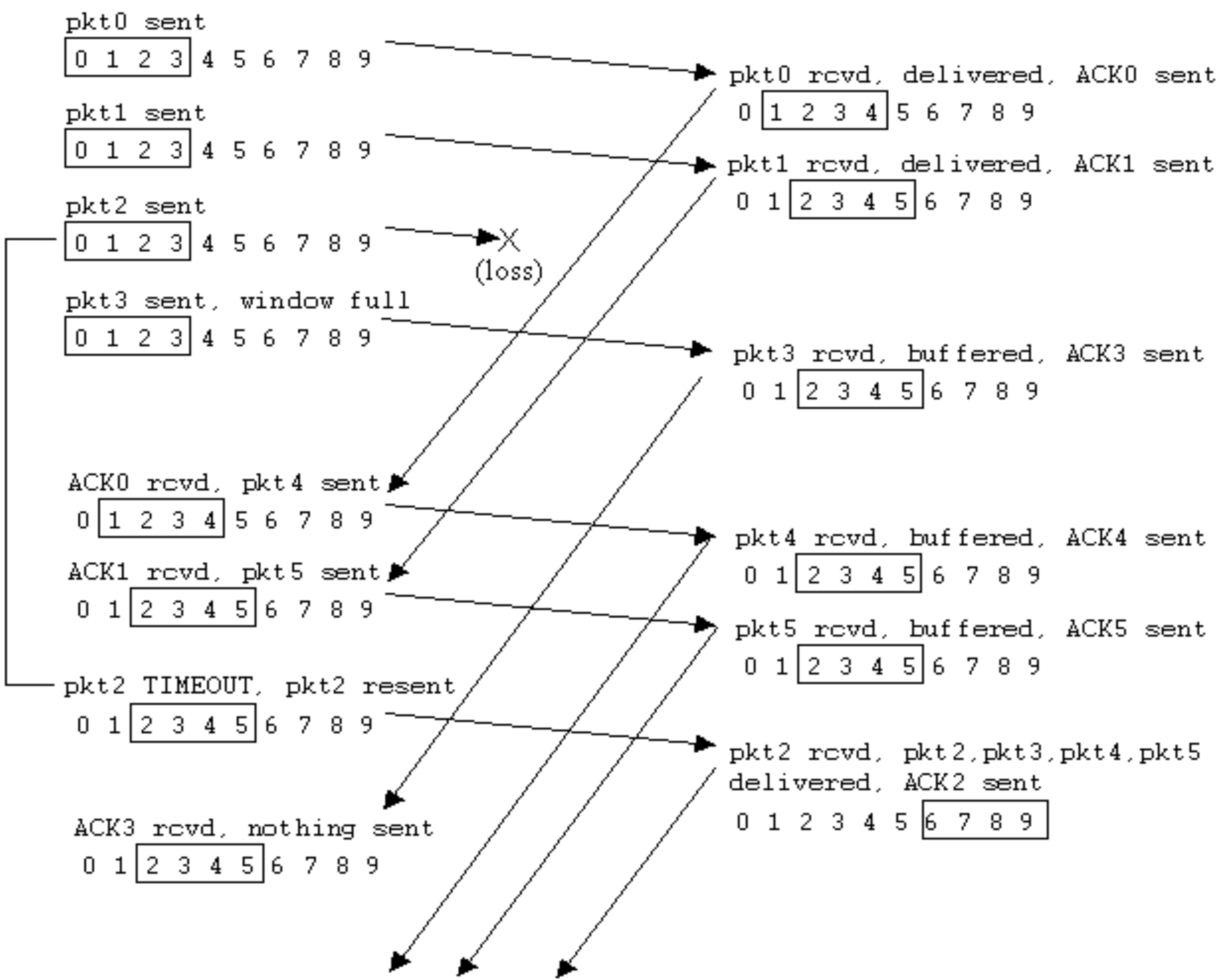
pkt n in [rcvbase, rcvbase+W-1]

- send ACK(n)
- if (out-of-order)
  mark and buffer pkt n
  else /*in-order*/
  deliver any in-order packets

otherwise:

- ignore

Selective Repeat in Action



Discussion: Efficiency of Selective Repeat

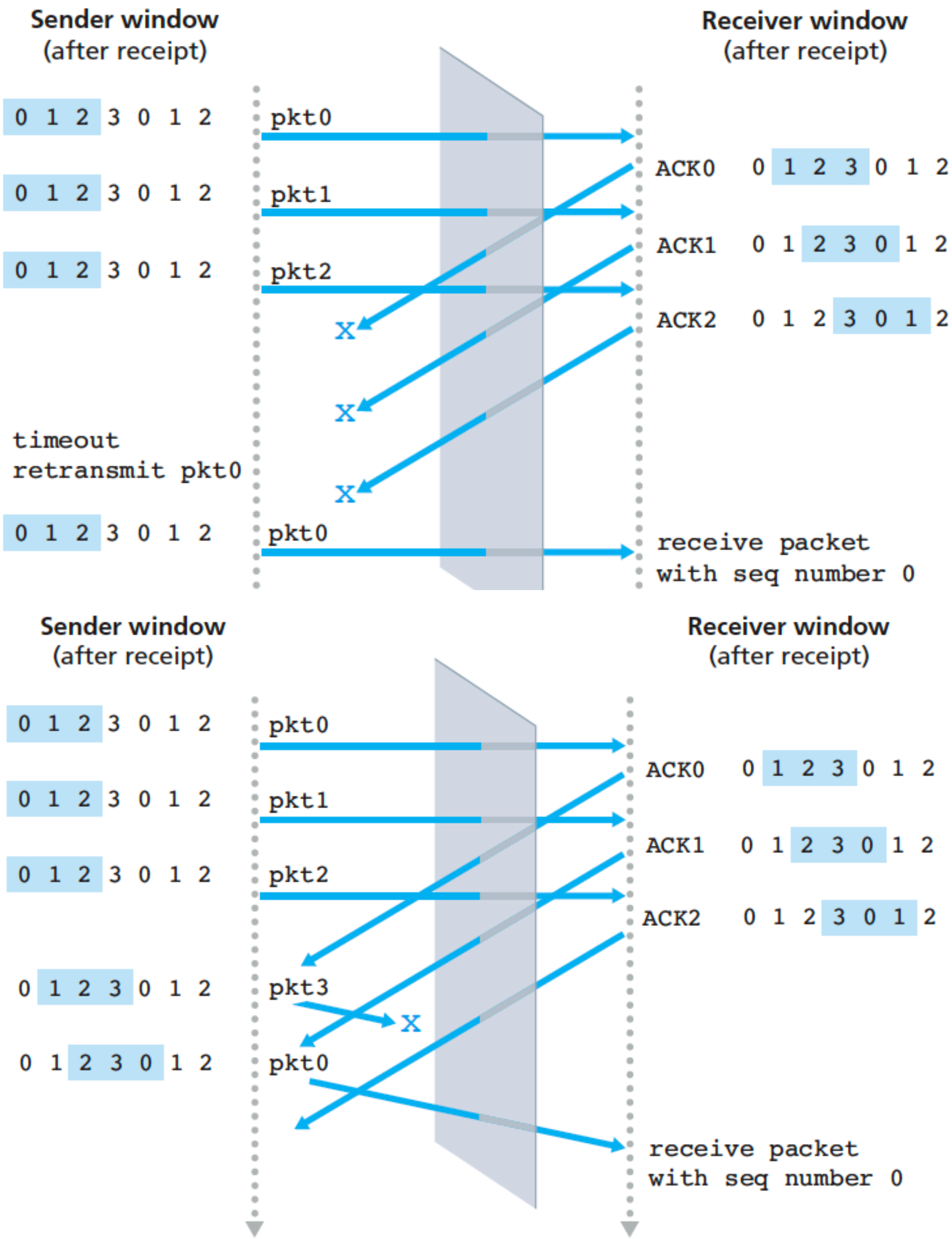
- Assume window size W
- Assume each packet is lost with probability p
- On average, how many packets do we send for each data packet received?

