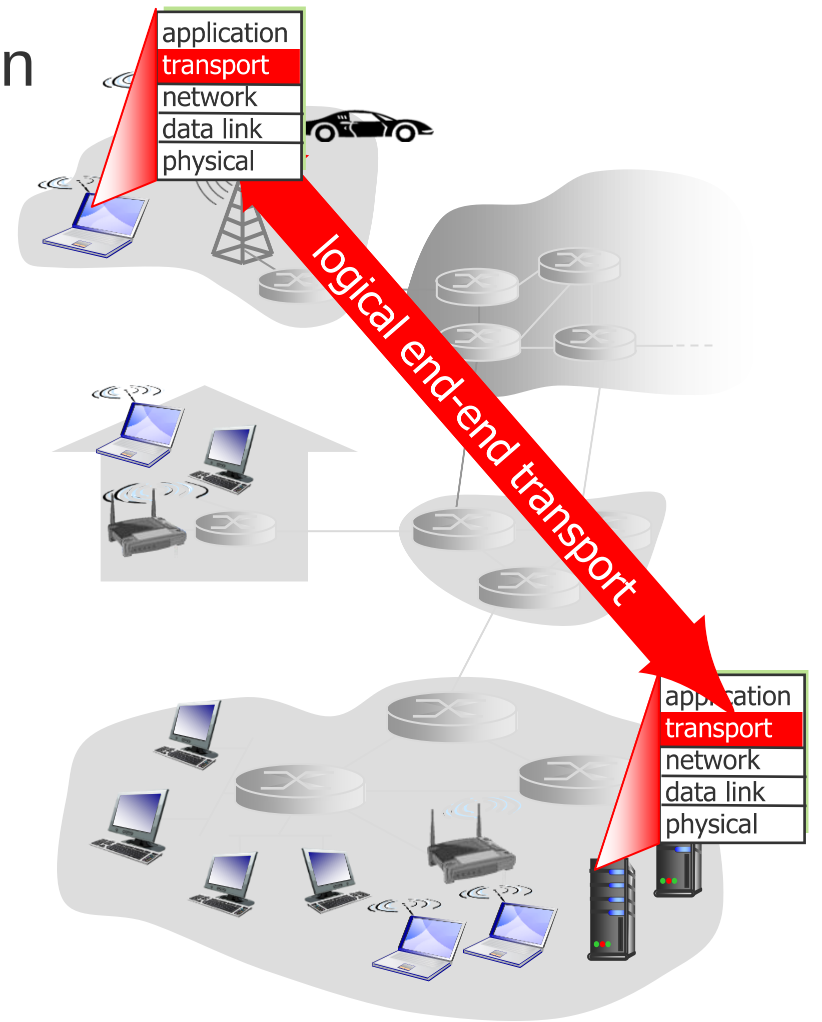
Transport Services and Protocols

Andy Carpenter

(Andy.Carpenter@manchester.ac.uk)

