Computer Networks and Applications

COMP 3331/COMP 9331

Week 4

Transport Layer Part 1

Reading Guide:
Chapter 3, Sections 3.1 – 3.5

Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport

Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

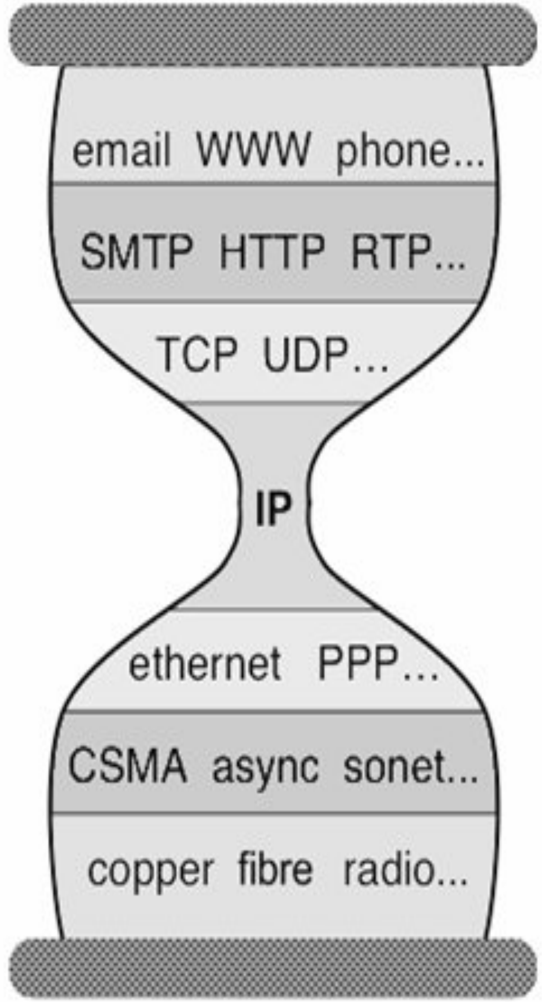
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Transport layer

- Moving "down" a layer
- Current perspective:
  - Application is the boss….
  - Usually executing within the OS Kernel
  - The network layer is ours to command !!



Network layer (context)

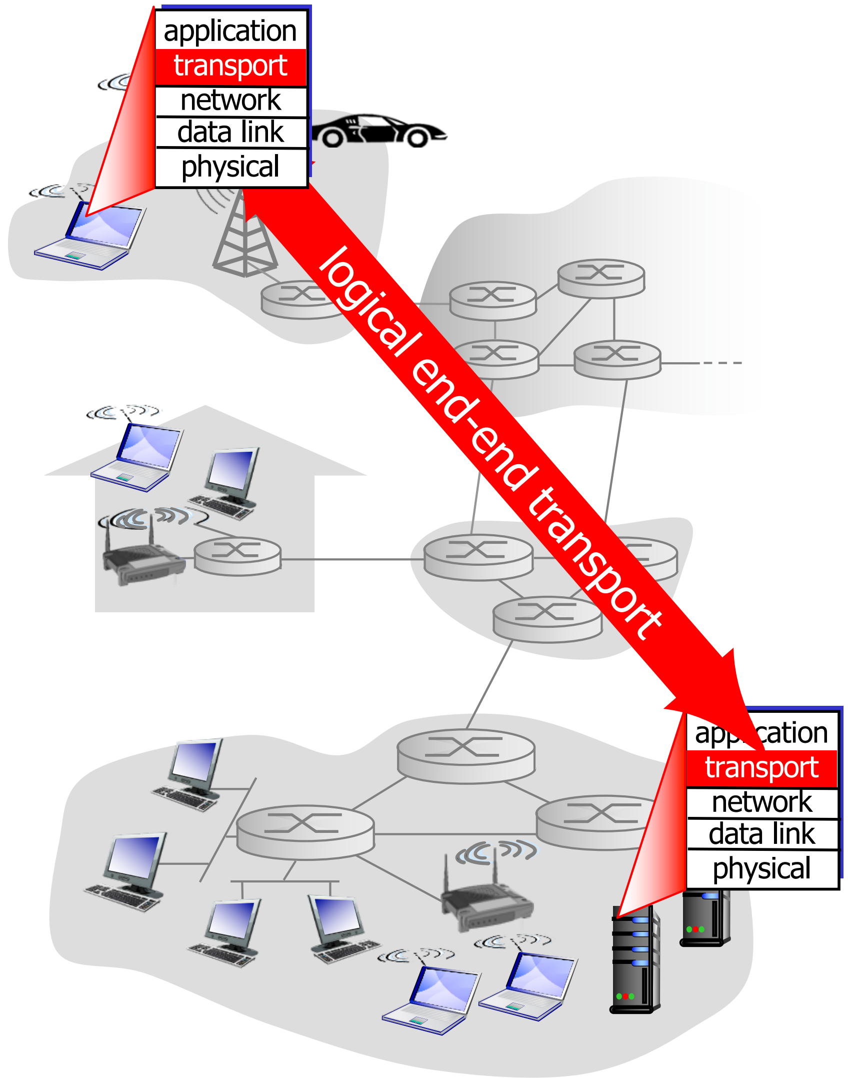
- What it does: finds paths through network
  - Routing from one end host to another

- What it doesn't:
  - Reliable transfer: "best effort delivery"
  - Guarantee paths
  - Arbitrate transfer rates

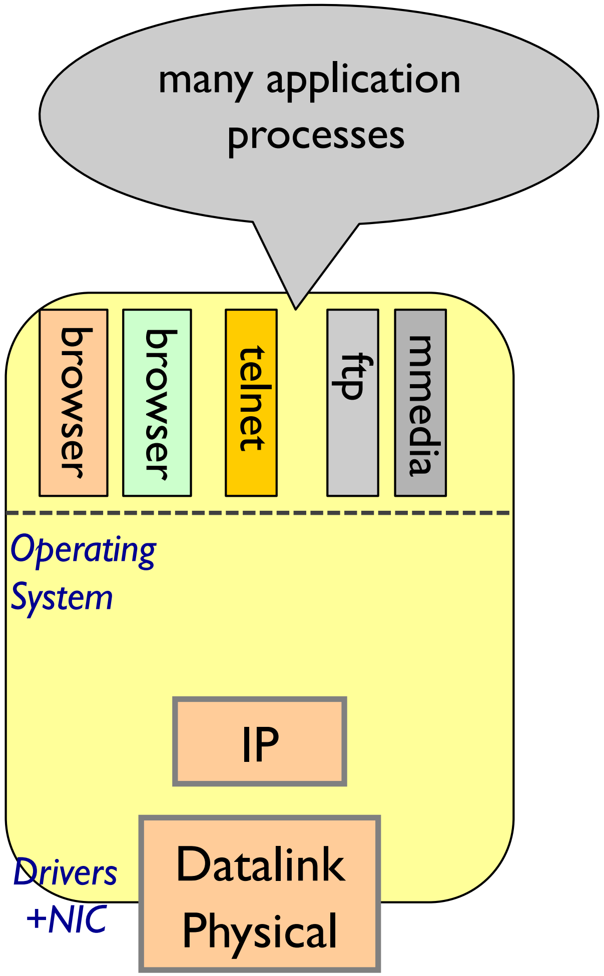
- For now, think of the network layer as giving us an "API" with one function: *sendtohost(data, host)*
  - Promise: the data will go to that (usually!!)

Transport services and protocols

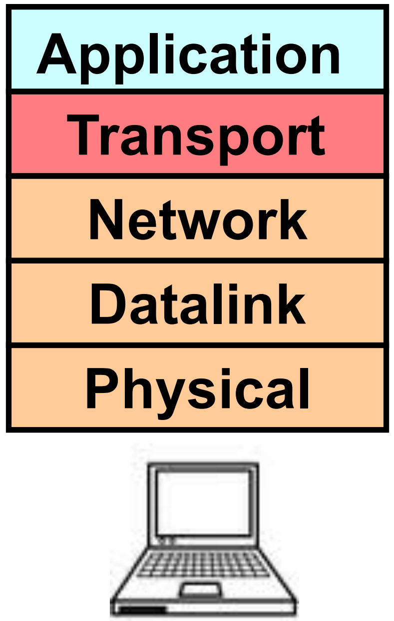
- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
  - Exports services to application that network layer does not provide



Why a transport layer?

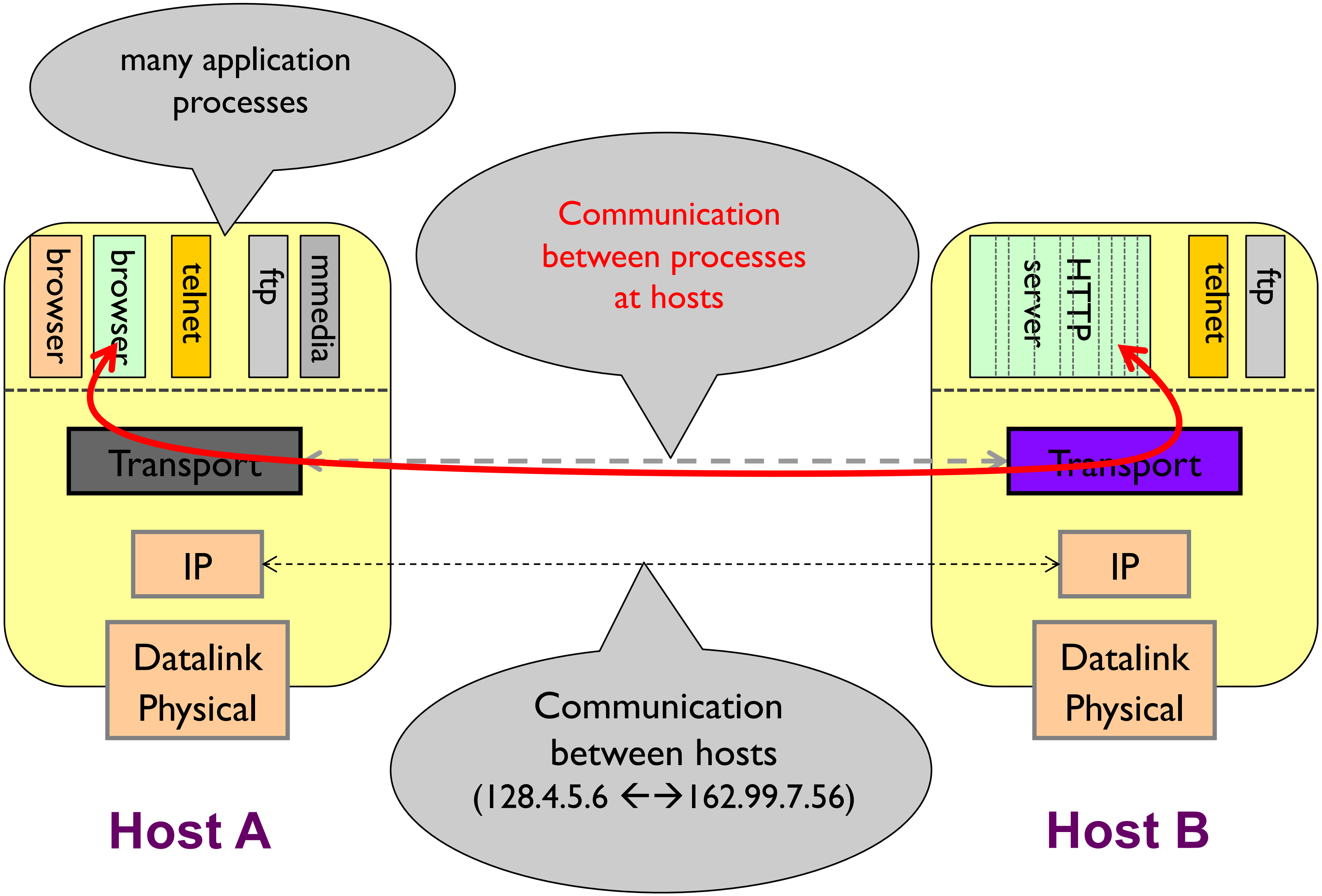


Host A



Host B

Why a transport layer?



Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

first service transport layer provide

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Each Socket has its own identifier, the format of the identifier determine it’s UDP socket or TCP socket

Multiplexing/demultiplexing

接收套接字，然后将报文段定向到该套接字。将运输层报文段中的数据交付到正确的套接字的工作称为**多路分解**（demultiplexing）。从源主机的不同套接字中收集数据块，并为每个数据块封装上首部信息（这将在多路分解时使用）从而生成报文段，然后将报文段传递到网络层的工作称为**多路复用**（multiplxing）。注意到图3-2的中间那台主机的运输层必须将从其下的

multiplexing at sender:

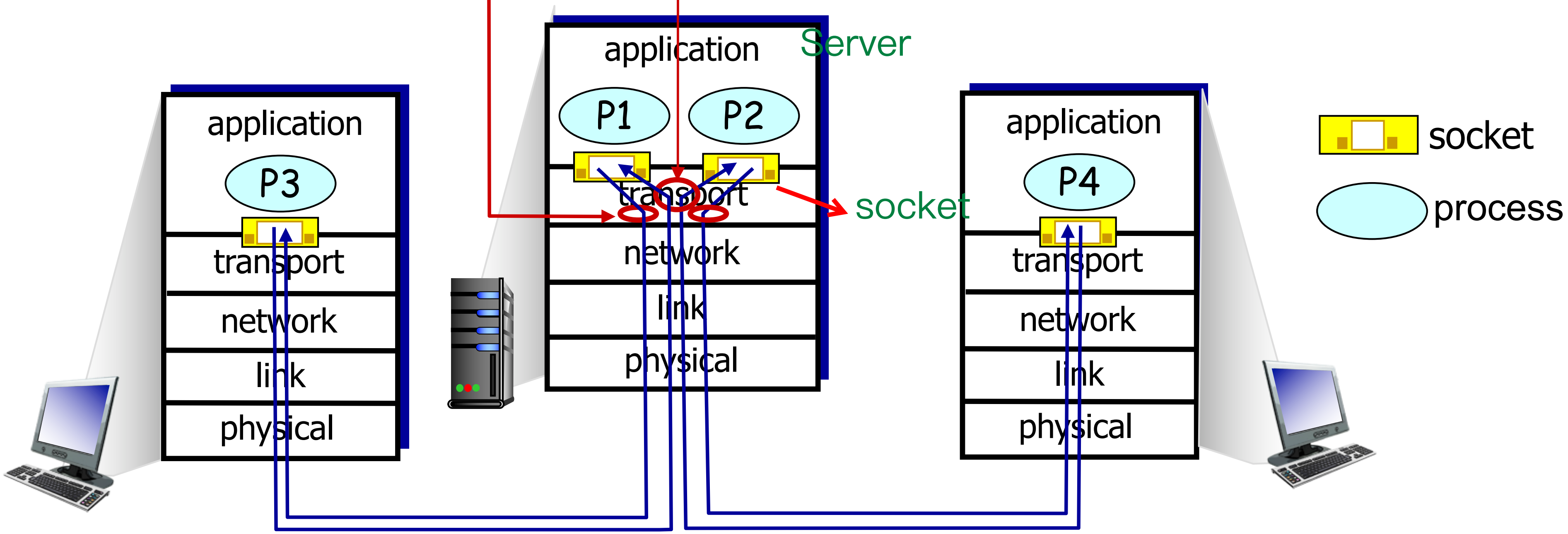
handle data from multiple sockets, add transport header (later used for demultiplexing)

Output management

demultiplexing at receiver:

use header info to deliver received segments to correct socket

Input management



Note: The network is a shared resource. It does not care about your applications, sockets, etc.

