Lecture 8:
Transport layer

Reading 6.2, 6.3, 6.4, 6.5

Computer Networks, Tanenbaum

Contents

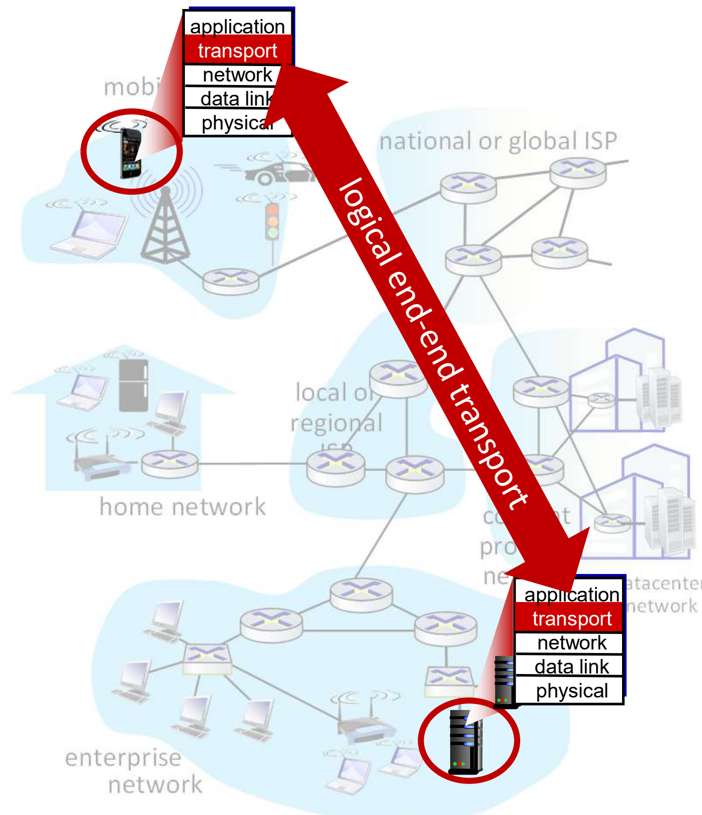
- Principles of transport layer
- UDP protocol
- Reliable data transfer
- TCP protocol

Transport layer in OSI model

Application (HTTP, Mail, …)	Support applications
Transport (UDP, TCP …)	**Transferring data between applications**
Network (IP, ICMP…)	Routing and forwarding data between hosts
Datalink (Ethernet, ADSL…)	
Physical (bits…)	

Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



Transport vs. network layer services and protocols

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes

Transport vs. network layer services and protocols

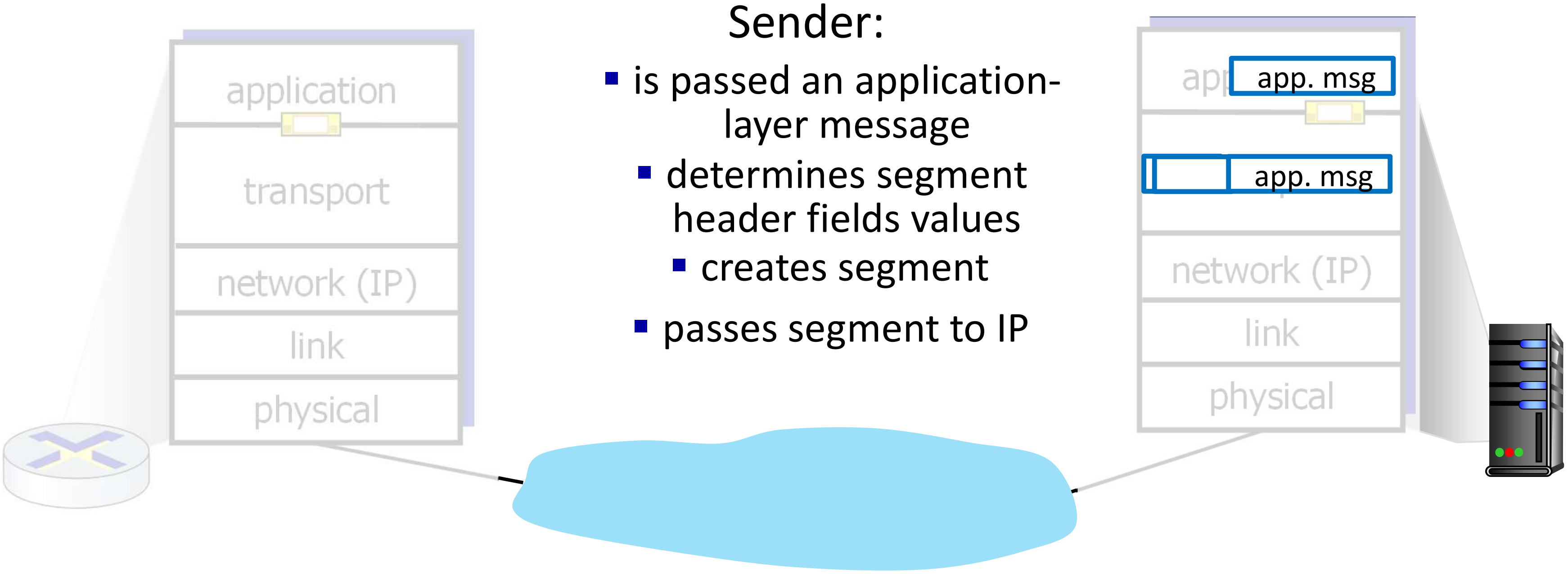
- network layer: logical communication between *hosts*
- transport layer*:* logical communication between *processes*
  - relies on, enhances, network layer services

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes

Transport Layer Actions



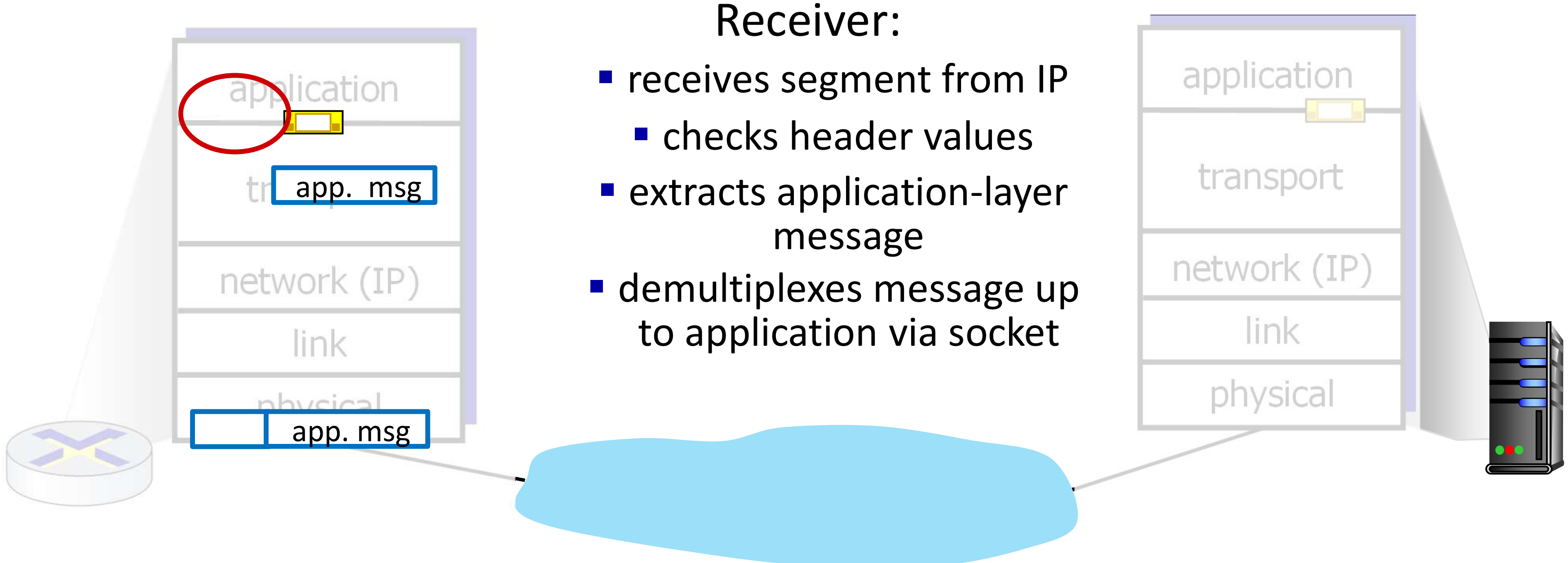
Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

Transport Layer Actions

Receiver:

- receives segment from IP
  - checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

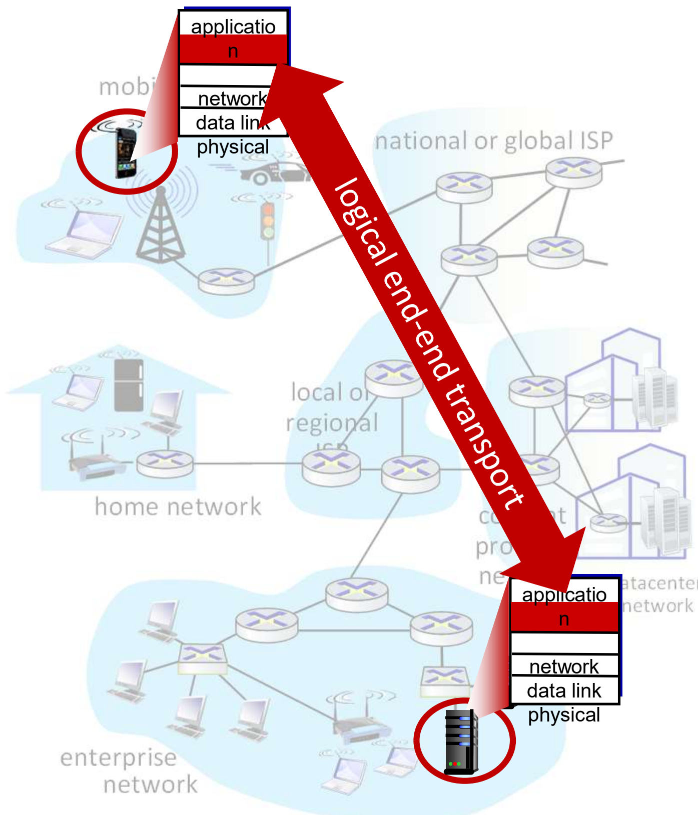


Why there are two kind of services?

- Various requirement about services from applications
- Applications that need 100% reliable data transfer, e.g. FTP, Mail…
  - Uses TCP (reliable) as transport services
- Application that need fast data transfer but can tolerate with packet lost, e.g. VoIP, Video Streaming
  - Uses UDP (best-effort) as transport services

Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Applications and transport services

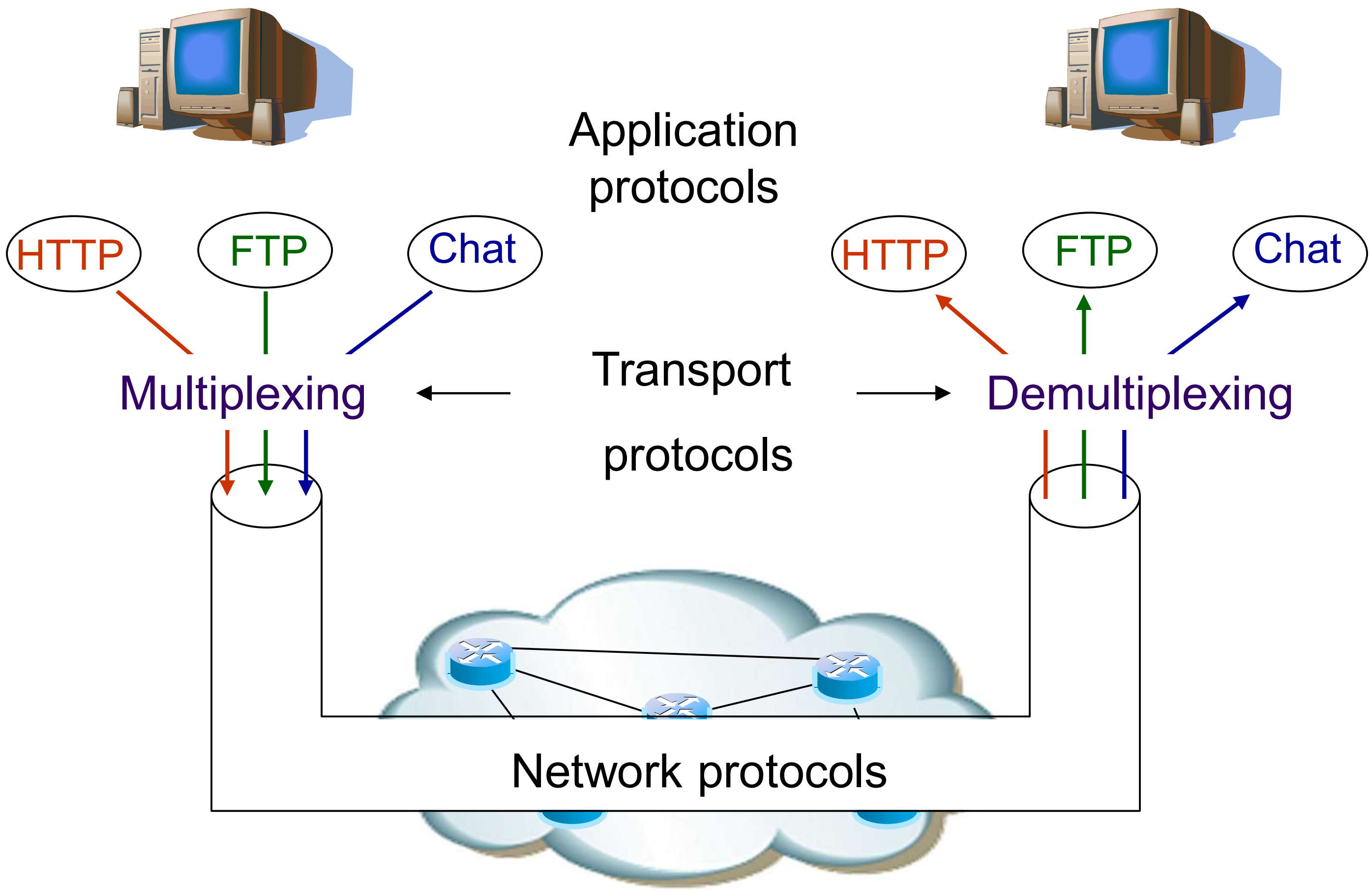
Application	Application protocols	Transport protocols
e-mail	SMTP	TCP
remote terminal access	Telnet	TCP
Web	HTTP	TCP
file transfer	FTP	TCP
streaming multimedia	Specific protocols (e.g. RealNetworks)	TCP or UDP
Internet telephony	Specific protocols (e.g., Vonage,Dialpad)	Usually UDP

Functionalities

MUX/DEMUX

Error control

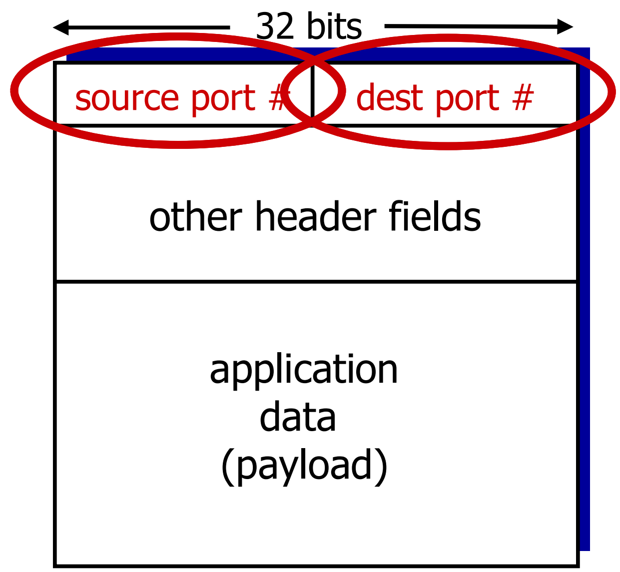
Mux/Demux



Application
protocols
HTTP
FTP
Chat
HTTP
FTP
Chat
Multiplexing
Transport
protocols
Demultiplexing
Network protocols

How demultiplexing works

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



TCP/UDP segment format

Connectionless demultiplexing

Recall:

- when creating socket, must specify *host-local* port #:

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

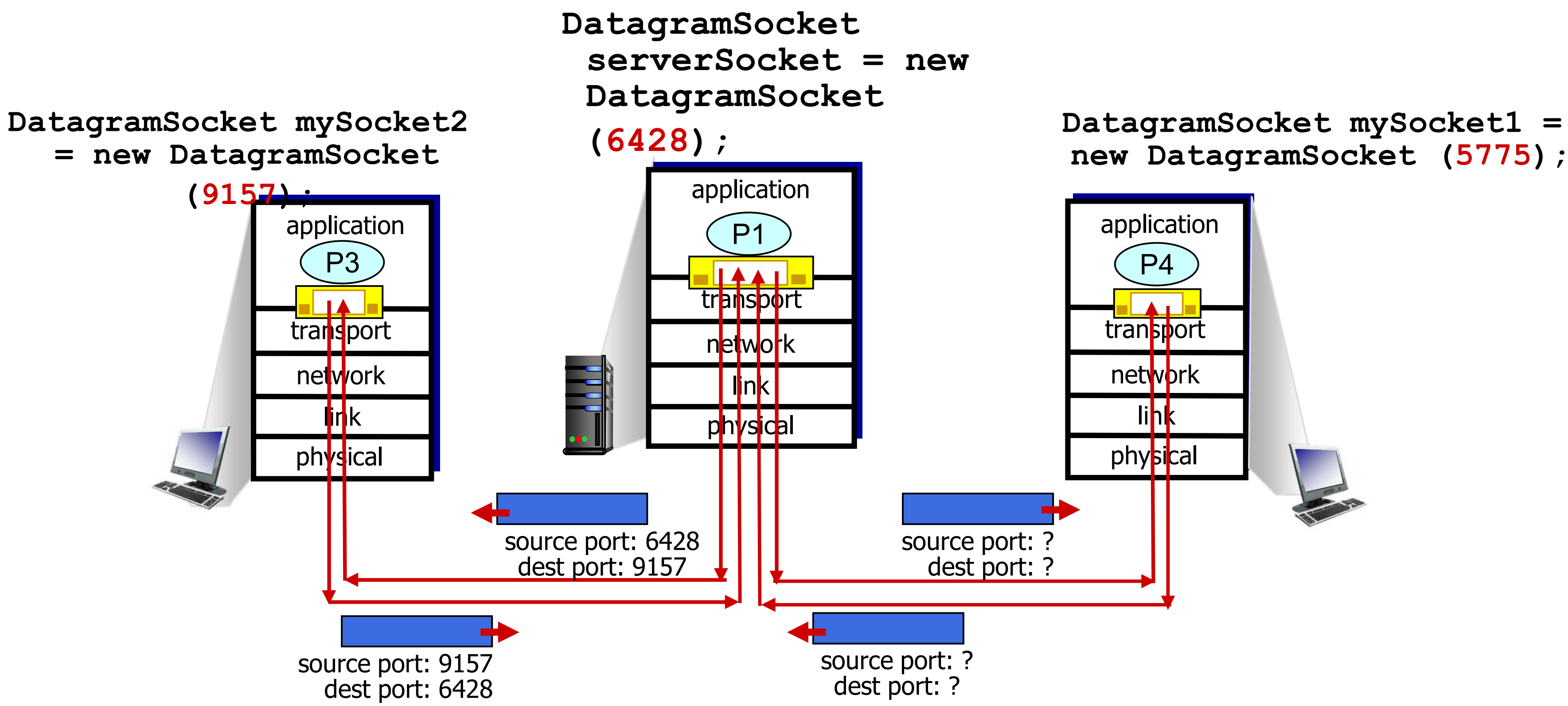
- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

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

Connectionless demultiplexing: an example

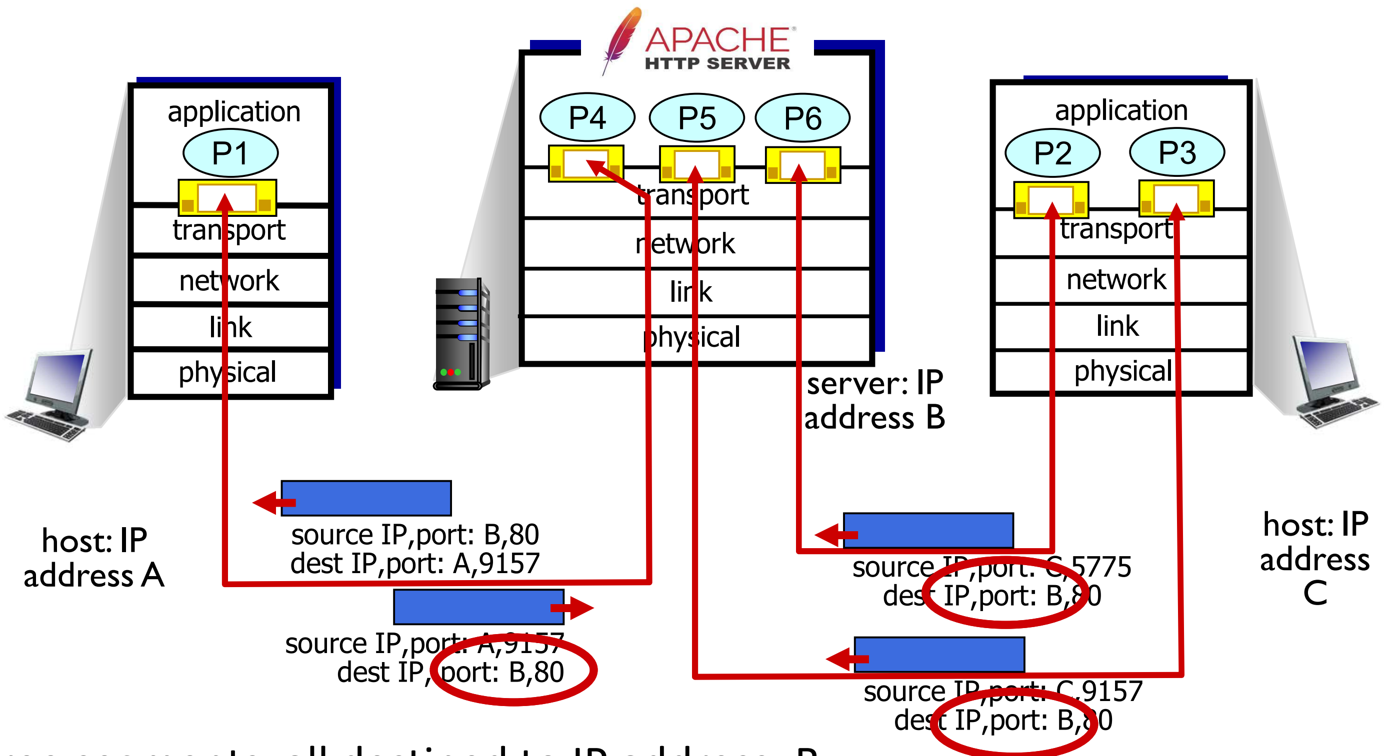


Connection-oriented demultiplexing

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

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

Connection-oriented demultiplexing: example



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

Error Control

- Use CRC or Checksum
- Checksum
  - Similar as checksum (16 bits) of IP
- Mechanism
  - Split data to 16-bit chunks
  - These chunks are then added, any generated carry is added back to the sum
  - Then, the 1's complement of the sum is performed and put in the checksum field

Example of checksum

	+ 1
	01110111
Frame 3:	+ 11110000
Partial Sum:	1 01100111
	+ 1
	01101000
Frame 4:	+ 11000011
Partial Sum:	1 00101011
	+ 1
Sum:	00101100
Checksum:	11010011

	+ 1
	01110111
Frame 3:	+ 11110000
Partial Sum:	1 01100111
	+ 1
	01101000
Frame 4:	+ 11000011
Partial Sum:	1 00101011
	+ 1
Sum:	00101100
Checksum:	11010011

Checksum

- Phát hiện lỗi bit trong các đoạn tin/gói tin
- Nguyên lý giống như checksum (16 bits) của giao thức IP
- Ví dụ:

		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
	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
Tổng		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

UDP
User Datagram Protocol

“Best effort” protocols

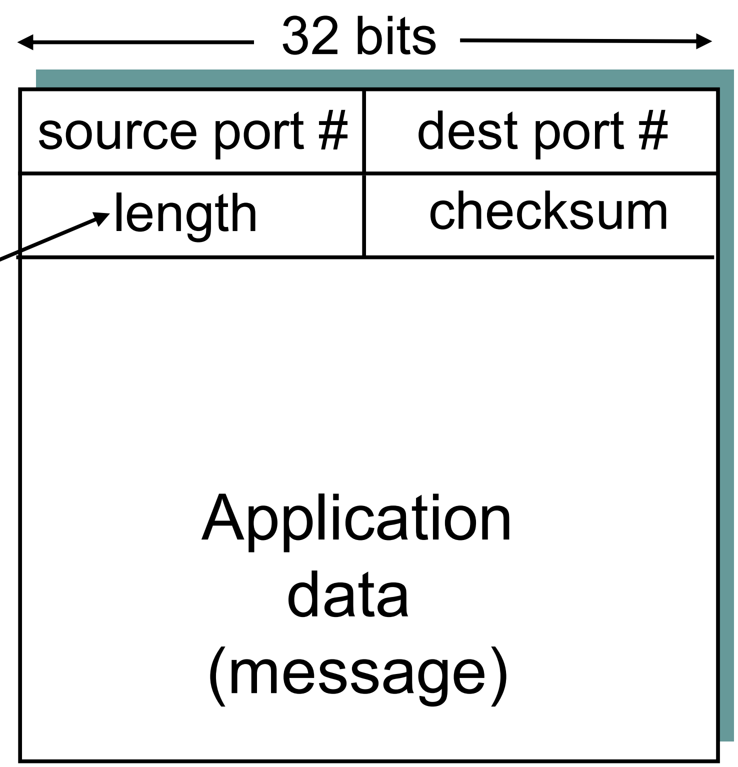
- Why UDP?
  - No need to establish connection (cause delay)
  - Simple
  - Small header
  - No congestion control → send data as fast as possible

- Main functionality of UDP?
  - MUX/DEMUX
  - Detect error by checksum

Datagram format

- Data unit in UDP is called datagram

Length of the datagram in byte



Internet checksum: an example

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
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

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet checksum: weak protection!

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
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

0 1
1 0

Even though numbers have changed (bit flips), *no* change in checksum!

Issues of UDP

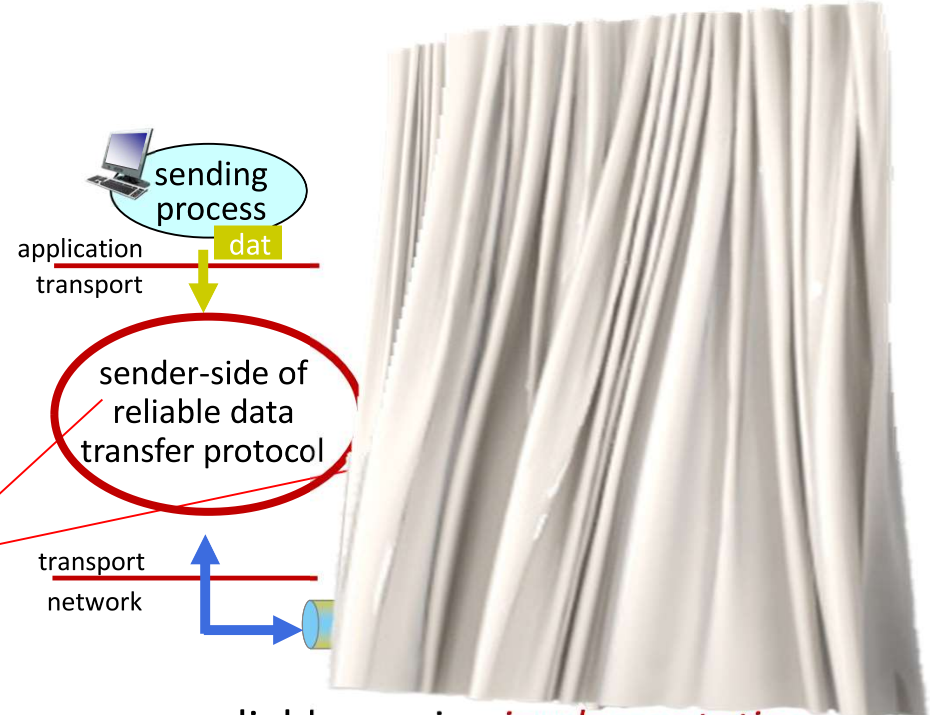
- No congestion control
  - Cause overload of the Internet
- No reliability
  - Applications have to implement themselves mechanisms to control errors

Summary: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Reliable data transfer

Principles of reliable data transfer



reliable service *implementation*

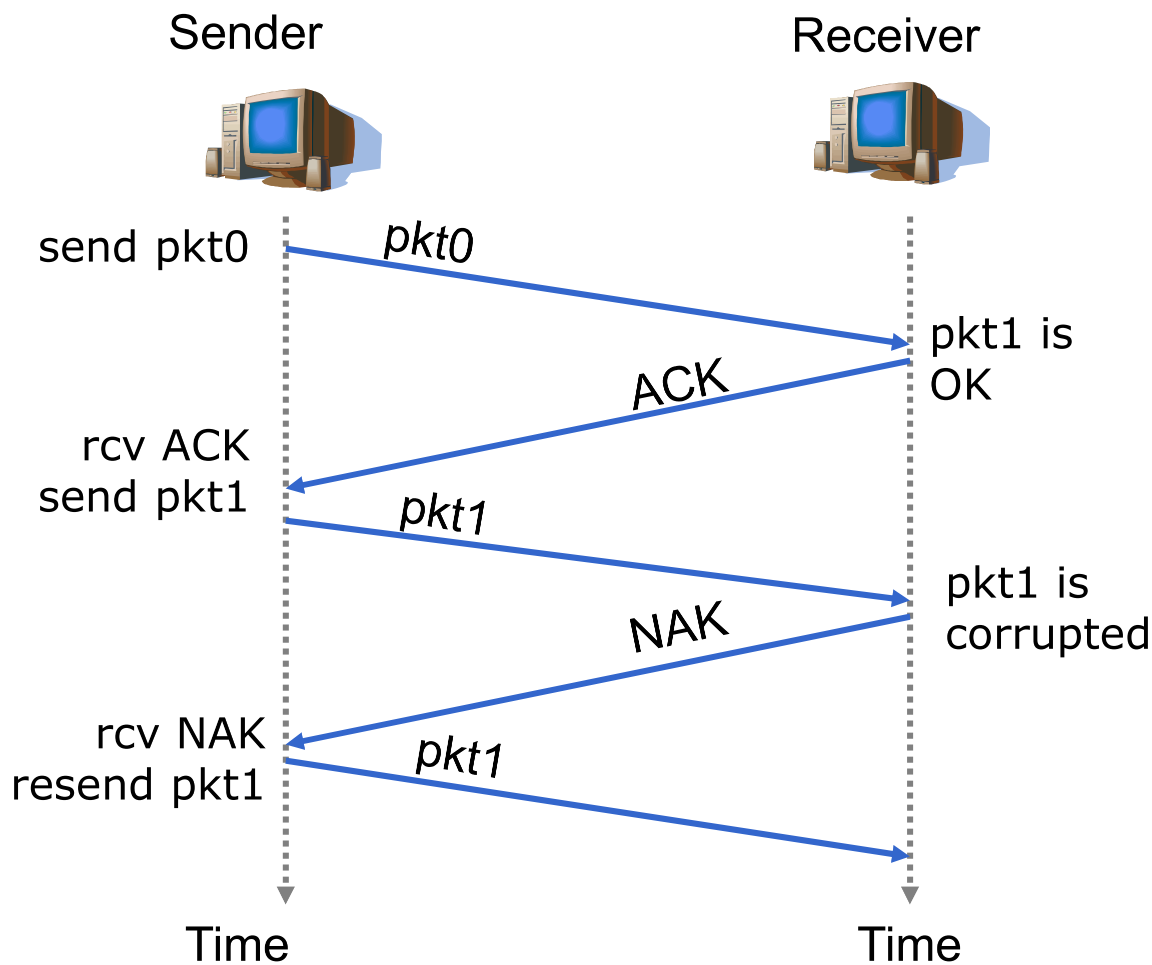
Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