Selective Repeat: Seq# Ambiguity

Example:

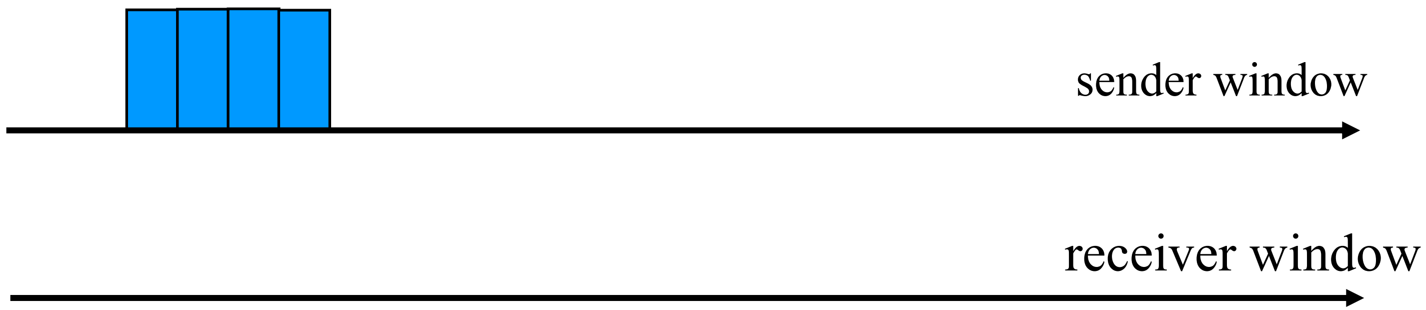
- seq #'s: 0, 1, 2, 3
- window size=3

- Error: incorrectly passes duplicate data as new.

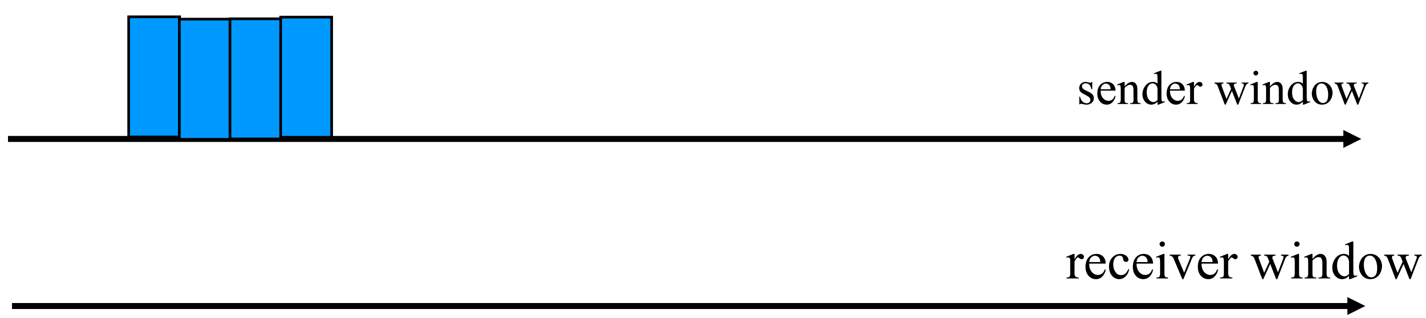


State Invariant: Window Location

- Go-back-n (GBN)



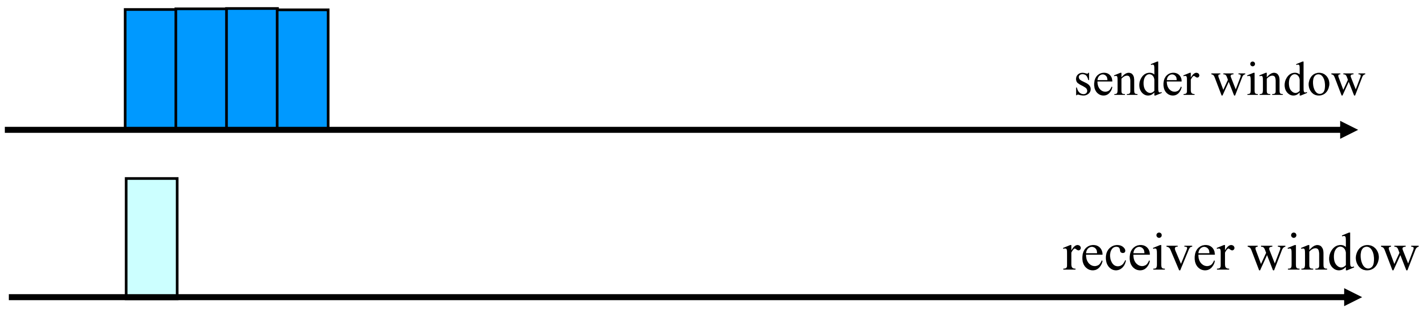
- Selective repeat (SR)



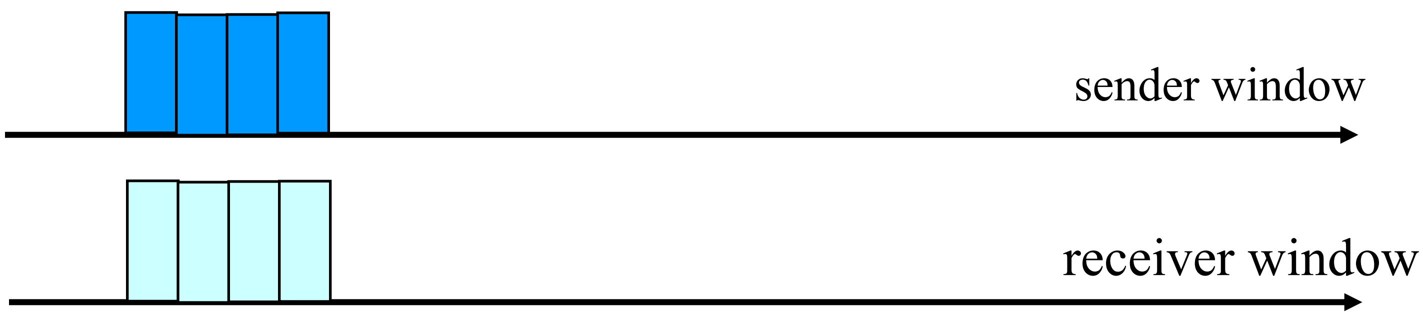
Window Location

Q: what relationship between seq # size and window size?