为了说明多路分解的工作过程，考虑前一节中家庭的例子。每个孩子通过其名字来标识。当Bill从邮递员处收到一批信件，并通过查看收信人名字而将信件亲手交付给他的兄弟姐妹们时，他执行的就是一个多路分解操作。当Ann从兄弟姐妹们那里收集信件并将它们交给邮递员

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

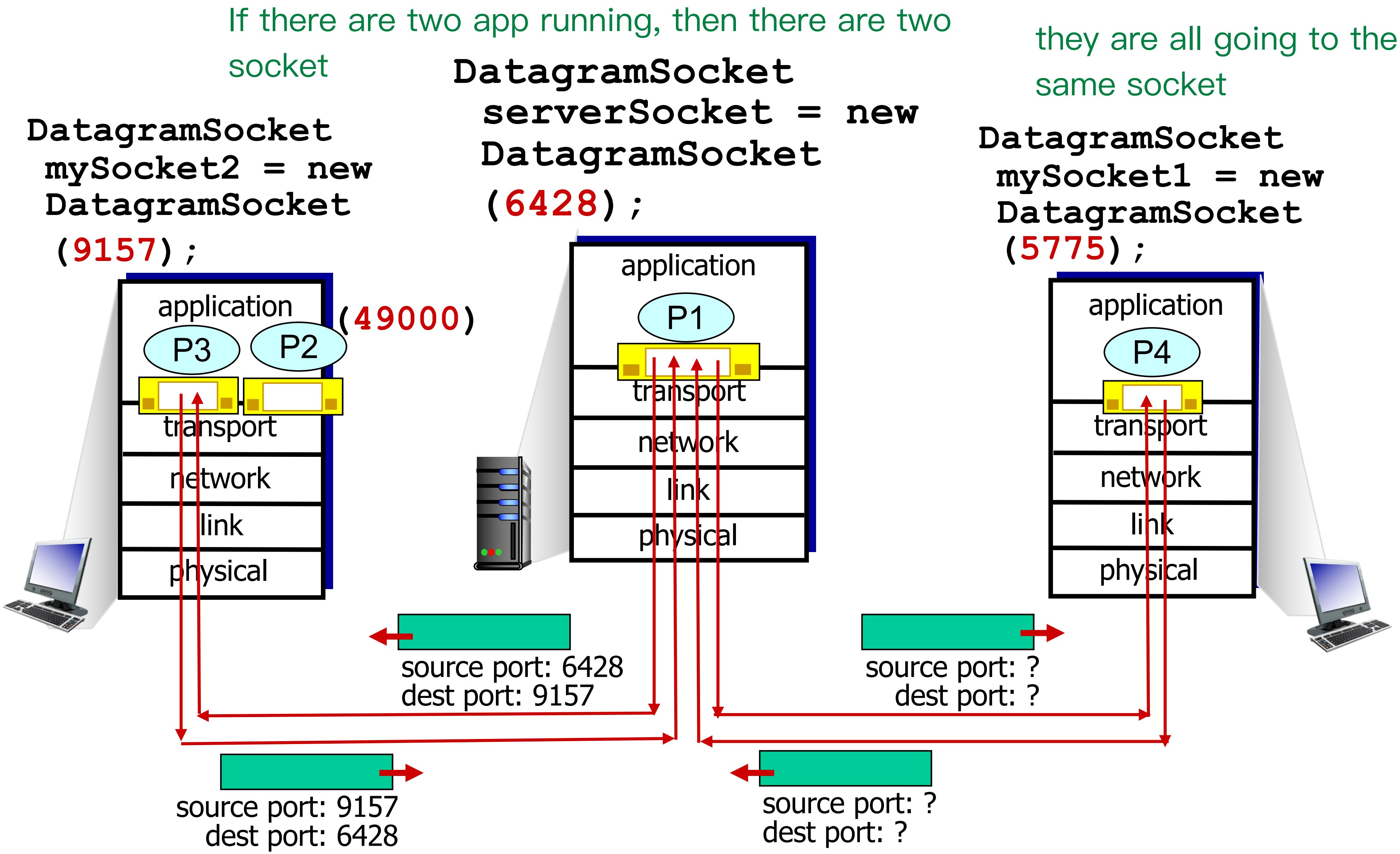
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example



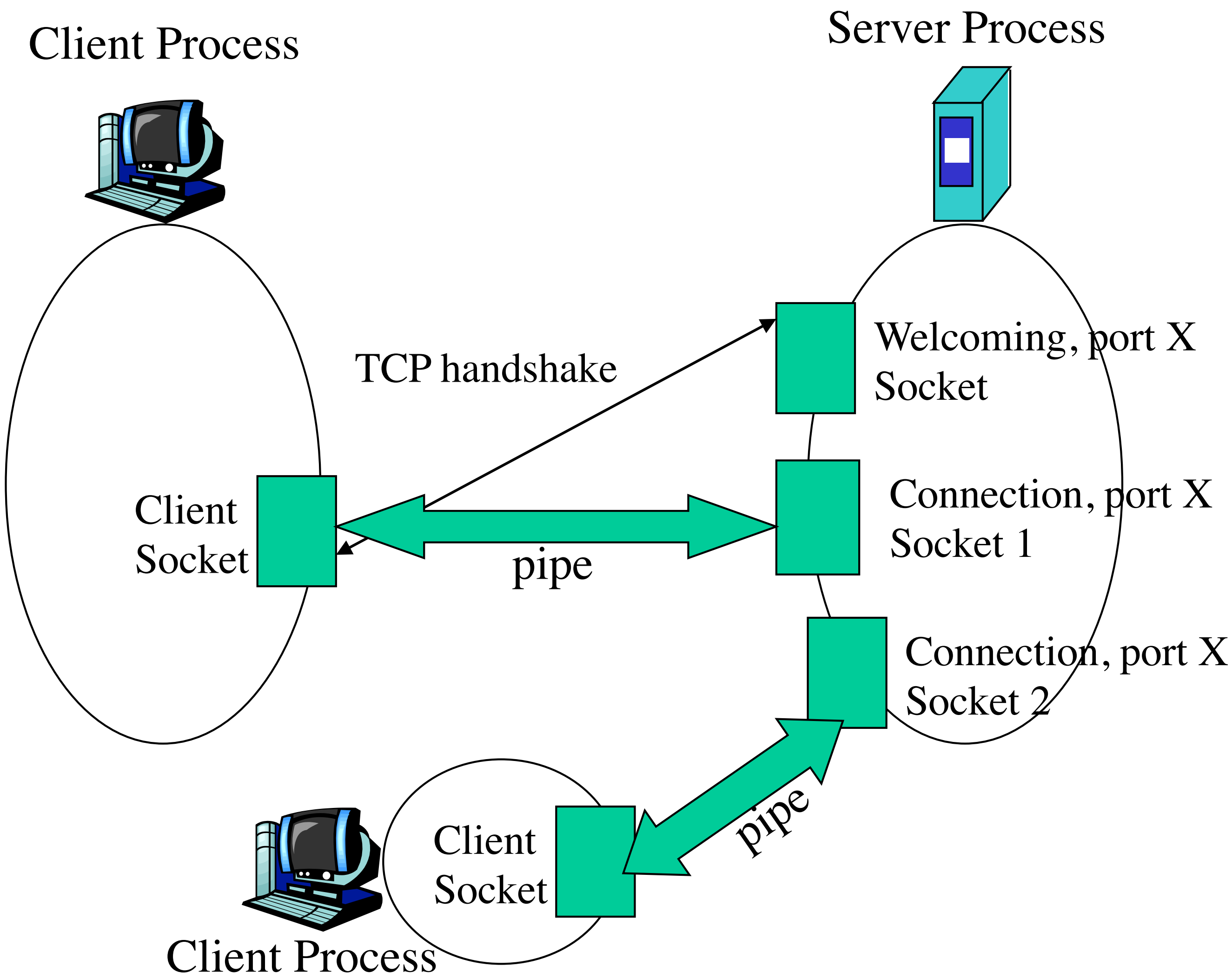
U have to maintain connection with all possible process in the world, to avoid clash, we just keep track with the 4–tuple

Connection-oriented demux

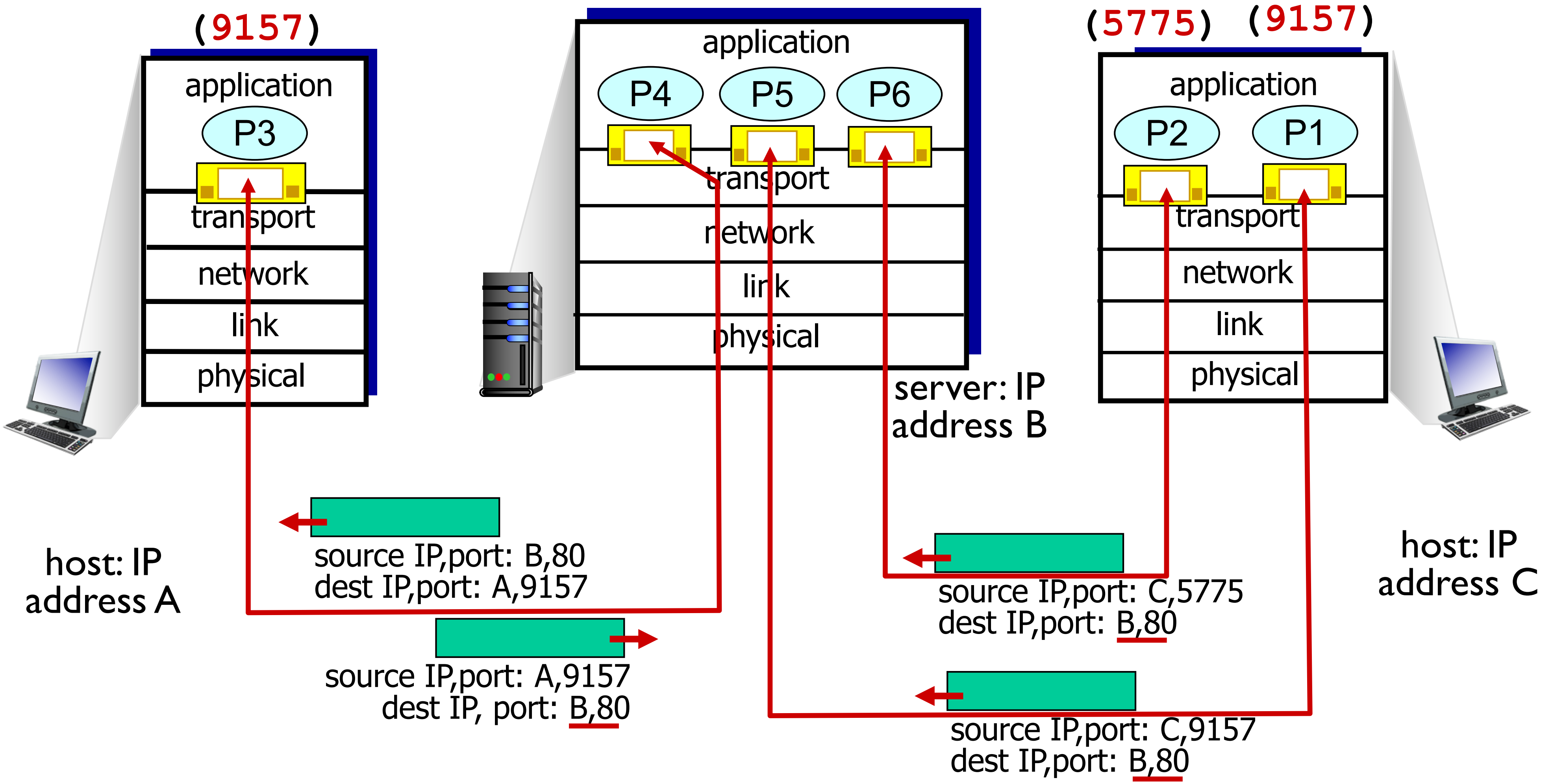
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Revisiting TCP Sockets



Connection-oriented demux: example



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

May I scan your ports?

http://netsecurity.about.com/cs/hackertools/a/aa121303.htm

- Servers wait at open ports for client requests
- Hackers often perform *port scans* to determine open, closed and unreachable ports on candidate victims
- Several ports are well-known
  - <1024 are reserved for well-known apps
  - Other apps also use known ports
    - MS SQL server uses port 1434 (udp)
    - Sun Network File System (NFS) 2049 (tcp/udp)
- Hackers can exploit known flaws with these known apps
  - Example: Slammer worm exploited buffer overflow flaw in the SQL server
- How do you scan ports?
  - Nmap, Superscan, etc

http://www.auditmypc.com/

https://www.grc.com/shieldsup

Quiz: UDP Sockets

- Suppose we use UDP instead of TCP for designing a web server where all requests and responses fit in a single UDP segment. Suppose 100 clients are simultaneously communicating with this web server. How many sockets are respectively at the server and each client?

a) 1, 1
b) 2, 1
c) 200, 2
d) 100, 1
e) 101, 1

A

Quiz: TCP Sockets

- Suppose 100 clients are simultaneously communicating with a traditional HTTP/TCP web server. How many sockets are respectively at the server and each client?

  a) 1, 1
  b) 2, 1
  c) 200, 2
  d) 100, 1
  e) 101, 1

E

TCP is connection-oriented, then there are 100 sockets.
Also, there is a welcome socket for hand-shake

Quiz: TCP Sockets

- Suppose 100 clients are simultaneously communicating with a traditional HTTP/TCP web server. Do all of the sockets at the server have the same server-side port number?

a) Yes

b) No

A

Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 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 app

- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

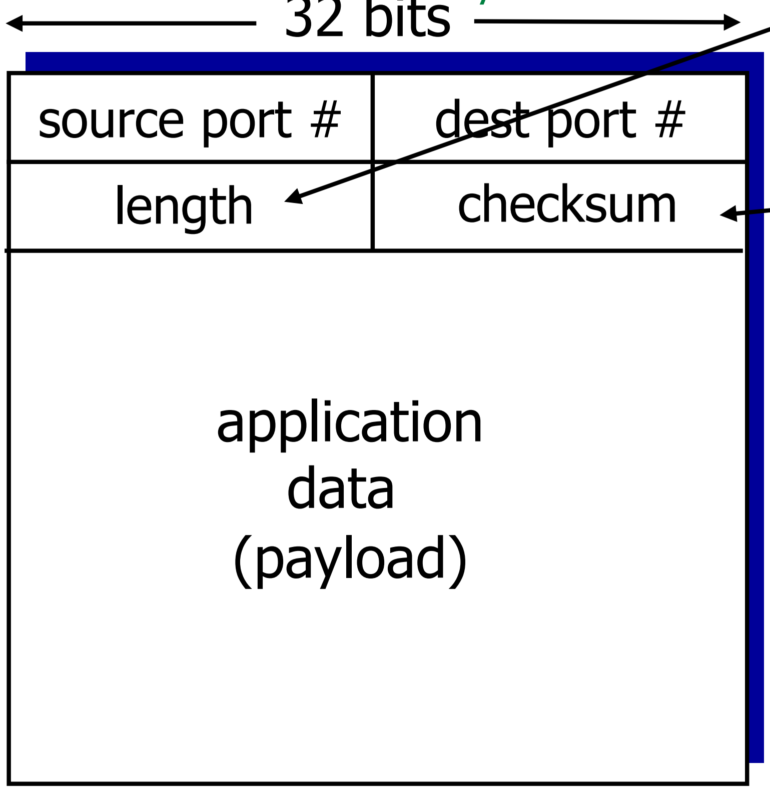
UDP: segment header

= 4 bytes

length, in bytes of UDP segment, including header

(Check if the segment is changed due to outer diturb)

2 bytes Optional Checksum



UDP segment format

8 bytes(2 bytes for each)

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP checksum

Check if there is big problem

- *Goal:* detect "errors" (e.g., flipped bits) in transmitted segment
  - Router memory errors
  - Driver bugs
  - Electromagnetic interference

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- Add all the received bits together as 16-bit integers
- Add that to the checksum
- If the result is not 1111 1111 1111 1111, there are errors !

Internet checksum: example

the header is also included, that is from the udp packet

example: add two 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
wraparound	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

1's complement

Checksum is 1's complement of *sum*

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result (wraparound)

It's efficient to check all possible error

Once we get a checksum, there will an algorithm to convert it into all 1, if not then it is wrong

UDP Applications

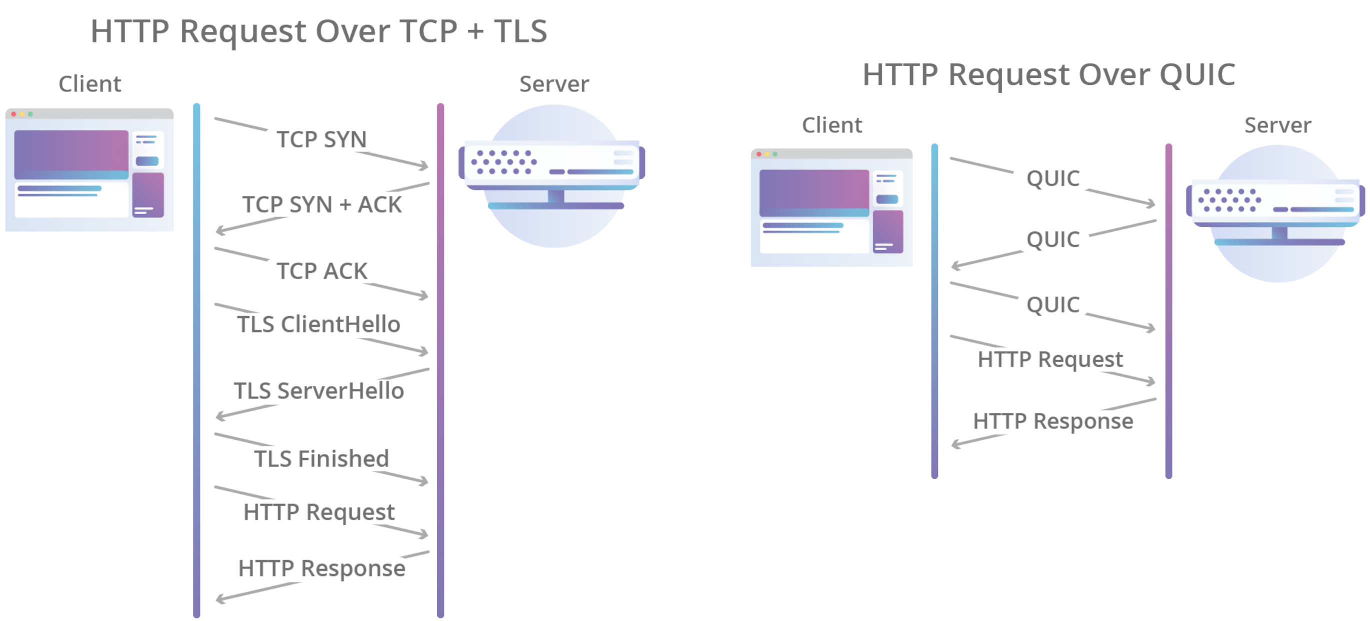
- Latency sensitive/time critical
  - Quick request/response (DNS, DHCP)
  - Network management (SNMP)
  - Routing updates (RIP)
  - Voice/video chat
  - Gaming (especially FPS)