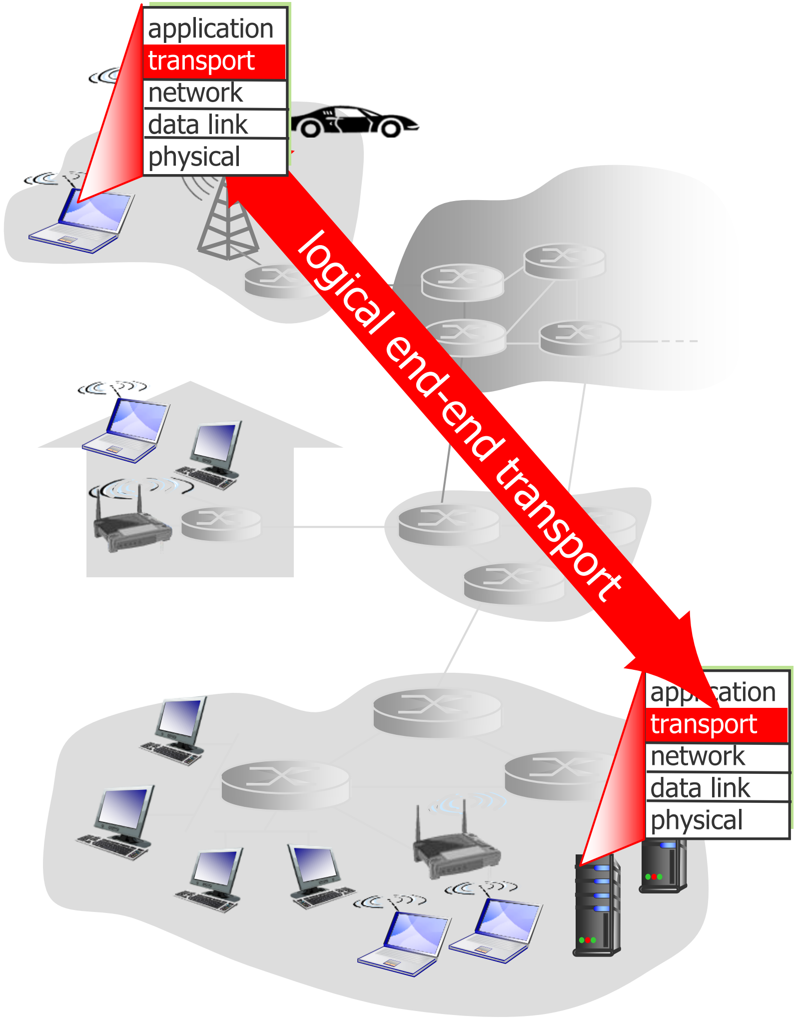
Transport Services and Protocols

- Provides logical communication between applications
  - process-to-process
- Runs in end systems
  - send side: breaks app messages into segments, passes to network layer
  - receiver side: reassembles segments into messages, passes to app layer
- Multiplex/ id end-points
- Protocols available:
  - Internet: TCP, UDP



Transport Service QoS Params

- Provides QoS to applications
- Options
  - Connectionless/connection-orientated
  - reliability
  - flow control
  - congestion control
- Alters underlying network QoS
- Services not available:
  - delay guarantees
  - bandwidth guarantees



End-points (Ports/Sockets)

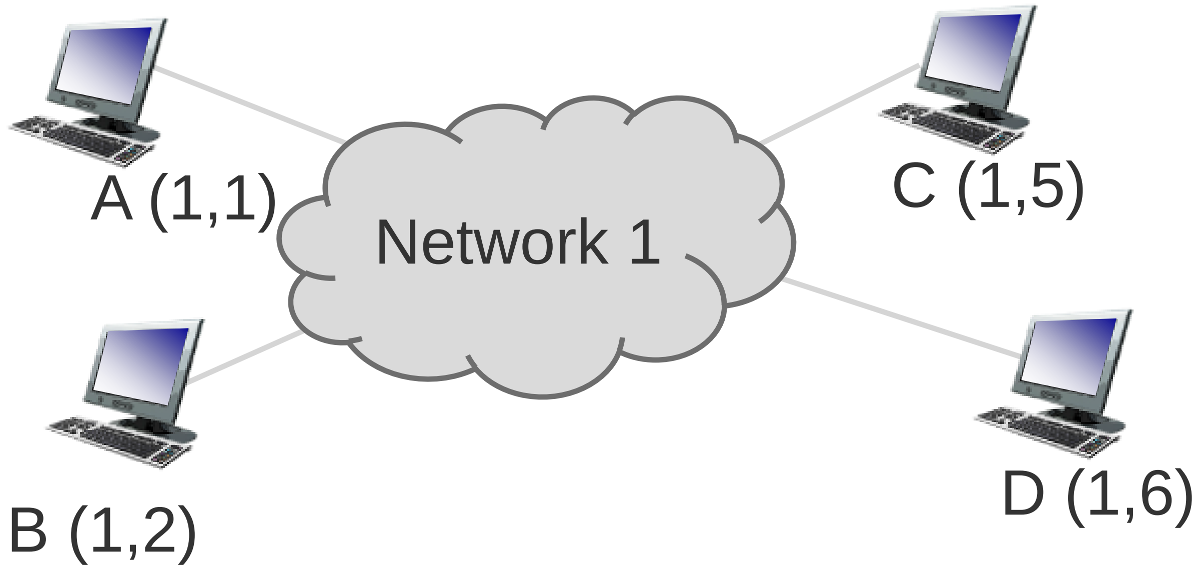
- Support multiple destinations
- Identify process via 16-bit identifier; port
- Applications use sockets which have associated port
- Well know services allocated ports 0-1024

