

# Transmission Control Protocol

Section 5.2 in the textbook

## TCP

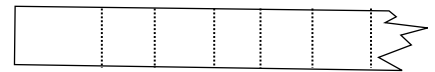
- TCP is arguably the second most important protocol in use today
- Does multiplexing just like UDP does, but it does more
  - Reliable transmission (auto-retransmission on lost or slow packets)
  - Reordering of packets
  - Fragmenting and assembling of packets

## TCP

- If UDP is a message-oriented protocol, then TCP is a byte-oriented protocol
- You might send data in sizes 4, 10 and 6
- The receiver might receive that same data in sizes 15, 2 and 3
- Every send()/write() does not need a matching recv()/read() and vice versa!

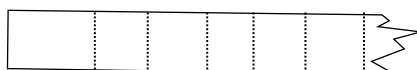


Hello there today is a very fin...



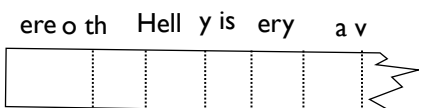
Hello there today is a very fin...

Hell o th ere toda y is a v ery

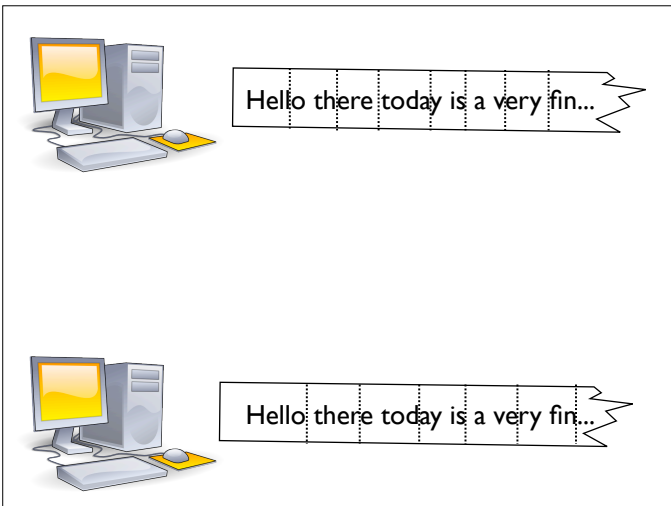
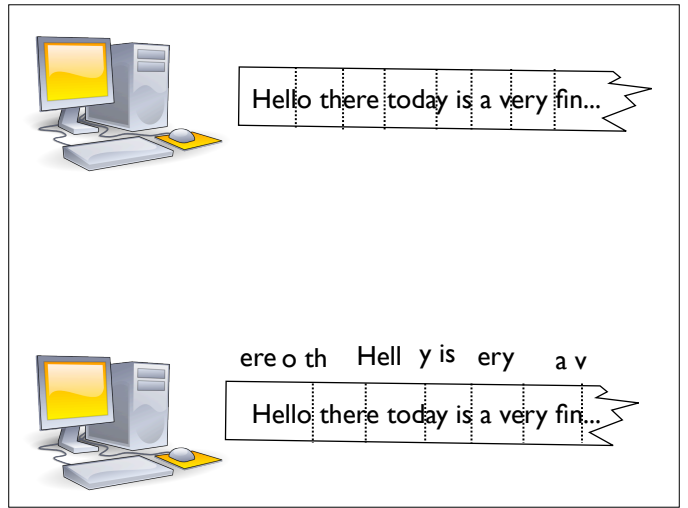
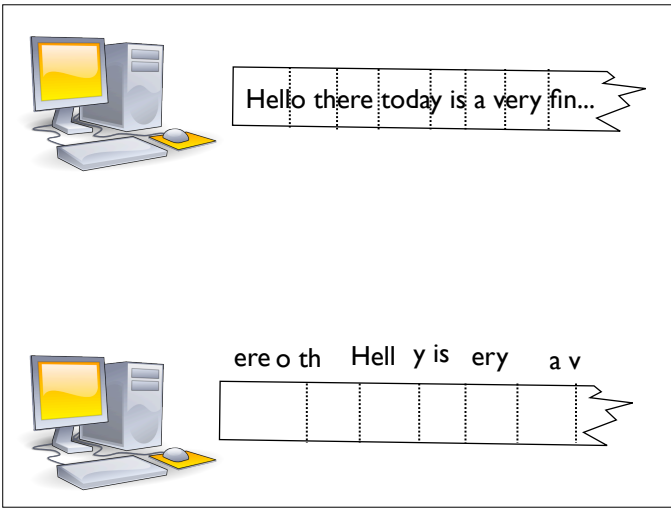


Hello there today is a very fin...