- unless communicated via a message

Reliable data transfer

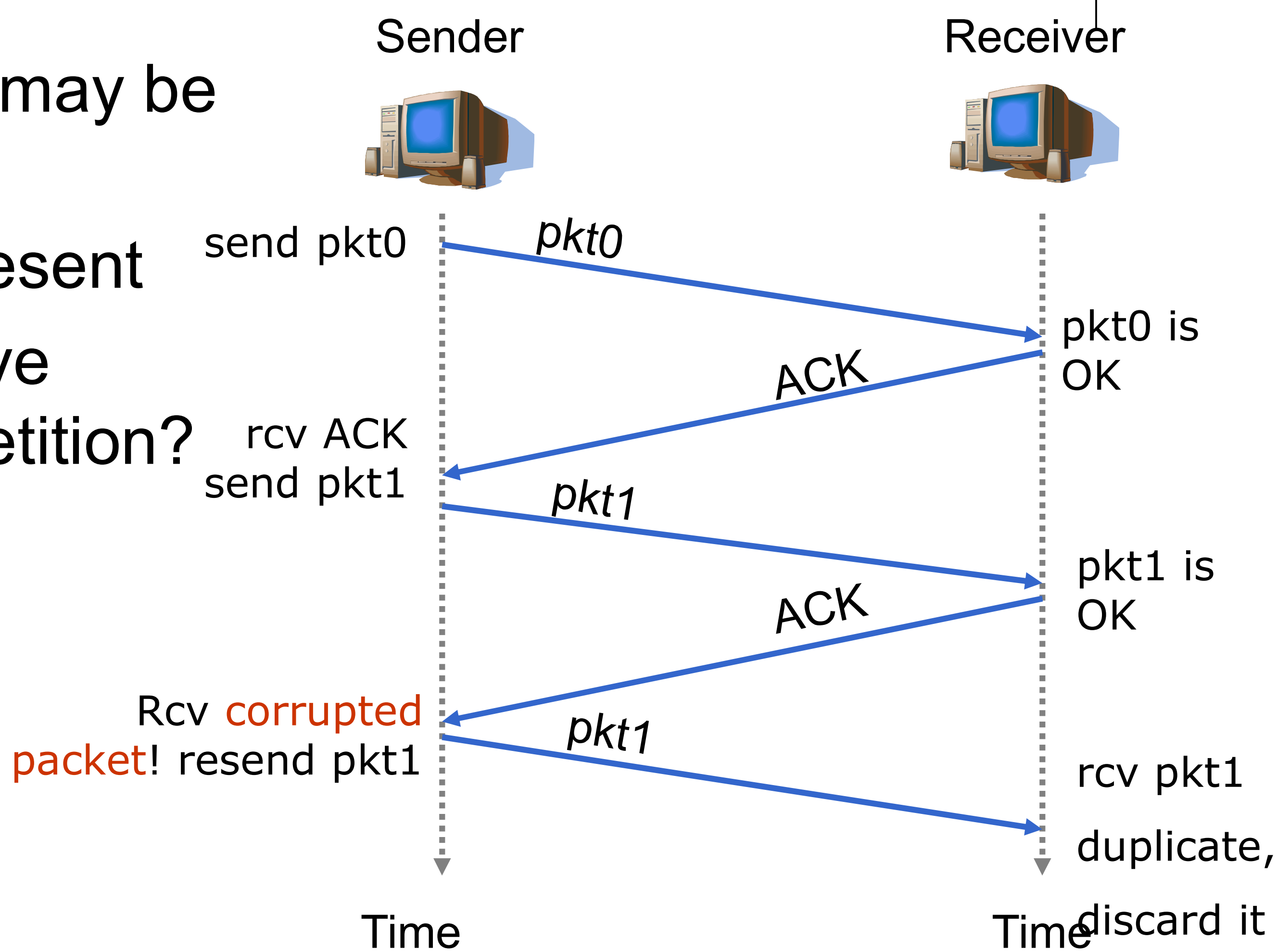
- How to detect error?
  - Checksum
- How to inform sender?
  - ACK (*acknowledgements*):
  - NAK (*negative acknowledgements*): tell sender that pkt has error
- Reaction of sender?
  - Retransmit the error packet once received NAK

Error control

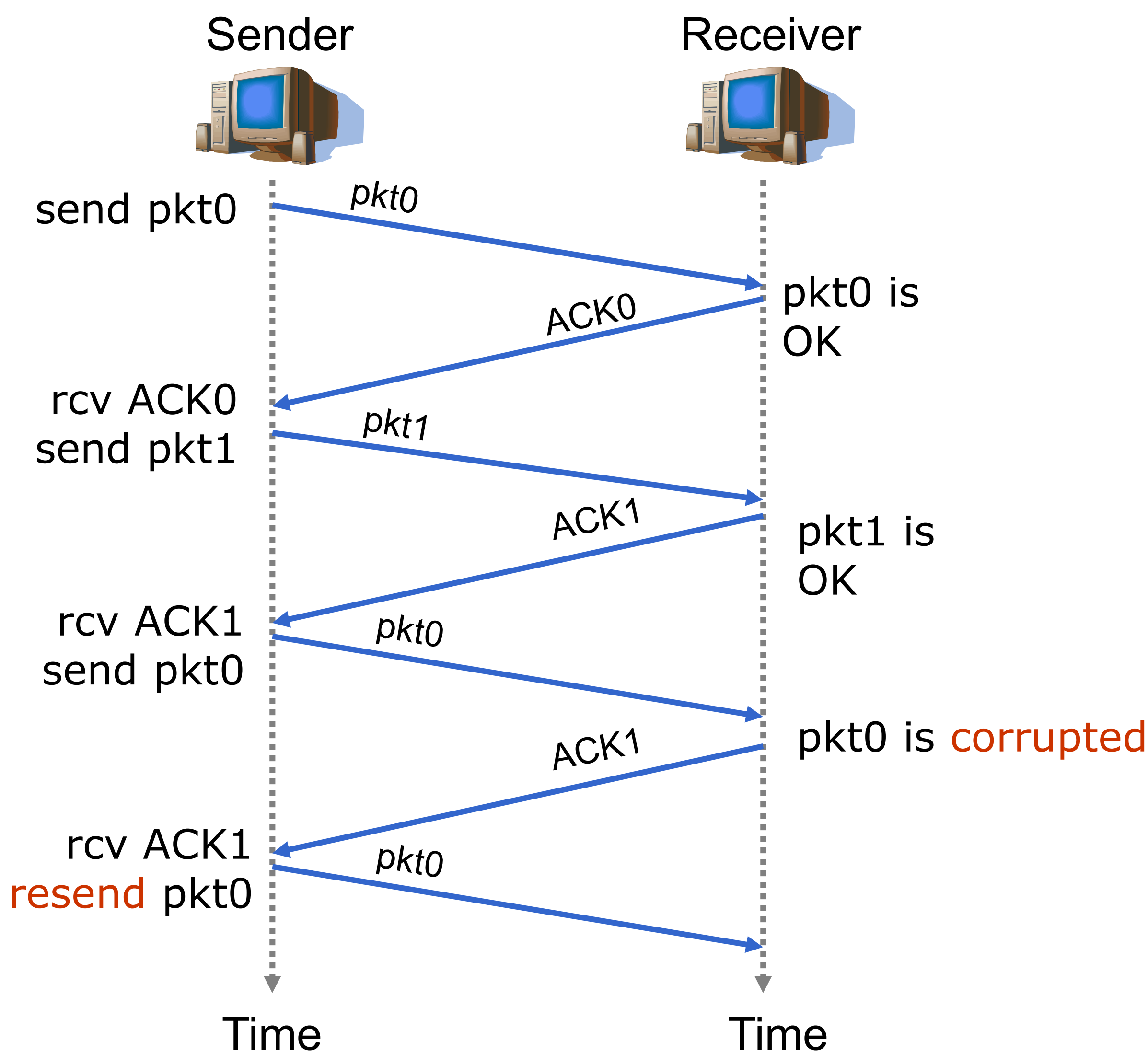


Error in ACK/NAK

- ACK/ NAK may be corrupted
- Packet is resent
- How to solve packet repetition?
- Use Seq.#



Error control without NAK



Chanel with error and packet lost

- Data and ACK can be lost
  - If no ACK is received? How sender knows and decides to resend data?
  - Sender should wait for ACK for a certain time. Timeout!
- How long should be timeout?
  - At least 1 RTT (Round Trip Time)
  - Need to start a timer each time sending a packet
- What if packet arrives and ACK is lost?
  - Packet should be numbered.

Illustration

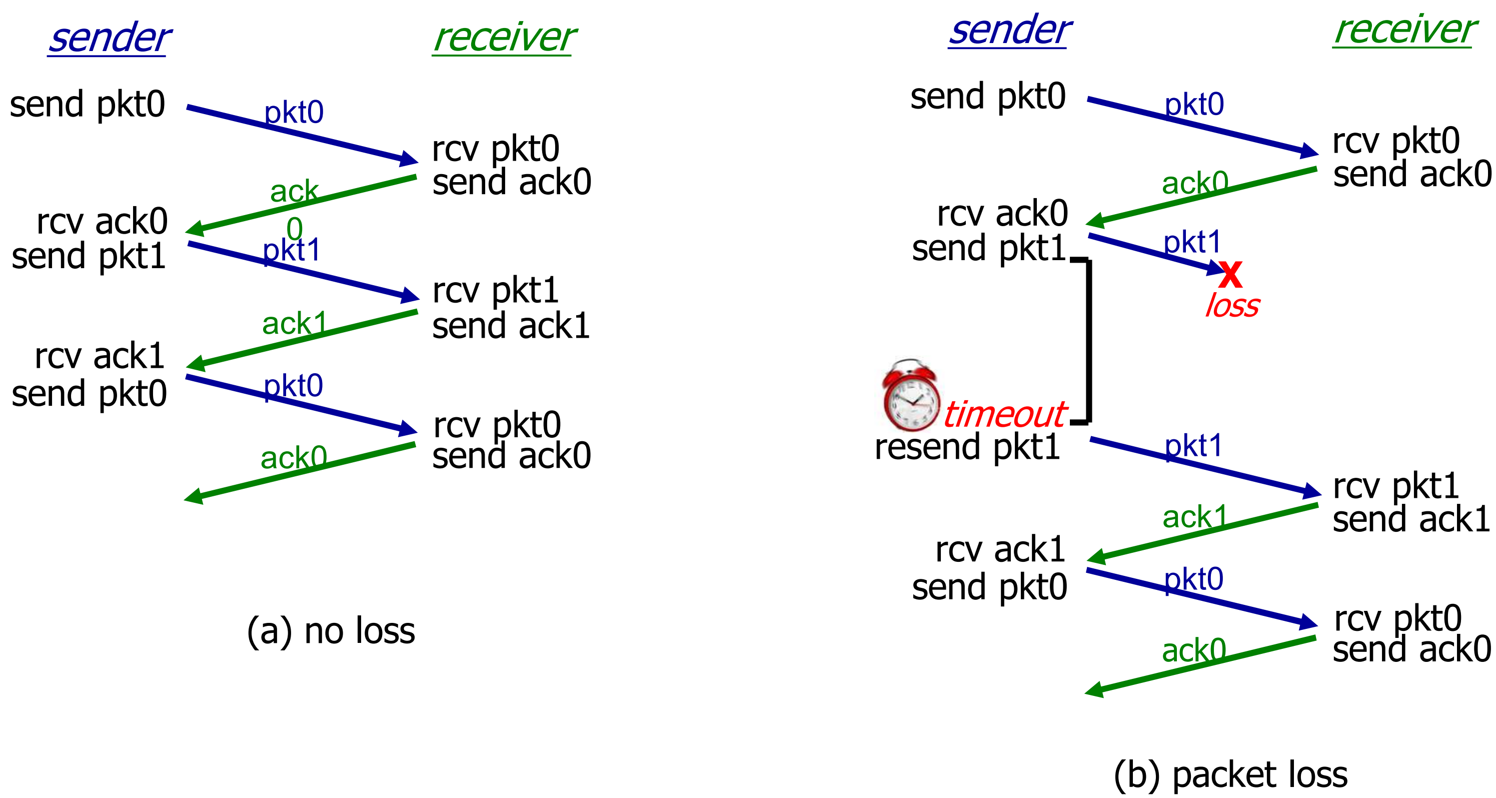
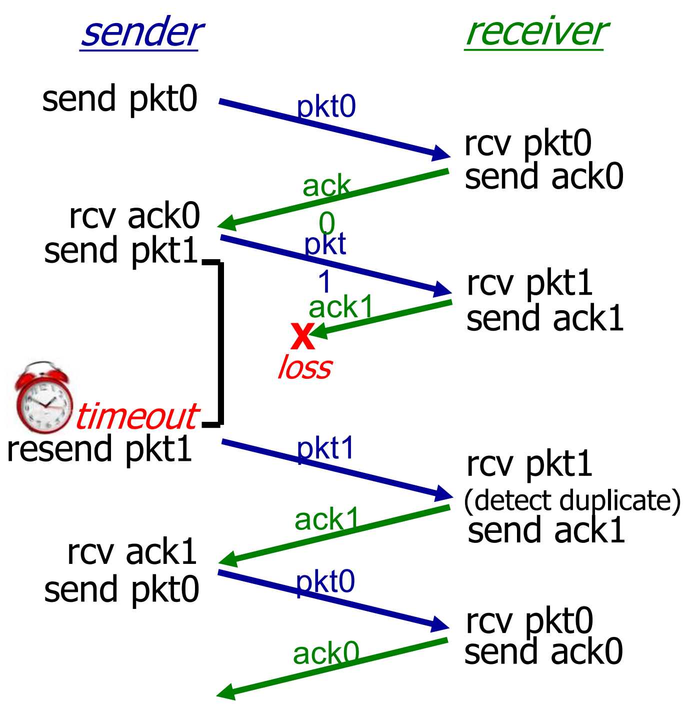
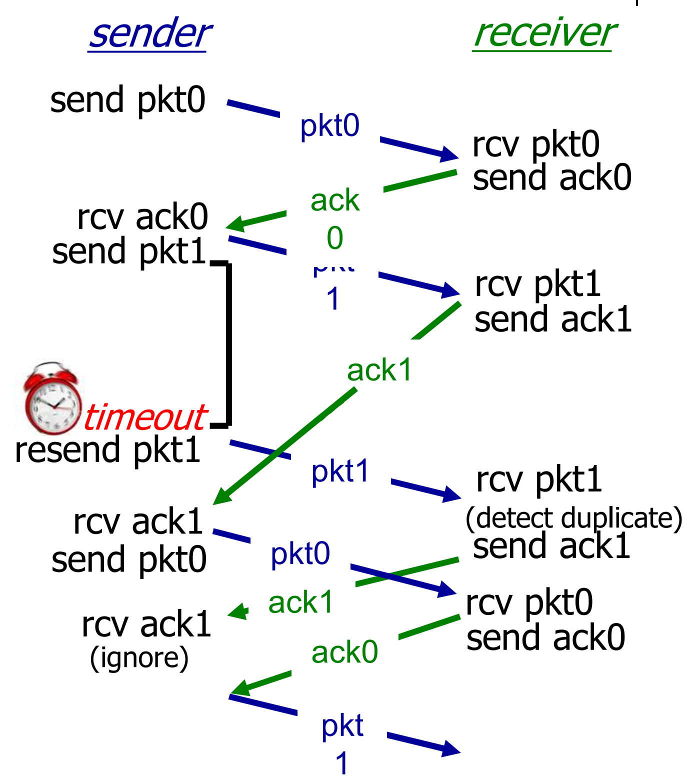


Illustration (2)

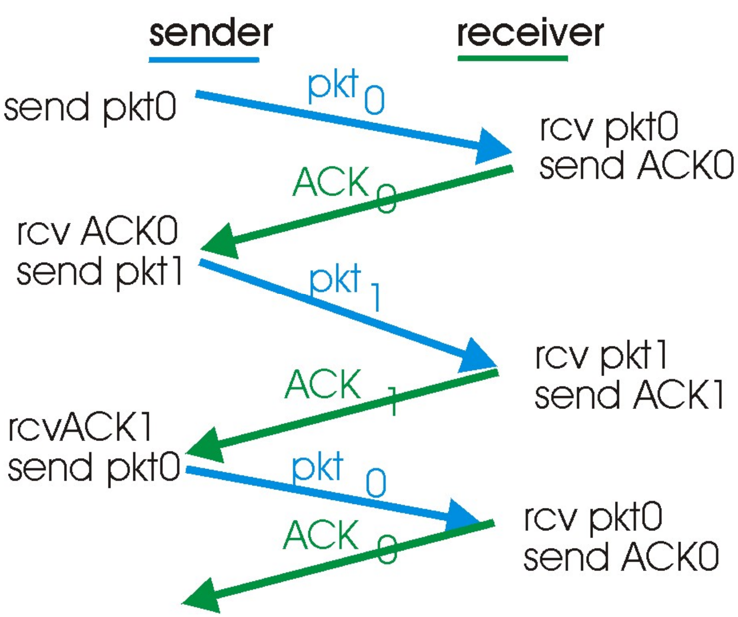


(c) ACK loss

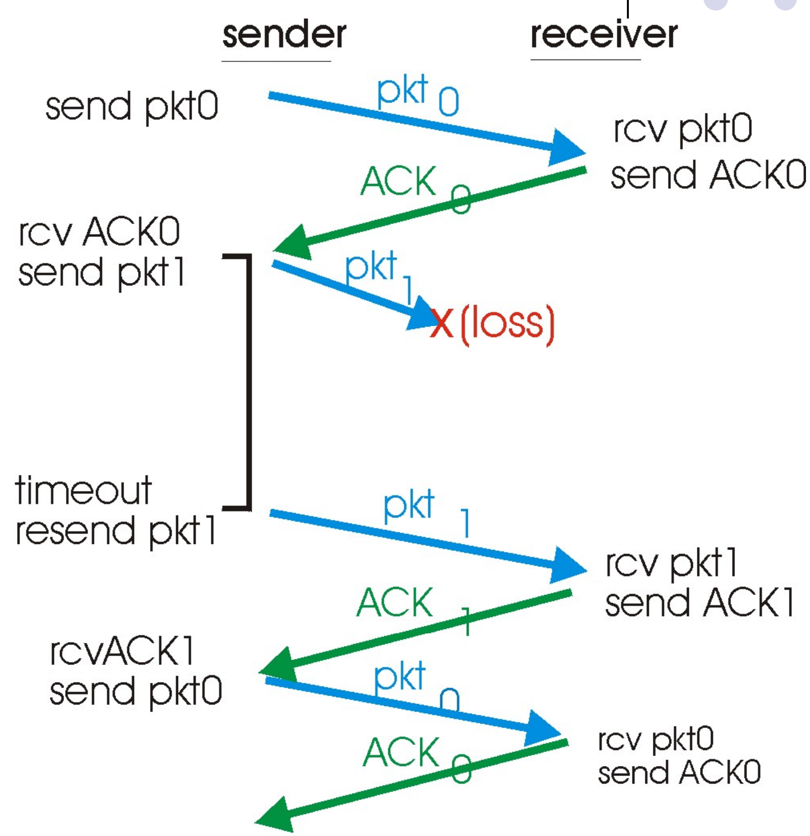


(d) premature timeout/ delayed ACK

Illustration



(a) operation with no loss



(b) lost packet

Performance of reliable data transfer (stop-and-wait)