- Error correction unnecessary (periodic messages)

QUIC: Quick UDP Internet Connections
A Google Experiment

Outside the scope of exams

- Core idea: HTTP/2 over UDP
  - Faster connection establishment
  - Overcomes HoL blocking due to lost packets
  - Improved congestion control
  - Forward error correction
  - Connection migration



https://www.chromium.org/quic

Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

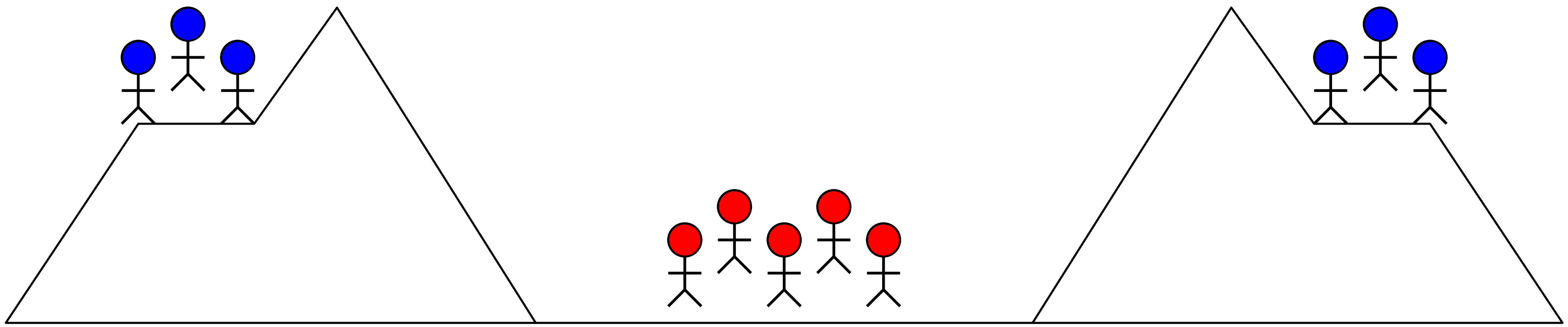
3.7 TCP congestion control

Reliable Transport

- In a perfect world, reliable transport is easy
- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost (*why?*) Queueing is heavy, and it's dropped
  - a packet is delayed (*why?*) Queueing
  - packets are reordered (*why?*) the packet is being send independently
  - a packet is duplicated (*why?*)

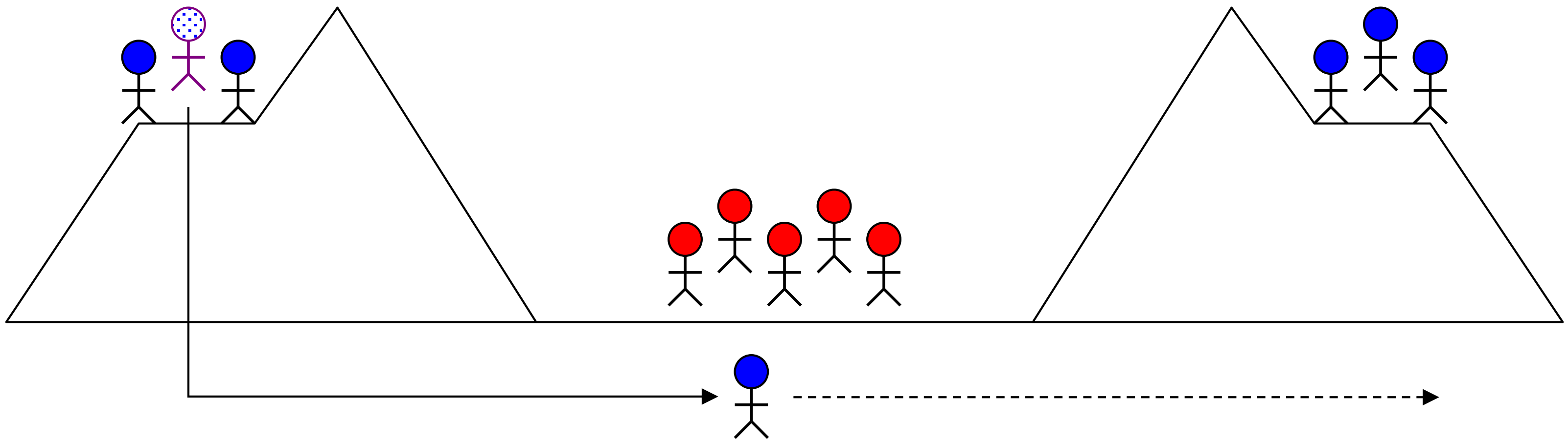
Didn't receive the response, and the packet is resent

The Two Generals Problem



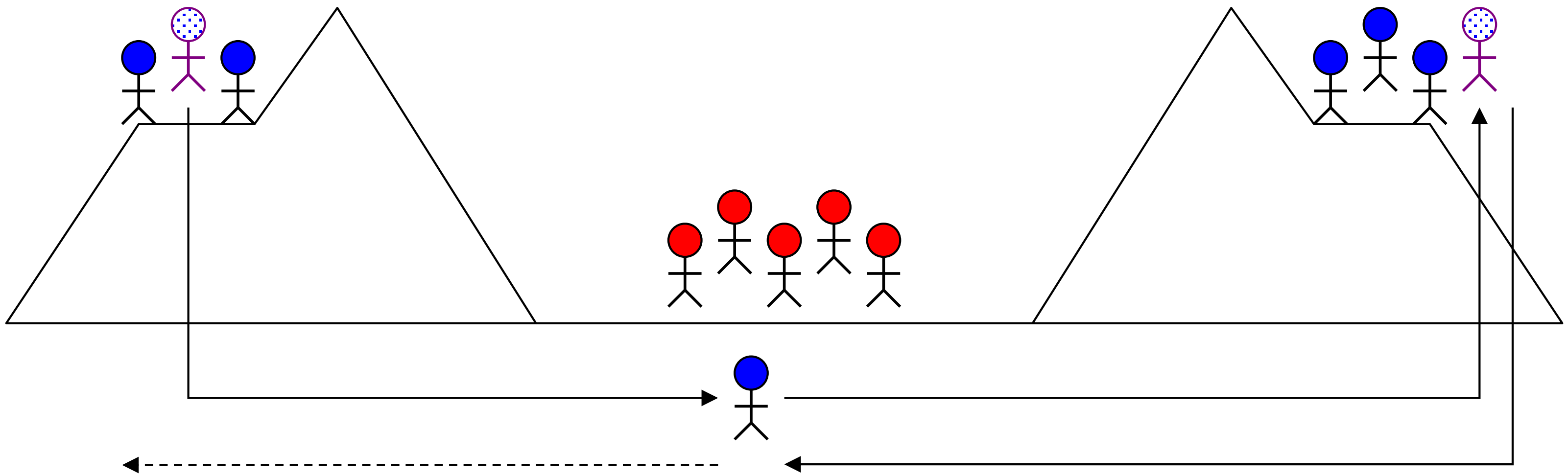
- Two army divisions (blue) surround enemy (red)
  - Each division led by a general
  - Both must agree when to simultaneously attack
  - If either side attacks alone, defeat
- Generals can only communicate via messengers
  - Messengers may get captured (unreliable channel)

The Two Generals Problem



- How to coordinate?
  - Send messenger: “Attack at dawn”
  - What if messenger doesn’t make it?

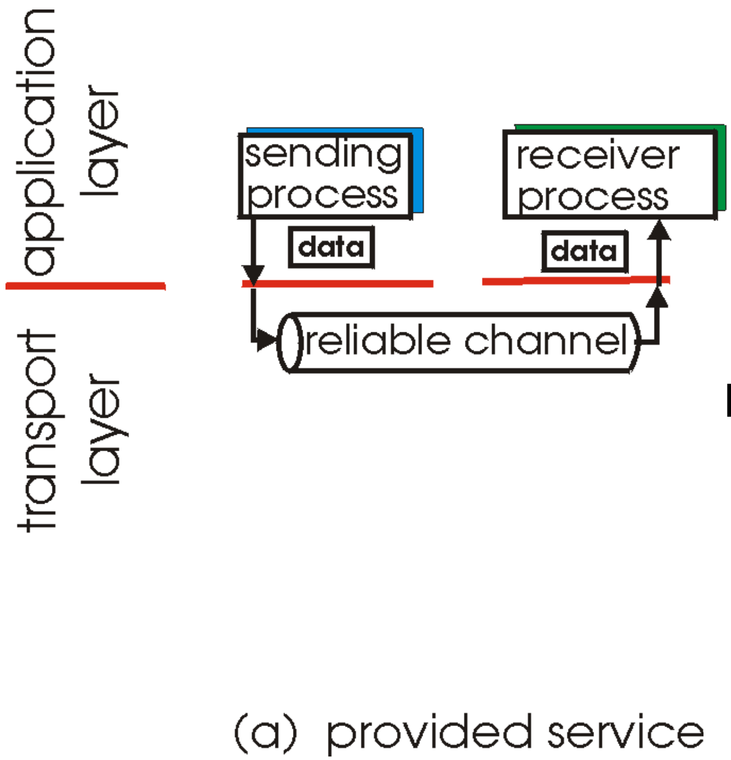
The Two Generals Problem



- How to be sure messenger made it?
  - Send acknowledgement: "We received message"

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

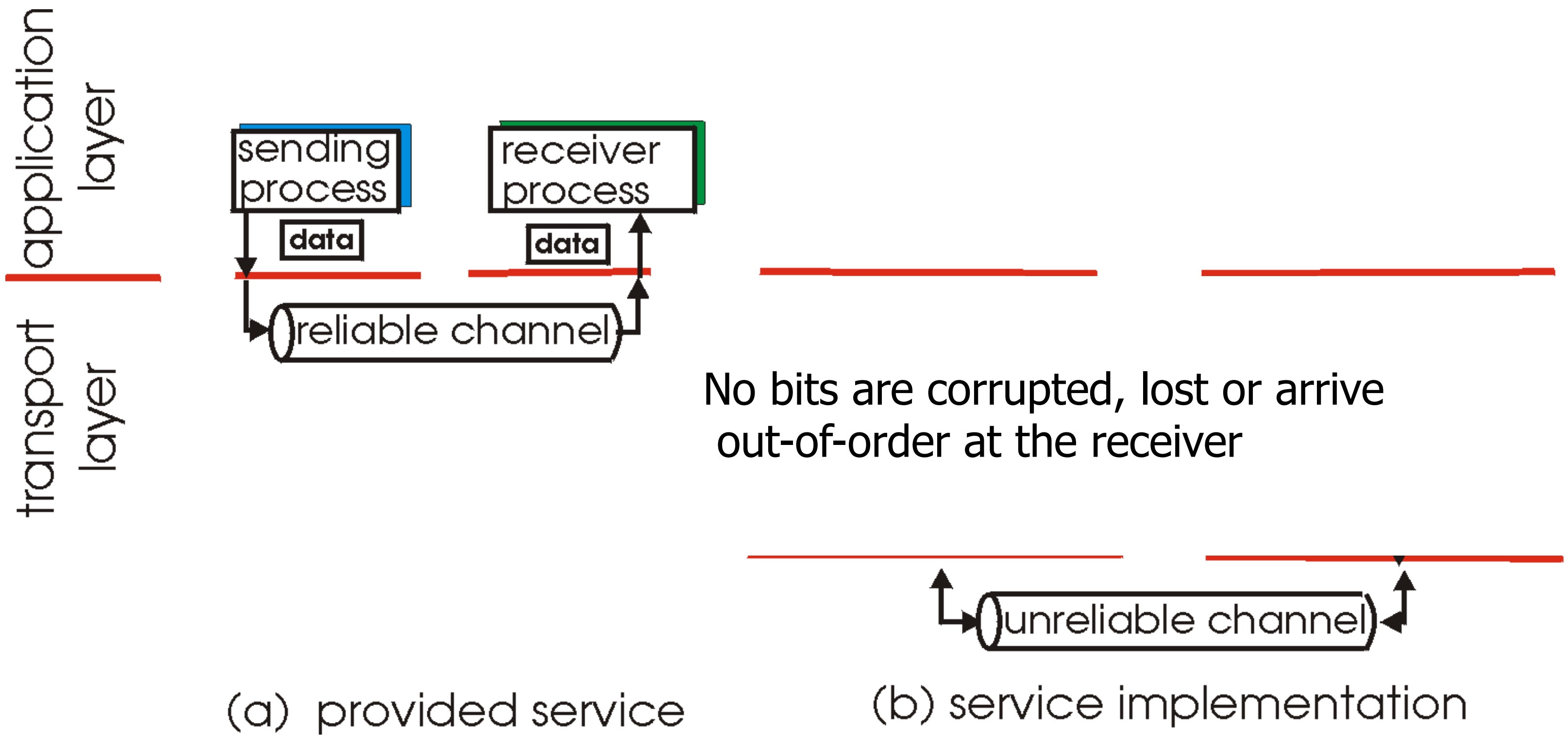


No bits are corrupted, lost or arrive out-of-order at the receiver

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

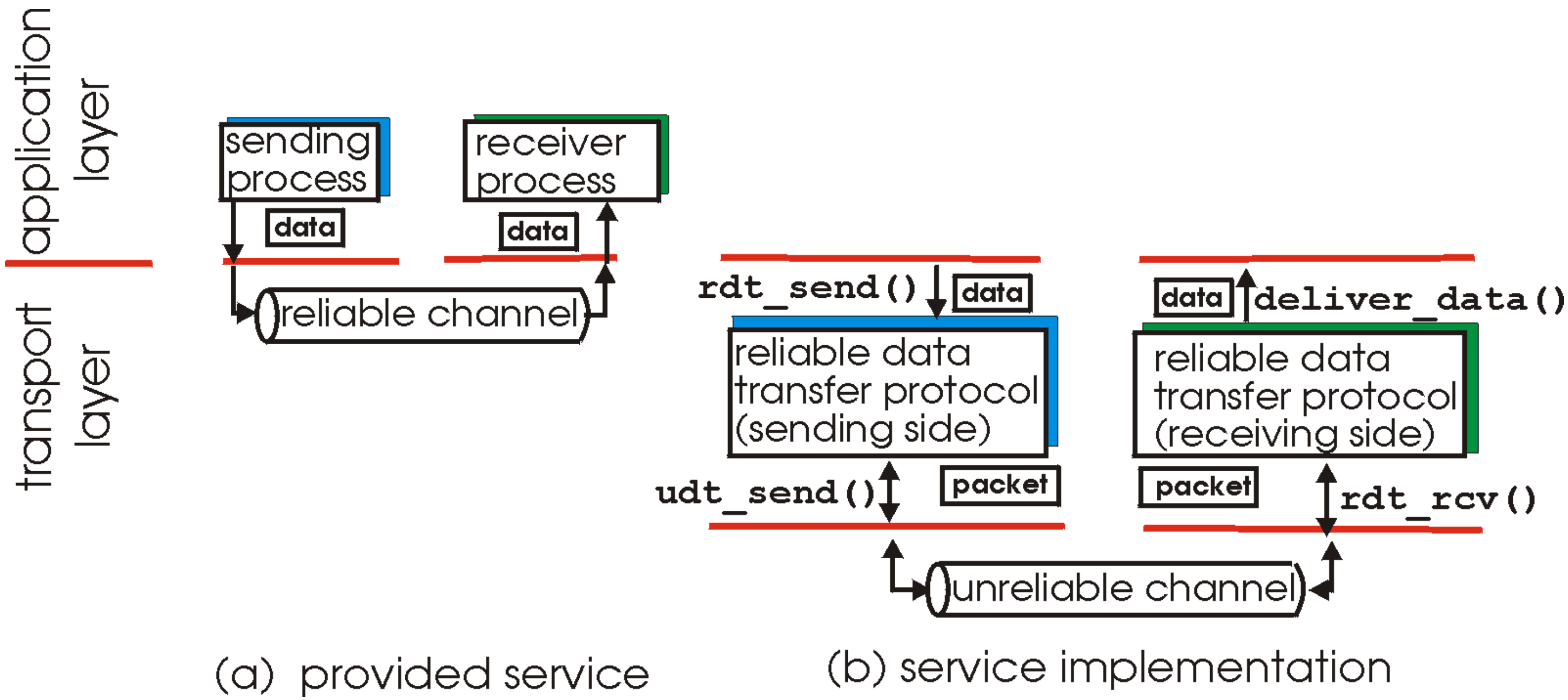


No bits are corrupted, lost or arrive out-of-order at the receiver

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

We' ll:

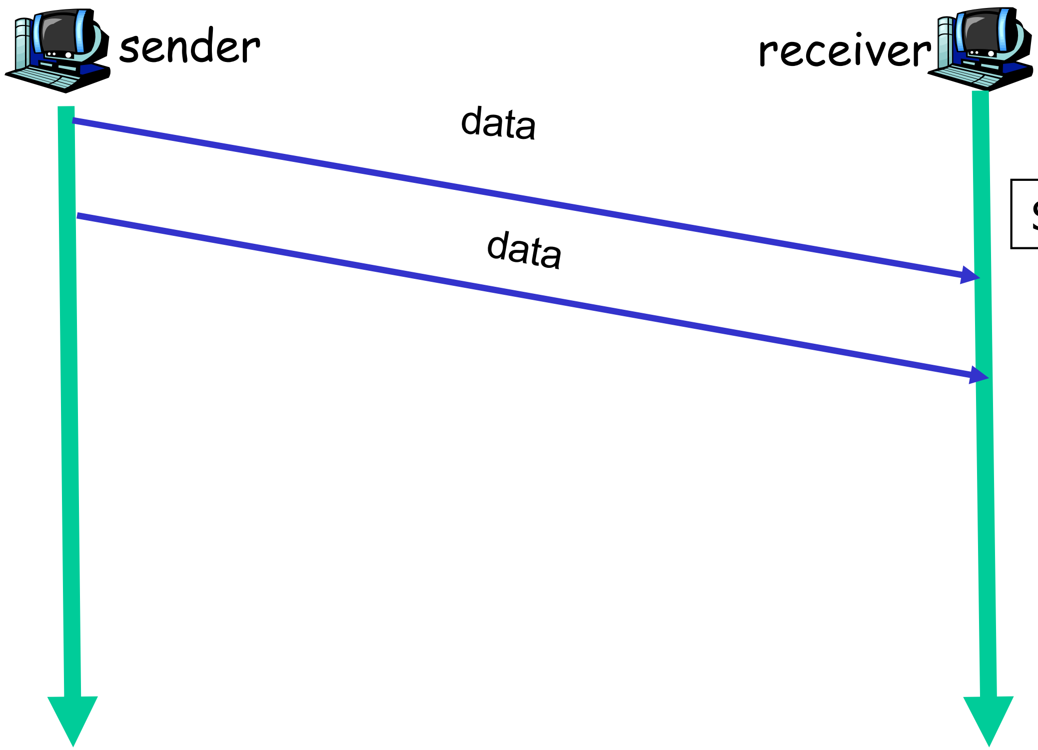
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - but control info will flow on both directions!
- Channel will not re-order packets

for convenience, just assume that the lower layers are unreliable

rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Transport layer does nothing !