toda



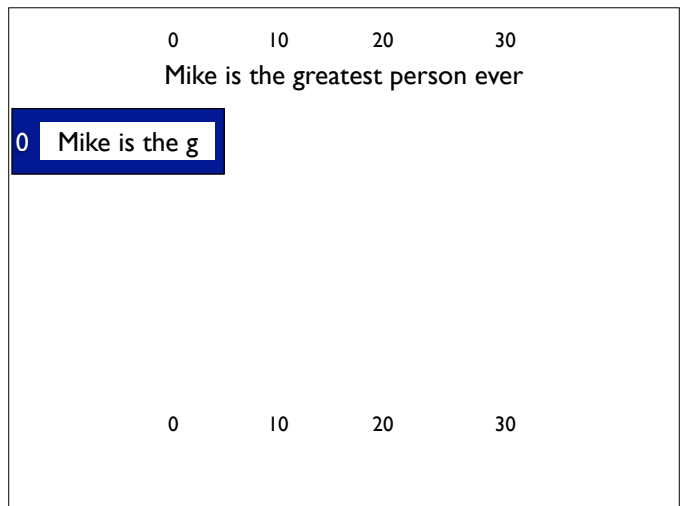
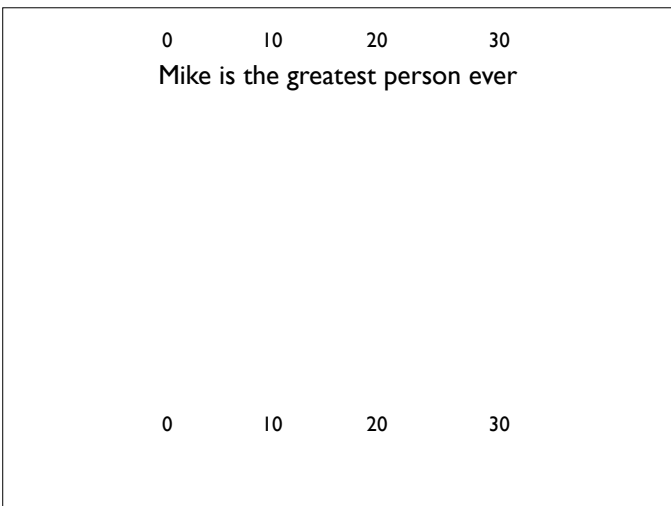
ere o th Hell y is ery a v