Port	Mnemonic	Service
7	ECHO	Echo
20	FTP-DATA	FTP (default data)
21	FTP	FTP (control)
53	DOMAIN	Domain name service
80	HTTP	Hypertext Transfer protocol

Centrally allocated

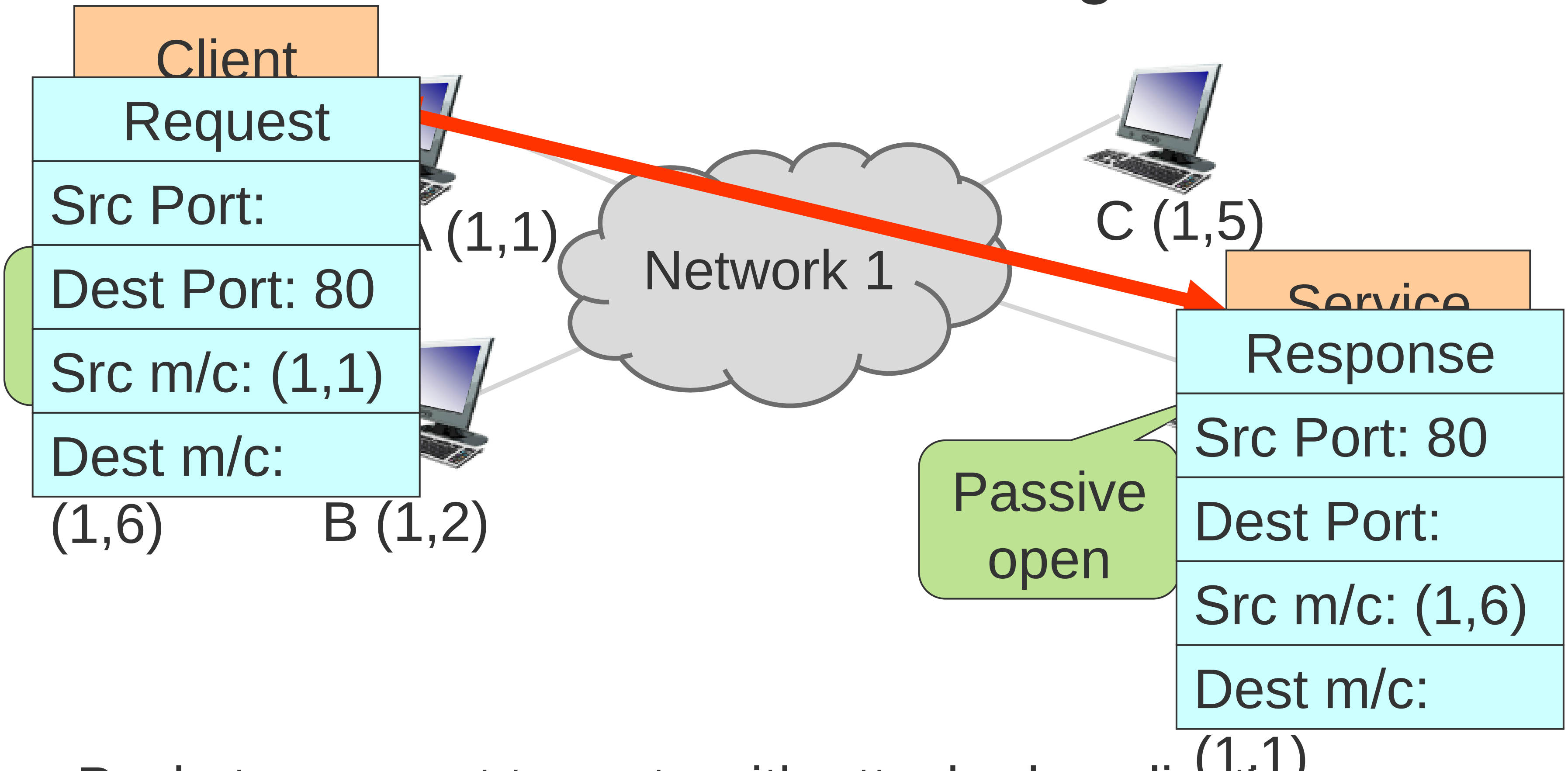
- Client ports are dynamically allocated by o/s