- Go-back-n (GBN)



- Selective repeat (SR)



Selective Repeat

sender

data from above :

- unACKed packets is less than window size W, send; otherwise block app.

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+W-1]:

- mark pkt n as received
- update sendbase to the first packet unACKed

receiver

pkt n in [rcvbase, rcvbase+W-1]

- send ACK(n)
- if (out-of-order)
  mark and buffer pkt n
  else /*in-order*/
  deliver any in-order packets

pkt n in [rcvbase-W, rcvbase-1]

- send ACK(n)

otherwise:

- ignore

Sliding Window Protocols: Go-back-n and Selective Repeat

	Go-back-n	Selective Repeat
data bandwidth: sender to receiver (avg. number of times a pkt is transmitted)	Less efficient $\frac{1-p+pw}{1-p}$	More efficient $\frac{1}{1-p}$
ACK bandwidth (receiver to sender)	More efficient	Less efficient
Relationship between M (the number of seq#) and W (window size)	M > W	M ≥ 2W
Buffer size at receiver	1	W
Complexity	Simpler	More complex

p: the loss rate of a packet; M: number of seq# (e.g., 3 bit M = 8); W: window size

Outline

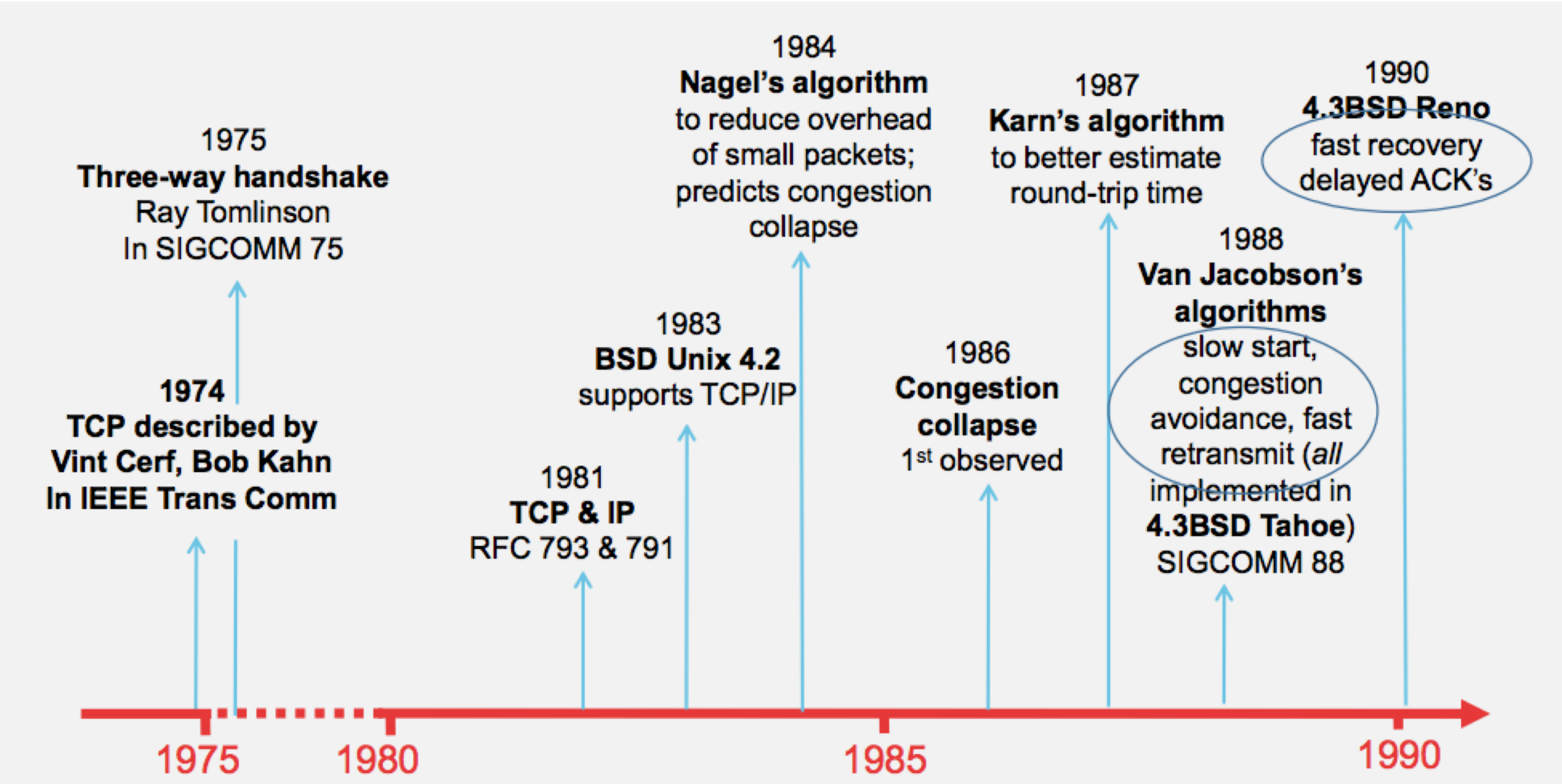
- Admin and Recap
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses
  - sliding window: reliability with throughput
- *TCP reliability*