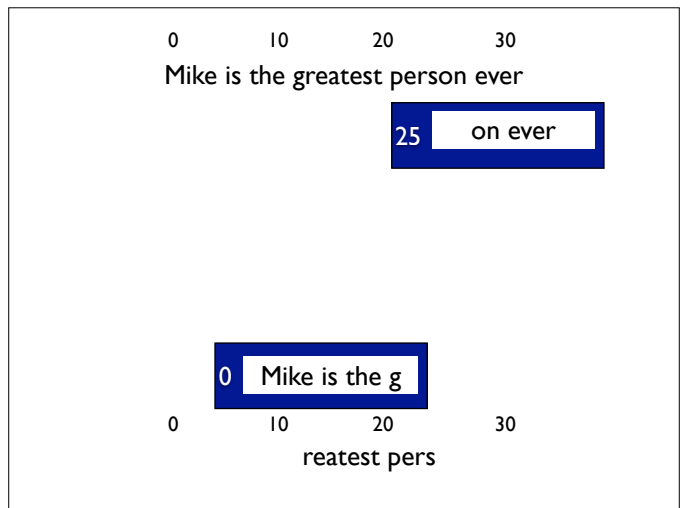
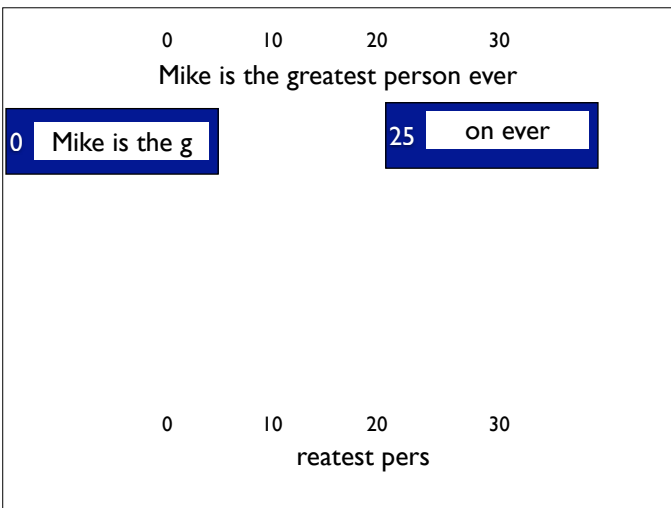
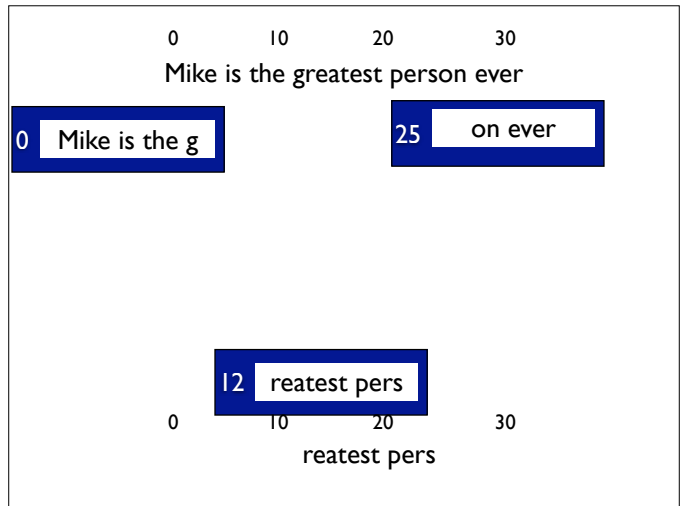
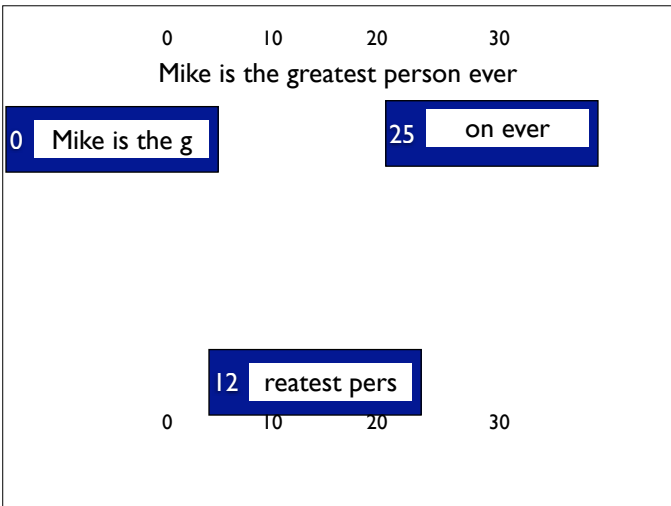
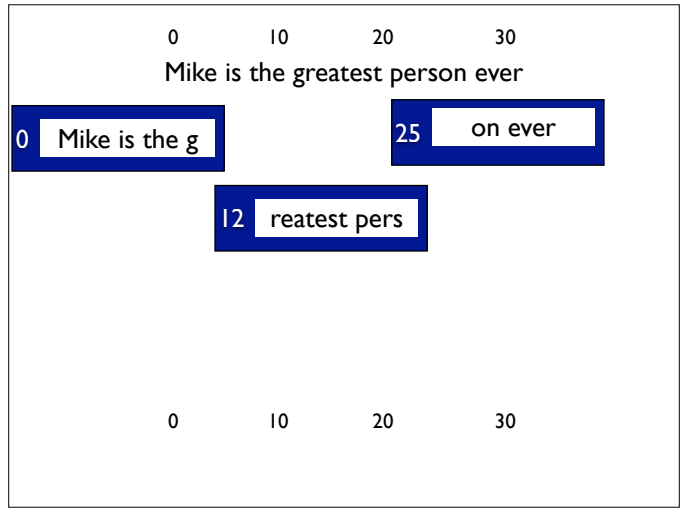
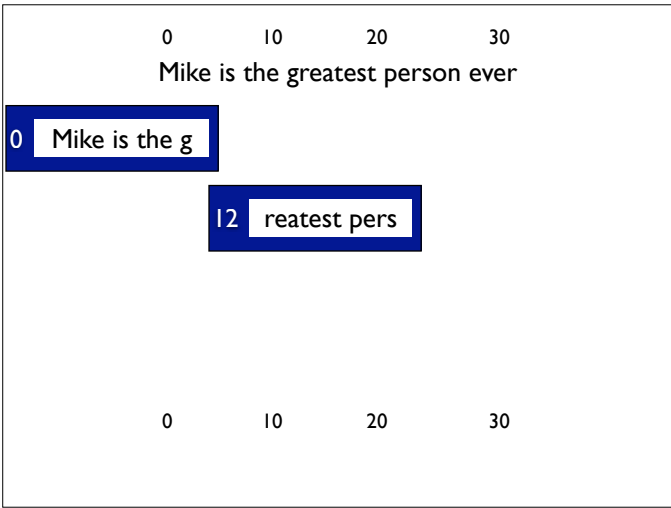