Global Picture of rdt1.0



Solid line: error-free transmission

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

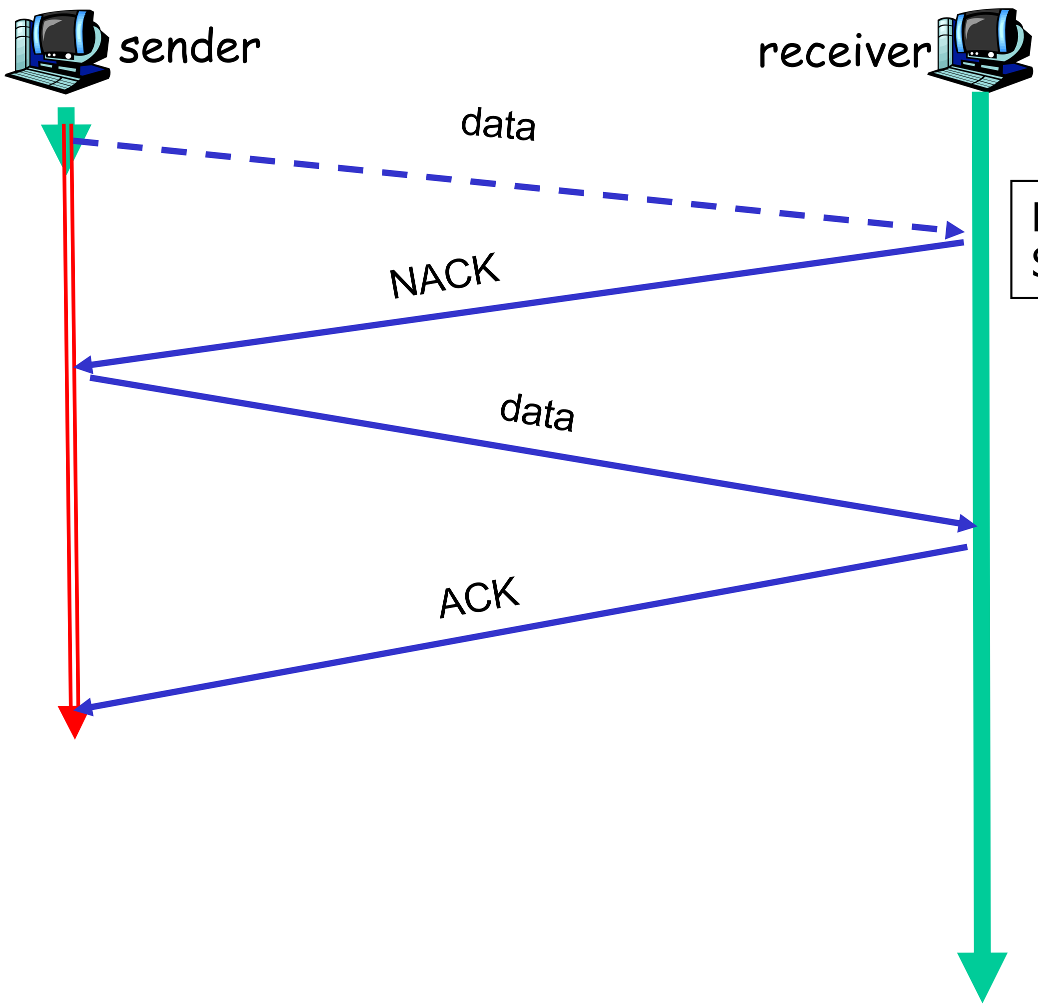
How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

Here checksum in the header of segment make sense

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK (positive acknowledgment)
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors Used for recover of big error
  - sender retransmits pkt on receipt of NAK (negative acknowledgment)
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender
  - retransmission

Global Picture of rdt2.0



Dotted line: erroneous transmission
Solid line: error-free transmission

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: discussion

Stop and wait!!!!

sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

receiver:

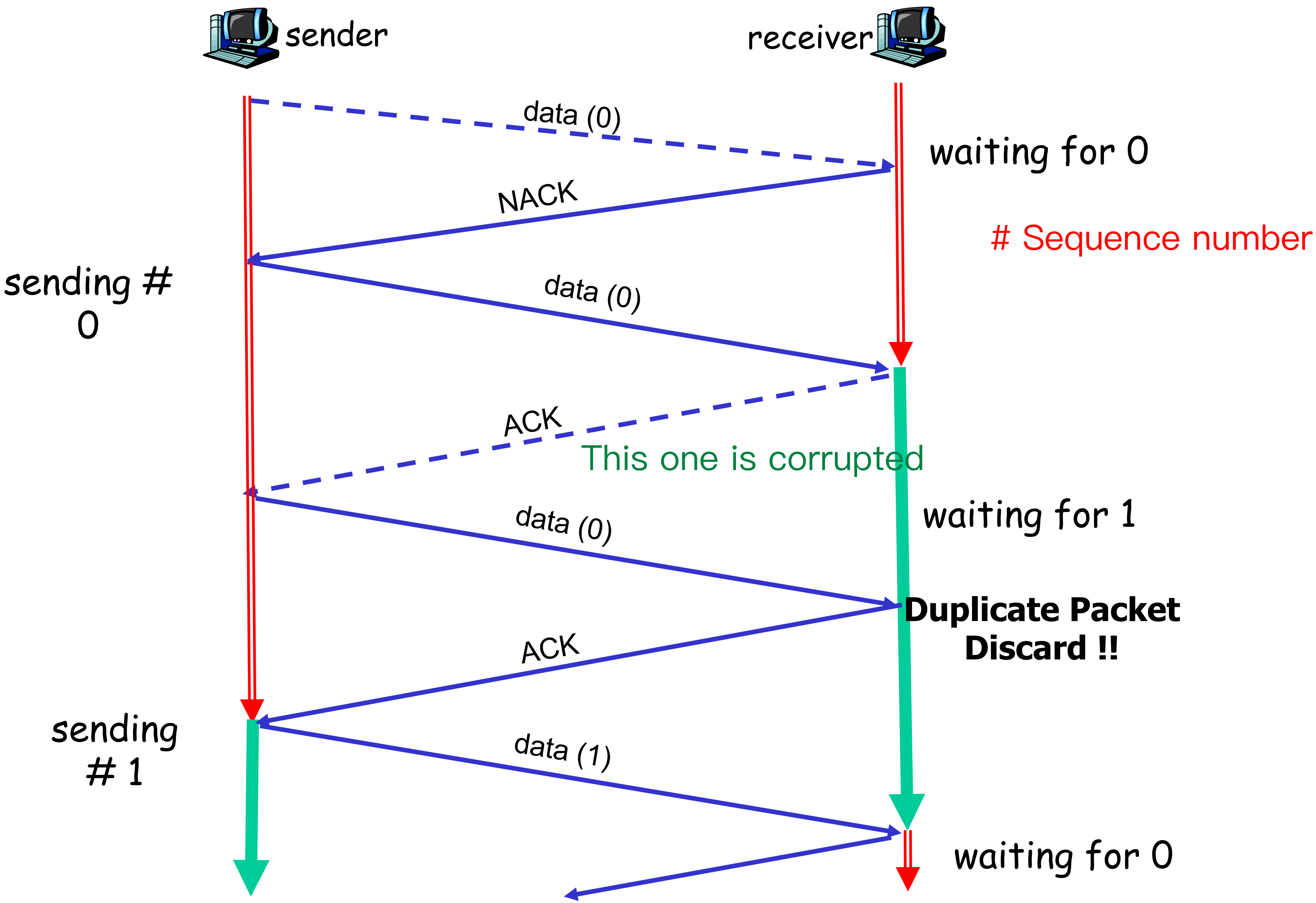
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

Simply check if the sequence number Changed, if unchange, just discard it

New Measures: Sequence Numbers, Checksum for ACK/NACK, Duplicate detection

Another Look at rdt2.1

Dotted line: erroneous transmission
Solid line: error-free transmission

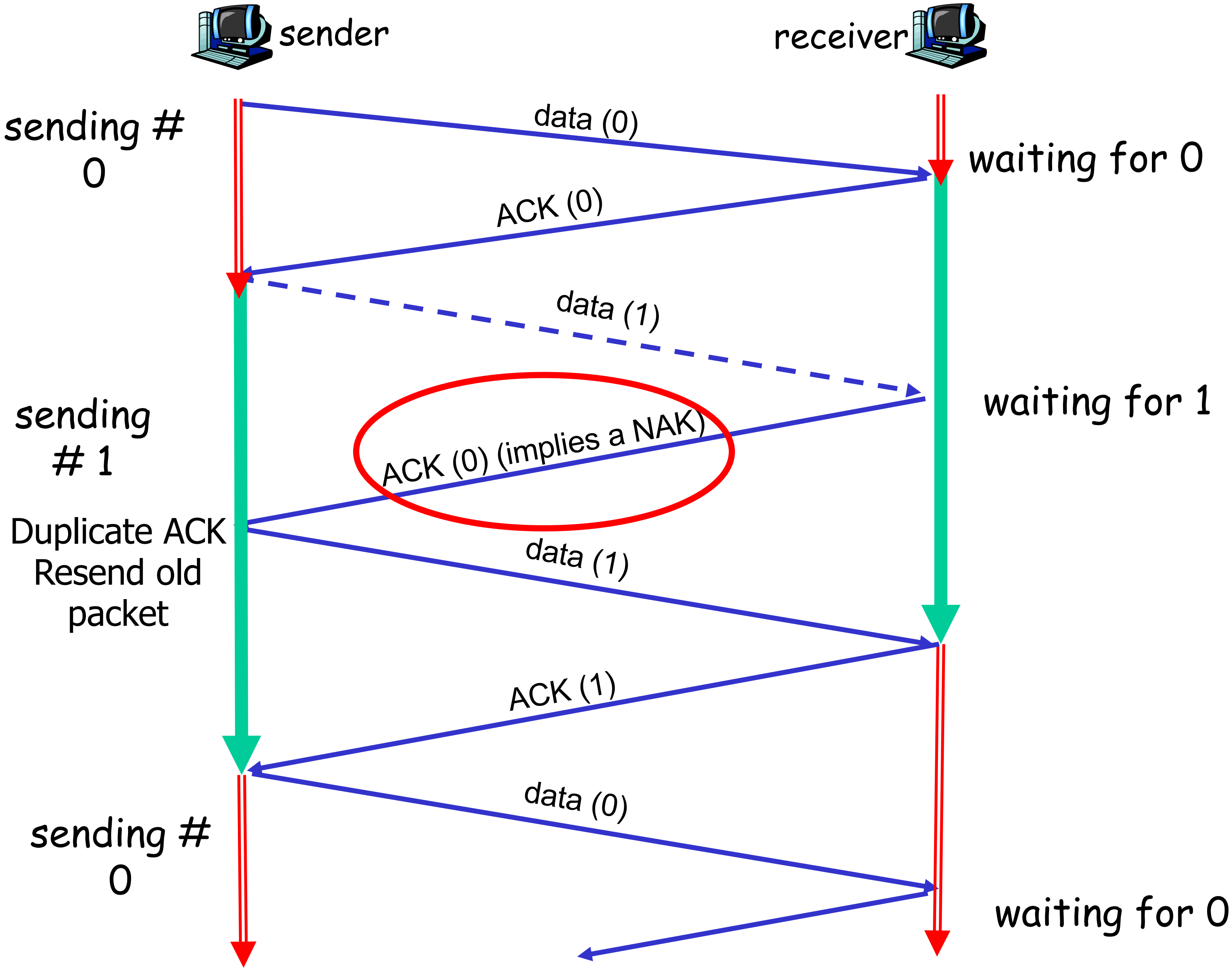


rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: Example

Dotted line: erroneous transmission
Solid line: error-free transmission



rdt3.0: channels with errors *and* loss

Packet error —> the receiver get the imformation

But loss may not

new assumption: underlying channel can also loose packets (data, ACKs)

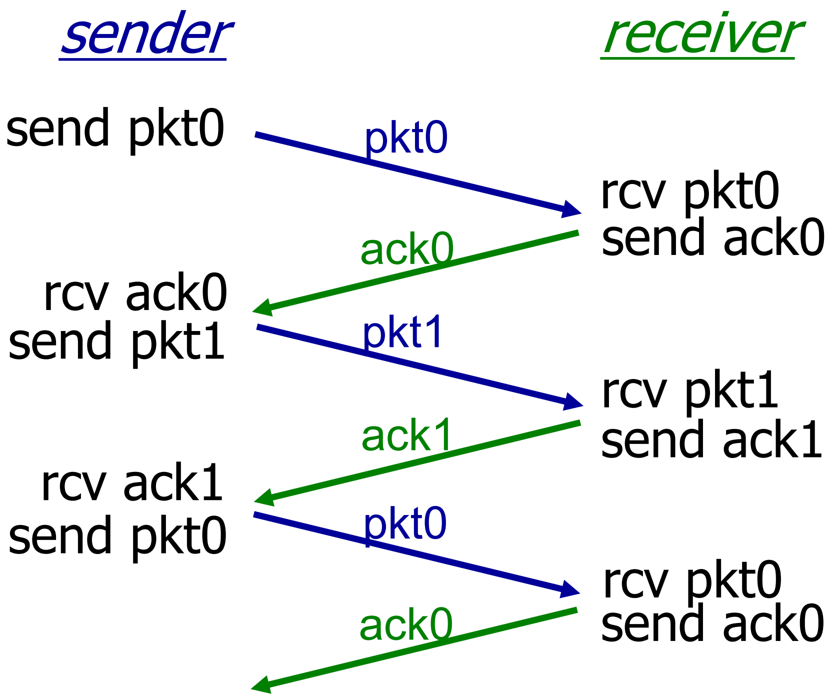
- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