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

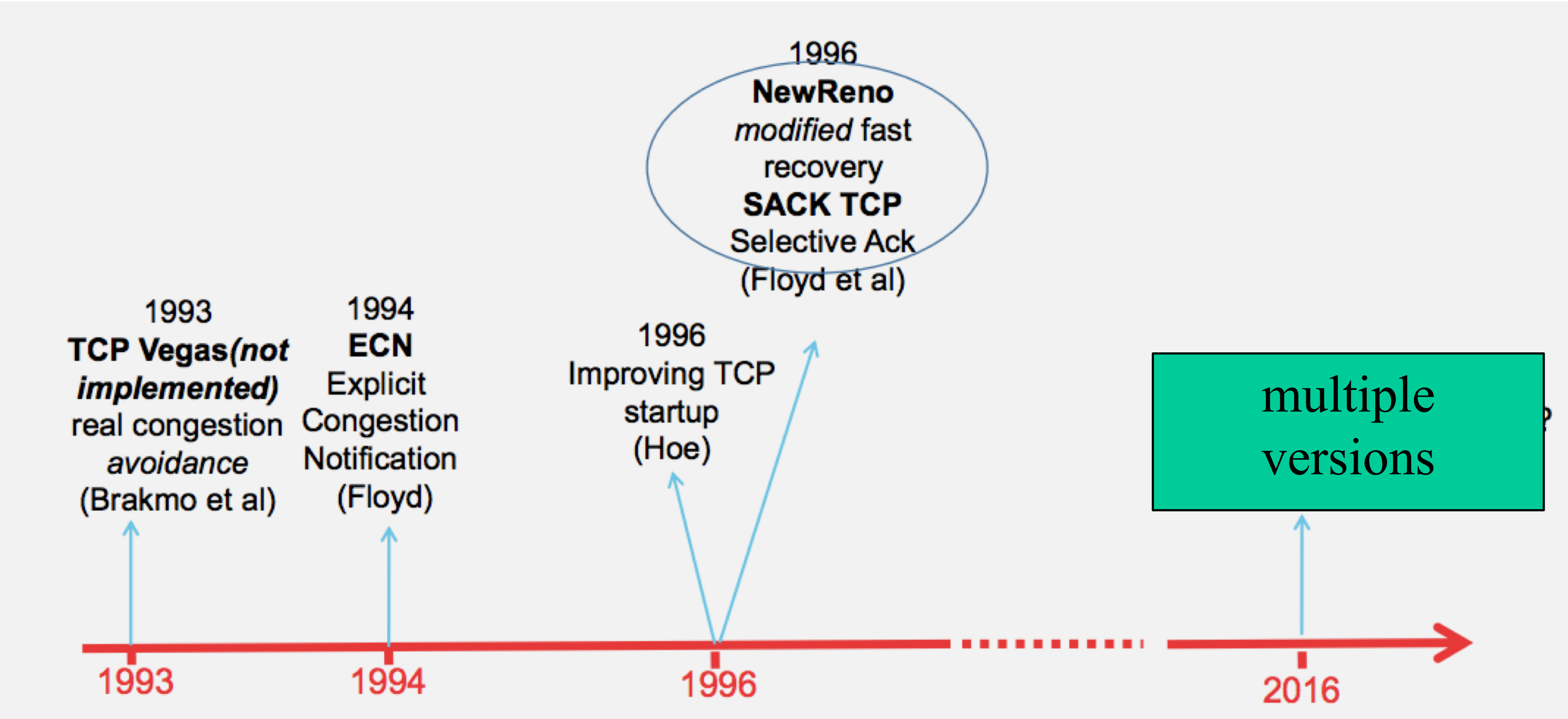
- Point-to-point reliability: one sender, one receiver
- Flow controlled and congestion controlled

Evolution of TCP



Source: http://webcourse.cs.technion.ac.il/236341/Winter2015-2016/ho/WCFiles/Tutorial10.pdf

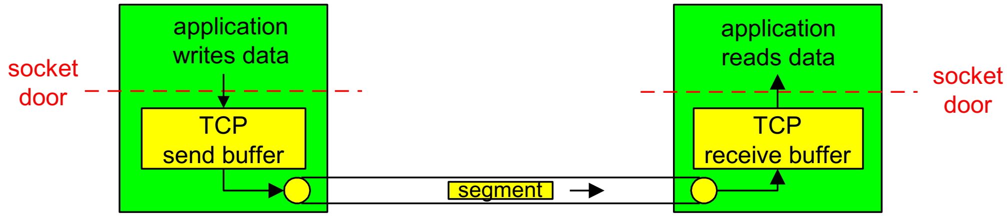
Evolution of TCP



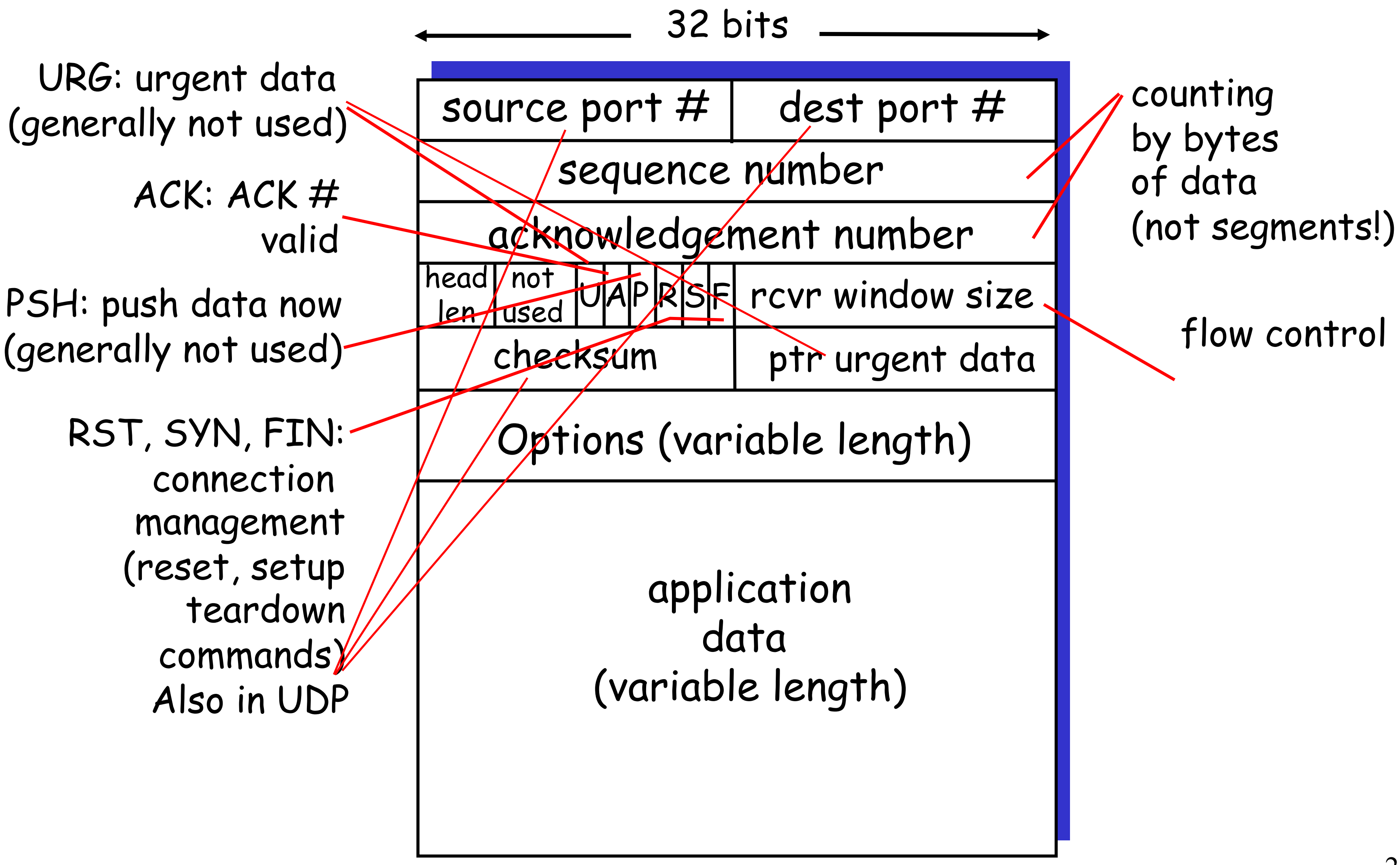
Source: http://webcourse.cs.technion.ac.il/236341/Winter2015-2016/ho/WCFiles/Tutorial10.pdf

TCP Reliable Data Transfer

- **Connection-oriented:**
  - connection management
    - setup (exchange of control msgs) init's sender, receiver state before data exchange
    - close
- **Full duplex data:**
  - bi-directional data flow in same connection
- **A sliding window protocol**
  - a combination of go-back-n and selective repeat:
    - send & receive buffers
    - cumulative acks
    - TCP uses a single retransmission timer
    - do not retransmit all packets upon timeout