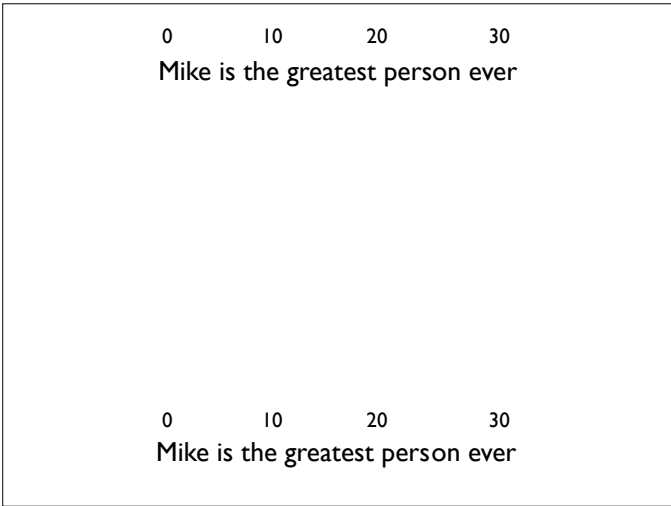
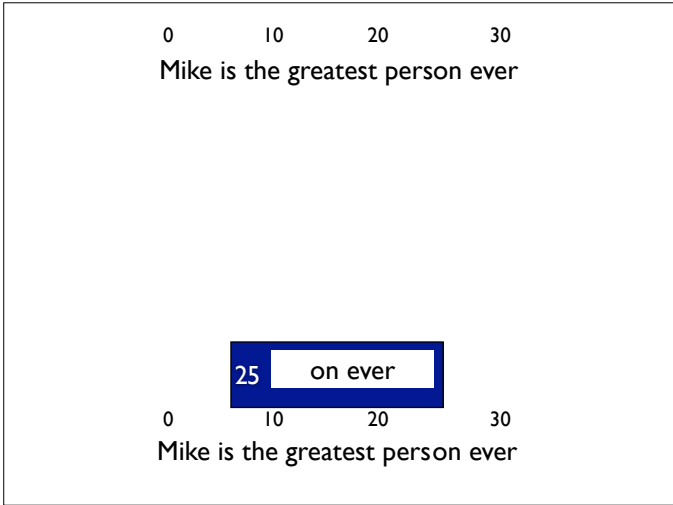
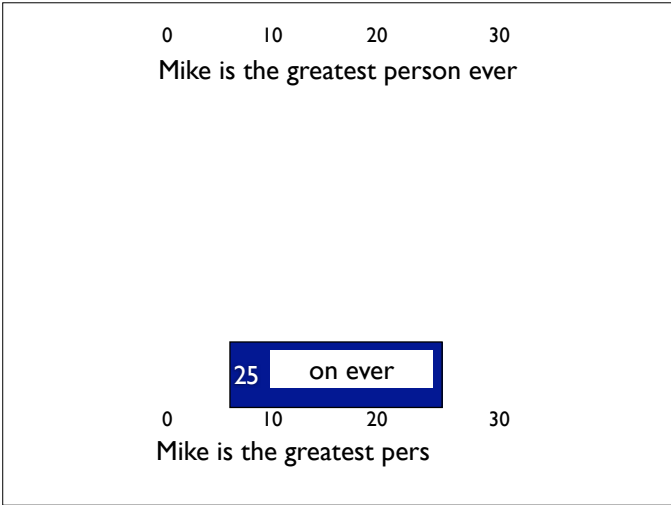
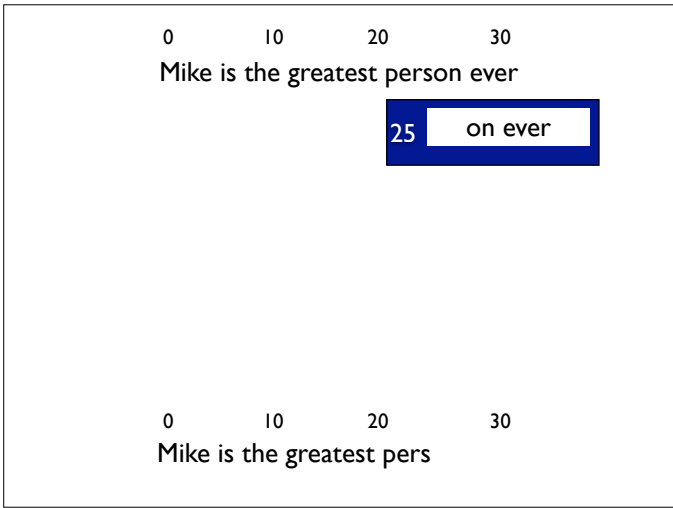
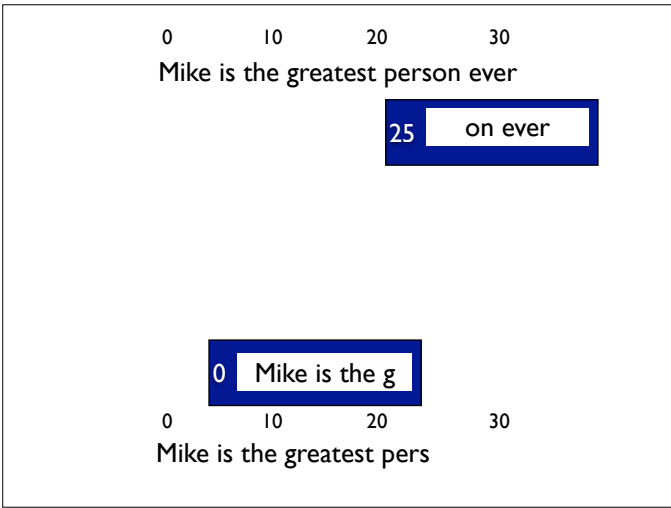
## Sliding windows

- Every byte is uniquely\* sequenced
- Whenever we send a packet, we label which part of the byte stream we're in
- The receiving end keeps a buffer of bytes it's received so far (even if it's received them out of order)

\* Sequencing not actually unique







0 10 20 30  
Mike is the greatest person ever

- When an application does a read(), what happens?
- If there is some data ready on the left of our buffer (we have something at sequence "0"), return as much as we have
- Otherwise (even if we have some out-of-order data saved), we block until we get some data at the left of our buffer
- What if we request more data than is ready?