End-Point: Connecting



- Packets are sent to ports with attached applications
- Data to other ports will generate an error message

End-Point: Connecting

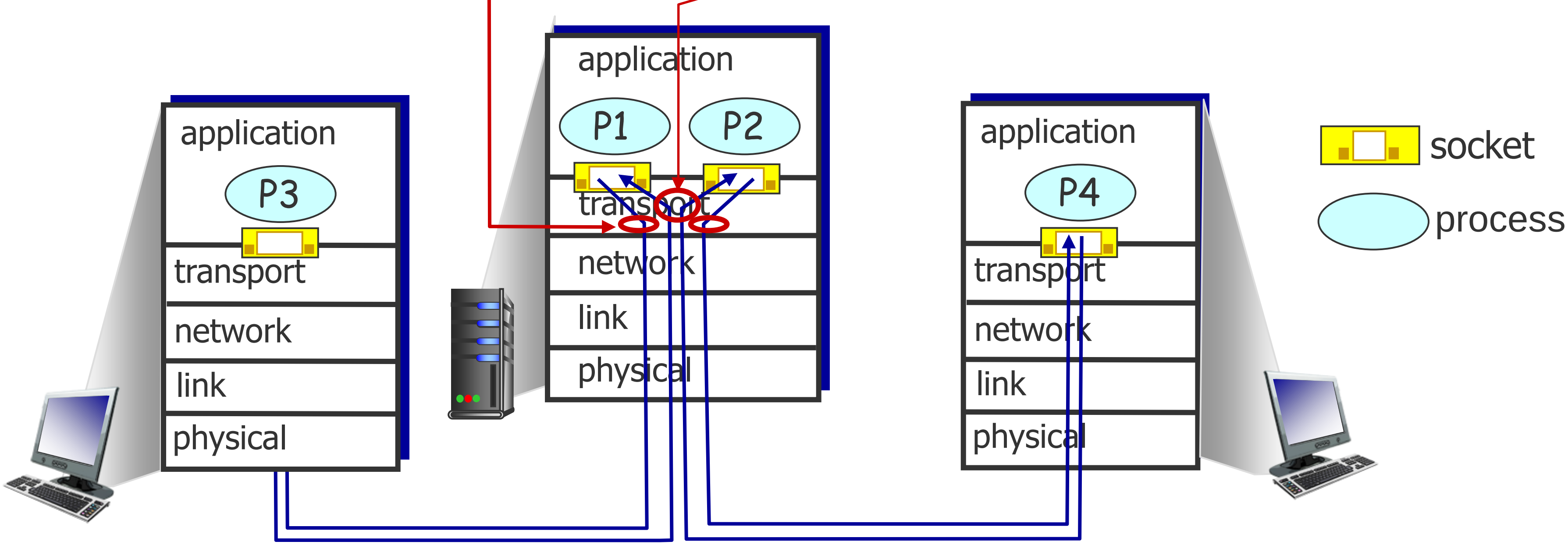


- Packets are sent to ports with attached applications
- Data to other ports will generate an error message