The timer is dynamic

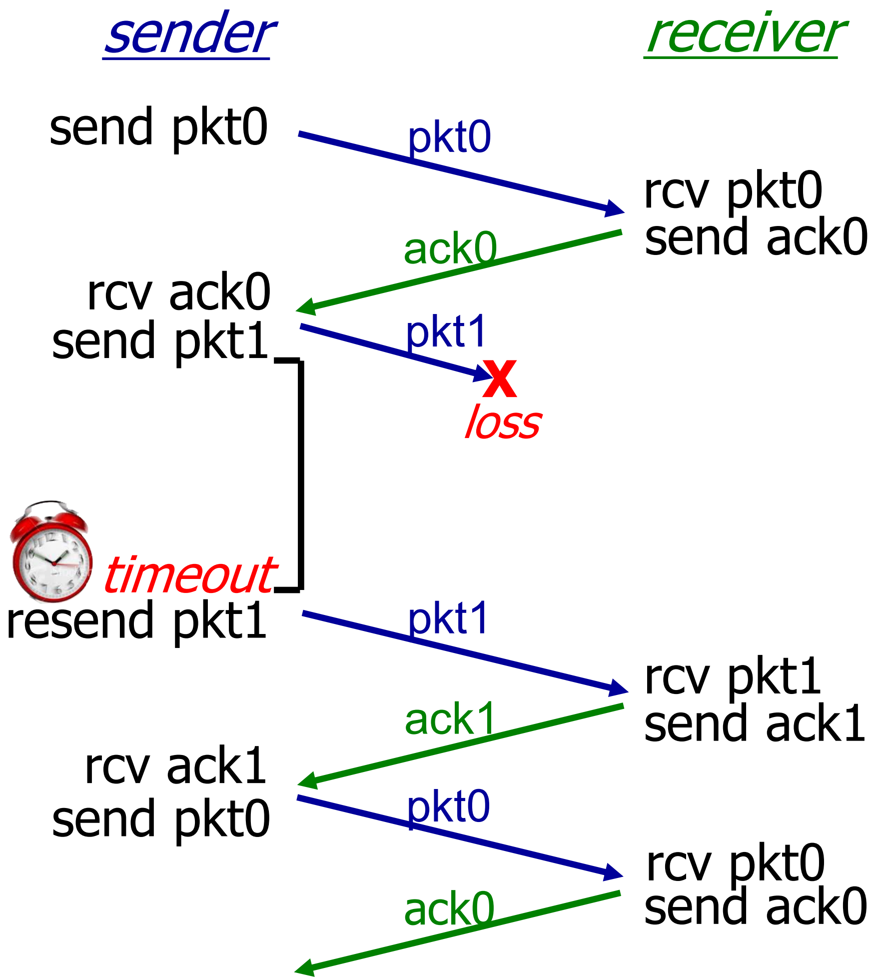
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 in action

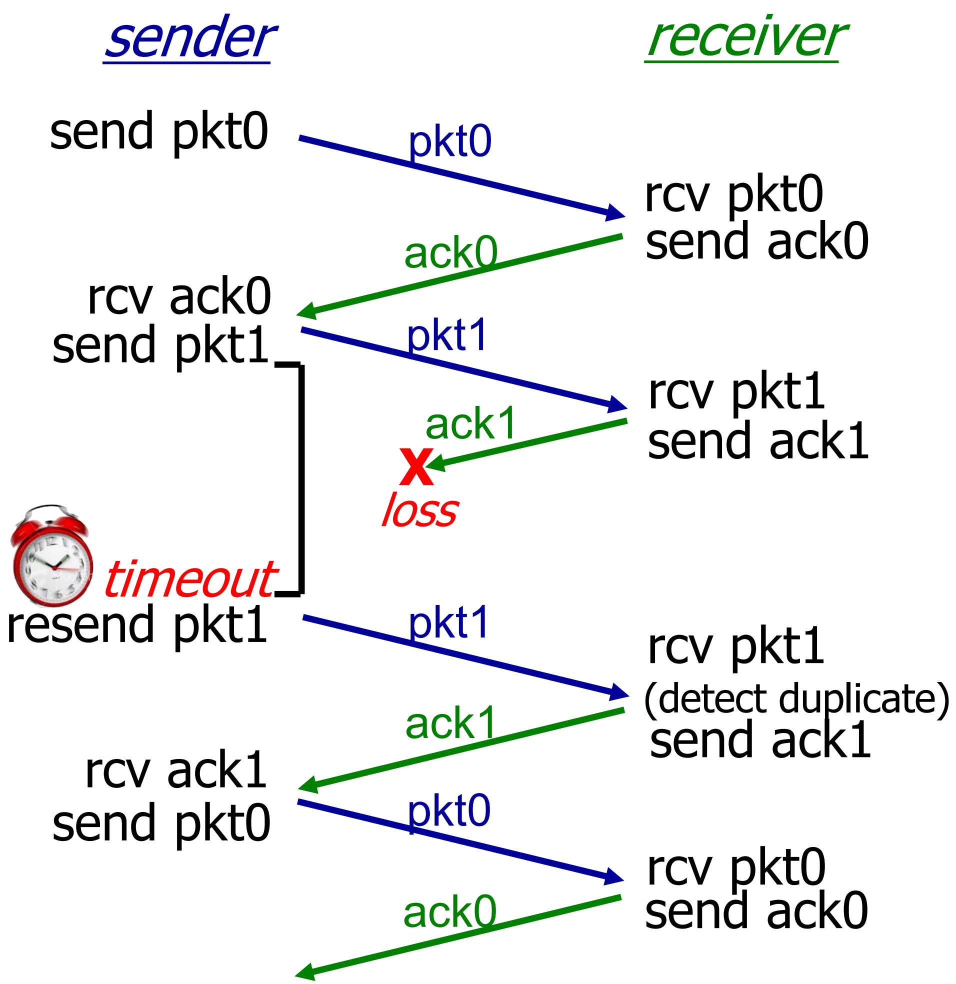


(a) no loss

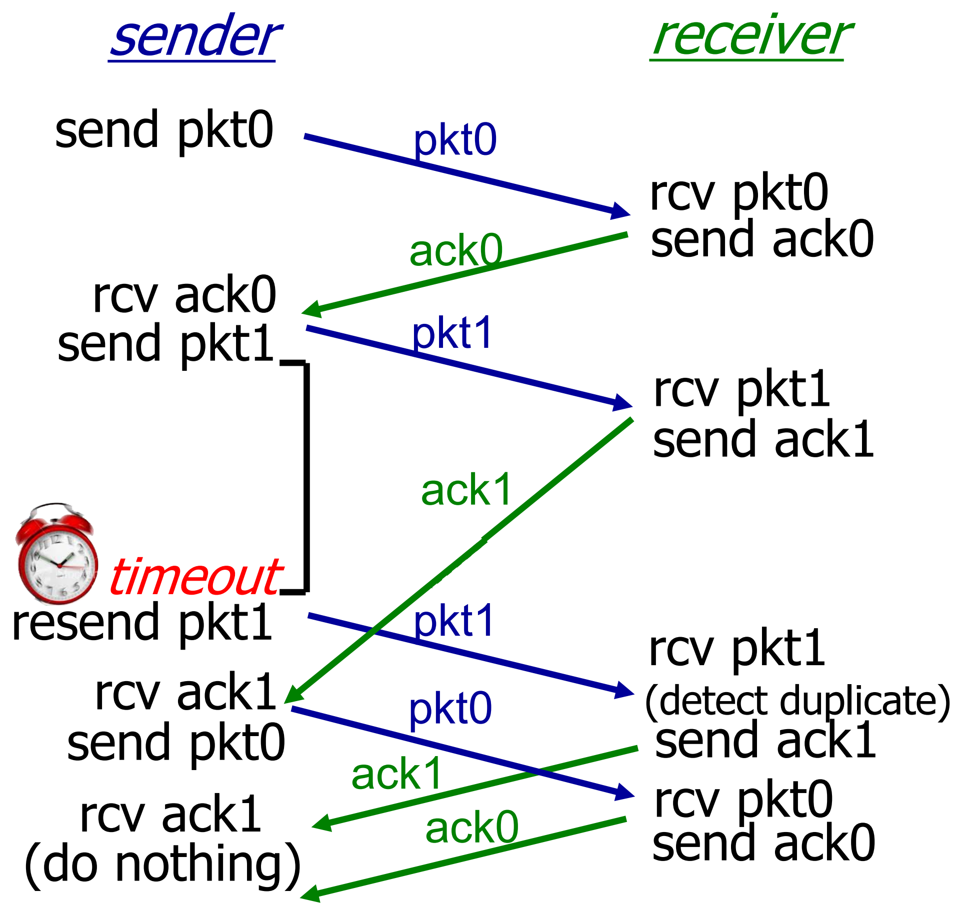


(b) packet loss

rdt3.0 in action



(c) ACK loss



Sender have to identify that this is a Kind of duplicates

(d) premature timeout/ delayed ACK

Quiz: Reliable Data Transfer

- Which of the following are needed for reliable data transfer with only packet corruption (and no loss or reordering)? Use only as much as is strictly needed.

d

a) Checksums
b) Checksums, ACKs, NACKs
c) Checksums, ACKs
d) Checksums, ACKs, sequence numbers
e) Checksums, ACKs, NACKs, sequence numbers

Quiz: Reliable Data Transfer

- If packets (and ACKs and NACKs) could be lost which of the following is true of RDT 2.1 (or 2.2)?

b

a) Reliable in-order delivery is still achieved
b) The protocol will get stuck
c) The protocol will continue making progress but may skip delivering some messages

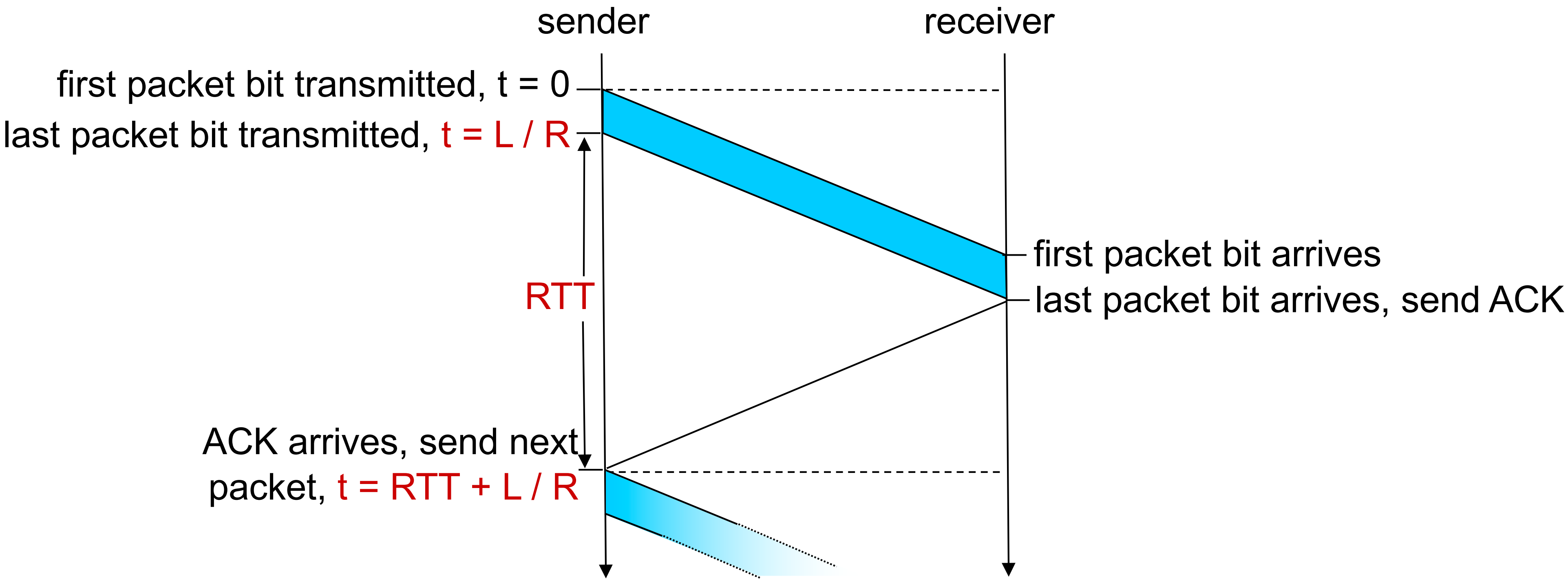
Quiz: Reliable Data Transfer

- Which of the following are needed for reliable data transfer to handle packet corruption and loss? Use only as much as is strictly needed.

D

a) Checksums, timeouts
b) Checksums, ACKs, sequence numbers
c) Checksums, ACKs, timeouts
d) Checksums, ACKs, timeouts, sequence numbers
e) Checksums, ACKs, NACKs, timeouts, sequence numbers

rdt3.0: stop-and-wait operation



$$U_{sender} = \frac{L / R}{RTT + L / R}$$

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 8000 bit packet and 30msec RTT:

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

- U $_{sender}$: *utilization* – fraction of time sender busy sending

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

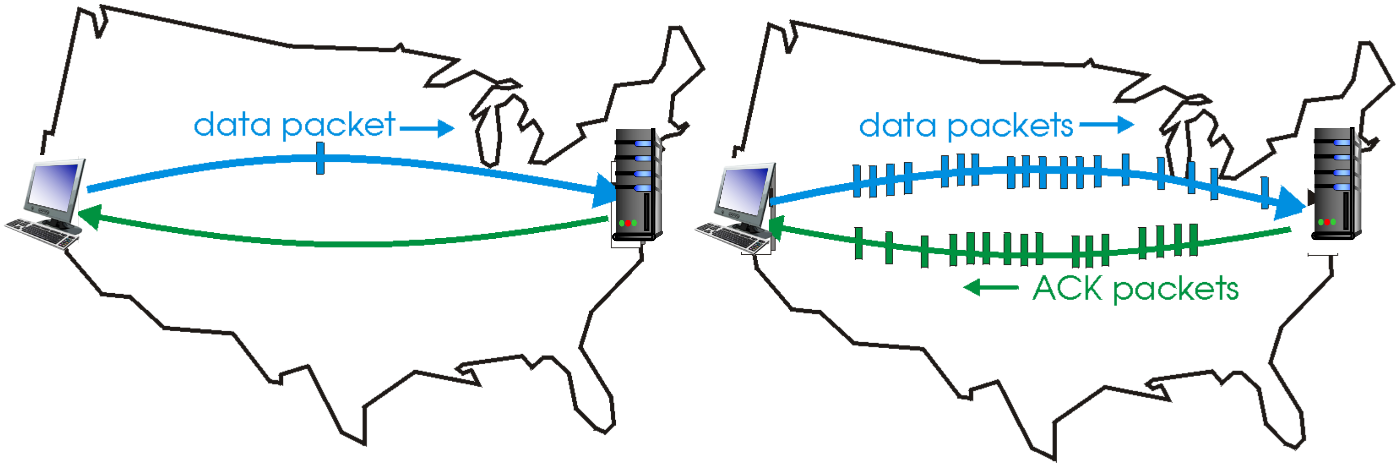
- RTT=30 msec, 1KB pkt every 30.008 msec: 33kB/sec thruput over 1 Gbps link

- Network protocol limits use of physical resources!

Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver



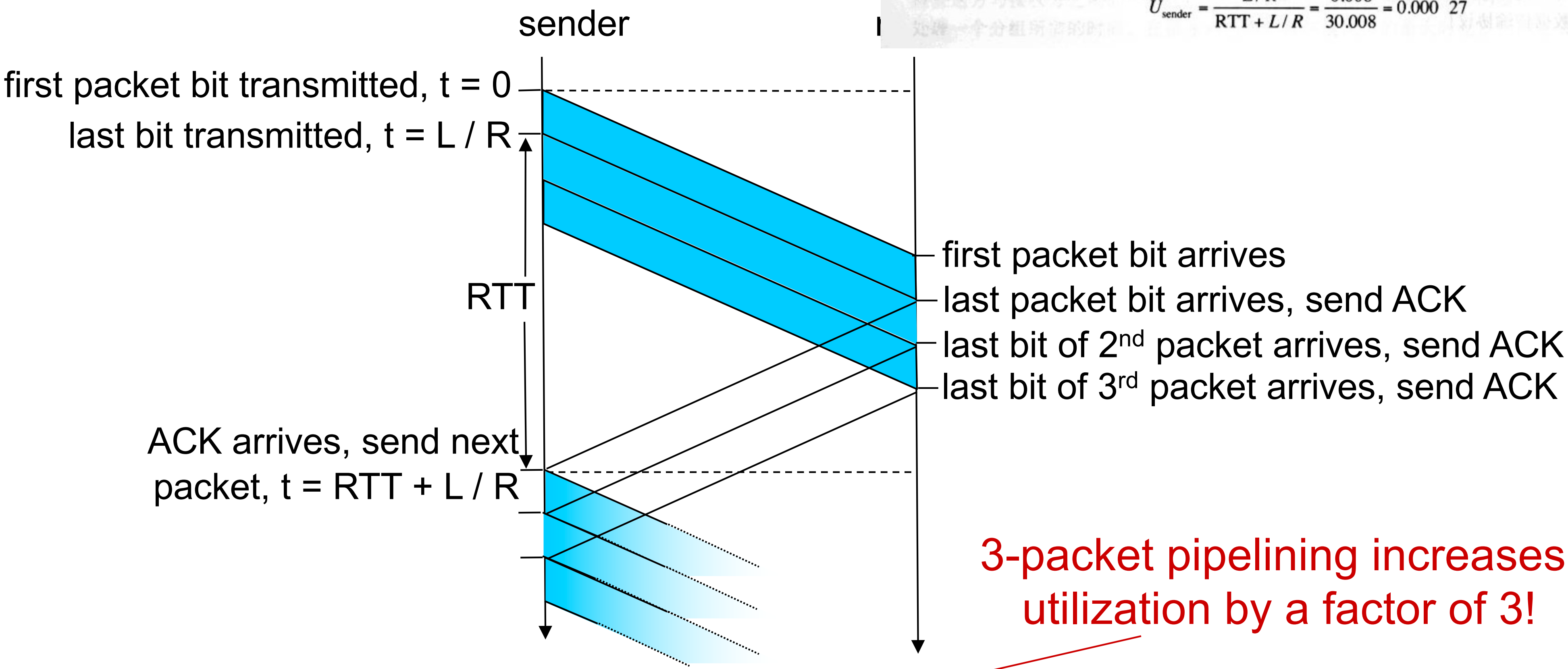
(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- two generic forms of pipelined (sliding window) protocols: *go-Back-N, selective repeat*