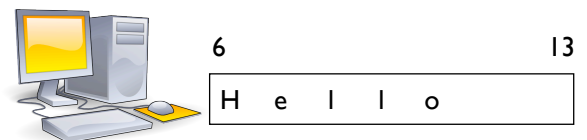
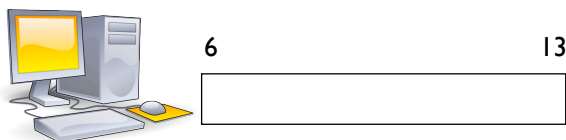
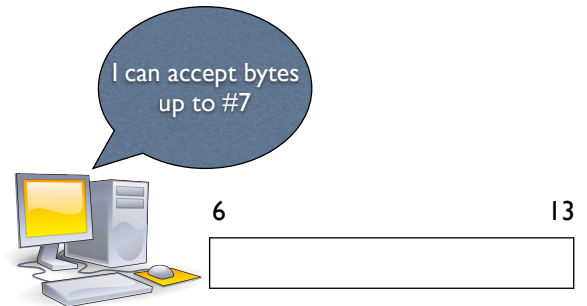
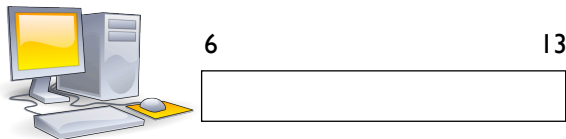
0 10 20 30  
Mike is the greatest person ever

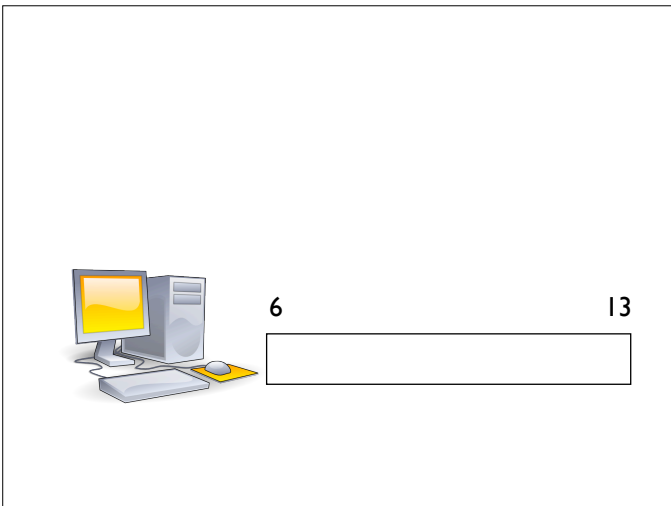
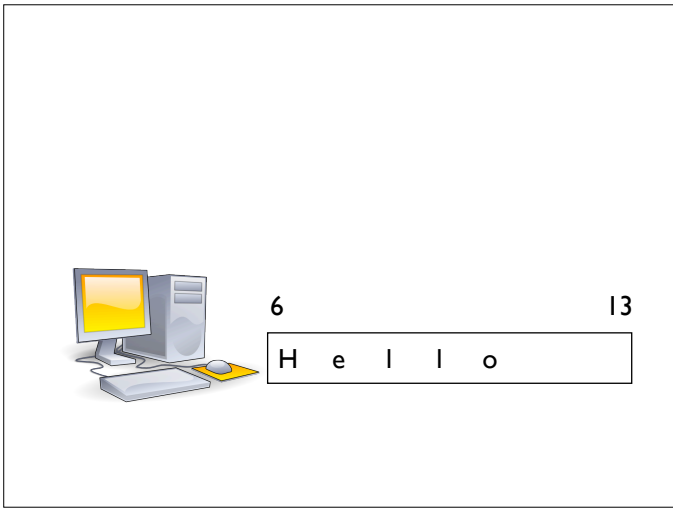
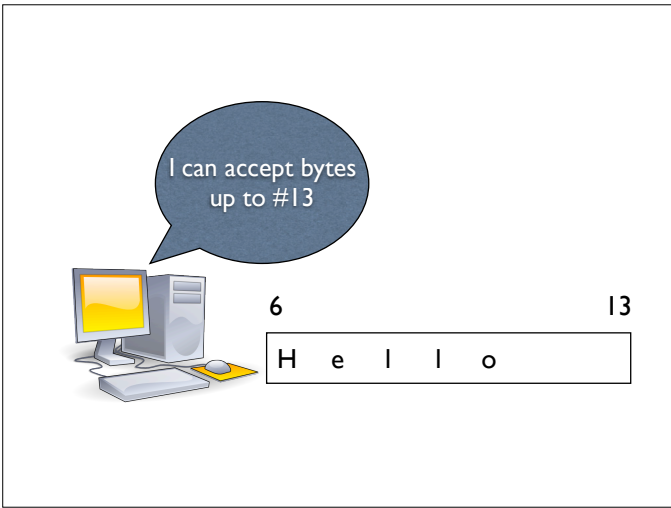
## Sliding windows

- Our range of sequence numbers is not infinite (in TCP it's 32 bits)
- Once we send 4GB over a connection, our sequence numbers wrap around!
- What if we receive every byte except the first one?
- Do we have to maintain gigabytes of buffer space just in case the first packet gets delayed?