- U_{sender}: *utilization* – fraction of time sender busy sending

- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

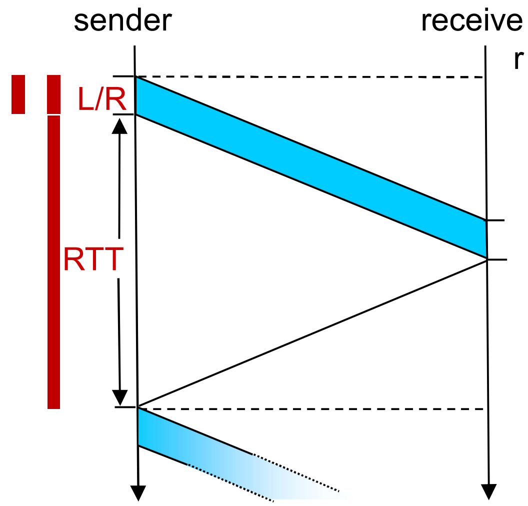
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

Reliable data transfer : stop-and-wait operation

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

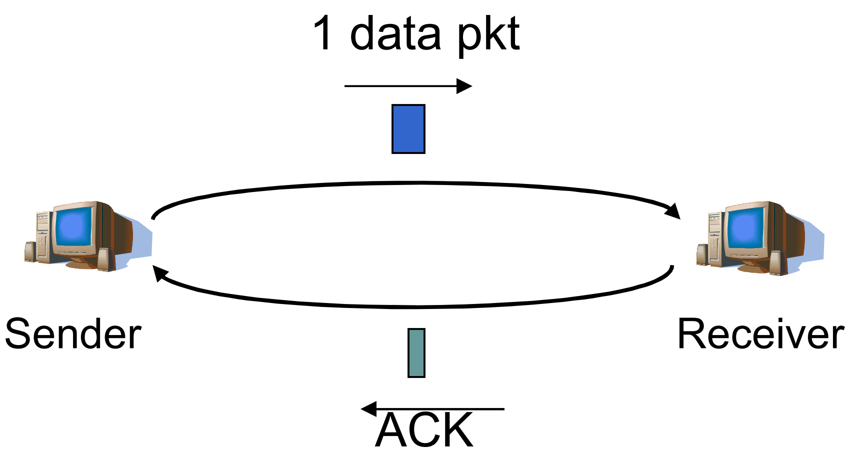
$$= \frac{.008}{30.008}$$

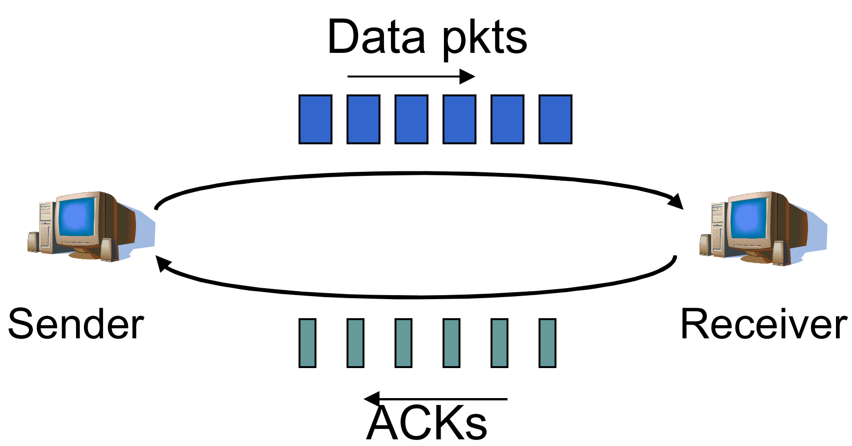
$$= 0.00027$$



- The performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

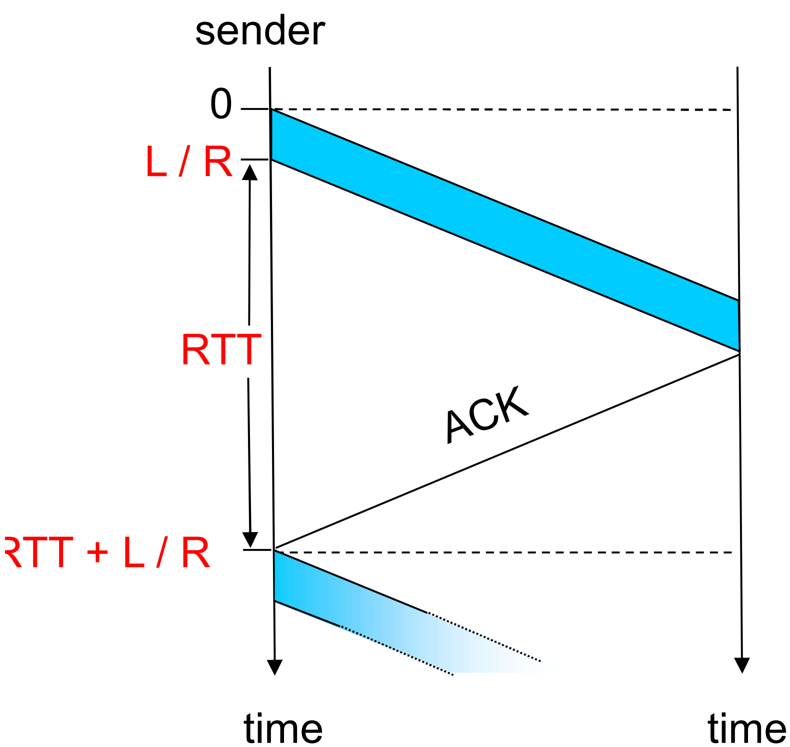
Transmission in pipeline



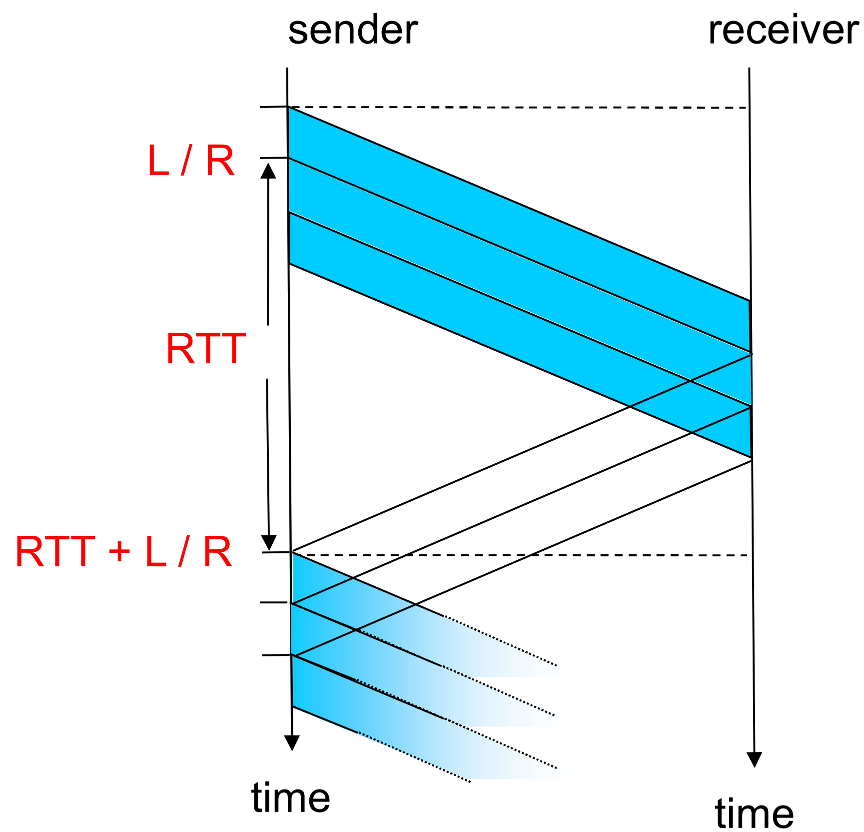


Comparison of efficiency

stop-and-wait



Pipeline

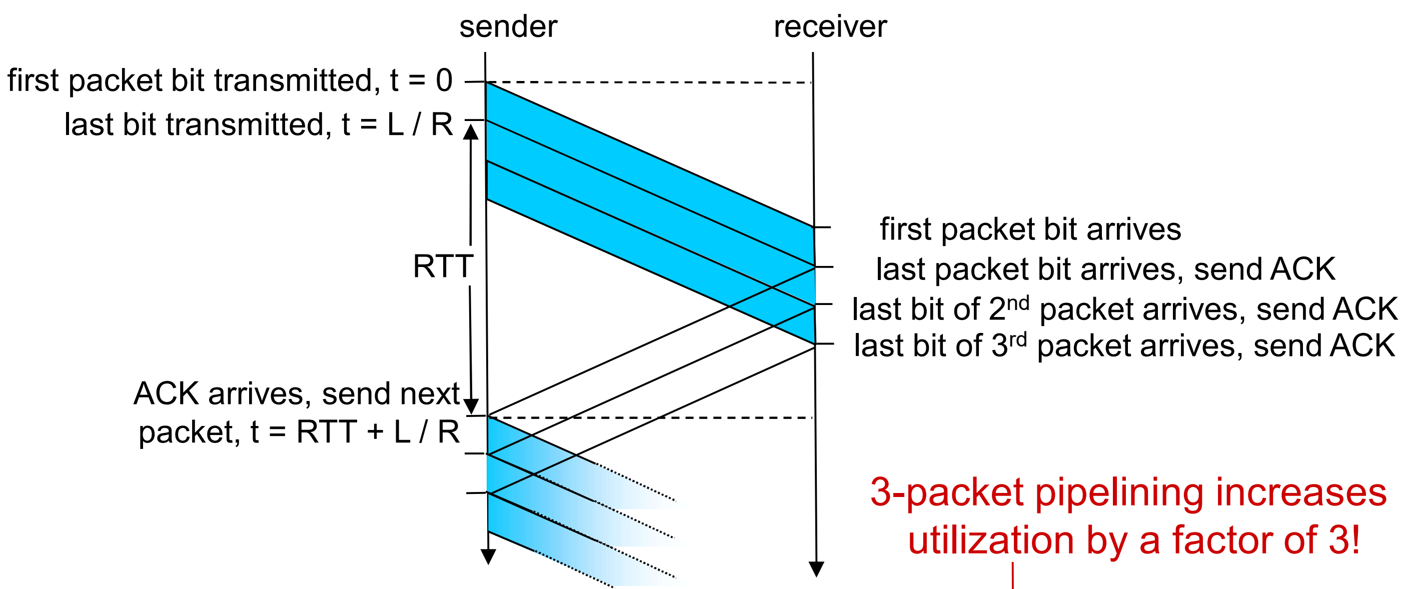


L:	Size of data pkt
R:	Link bandwidth
RTT:	Round trip time

$$\text{Performance} = \frac{L / R}{RTT + L / R}$$

$$\text{Performance} = \frac{3 * L / R}{RTT + L / R}$$

Pipelining: increased utilization



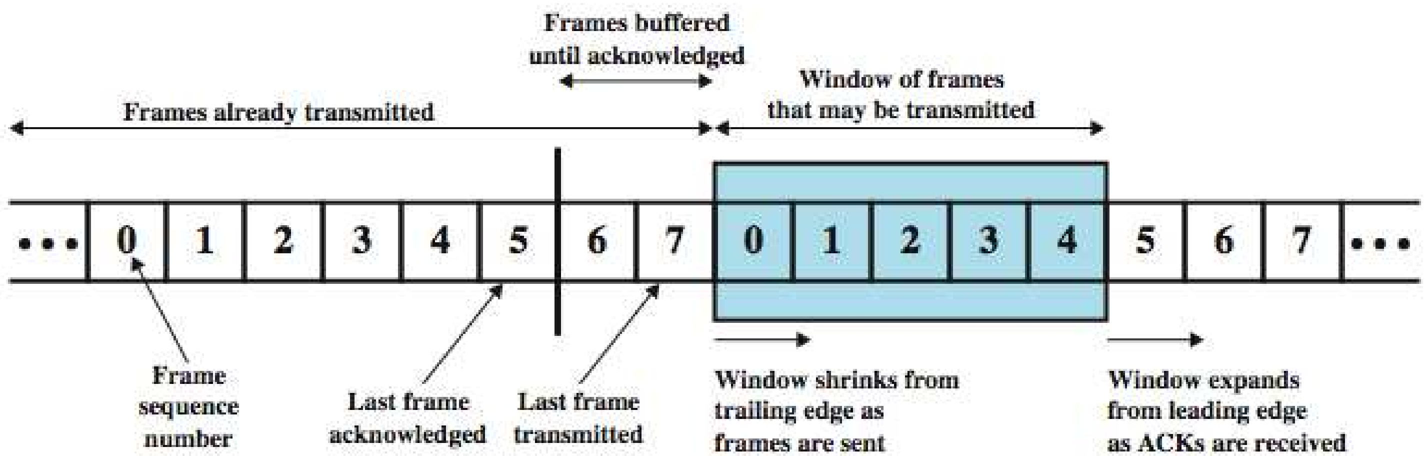
3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

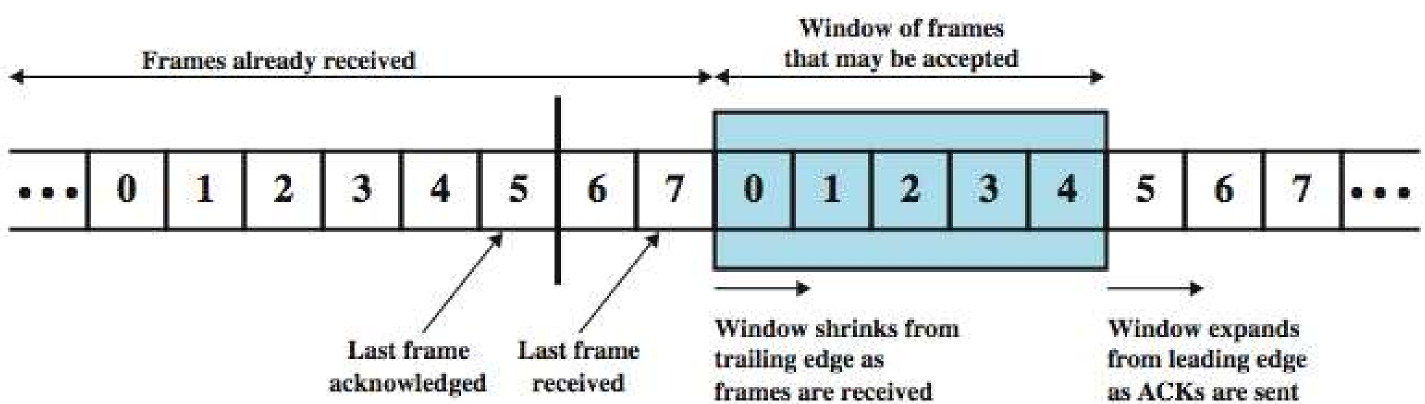
Sliding windows: mechanism

- Send multiple segments/frames simultaneously to reduce the waiting time
- Store transmitted frames while waiting for ACKs
- The number of transmitted frames is dependent on buffer
- After receiving ACK
  - Release the acknowledged (ACK) frame from the buffer
  - Send the next frame