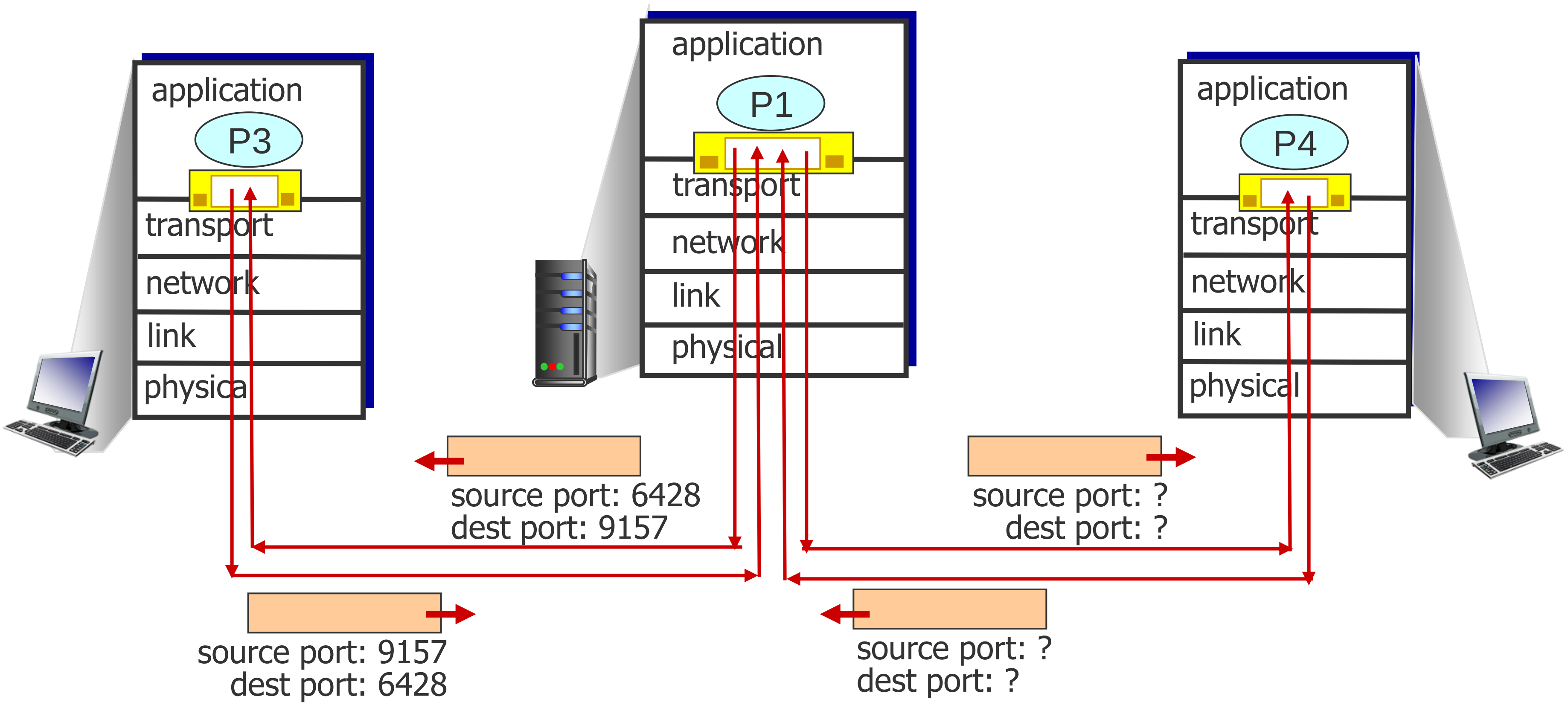
End-Point: Multiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