Pipelining: increased u

图3-18a显示了对于停等协议，如果发送方在$t = 0$时刻开始发送分组，则在$t = L/R = 8\ \mu s$后，最后1 比特数据进入发送方信道。该分组经过15 ms跨美国的旅行后到达接收方，该分组的最后1比特在时刻$t = RTT/2 + L/R = 15.008$ ms时到达接收方。为了简化起见，假设ACK分组很小（以便我们可以忽略其发送时间），接收方收到一个数据分组的最后1 比特后立即发送ACK，ACK在时刻$t = RTT + L/R = 30.008$ ms时到达发送方。此时，发送方可以发送下一个报文。因此，在30.008 ms内，发送方的发送只用了0.008 ms。定义发送方（或信道）的**利用率**（utilization）为：发送方实际忙着将发送比特送进信道的那部分时间与发送时间之比。图3-18a中的分析表明了停等协议有着非常低的发送方利用率U_{sender}：

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30.008} = 0.000\ 27$$



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

$$U_{sender} = \frac{3L / R}{RTT + L / R}$$

Here is the drawbacks of stop and wait

Pipelined protocols: overview

Go-Back-N:

- Sender can have up to N unacked packets in pipeline
- Sender has single timer for oldest unacked packet, when timer expires, retransmit *all* unacked packets
- There is no buffer available at Receiver, out of order packets are discarded
- Receiver only sends *cumulative ack*, doesn't ack new packet if there's a gap

Selective Repeat:

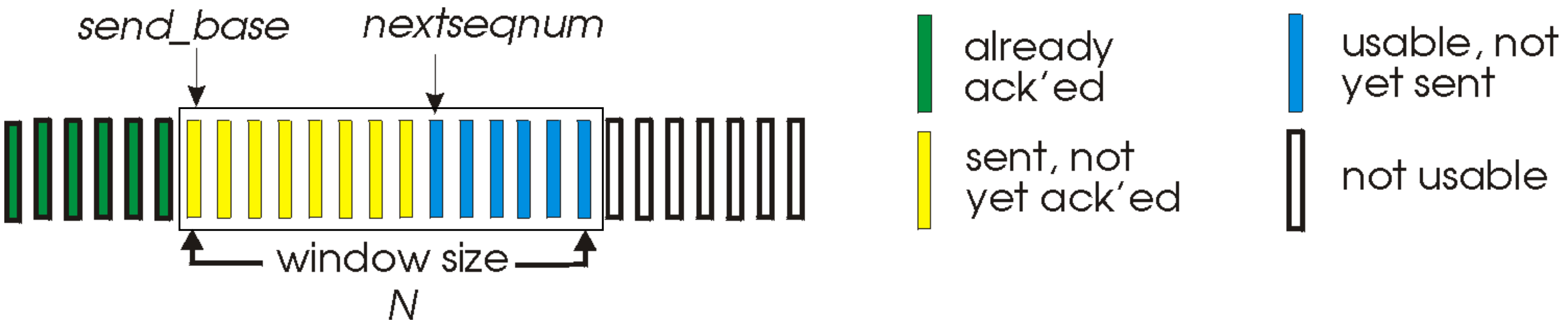
- Sender can have up to N unacked packets in pipeline

Vary as the size of packet

- Sender maintains timer for each unacked packet, when timer expires, retransmit only that unacked packet
- Receiver has buffer, can accept out of order packets
- Receiver sends *individual ack* for each packet

Go-Back-N: sender

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



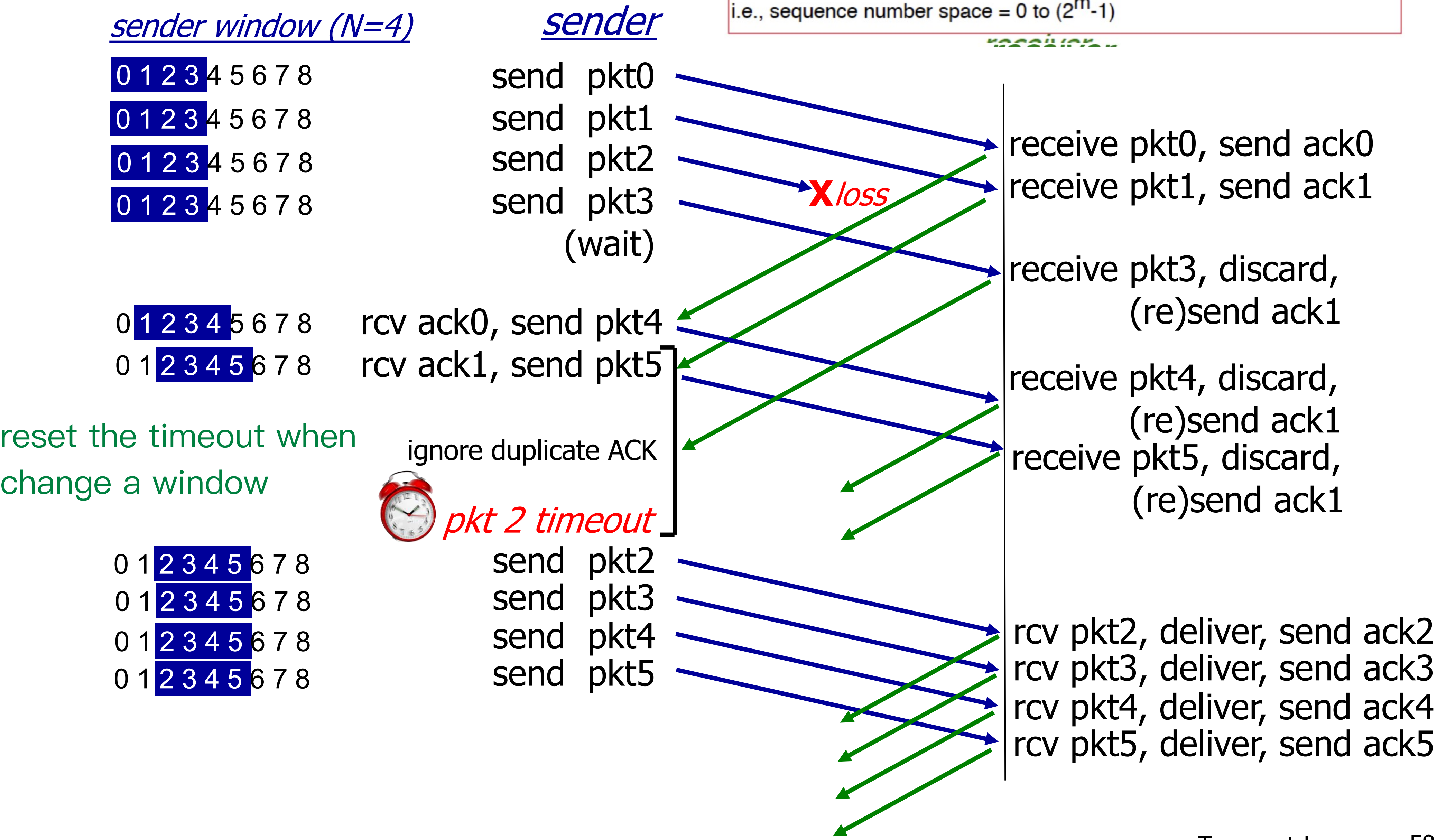
- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n)*: retransmit packet n and all higher seq # pkts in window

Applets: http://media.pearsoncmg.com/aw/aw_kurose_network_2/applets/go-back-n/go-back-n.html
http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

m is the number of bit in sequence number field

GBN in action

GBN Window size restrictions:
Receiver Window Size = 1
Sender Window Size (N) < 2^m (why not 2^m)Hint: what if all ACKs are lost!
m is the number of bits in sequence number field
i.e., sequence number space = 0 to (2^m-1)



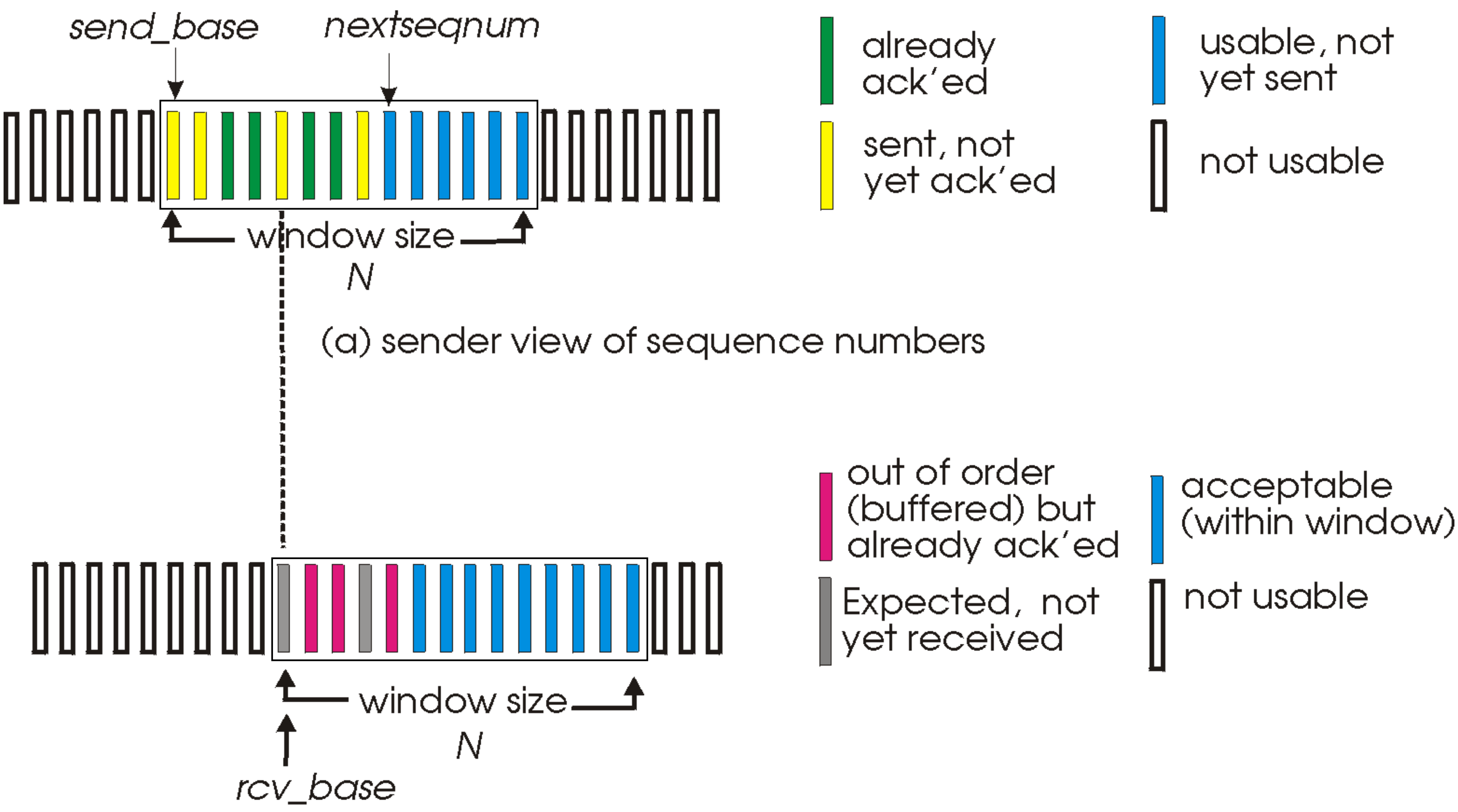
SR sender window size = receiver window size <=(2^{m-1})

Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - *N* consecutive seq #'s
  - limits seq #s of sent, unACKed pkts

Applet: http://media.pearsoncmg.com/aw/aw_kurose_network_3/applets/SelectRepeat/SR.html

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

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

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

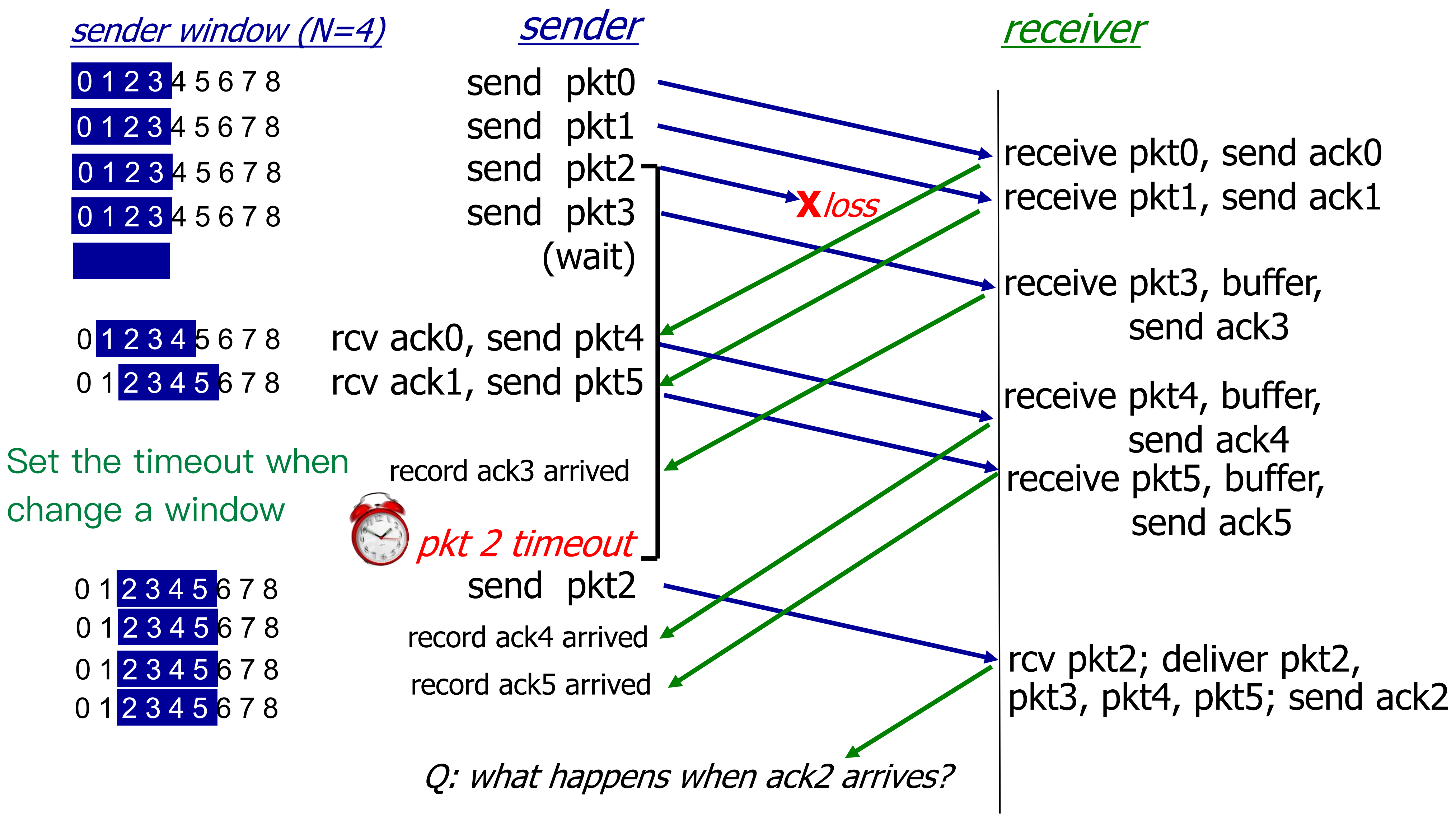
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



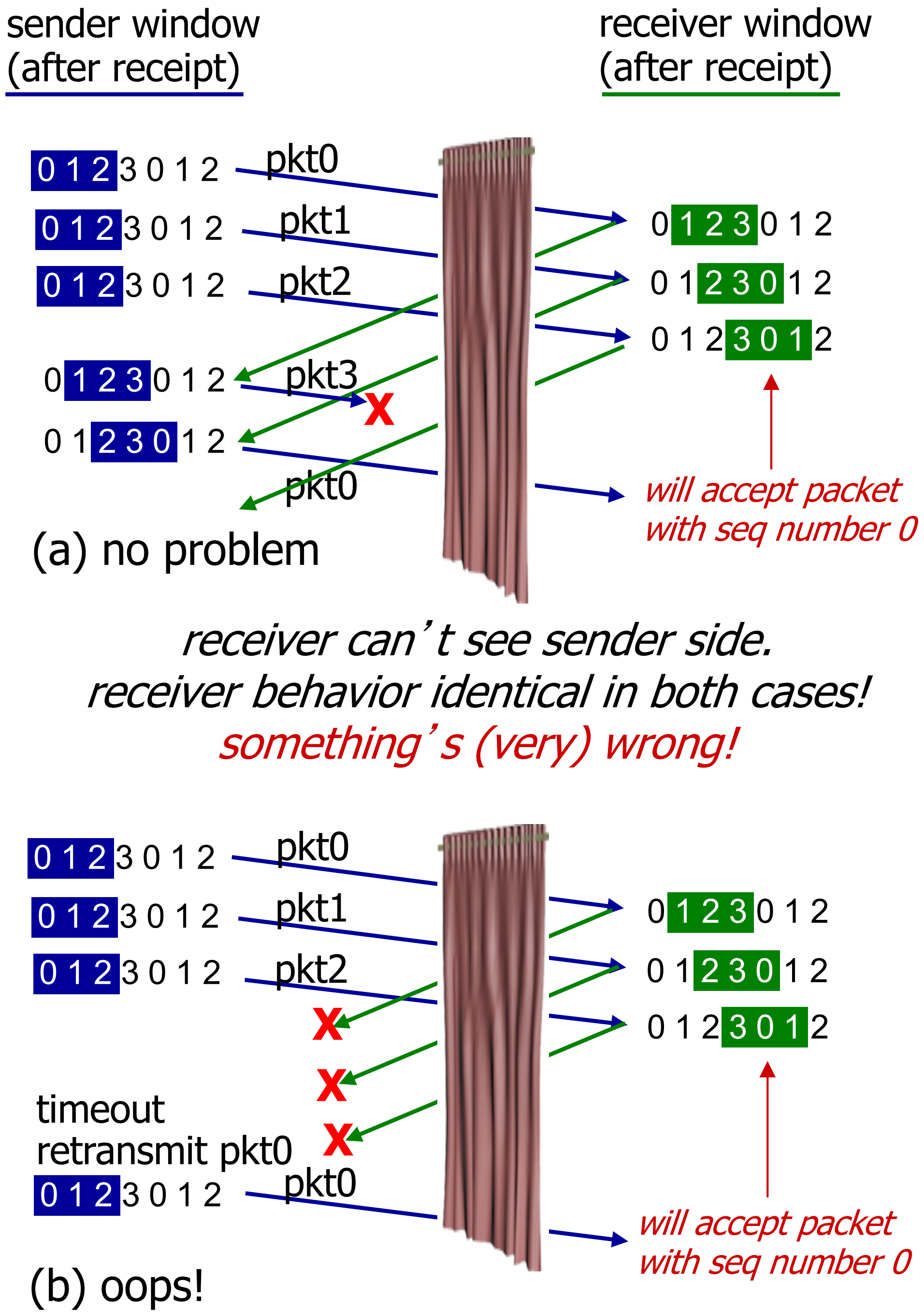
Selective repeat: dilemma

example:

- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?