TCP Segment Structure

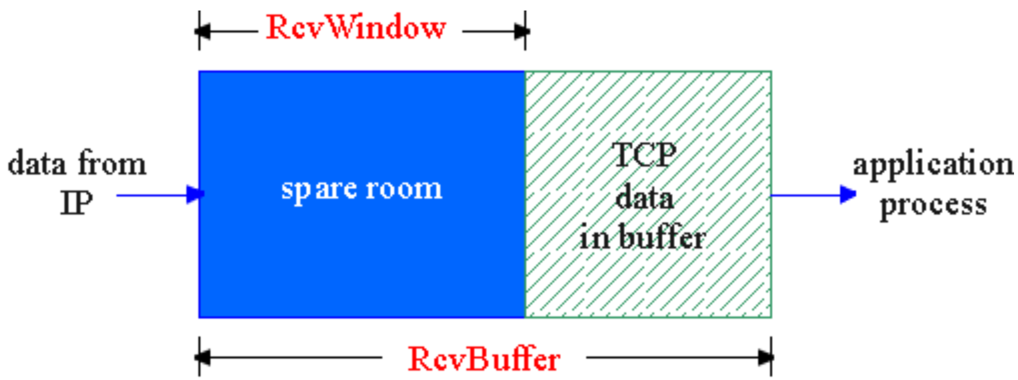


Outline

- Admin and Recap
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses
  - sliding window: reliability with throughput
- TCP reliability
  - *data seq#, ack, buffering*

Flow Control

- receive side of a connection has a receive buffer:



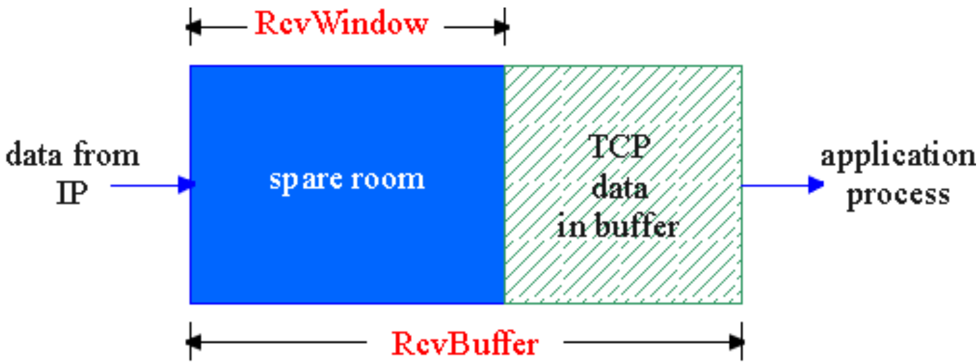
- app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

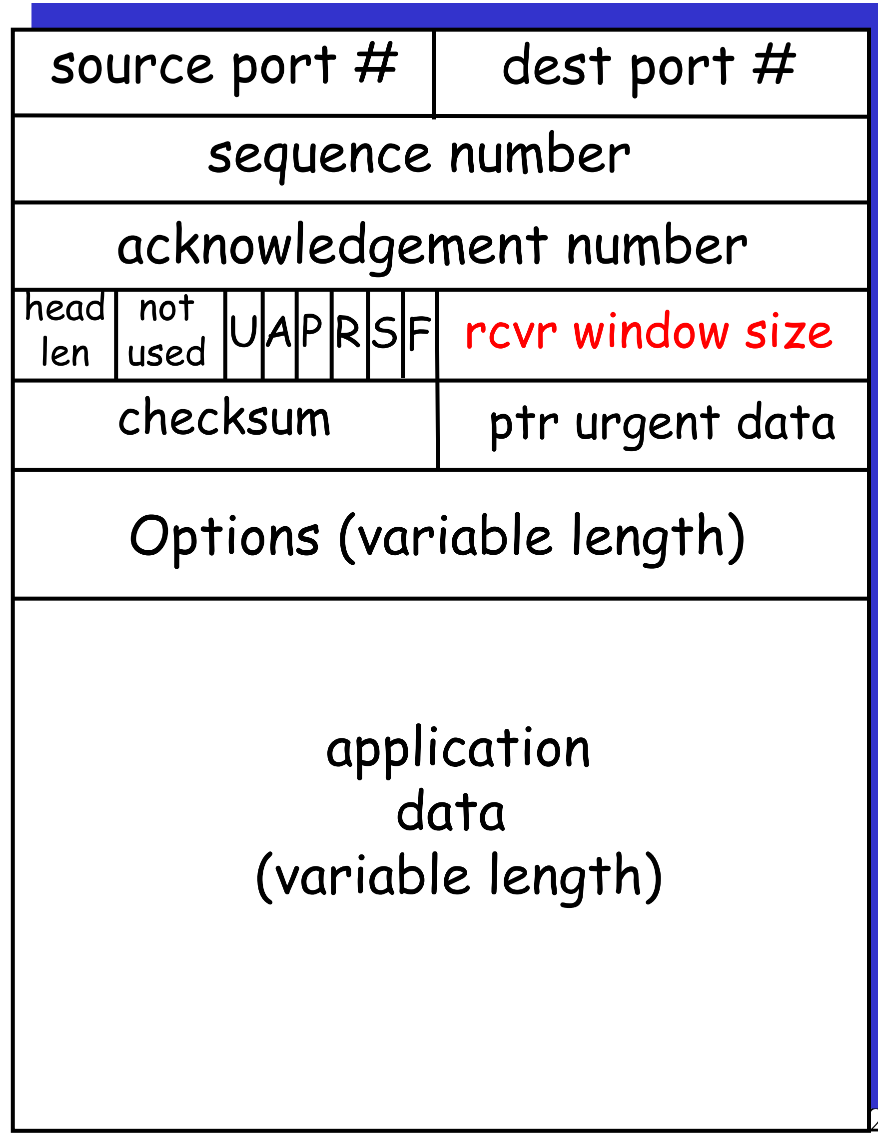
- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow Control: How it Works