Sliding window mechanism



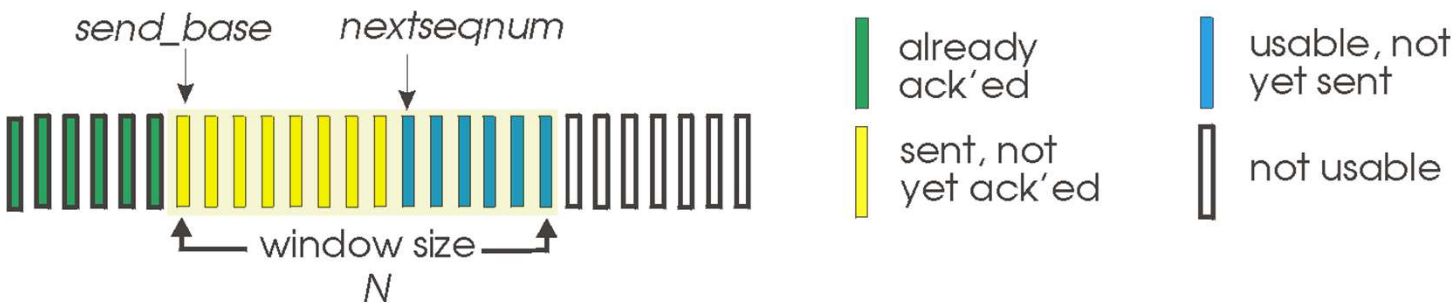
(a) Sender's perspective



(b) Receiver's perspective

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

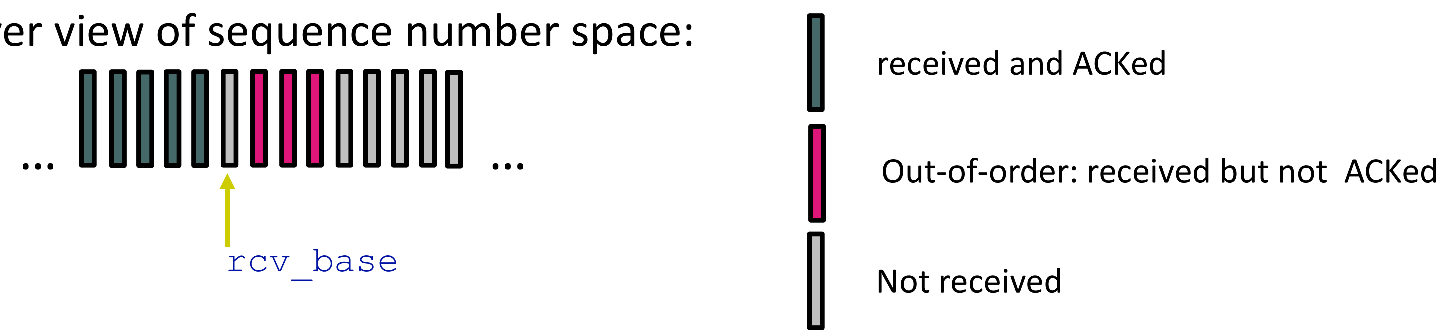


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

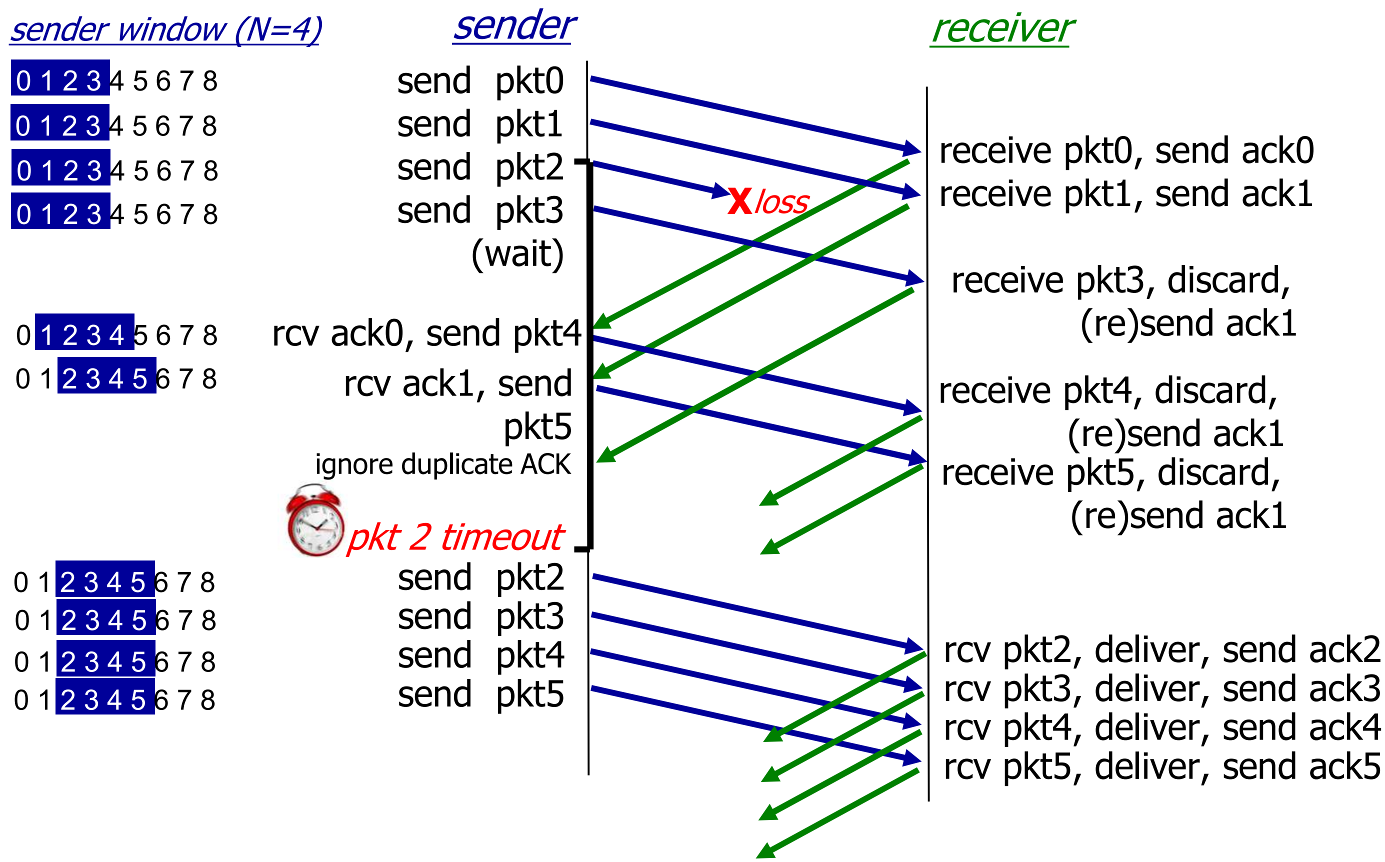
Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



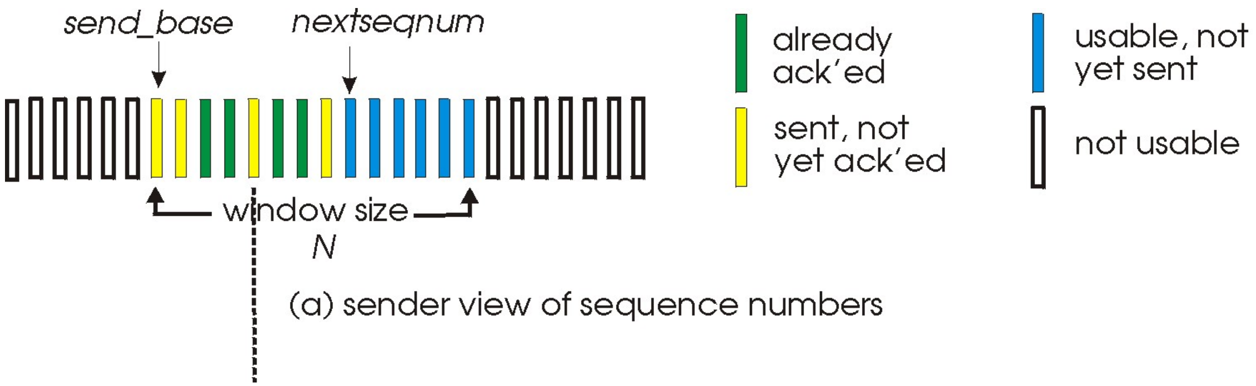
Go-Back-N in action



Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

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

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

receiver

packet *n* in [rcvbase, rcvbase+N-1]

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

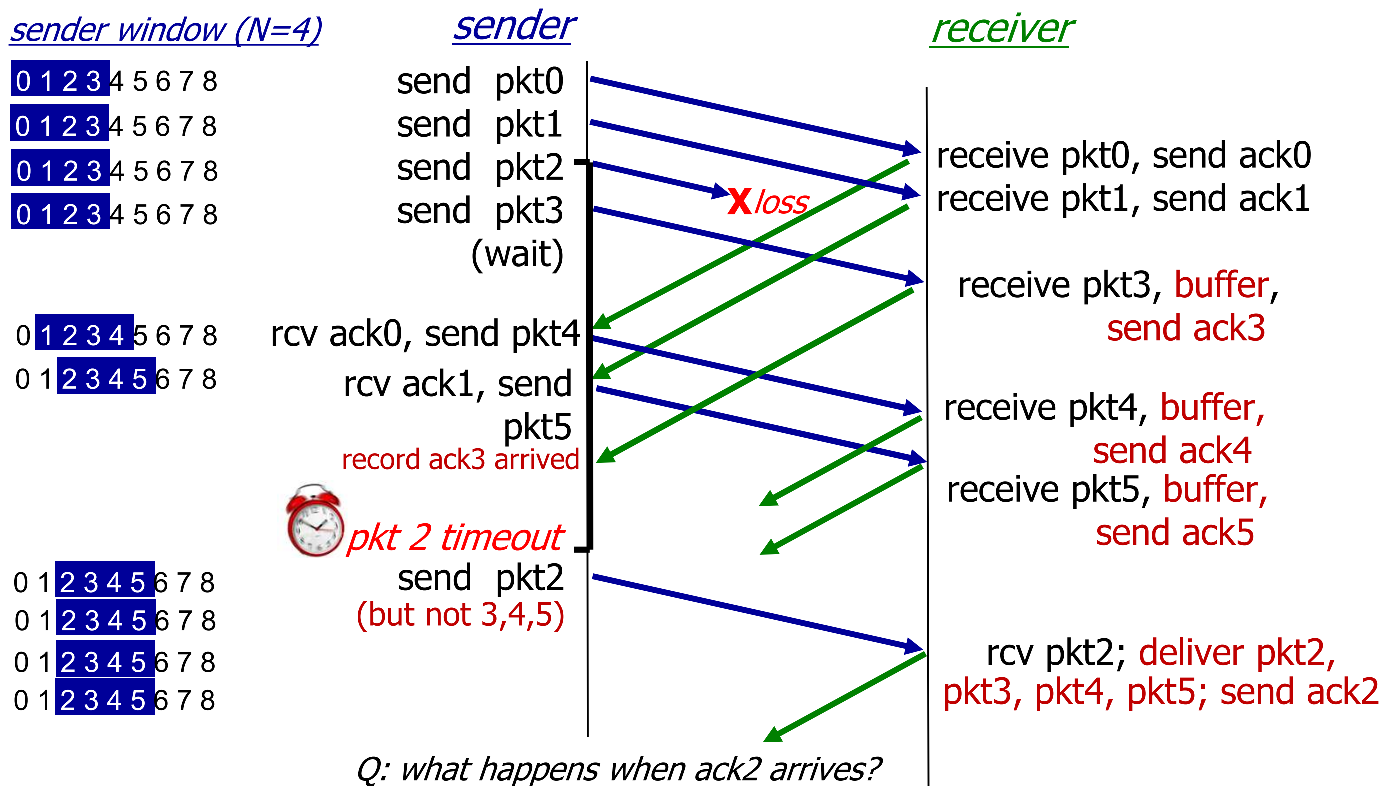
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

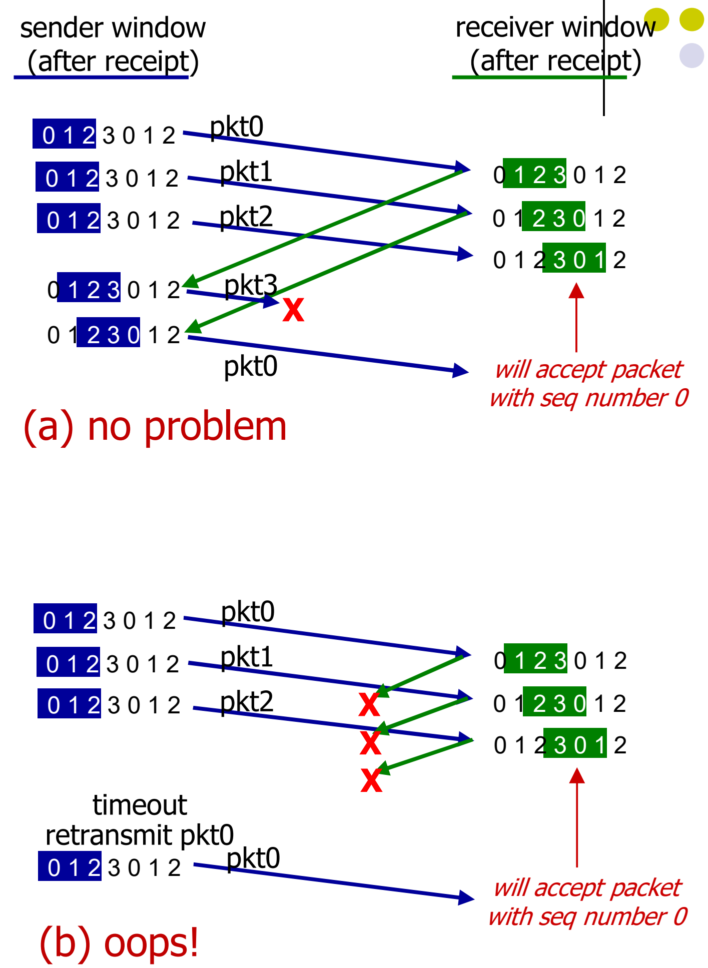
Selective Repeat in action



Selective repeat: a dilemma!

example:

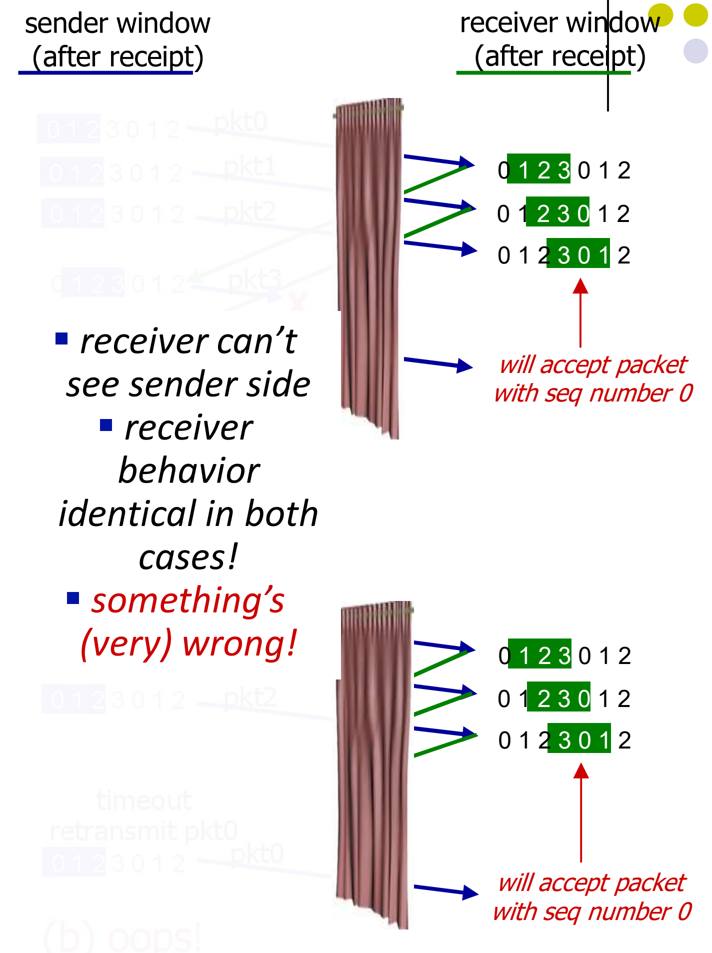
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



Selective repeat: a dilemma!

example:

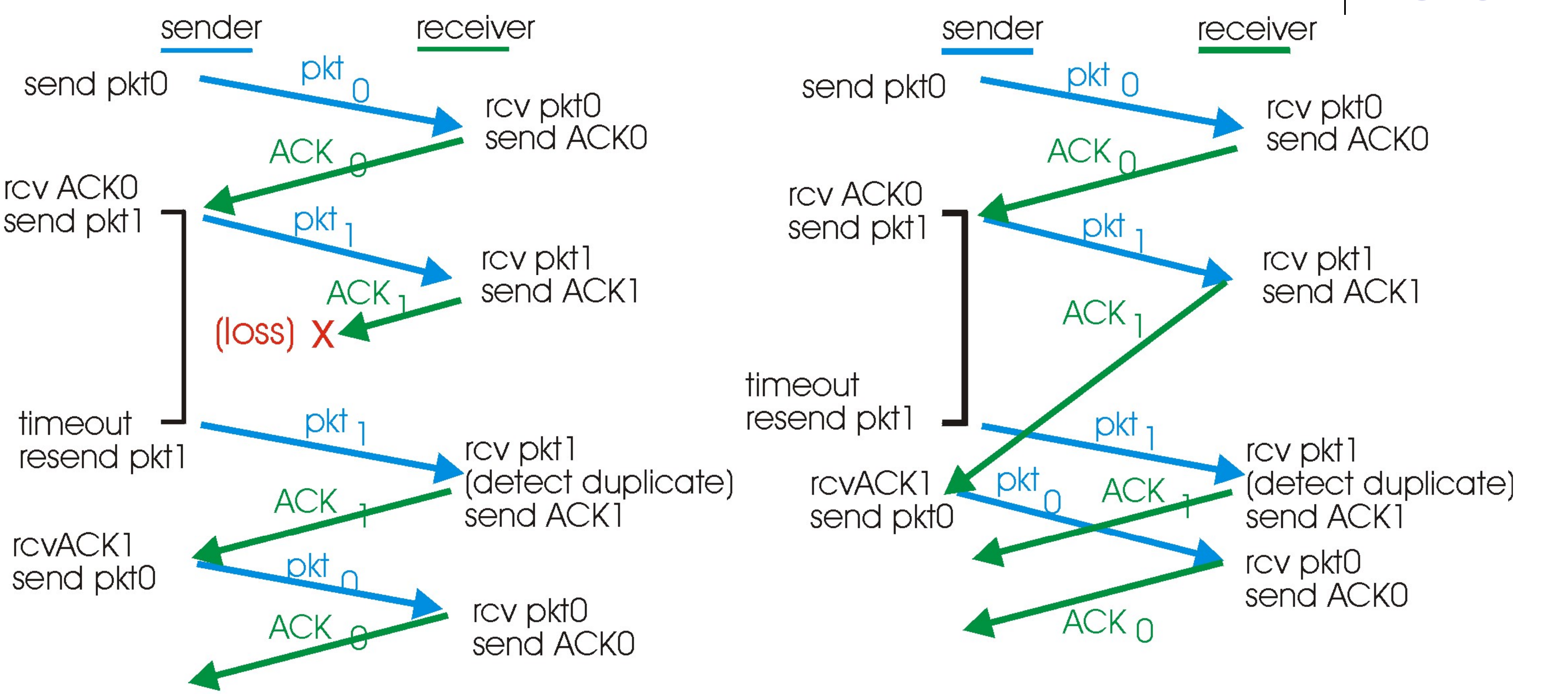
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



- *receiver can't see sender side*
  - *receiver behavior identical in both cases!*
- *something's (very) wrong!*

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

Illustration



(c) lost ACK

(d) premature timeout

TCP
Transmission Control Protocol

TCP segment structure

Connection management

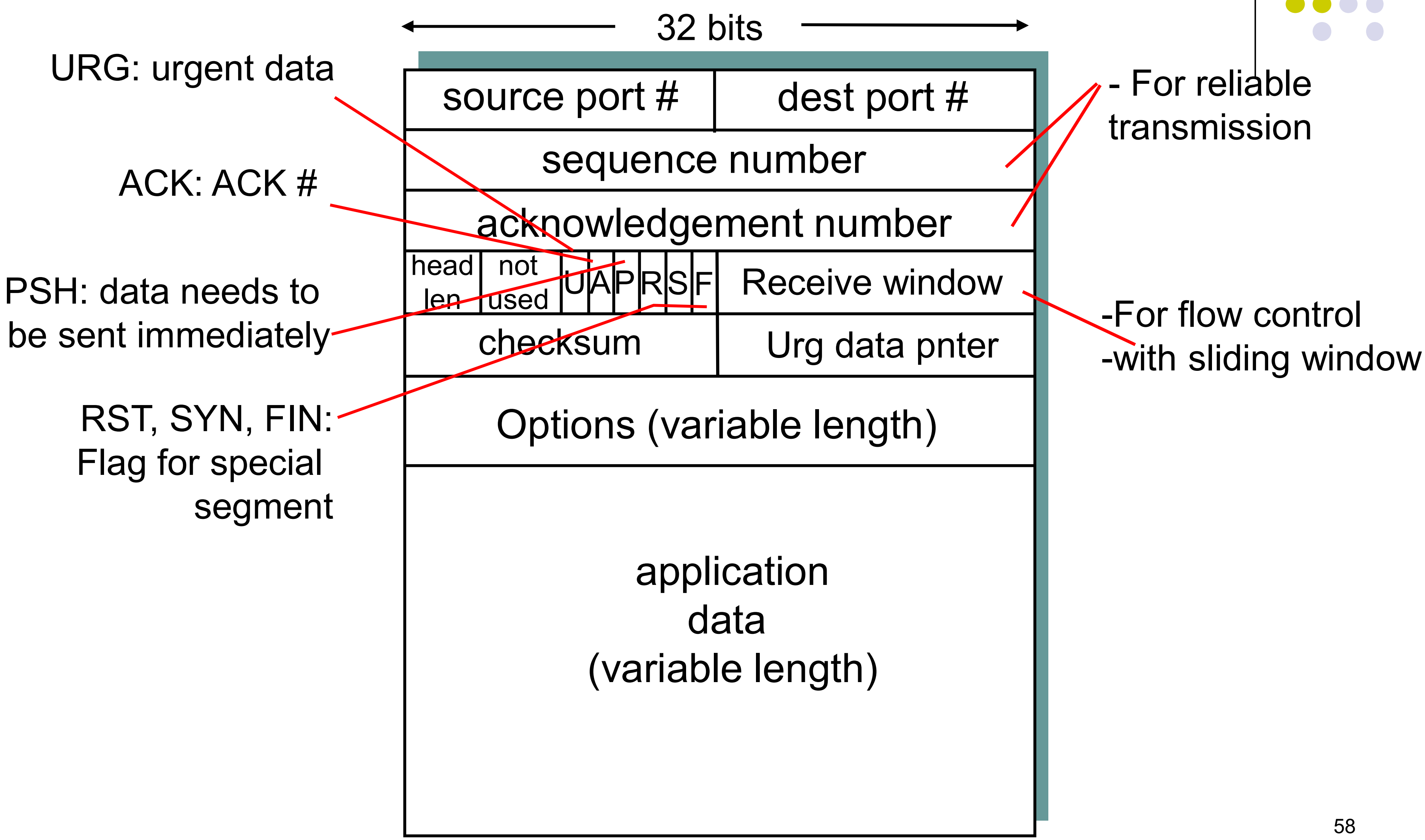
Flow control

Congestion control

Overview of TCP

- Connection oriented
  - 3 steps hand-shake
- Data transmission in stream of byte, reliable
  - Use buffer
- Transmit data in pipeline
  - Increase the performance
- Flow control
  - Sliding windows
- Congestion control
  - Detect congestion and solve

TCP segment



32 bits
URG: urgent data
ACK: ACK #
PSH: data needs to be sent immediately
RST, SYN, FIN: Flag for special segment
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
Receive window
checksum
Urg data pnter
Options (variable length)
application data (variable length)
- For reliable transmission
-For flow control
-with sliding window

How TCP provide reliable service?

- In order to assure if data arrives to destination:
  - Seq. #
  - Ack
- TCP cycle life:
  - Connection establishing
    - 3 steps
  - Data transmission
  - Close connection

TCP sequence numbers, ACKs

Sequence numbers:

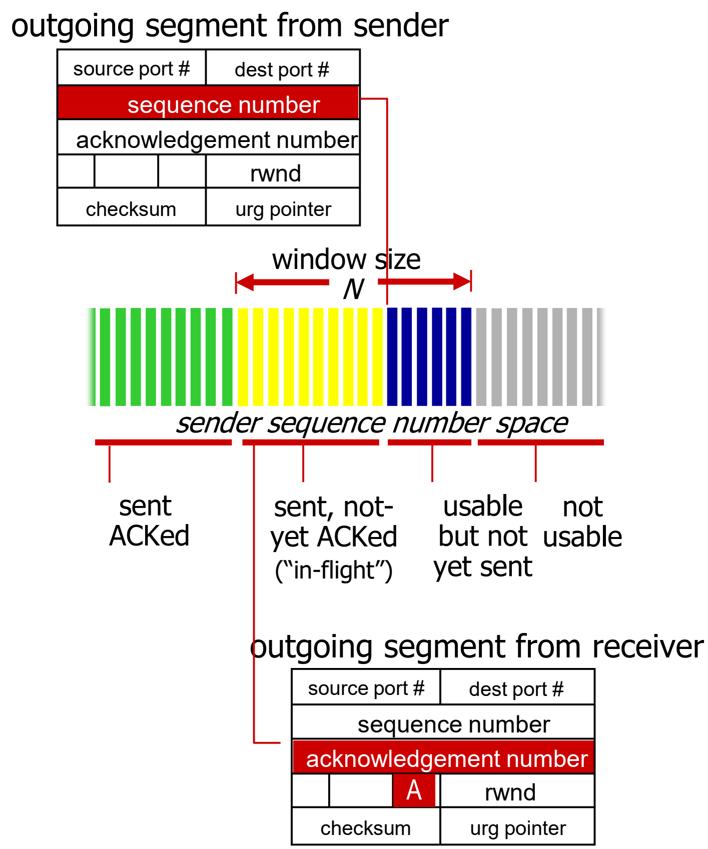
- byte stream "number" of first byte in segment's data

Acknowledgements:

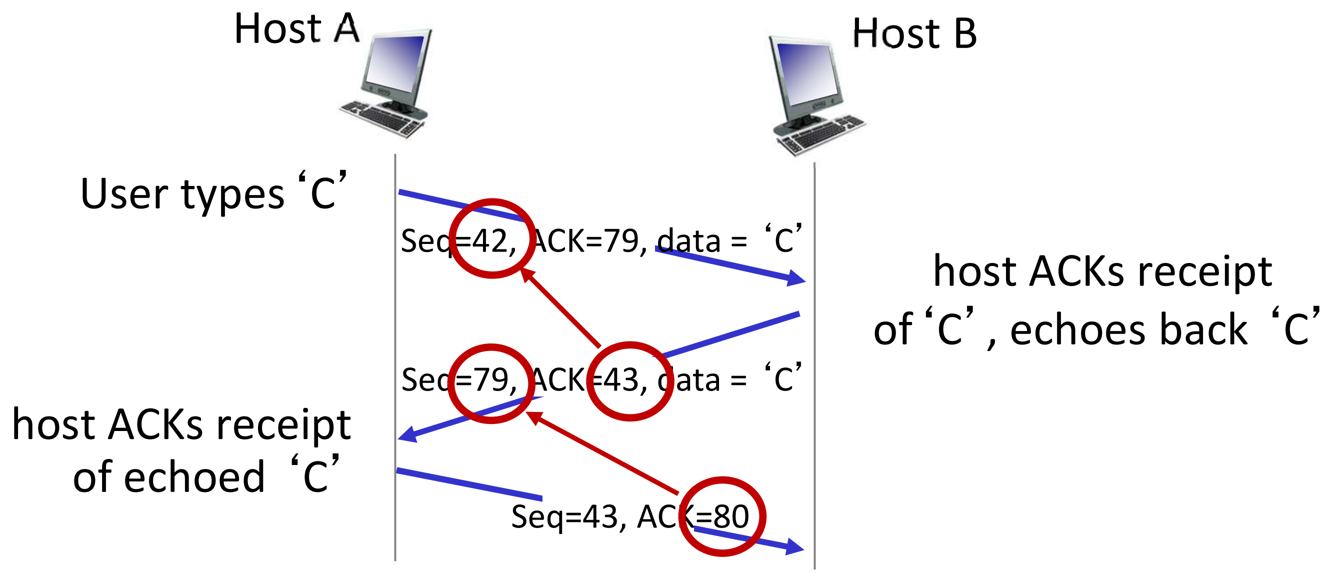
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor

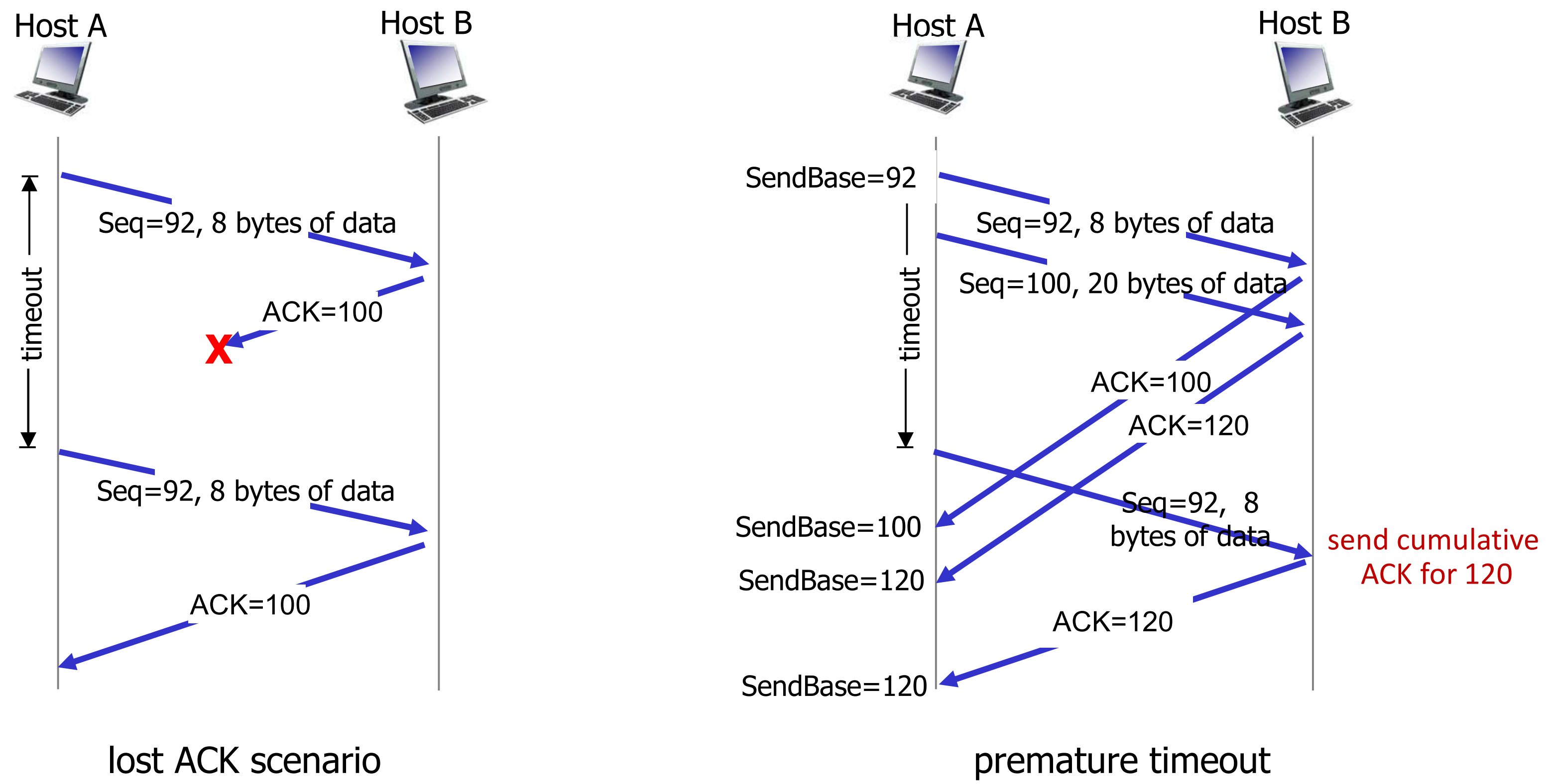


TCP sequence numbers, ACKs

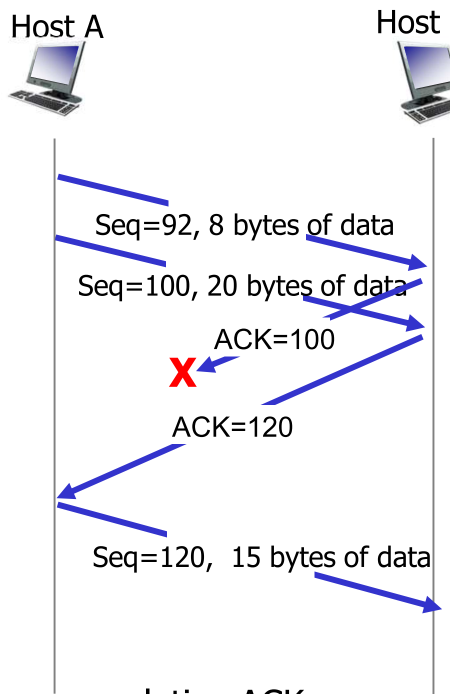


simple telnet scenario

TCP: retransmission scenarios



TCP: retransmission scenarios



cumulative ACK covers
for earlier lost ACK

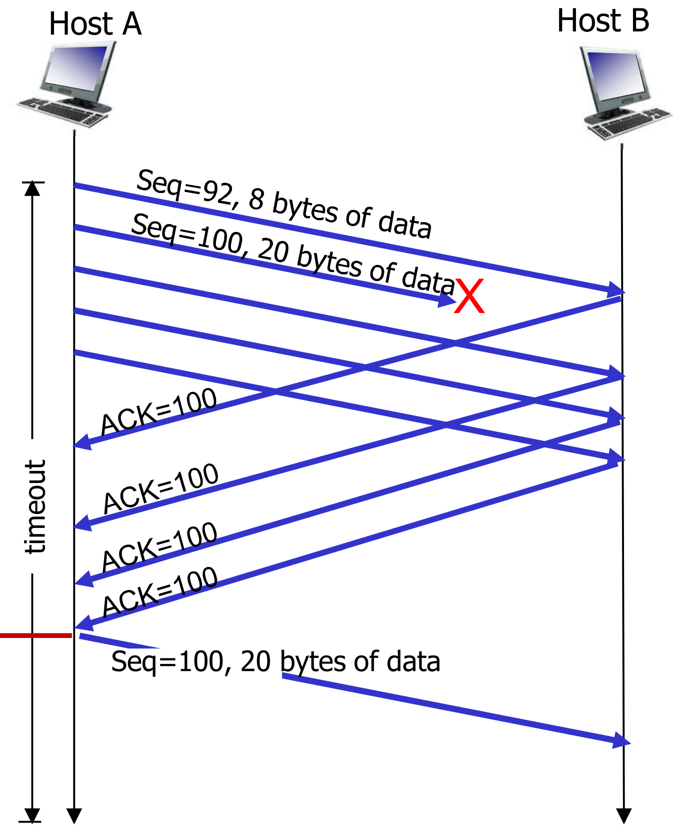
TCP fast retransmit

TCP fast retransmit

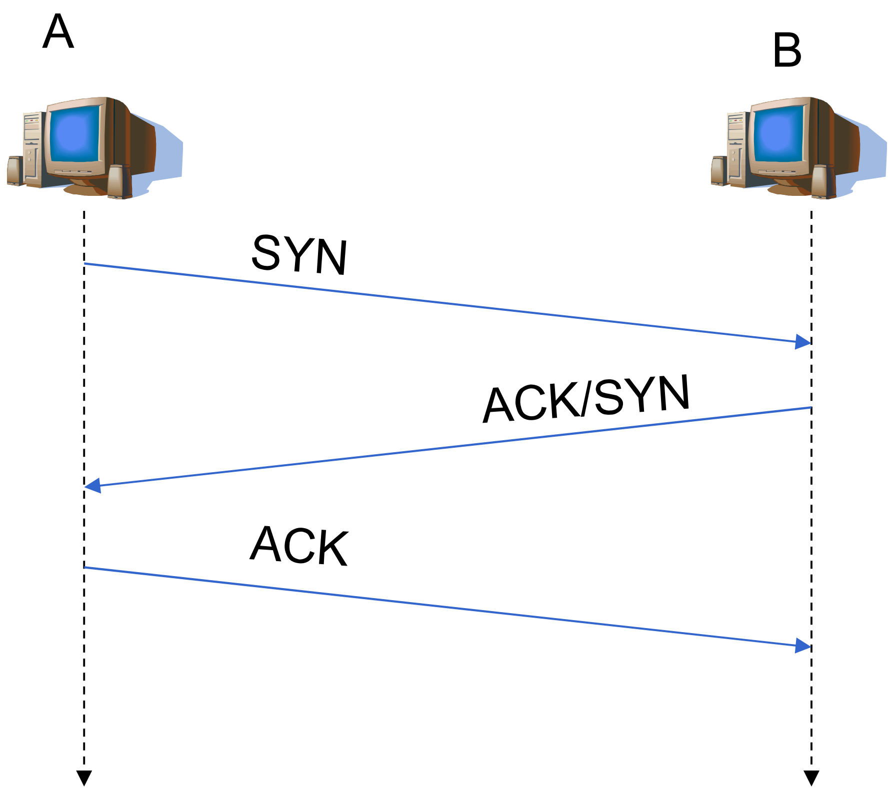
if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



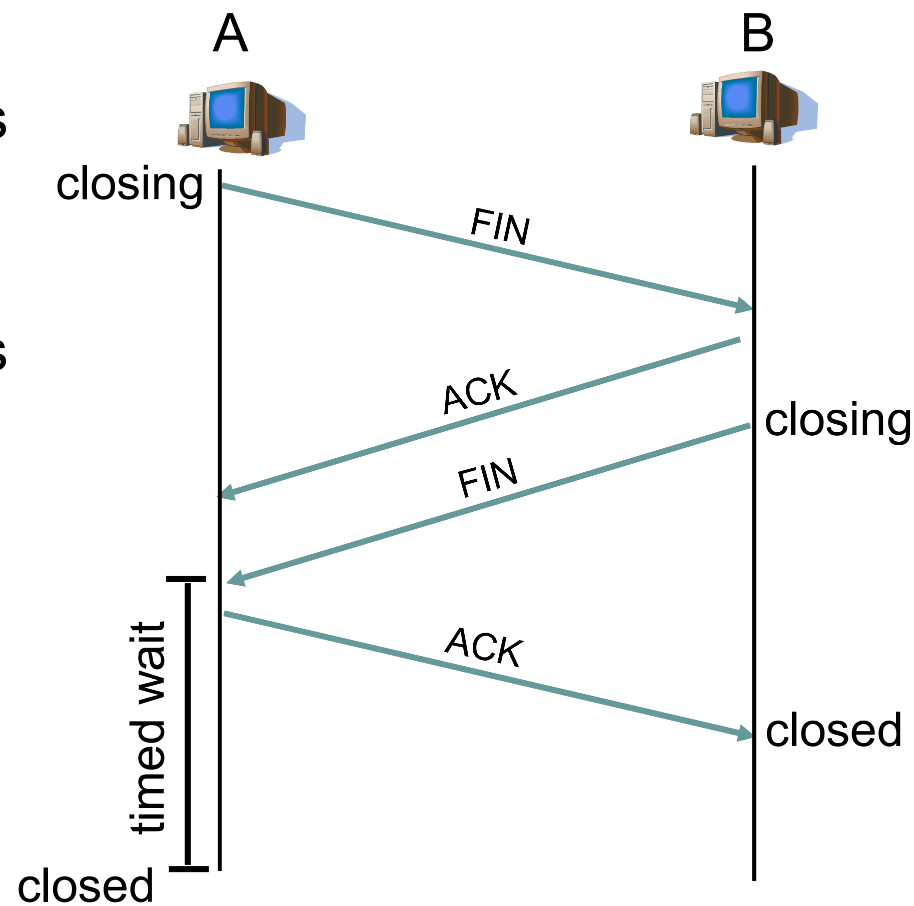
Connection establishing in TCP : 3 steps (3-way handshake)



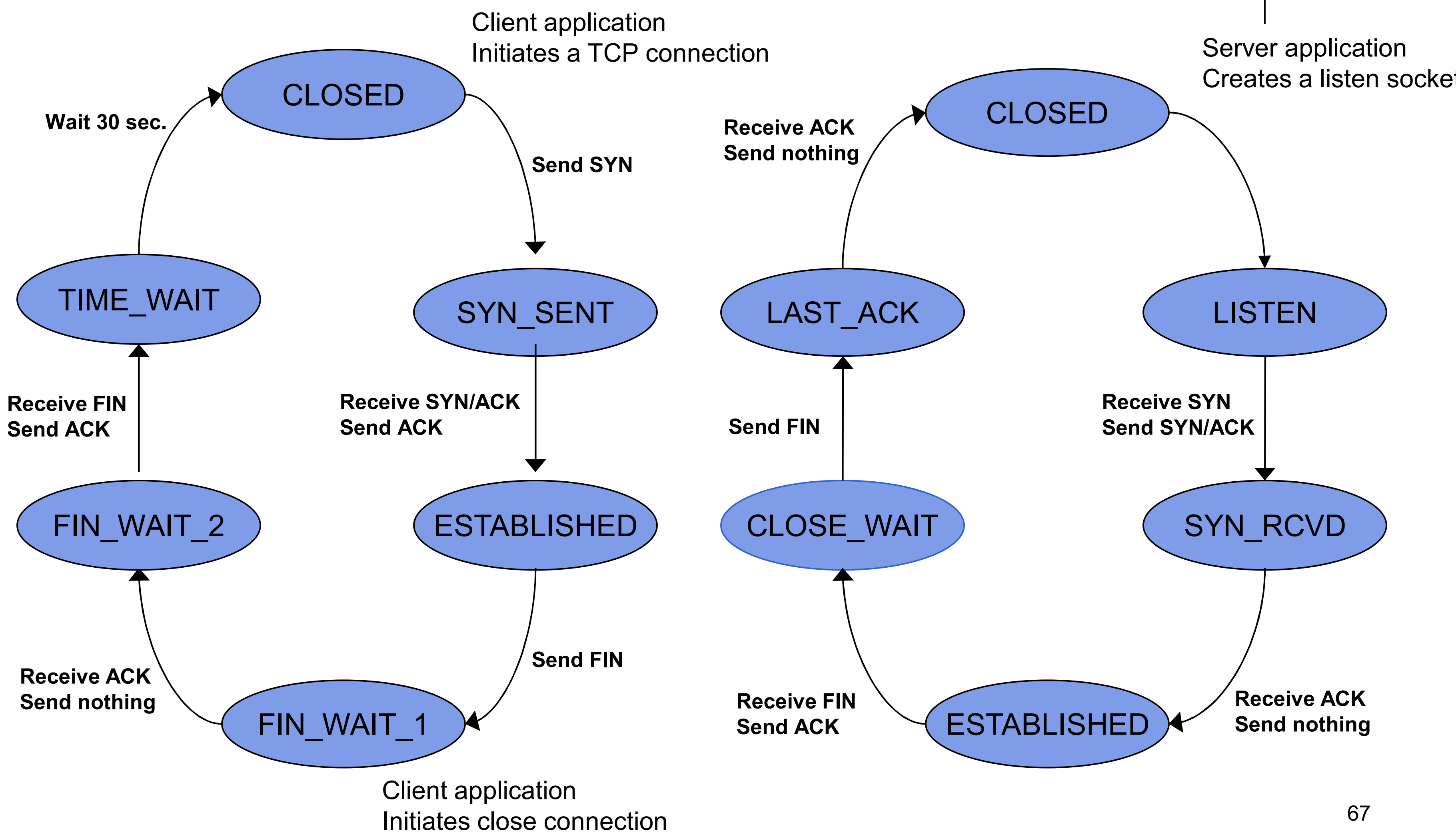
- Step 1: A sends SYN to B
  - Indicate initial value of seq # of A
  - No data
- Step 2: B receives SYN, replies by SYNACK
  - B initiates the buffer on its side
  - Indicate initial value of seq. # of B
- Step 3: A receives SYNACK, replies ACK, maybe with data.

Close connection

- Step 1: Send FIN to B
- Step 2: B receives FIN, replies ACK, closes the connection and sends FIN.
- Step 3: A receives FIN, replies ACK, go to “waiting”.
- Bước 4: B receives ACK. close connection



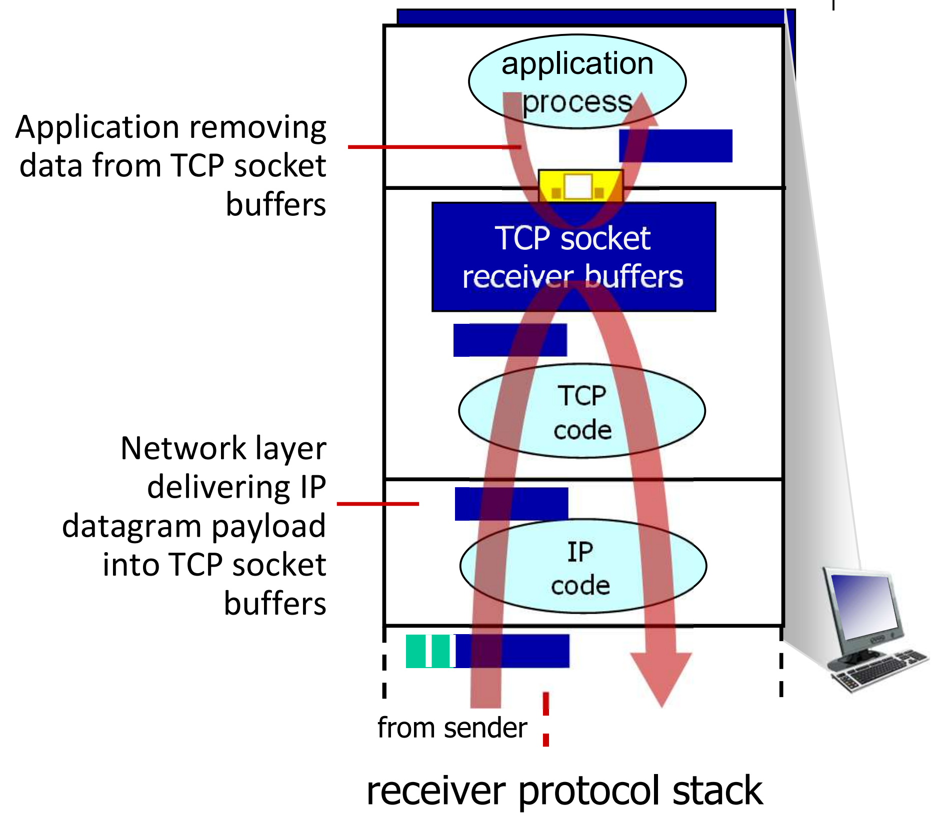
Symplified life cycle of TCP



Flow control in TCP

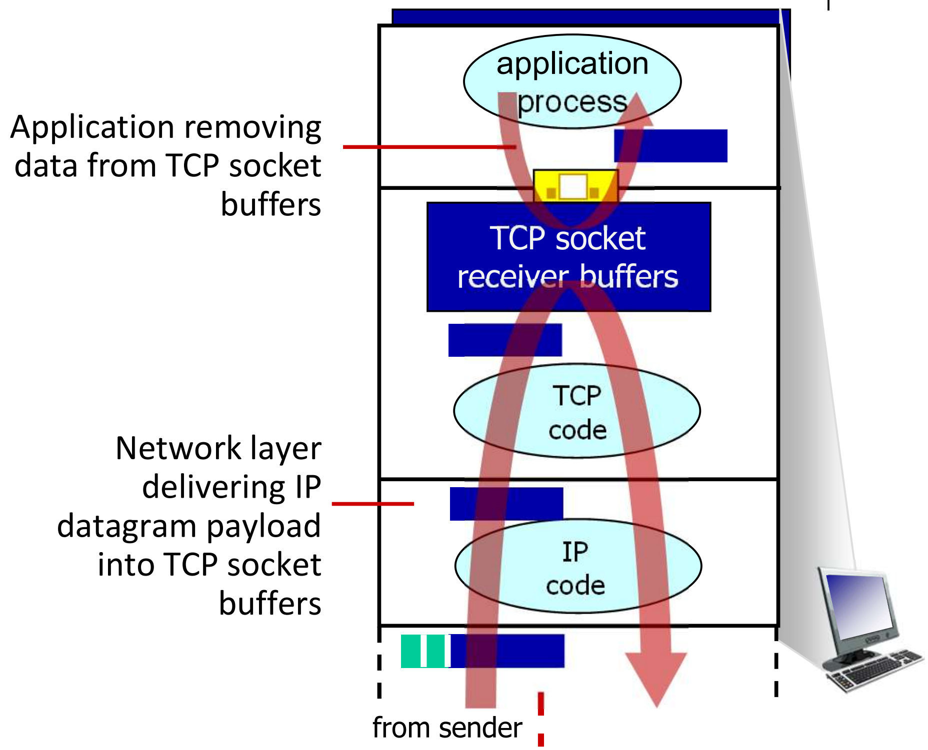
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP flow control

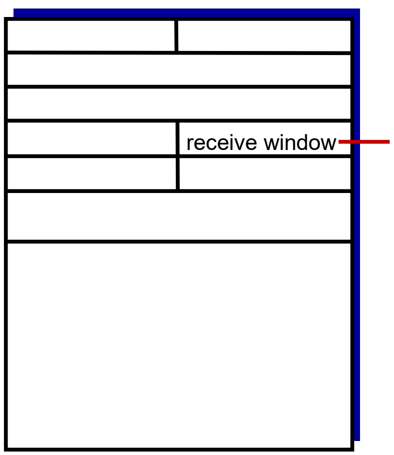
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

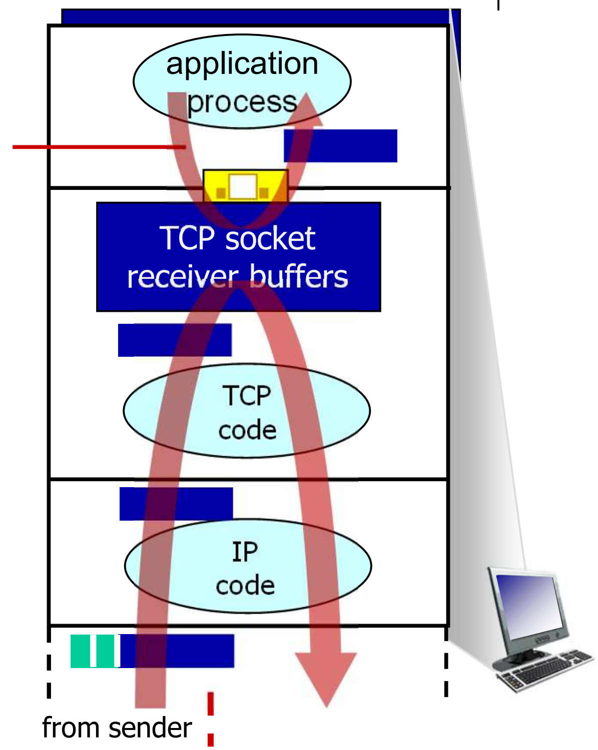
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept

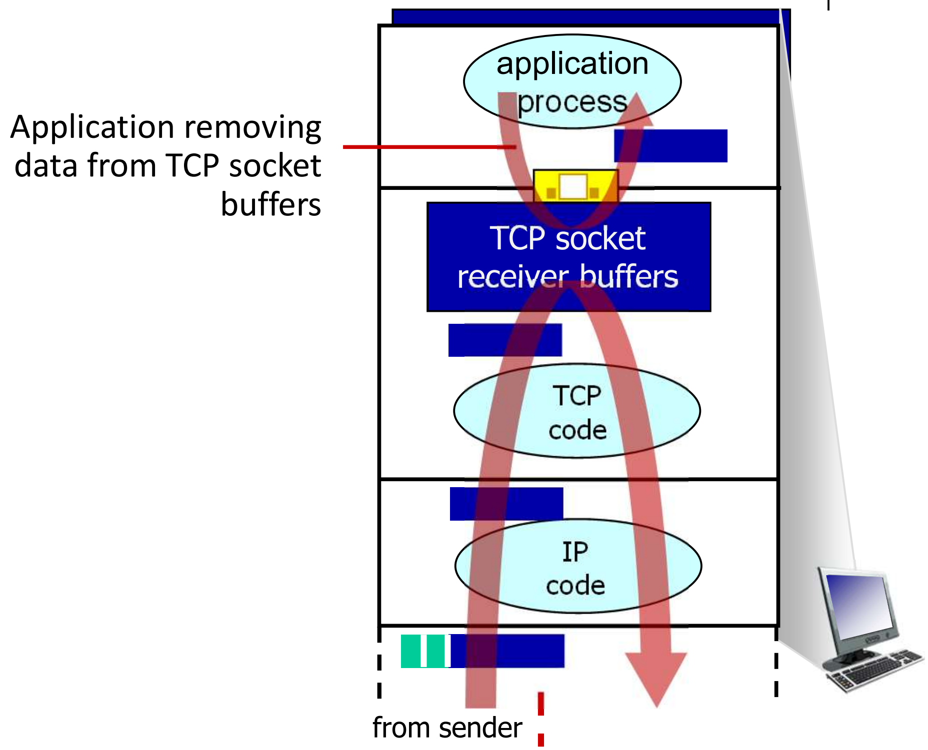
Application removing data from TCP socket buffers



receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

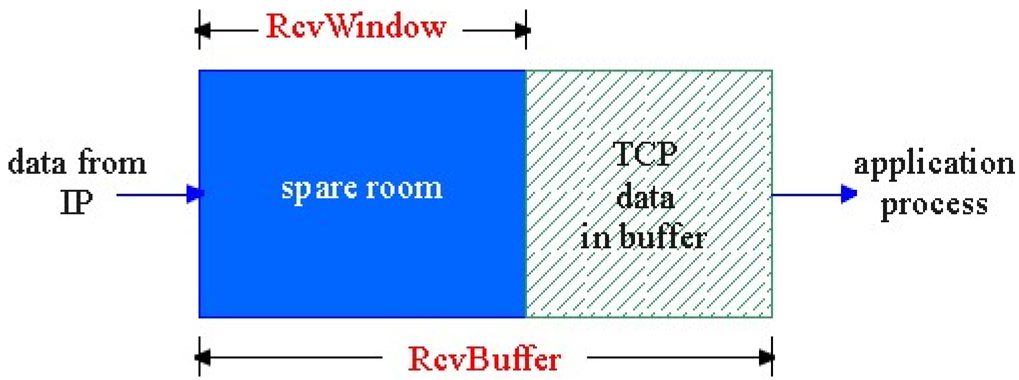


receiver protocol stack

flow control

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

TCP flow control

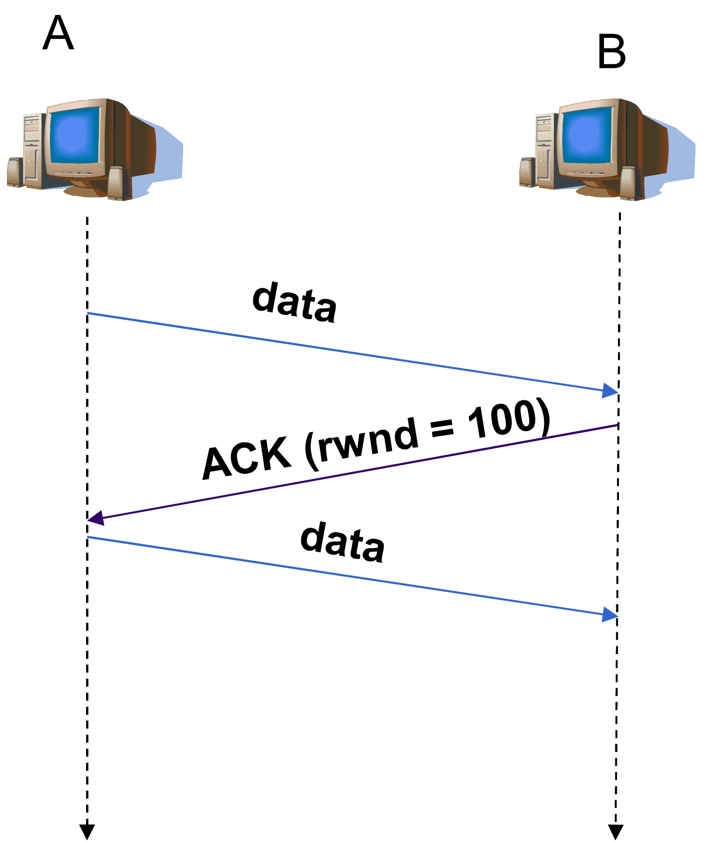


- Size of free buffer

= Rwnd

= **`RcvBuffer-[LastByteRcvd`**
`- LastByteRead]`

Information exchanged on Rwnd



- Receiver inform regularly to senders the value of `Rwnd` in acknowledgment segments

Congestion control in TCP

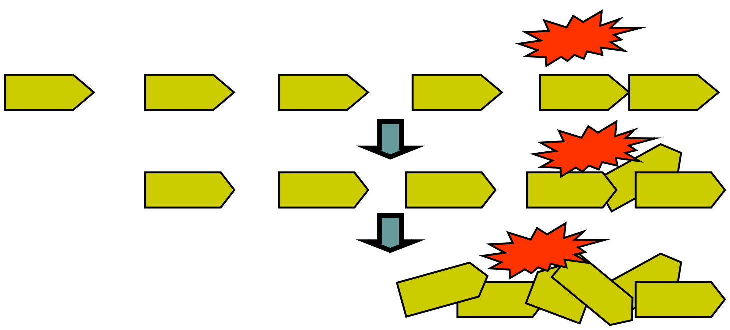
Principles of congestion control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver



Congestion occur

TCP congestion control: AIMD

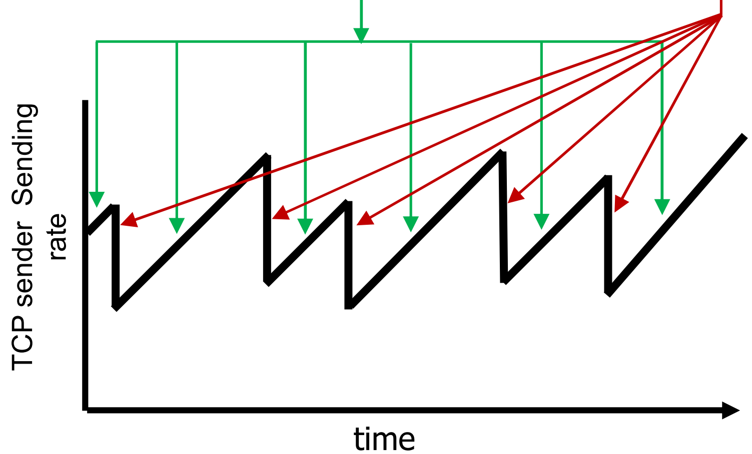
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

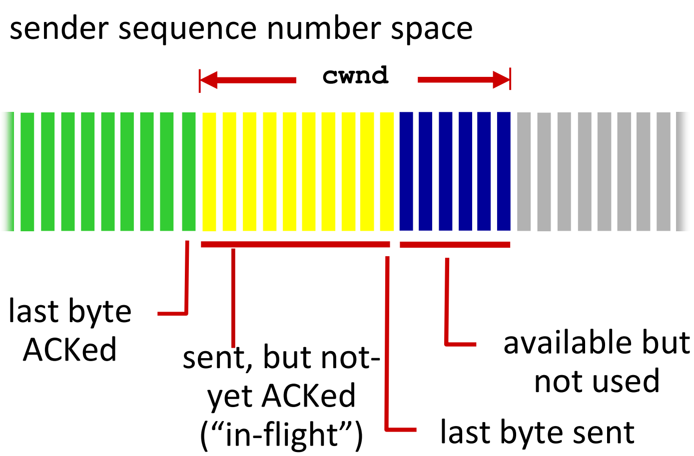
Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP congestion control: details



TCP sending behavior:

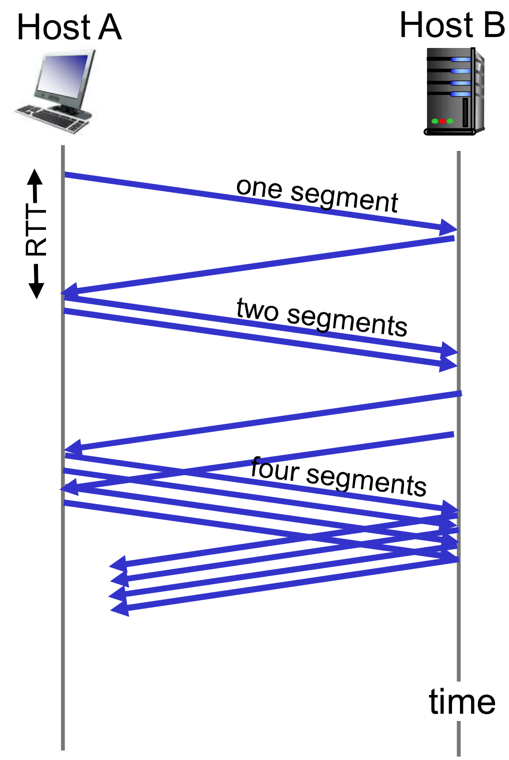
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: `LastByteSent- LastByteAcked` $\leq$ `cwnd`
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

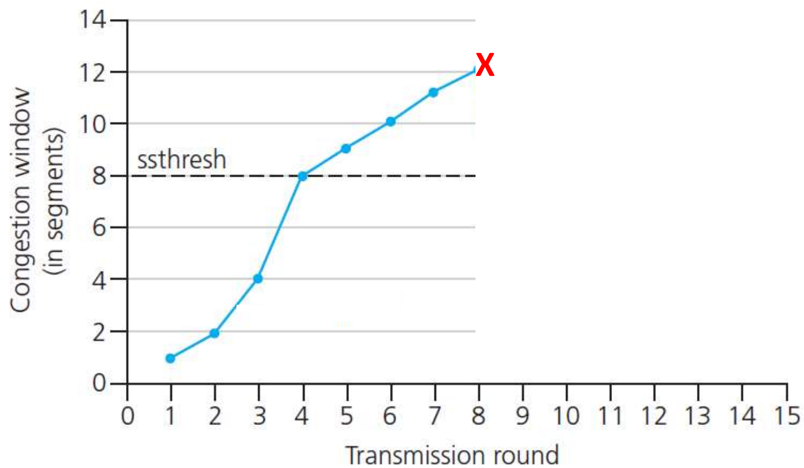
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

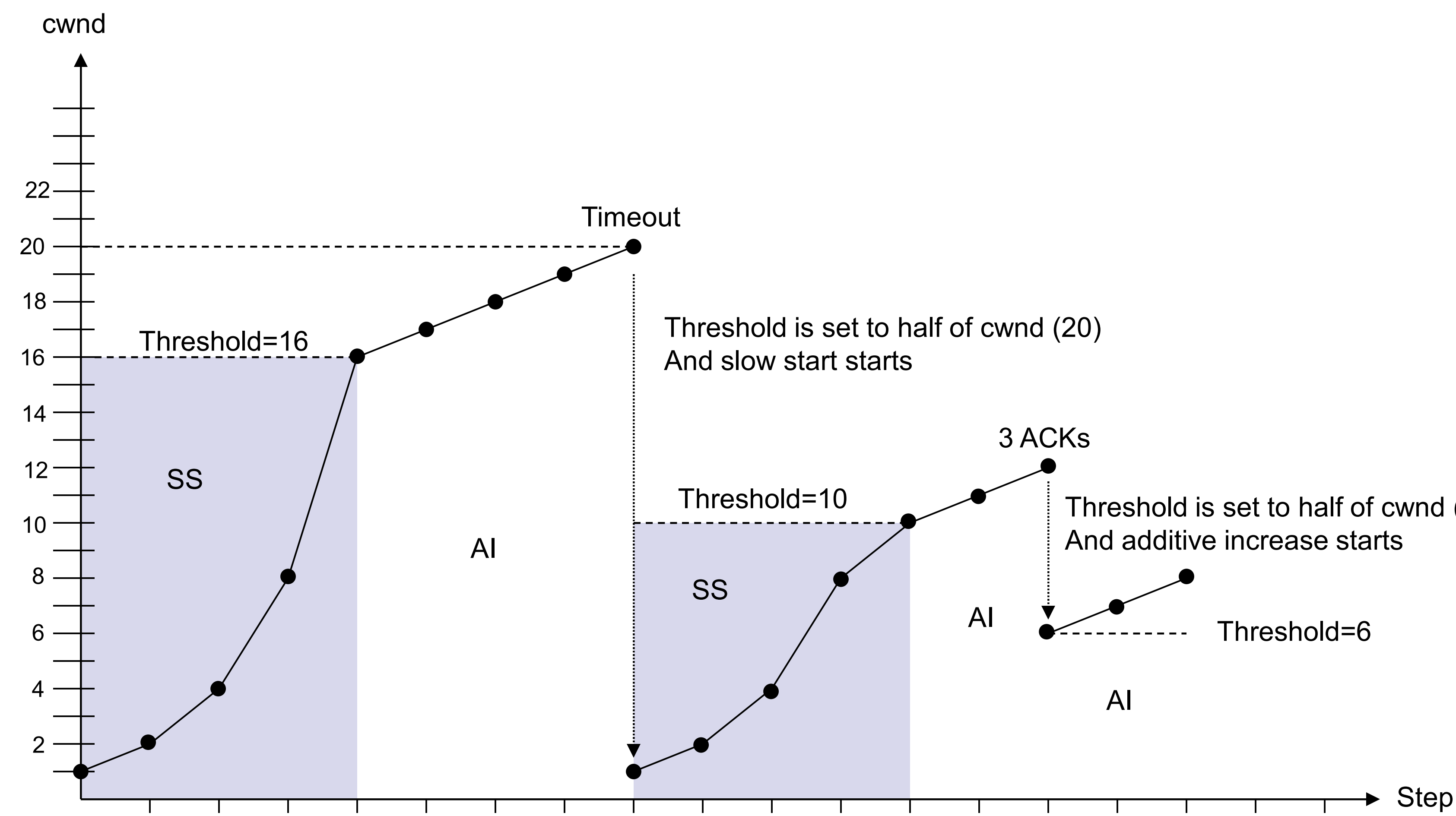
A: when **cwnd** gets to 1/2 of its value before timeout.



Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Congestion control – illustration



Exercise

- Assume that we need transmit 1 file
  - File size O =100KB over TCP connection
  - S is the size of each TCP segment, S = 536 byte
  - RTT = 100 ms.
- Assume that the congestion window size of TCP is fixed with value W.

  What is the minimum transmission time? If the transmission speed is
  - R = 10 Mbit/s;
  - R= 100 Mbits/s.

Solution (cont.)

- T transmit (W packet) = W * S/R
- Transmit without waiting:
- => (W-1)*S/R >= RTT
- => W >= RTT*R/S +1
- Time to transmit all data L = L/R + RTT
- R=100 Mbps
  - W>= 100ms * 100 Mbps/ (536*8) + 1

Exercise

- Assume that we need transmit 1 file
  - File size *O* =100KB over TCP connection
  - S is the size of each TCP segment, *S* = 536 byte
  - *RTT* = 100 *ms*.
- Assume that the congestion window of TCP works according to slow-start mechanism.
- What is the size of the congestion window when the whole file is transmited.
- How much of time is required for transmitting the file? If R = 10 Mbit/s; R= 100 Mbits/s.