A: Sender window size <= ½ of Sequence number space



Recap: components of a solution

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)

- Reliability protocols use the above to decide when and what to retransmit or acknowledge

Quiz: GBN, SR

- Which of the following is not true? c

a) GBN uses cumulative ACKs, SR uses individual ACKs
b) Both GBN and SR use timeouts to address packet loss
c) GBN maintains a separate timer for each outstanding packet
d) SR maintains a separate timer for each outstanding packet
e) Neither GBN nor SR use NACKs

Quiz: GBN, SR

- Suppose a receiver that has received all packets up to and including sequence number 24 and next receives packet 27 and 28. In response, what are the sequence numbers in the ACK(s) sent out by the GBN and SR receiver, respectively?

a) [27, 28], [28, 28]
b) [24, 24], [27, 28]
c) [27, 28], [27, 28]
d) [25, 25], [25, 25]
e) [nothing], [27, 28]

B

Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

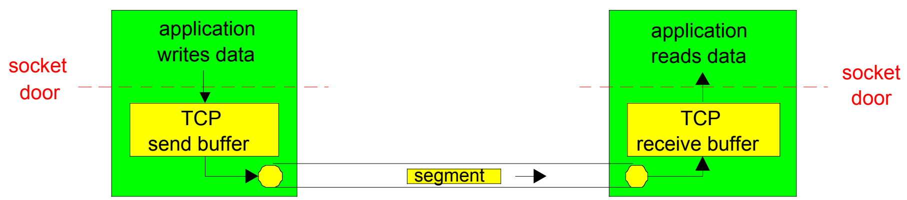
3.7 TCP congestion control

Practical Reliability Questions

- How do the sender and receiver keep track of outstanding pipelined segments?
- How many segments should be pipelined?
- How do we choose sequence numbers?
- What does connection establishment and teardown look like?
- How should we choose timeout values?

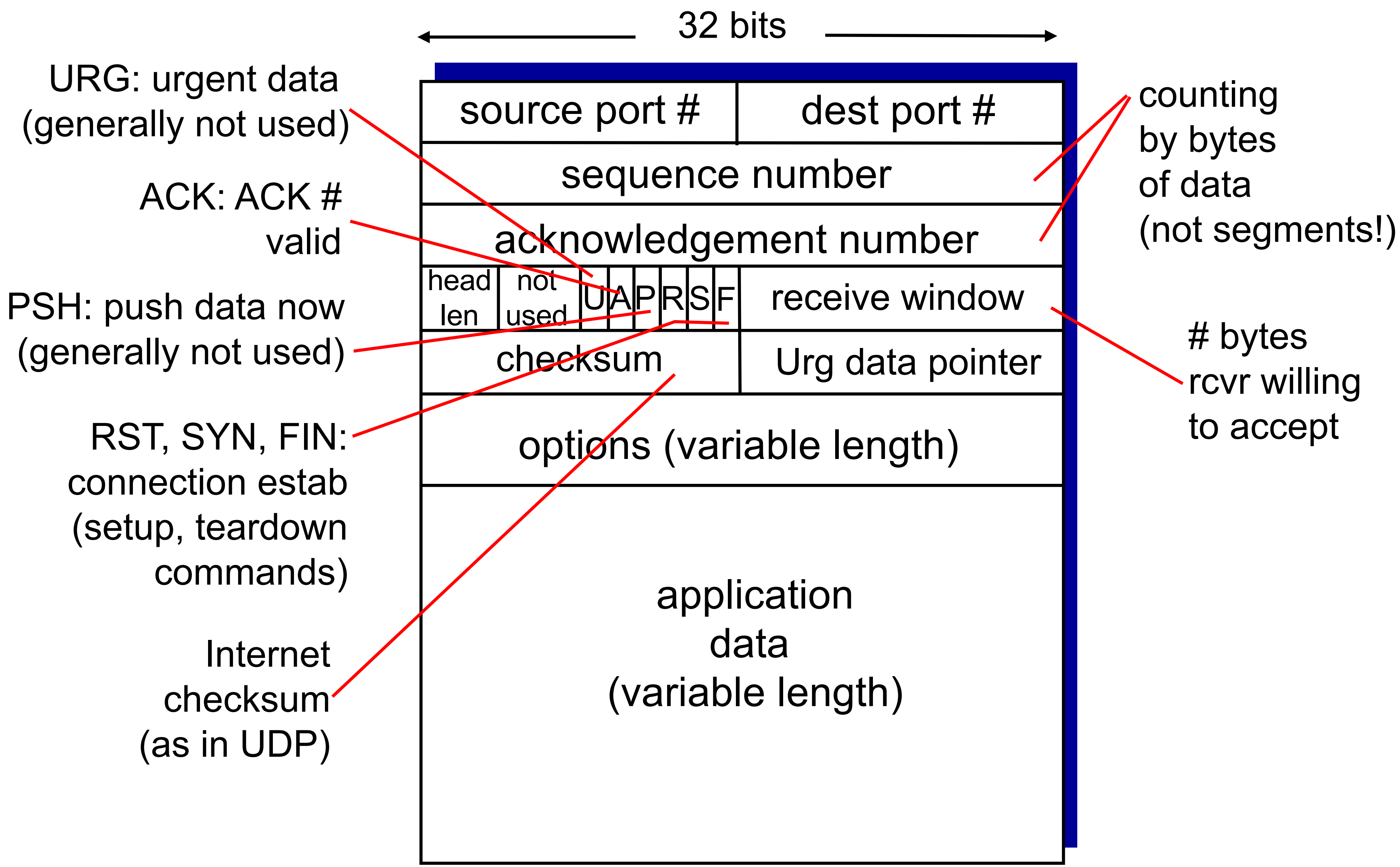
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- send and receive buffers



- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

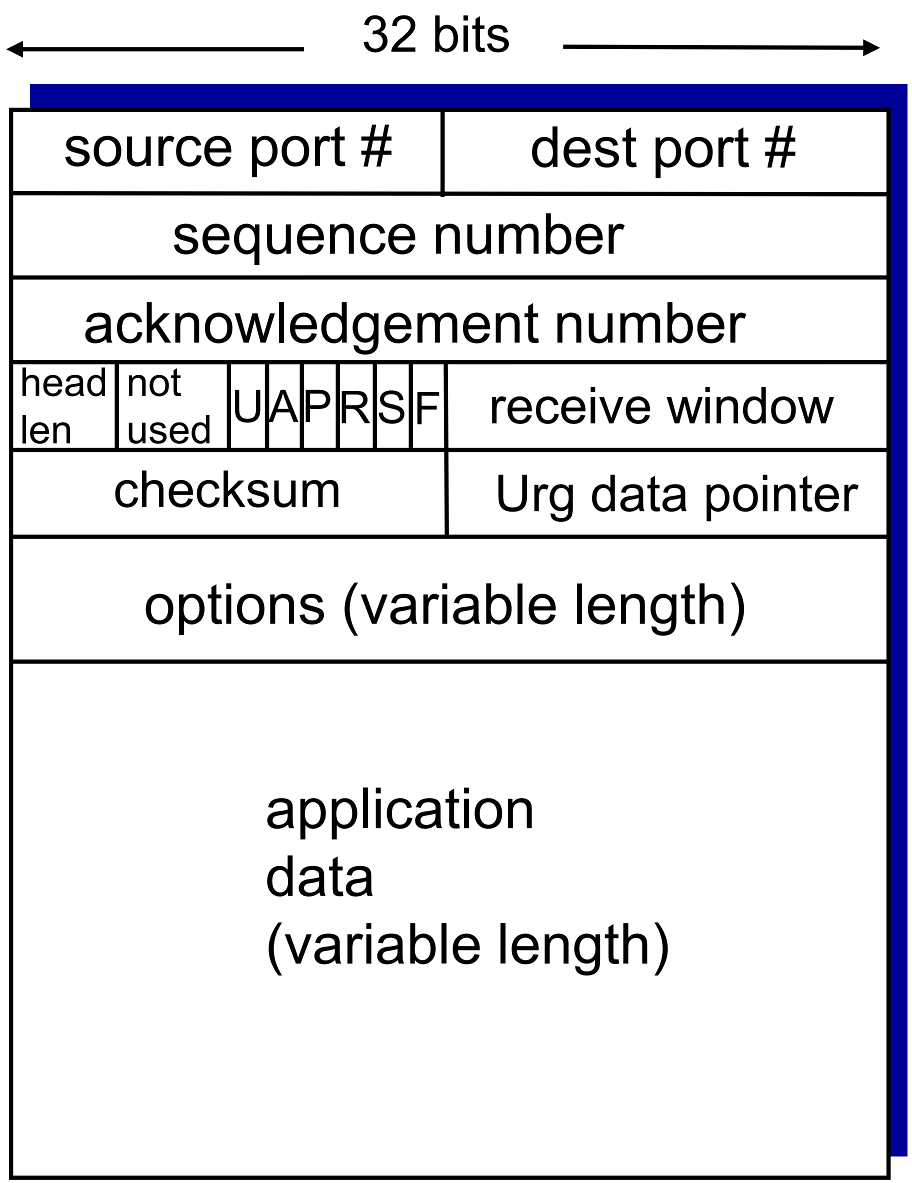
TCP segment structure



TCP segment structure

20 Bytes

(UDP was 8)



Transport Layer Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Recall: Components of a solution for reliable transport

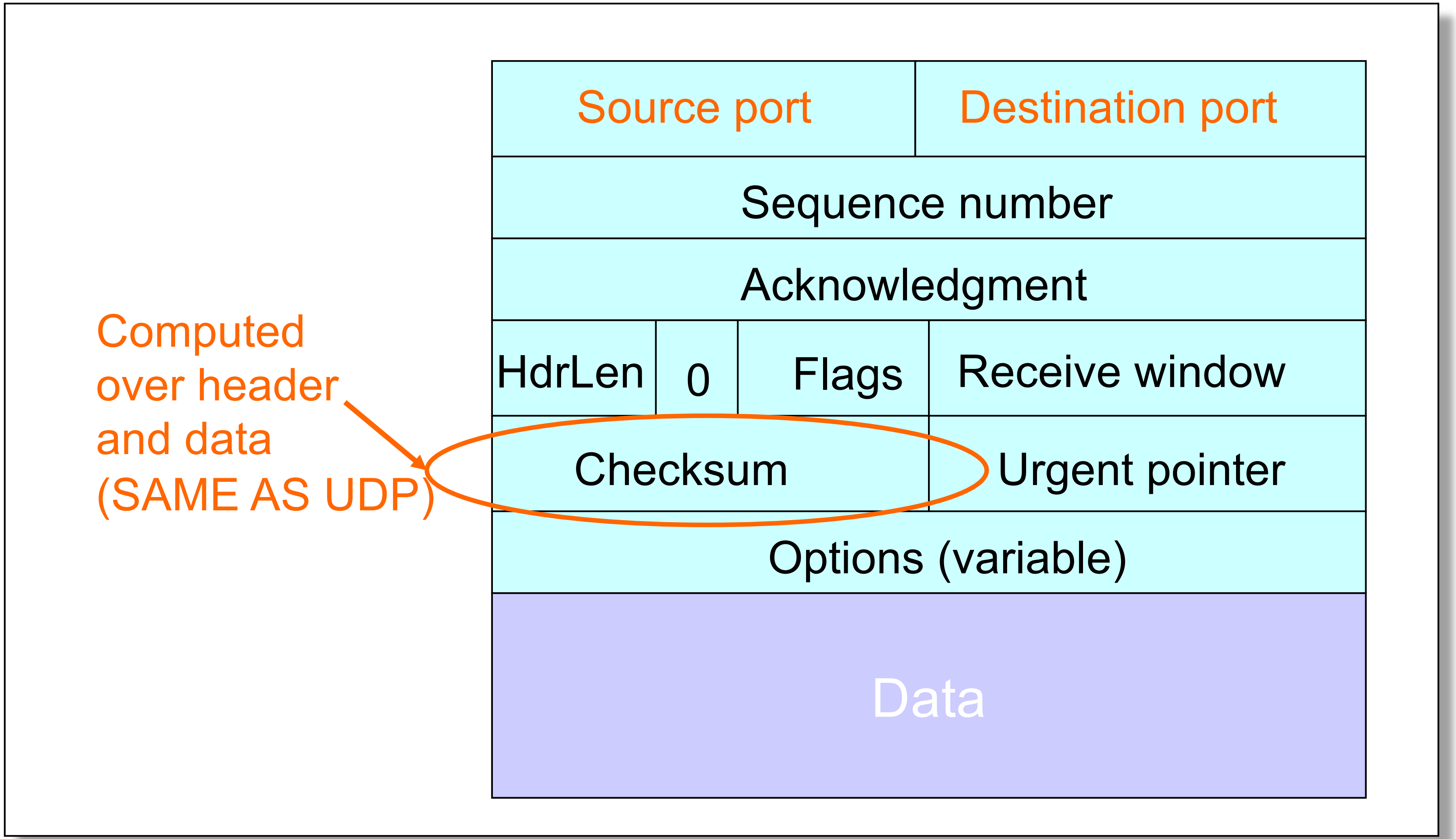
- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
  - Go-Back-N (GBN)
  - Selective Repeat (SR)

What does TCP do?