## Sliding windows

- TCP is necessarily a symmetric, full-duplex protocol
- It would be impossible to do the things TCP does without the receiving end sending things back
- One of the things the receiving end sends back is a window



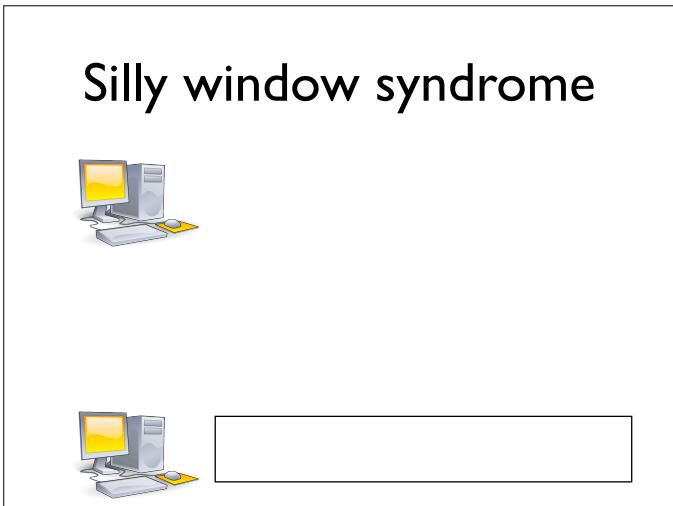


## Sliding window notes

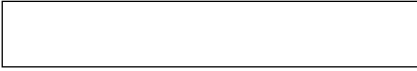
- It got annoying trying to animate that all after a while, so some other points:
  - The receiver's buffer is typically a circular buffer
    - Application code might read out some of what's in the buffer, but not all of it
  - The sender also maintains a buffer of stuff ready to send
    - The sender doesn't send unless the receiver's window can handle it

## Sliding window niceties

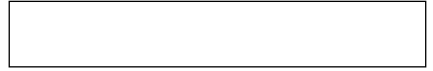
- Different bytes might have the same sequence number (after things wrap around)
  - Not a problem! Only one will be "in the window"
- One party's on a fast connection and one party's on a slow connection?
  - Not a problem! The receiving end automatically dictates the maximum speed



# Silly window syndrome



# Silly window syndrome



# Silly window syndrome



# Silly window syndrome



# Silly window syndrome



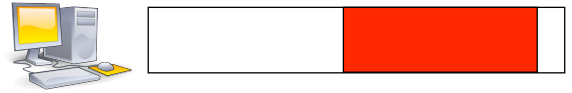
# Silly window syndrome



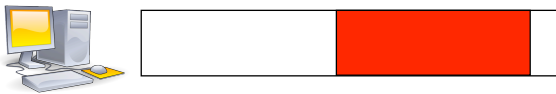
## Silly window syndrome



## Silly window syndrome



## Silly window syndrome



## Nagle's Algorithm

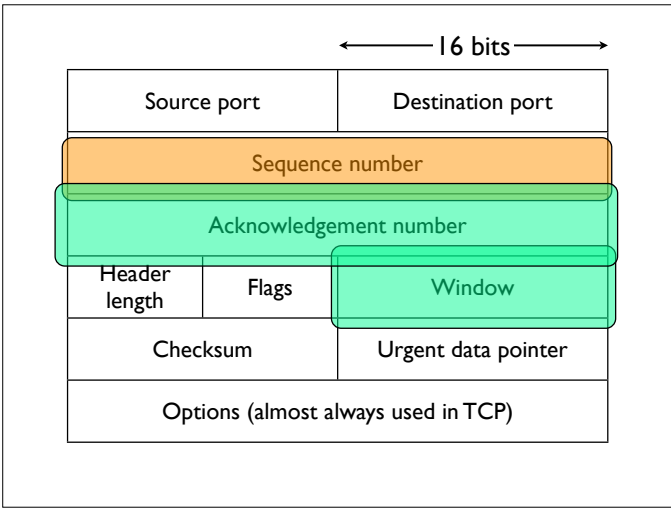
- Solution to Silly Window Syndrome problem
  - If a small window is open, do we ship enough to fit that window, or do we wait for the window to open further?
- Nagle's Algorithm: if there is an unACKed (more on this later) packet in flight, wait for the window to open further