- spare room in buffer

= **RcvWindow**



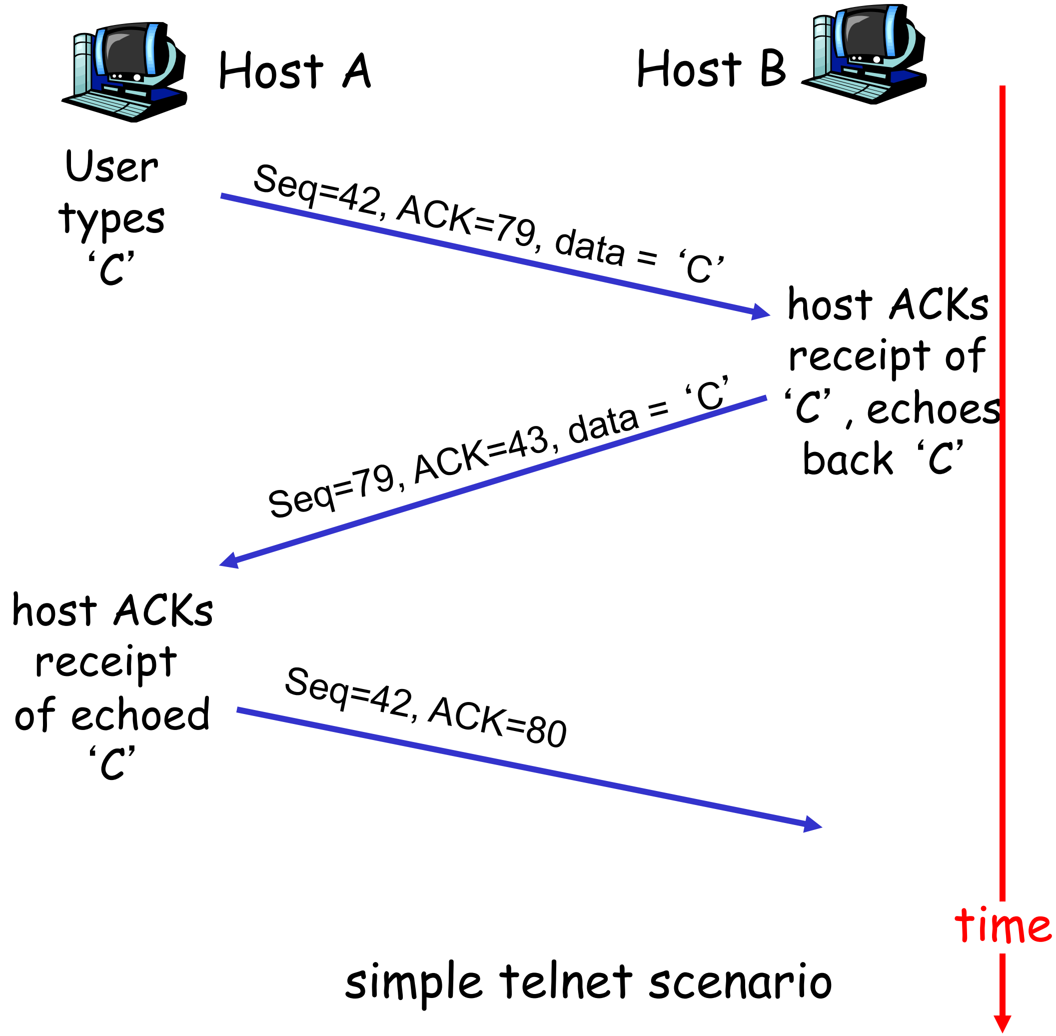
TCP Seq. #'s and ACKs

Seq. #'s:

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

ACKs:

- seq # of next byte expected from other side
- cumulative ACK in standard header
- selective ACK in options



simple telnet scenario

TCP Send/Ack Optimizations

- TCP includes many tune/optimizations, e.g.,
  - the "small-packet problem": sender sends a lot of small packets (e.g., telnet one char at a time)
    - Nagle's algorithm: do not send data if there is small amount of data in send buffer and there is an unack'd segment
  - the "ack inefficiency" problem: receiver sends too many ACKs, no chance of combing ACK with data
    - Delayed ack to reduce # of ACKs/combine ACK with reply

TCP Receiver ACK Generation [RFC 1122, RFC 2581]

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

Outline

- Admin and Recap
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses
  - sliding window: reliability with throughput
- TCP reliability
  - data seq#, ack, buffering
  - *timeout realization*

TCP Reliable Data Transfer

- Basic structure: sliding window protocol
- Remaining issue: How to determine the "right" parameters?
  - timeout value?
  - sliding window size?

History

- Key parameters for TCP in mid-1980s
  - fixed window size W
  - timeout value = 2 RTT

- Network collapse in the mid-1980s
  - UCB ←→ LBL throughput dropped by 1000X !
- The intuition was that the collapse was caused by wrong parameters…

Timeout: Cost of Timeout Param

Why is good timeout value important?

- too short
  - premature timeout
  - unnecessary retransmissions; many duplicates
- too long
  - slow reaction to segment loss

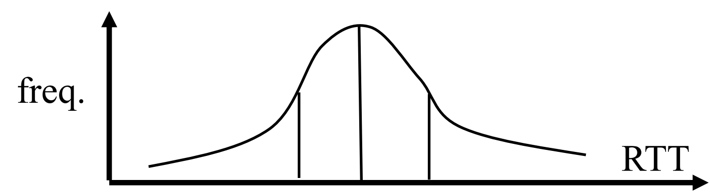
Q: Is it possible to set Timeout as a constant?

Q: Any problem w/ the early approach: Timeout = 2 RTT

Setting Timeout

Problem:

- Ideally, we set timeout = RTT,
  but RTT is not a fixed value
  =>
  using the average of **RTT** will generate
  many timeouts due to network variations
- Possibility: using the average/median of RTT
- Issue: this will generate many timeouts due to network variations



Solution:

- **Set Timeout RTO = avg +** "safety margin" based on variation

TCP approach:

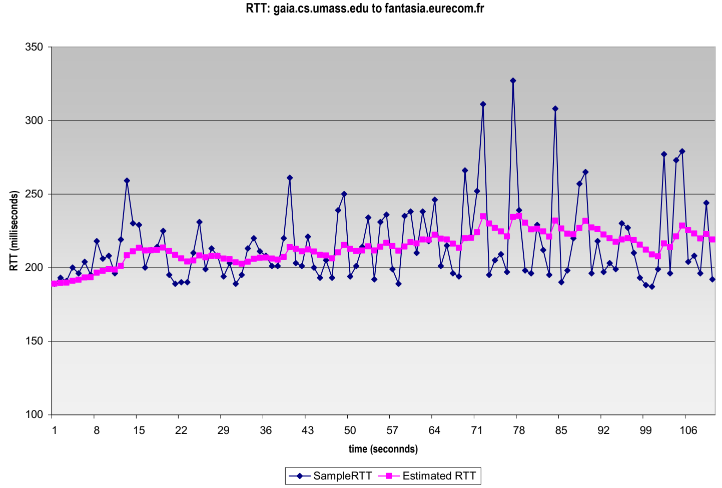
Timeout = EstRTT + 4 * DevRTT

Compute EstRTT and DevRTT

- Exponential weighted moving average (EWMA)
  - influence of past sample decreases exponentially fast

```
EstRTT = (1-alpha)*EstRTT + alpha*SampleRTT
```