Many of our previous ideas, but some key differences

- Checksum

TCP Header


Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Receive window
Checksum
Urgent pointer
Options (variable)
Data
Computed over header and data (SAME AS UDP)

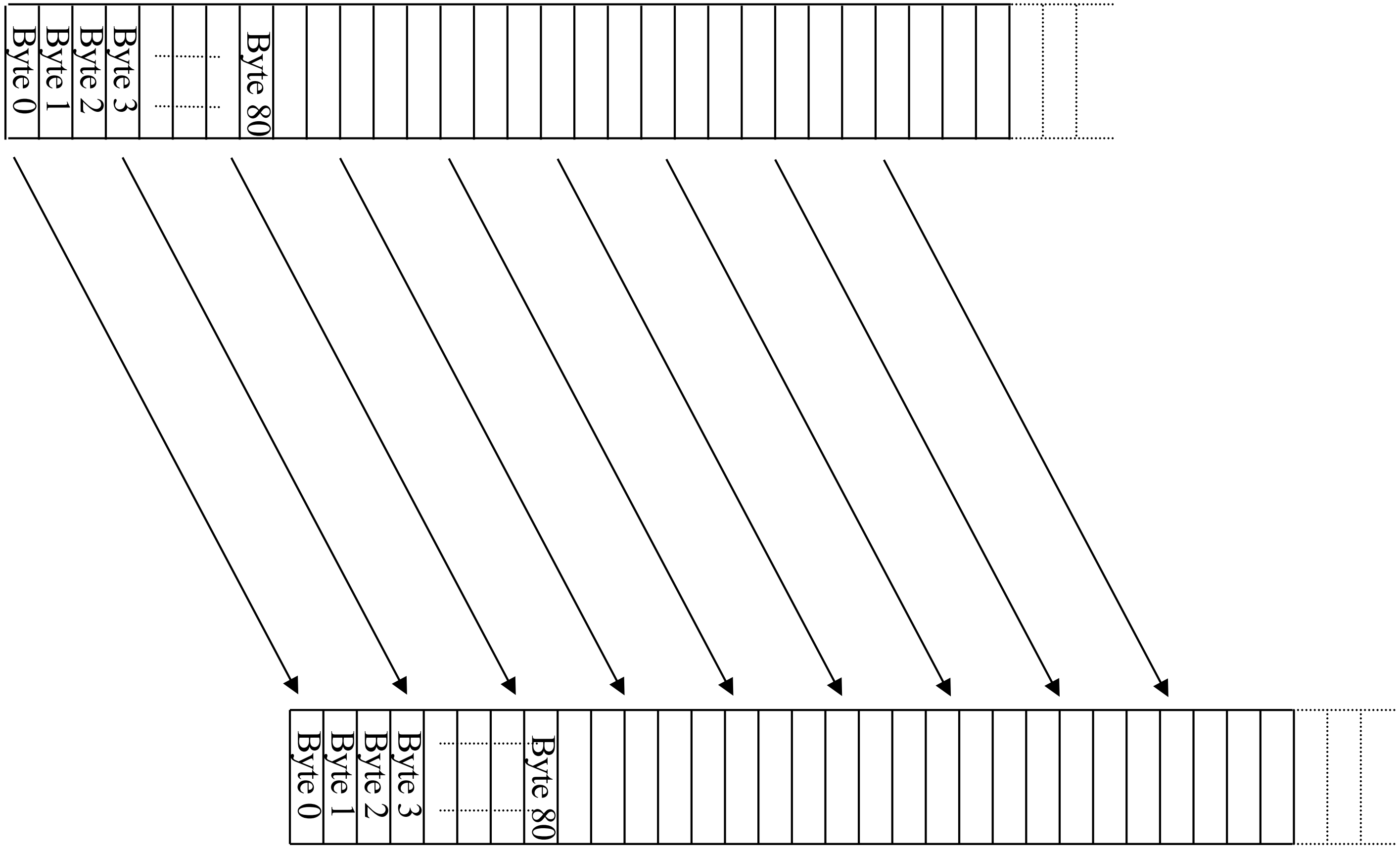
What does TCP do?

Many of our previous ideas, but some key differences

- Checksum
- **Sequence numbers are byte offsets**

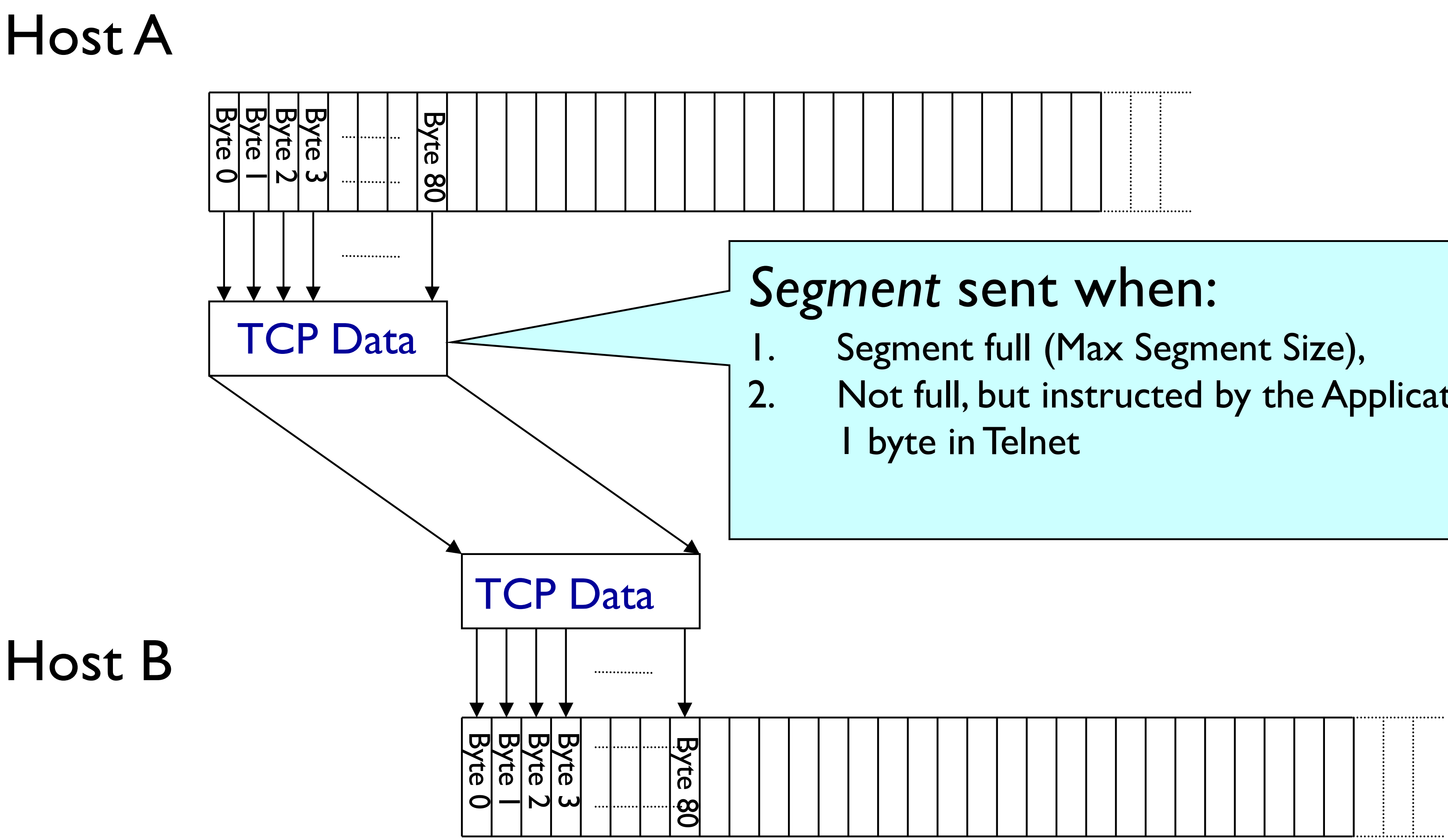
TCP "Stream of Bytes" Service ..

Application @ Host A



Application @ Host B

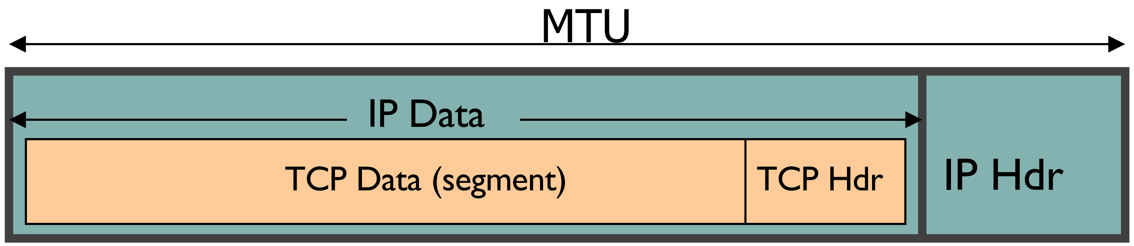
.. Provided Using TCP "Segments"



Segment sent when:

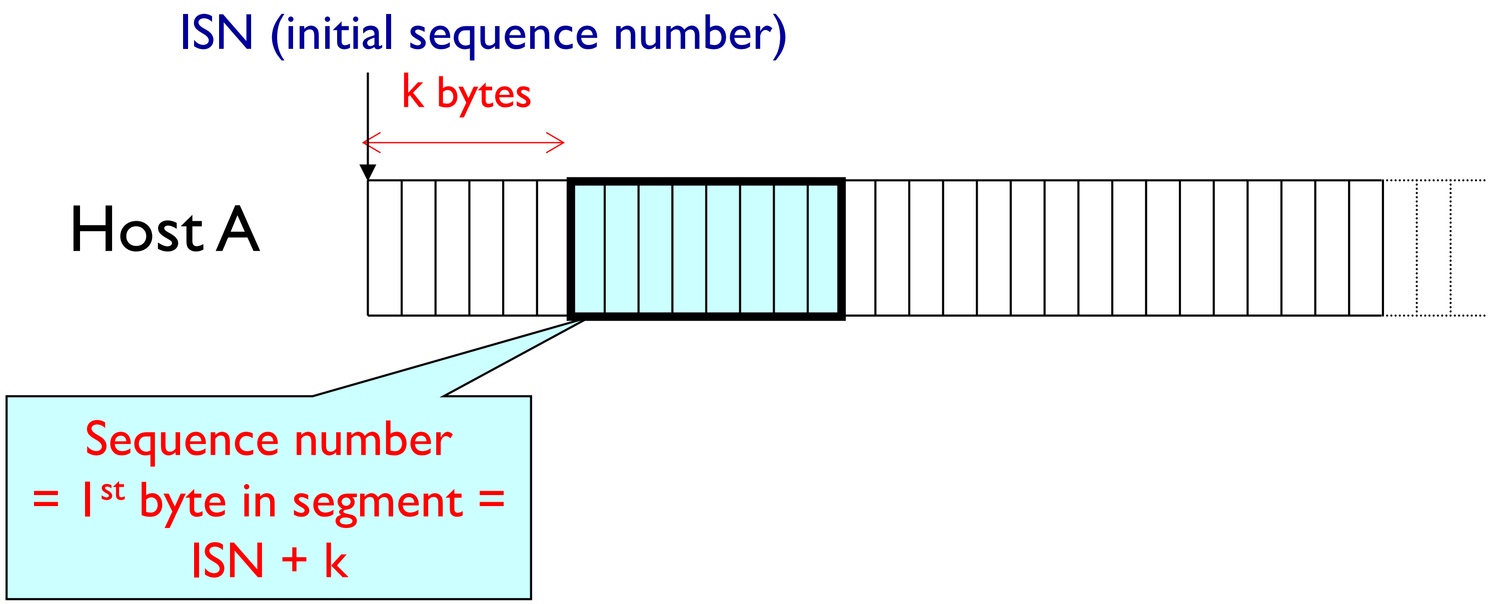
1. Segment full (Max Segment Size),
2. Not full, but instructed by the Application e.g., 1 byte in Telnet

TCP Maximum Segment Size



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

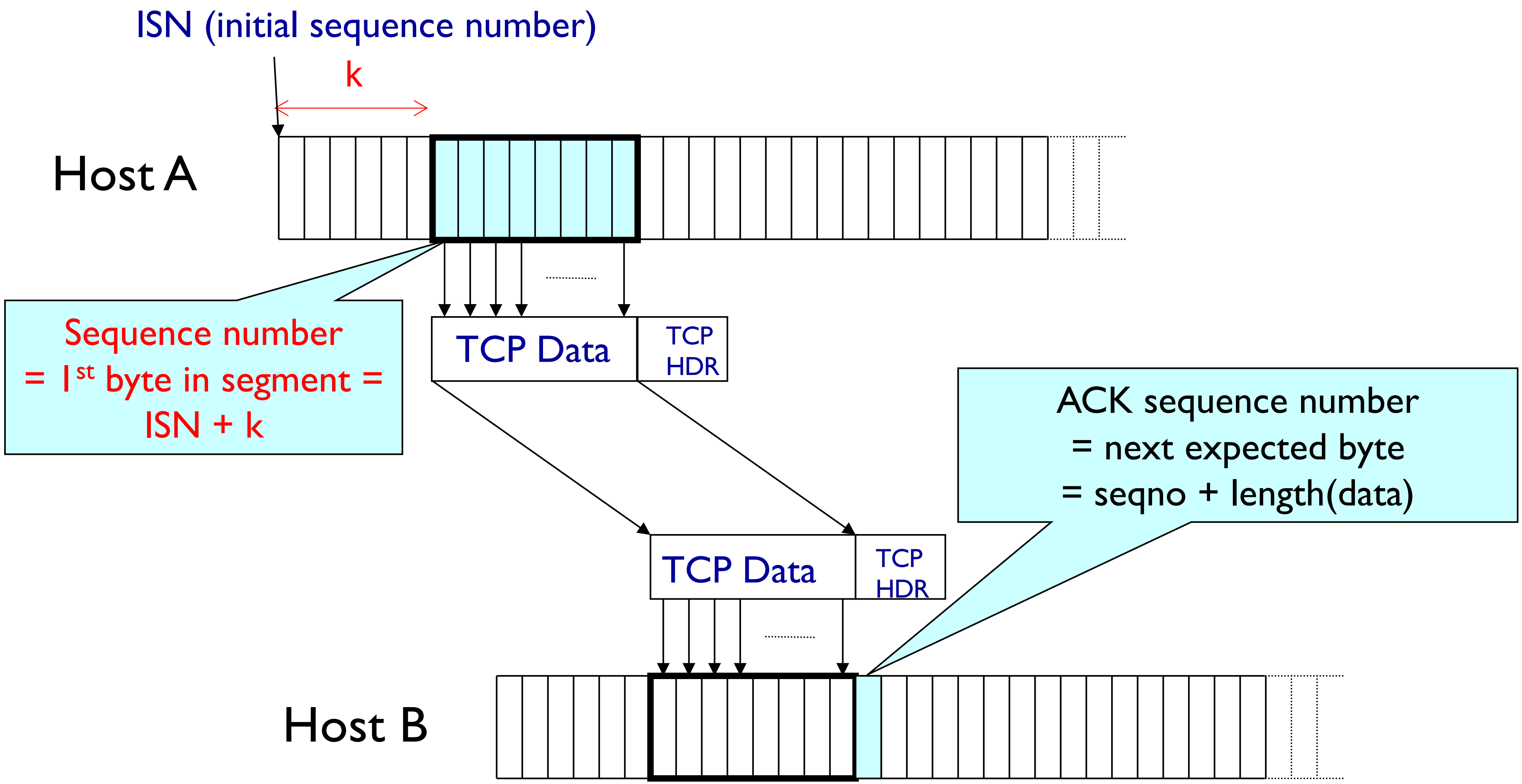
Sequence Numbers



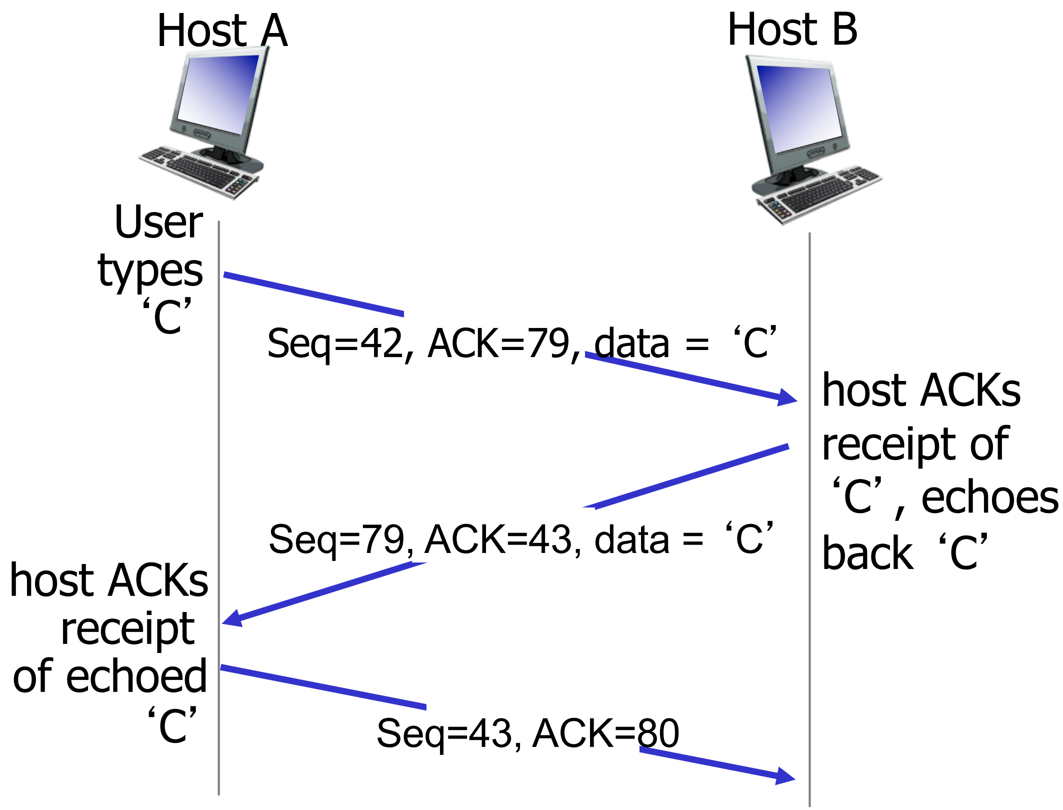
Sequence numbers:

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

Sequence & Ack Numbers



TCP seq. numbers, ACKs



simple telnet scenario

Transport Part 1: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - UDP
  - reliable data transfer
  - Pipelined Protocols for reliable data transfer

- **Next Week:**
  - TCP
    - TCP Flow Control
    - TCP Connection Management