← 16 bits →

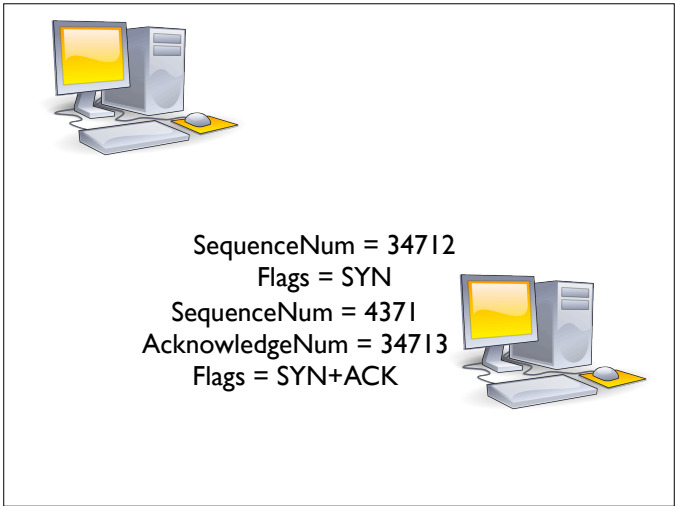
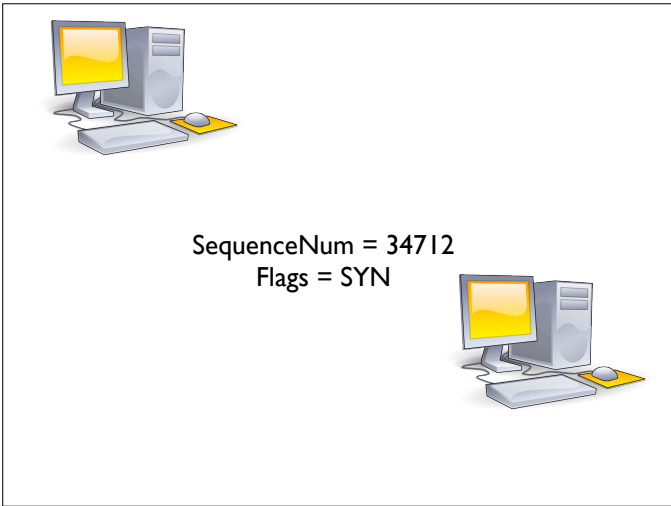
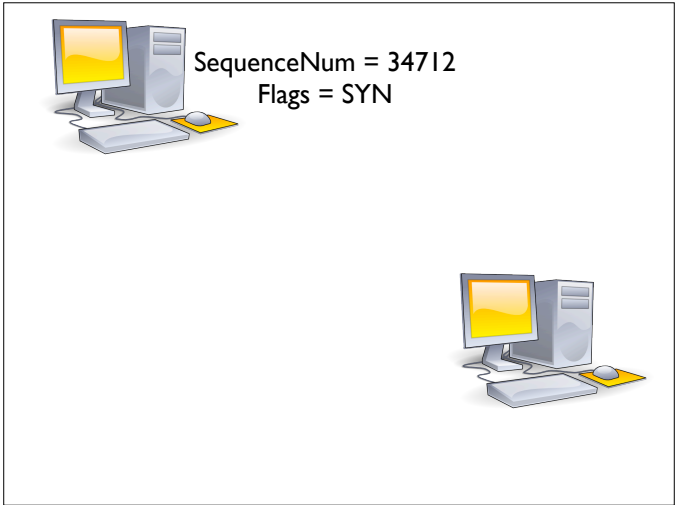
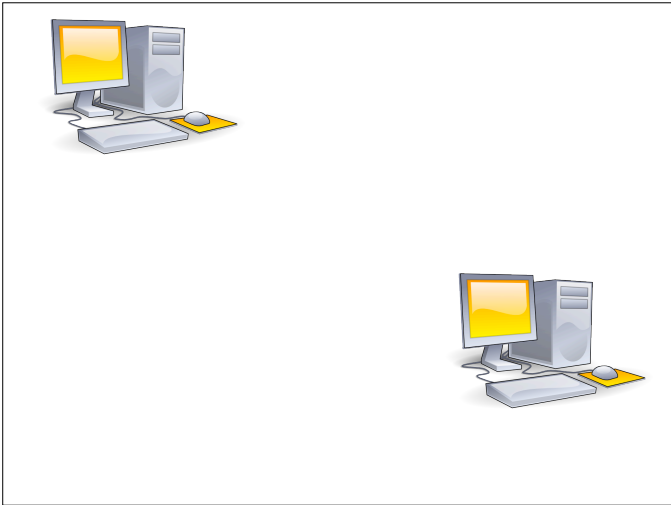
|                                     |       |                     |  |
|-------------------------------------|-------|---------------------|--|
| Source port                         |       | Destination port    |  |
| Sequence number                     |       |                     |  |
| Acknowledgement number              |       |                     |  |
| Header length                       | Flags | Window              |  |
| Checksum                            |       | Urgent data pointer |  |
| Options (almost always used in TCP) |       |                     |  |

← 16 bits →

|                                     |       |                     |  |
|-------------------------------------|-------|---------------------|--|
| Source port                         |       | Destination port    |  |
| Sequence number                     |       |                     |  |
| Acknowledgement number              |       |                     |  |
| Header length                       | Flags | Window              |  |
| Checksum                            |       | Urgent data pointer |  |
| Options (almost always used in TCP) |       |                     |  |