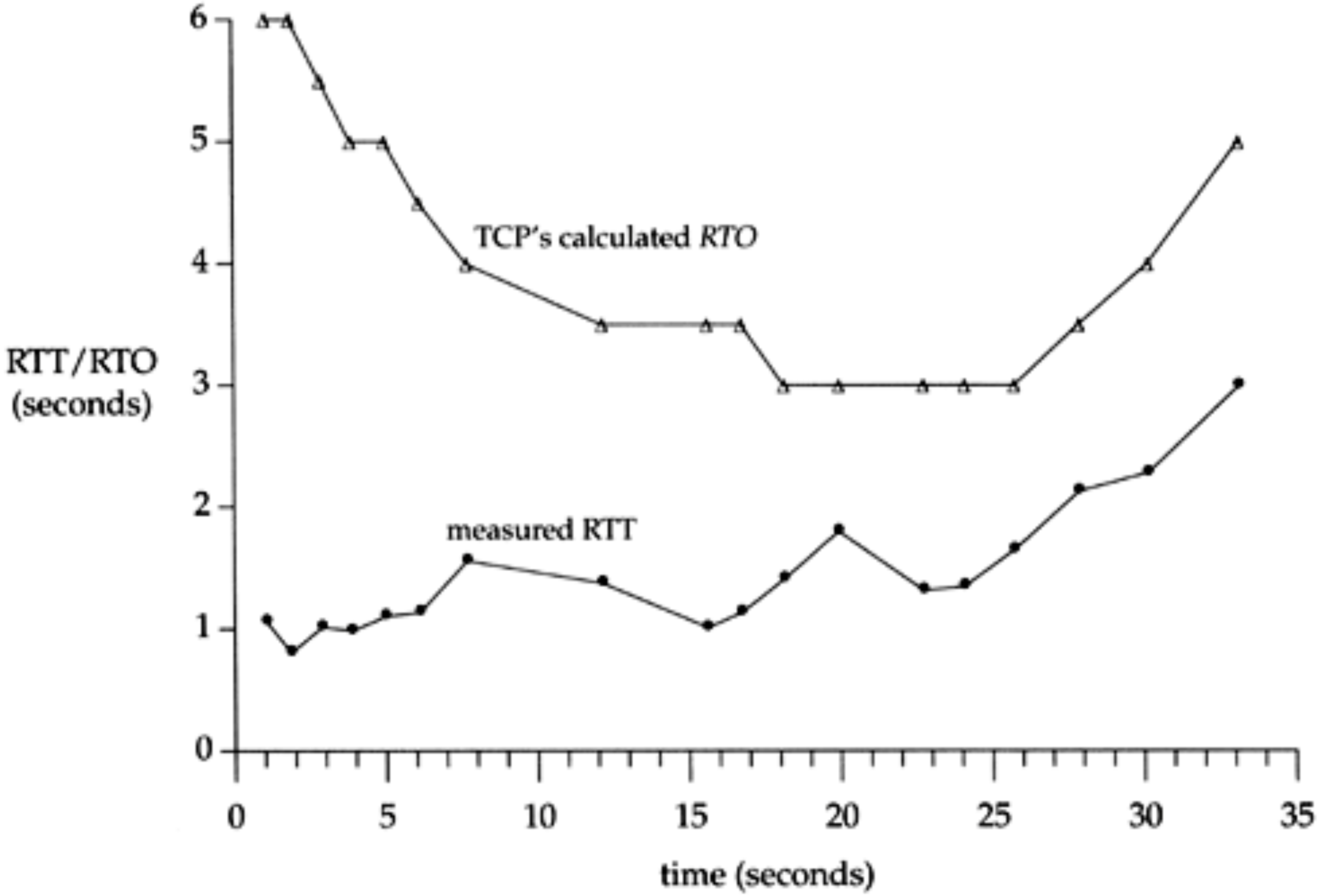
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
- typical value: **`alpha`** = 0.125




```
DevRTT = (1-beta)*DevRTT + beta|SampleRTT-EstRTT|

 (typically, beta = 0.25)
```

An Example TCP Session



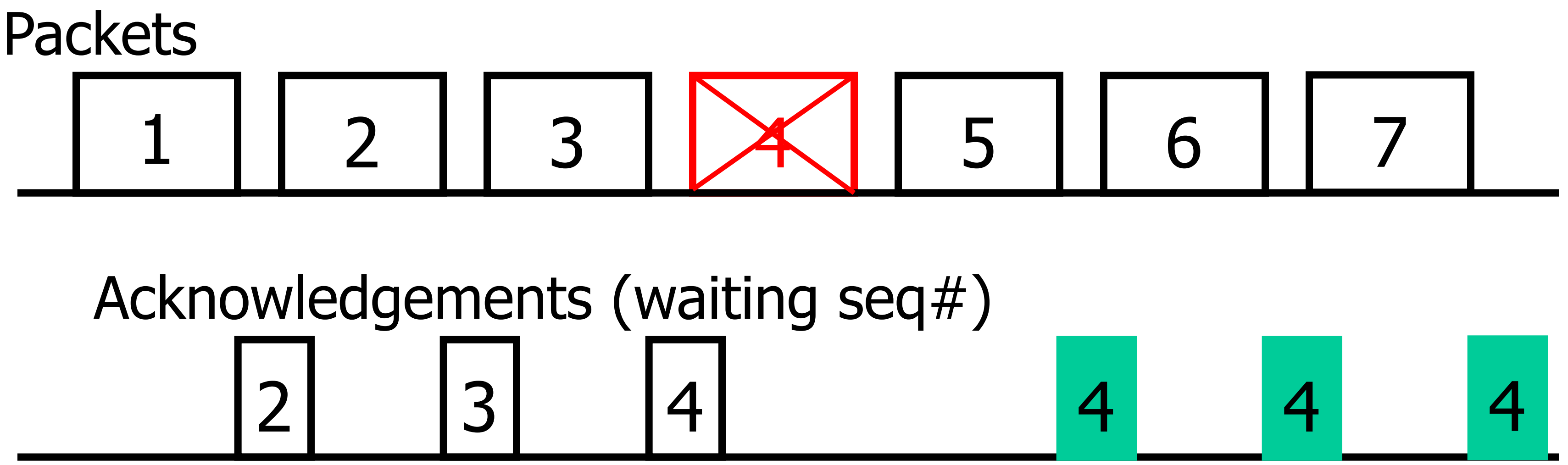
Fast Retransmit

- Issue: Timeout period often relatively long:
  - long delay before resending lost packet
- Question: Can we detect loss faster than RTT?

- Detect lost segments via duplicate ACKs
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - resend segment before timer expires

Triple Duplicate Ack



Fast Retransmit:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              …
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
              …
            }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
                …
```

a duplicate ACK for already ACKed segment

fast retransmit

TCP: reliable data transfer

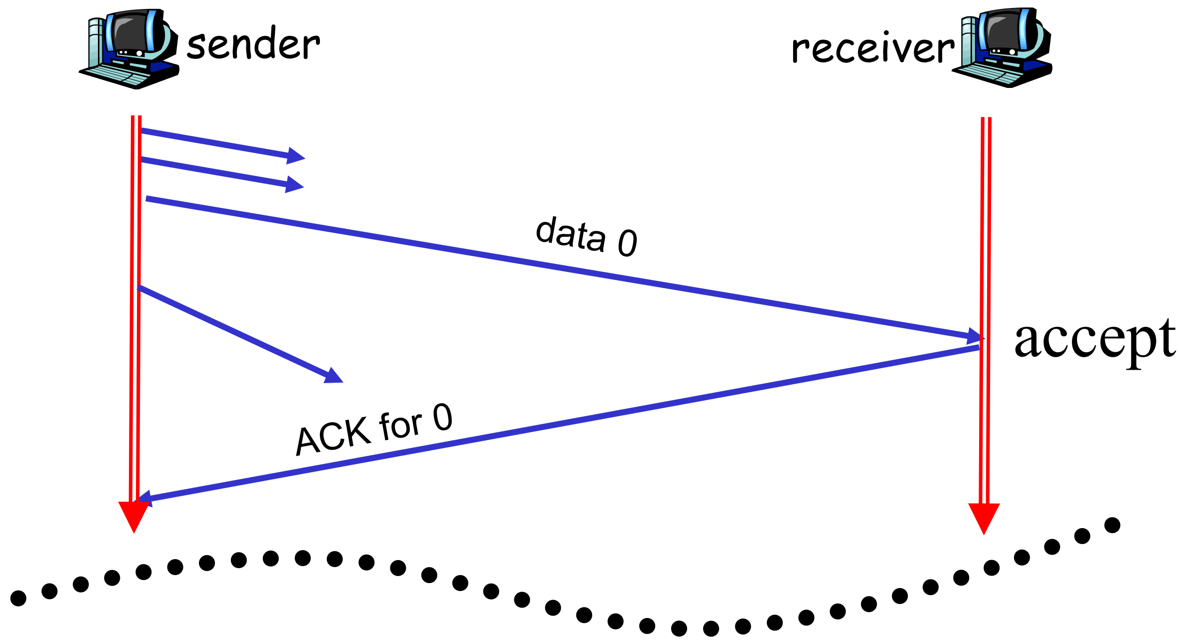
Simplified TCP sender

```
sendbase = initial_sequence number agreed by TWH
nextseqnum = initial_sequence number by TWH
loop (forever) {
  switch(event)
  event: data received from application above
        if (window allows send)
          create TCP segment with sequence number nextseqnum
          if (no timer) start timer
          pass segment to IP
          nextseqnum = nextseqnum + length(data)
        else put packet in buffer
  event: timer timeout for sendbase
      retransmit segment
      compute new timeout interval
      restart timer
  event: ACK received, with ACK field value of y
      if (y > sendbase) { /* cumulative ACK of all data up to y */
         cancel the timer for sendbase
         sendbase = y
         if (no timer and packet pending) start timer for new sendbase
         while (there are segments and window allow)
            sent a segment;
      }
      else { /* y==sendbase, duplicate ACK for already ACKed segment */
         increment number of duplicate ACKs received for y
         if (number of duplicate ACKS received for y == 3) {
            /* TCP fast retransmit */
            resend segment with sequence number y
            restart timer for segment y
         }
  } /* end of loop forever */
```

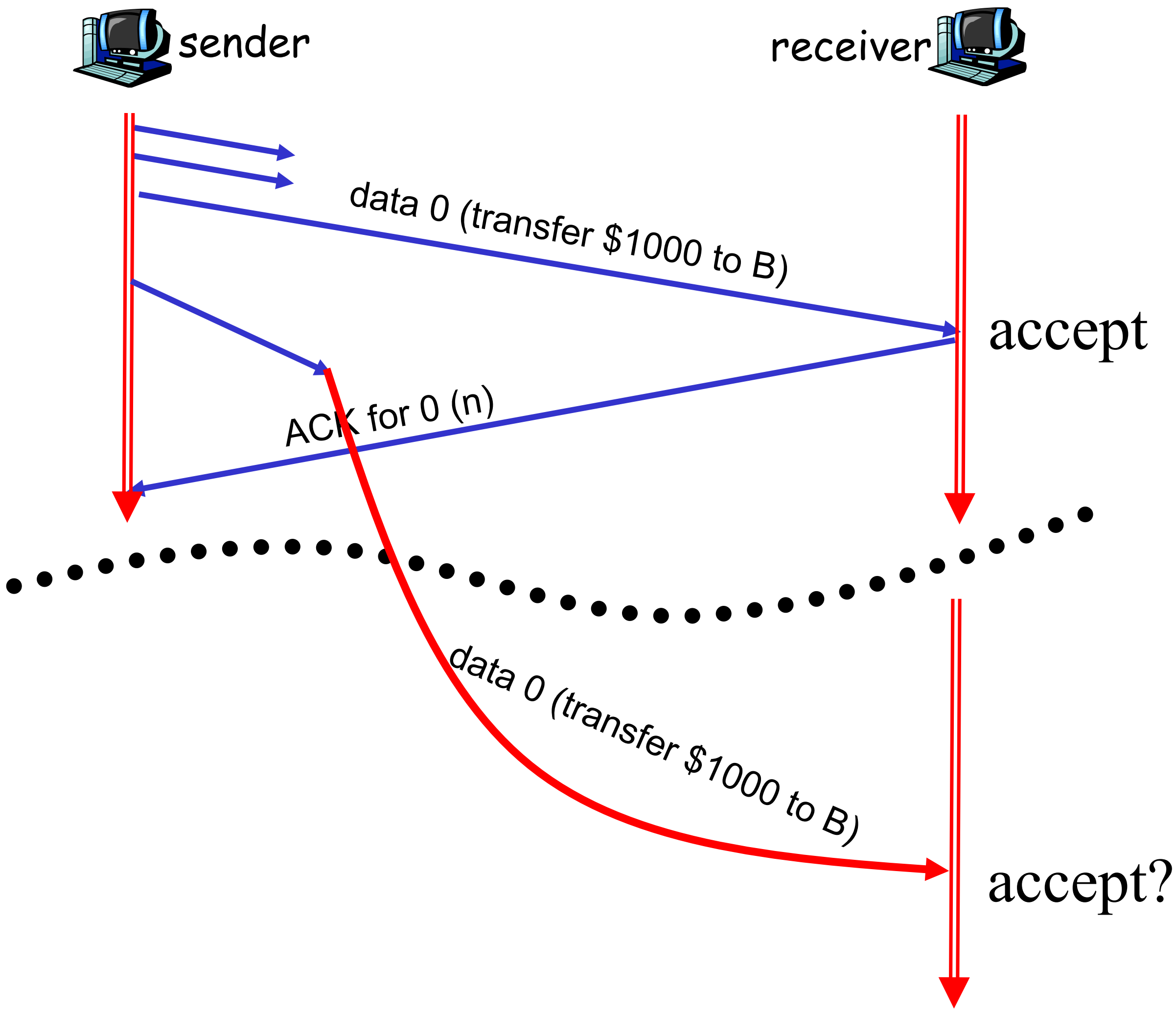
Outline

- Admin and Recap
- Reliable data transfer
  - perfect channel
  - channel with bit errors
  - channel with bit errors and losses
  - sliding window: reliability with throughput
- TCP reliability
  - data seq#, ack, buffering
  - timeout realization
  - *connection management*

Why Connection Setup/When to Accept (Safely Deliver) First Packet?



Why Connection Setup/When to Accept (Safely Deliver) First Packet?



Transport "Safe-Setup" Principle

- A general safety principle for a receiver R to accept a message from a sender S is the general "authentication" principle, which consists of two conditions:

Transport authentication principle:
- [p1] Receiver can be sure that what Sender says is **fresh**
- [p2] Receiver receives something that ***only*** Sender can say

We first assume a secure setting: no malicious attacks.

Exercise: Techniques to allow a receiver to check for freshness (e.g., add a time stamp)?