- ## TCP flags
- 6 bits
  - SYN (synchronize)
  - FIN (finalize)
  - RESET (duh)
  - PUSH
  - URG (urgent data is included)
  - ACK (acknowledge)
  - These can be bitwise OR'd together in almost any combination
  - SYN+FIN would not make any sense





SequenceNum = 4371  
 AcknowledgeNum = 34713  
 Flags = SYN+ACK



SequenceNum = 4371  
 AcknowledgeNum = 34713  
 Flags = SYN+ACK



SequenceNumber = 34713  
 AcknowledgeNum = 4373  
 Flags = ACK

SequenceNum = 4371  
 AcknowledgeNum = 34713  
 Flags = SYN+ACK



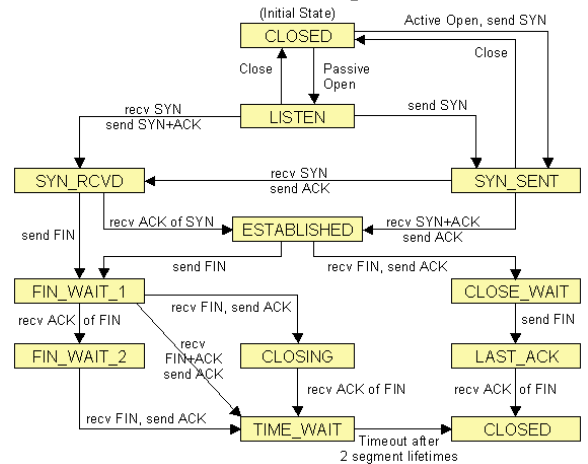
SequenceNumber = 34713  
 AcknowledgeNum = 4373  
 Flags = ACK



SequenceNumber = 34713  
 AcknowledgeNum = 4373  
 Flags = ACK



### TCP State Diagram



## Sliding windows revisited

- Every time the receiver receives data, it must send back an ACK
- The Acknowledgement Number field must indicate the next byte expected (i.e., the highest numbered contiguous byte + 1)
- The Advertized Window field must indicate the maximum numbered byte the sender is allowed to send

## Timeouts and retransmissions

- TCP assumes that if a packet is received, an ACK should be received within 120 seconds
- If the window for sequence numbers wraps around before then, we have a problem
- If a packet was slow/dropped in the past and we get a packet with its sequence number, is it the old packet being slow or the new packet being fast?

## 32-bit timestamp extension

- Most TCP implementations these days use an extension which puts a 32-bit timestamp on each packet
- This is handy for another purpose coming up, but it also solves the wrapping window problem
- We distinguish a packet by its sequence number and 32-bit timestamp

## Adaptive retransmission

- We don't want to have to wait 120 seconds before retransmitting a lost packet
- On most connections, if a packet's dropped, you'll know within milliseconds, not seconds

## Original algorithm

- Start off *EstimatedRTT* at 120 seconds
- Each time you receive an ACK, set *SampleRTT* to be the round-trip time for that packet
- $EstimatedRTT = \alpha EstimatedRTT + (1 - \alpha)SampleRTT$
- $\alpha$  is a "smoothing factor"
- Timeout is double *EstimatedRTT*

## Karn/Partridge Algorithm

- In 1987, a tweak was made
- To avoid inaccurate data, *SampleRTT* is only updated when we receive an ACK corresponding to a packet which has not been resent
- This was before the 32-bit timestamp extension (not really useful now)

## Jacobson/Karels algorithm

- J/K was introduced mainly to fight congestion (which we'll talk about later)
- As a side effect, it does a much better job than Karn/Partridge
- Idea: doubling *EstimatedRTT* to get a bound on the RTT is totally arbitrary. Can we do better?

## Jacobson/Karels algorithm

$$d = \text{SampleRTT} - \text{EstimatedRTT}$$
$$\text{EstimatedRTT} = \text{EstimatedRTT} + \alpha d$$
$$\text{Deviation} = \text{Deviation} + \alpha(|d| - \text{Deviation})$$
$$\text{TimeOut} = \mu \text{EstimatedRTT} + \phi \text{Deviation}$$

- $\alpha$ ,  $\mu$  and  $\phi$  are still made up
- Typically  $\mu=1$  and  $\phi=4$

## TCP wrap-up

- Connectionful! Explicit set-up and tear-down
- Reordering of out-of-order packets via a sliding window
- The sliding window also throttles a faster connection to the speed of a slower one!
- Automatic retransmission based on guessing a timeout

## Overhead up until now

- Link layer (Ethernet): 18 bytes per packet
- Network layer (IP): 20+ bytes per packet
- Transport layer (TCP): 20+ bytes (usually 24+) per packet
- Let's say 65 bytes of overhead per packet (usually about 1200 bytes)

## Overhead up until now

- $65/1200 = 5.5\%$  overhead
- $100\text{Mbit/s} = 5.5\text{Mbit/s}$  overhead +  $94.5\text{Mbit/s}$  payload
- Maximum throughput =  $11.8\text{MB/s}$