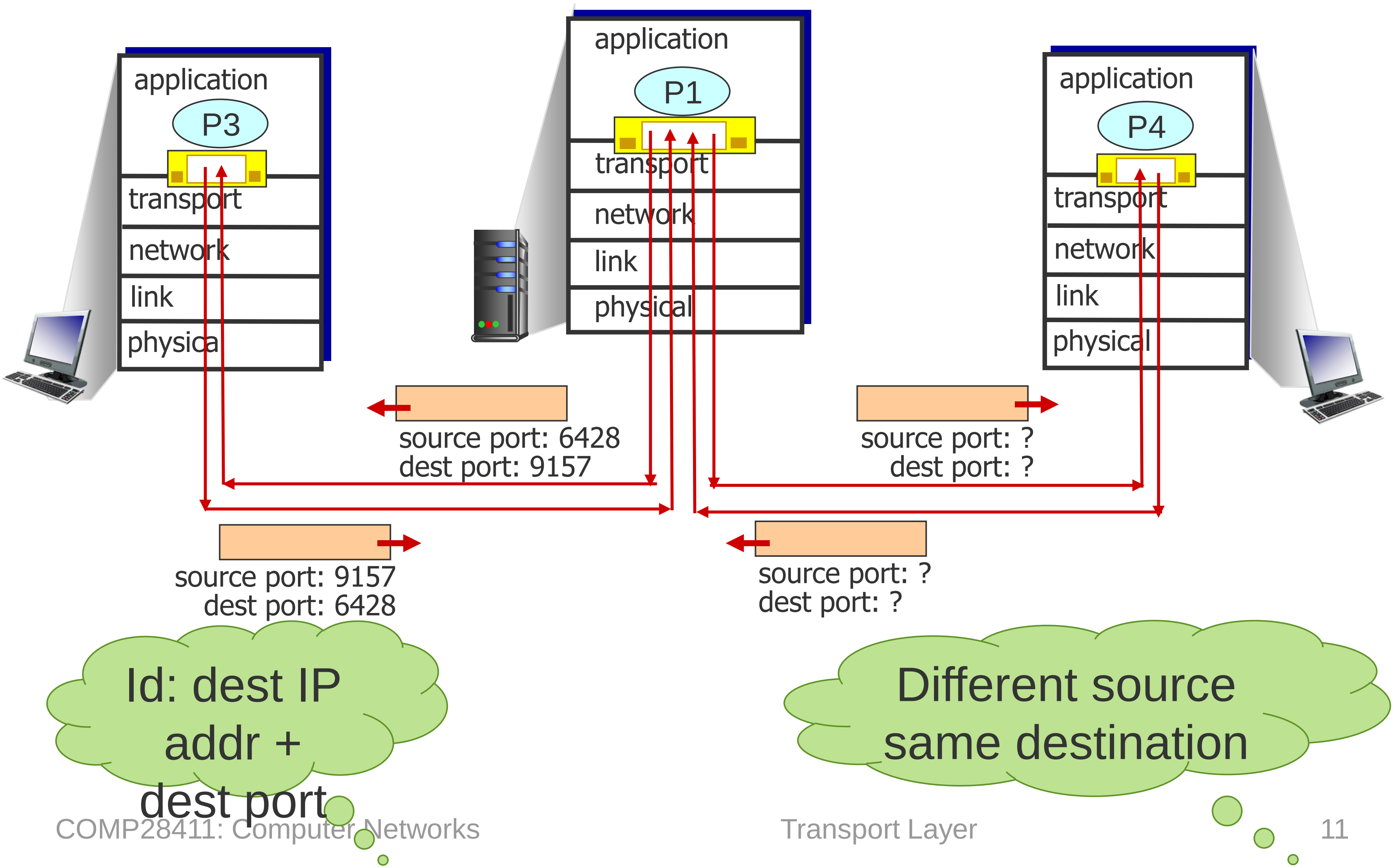
demultiplexing at receiver:
use header info to deliver received segments to correct socket



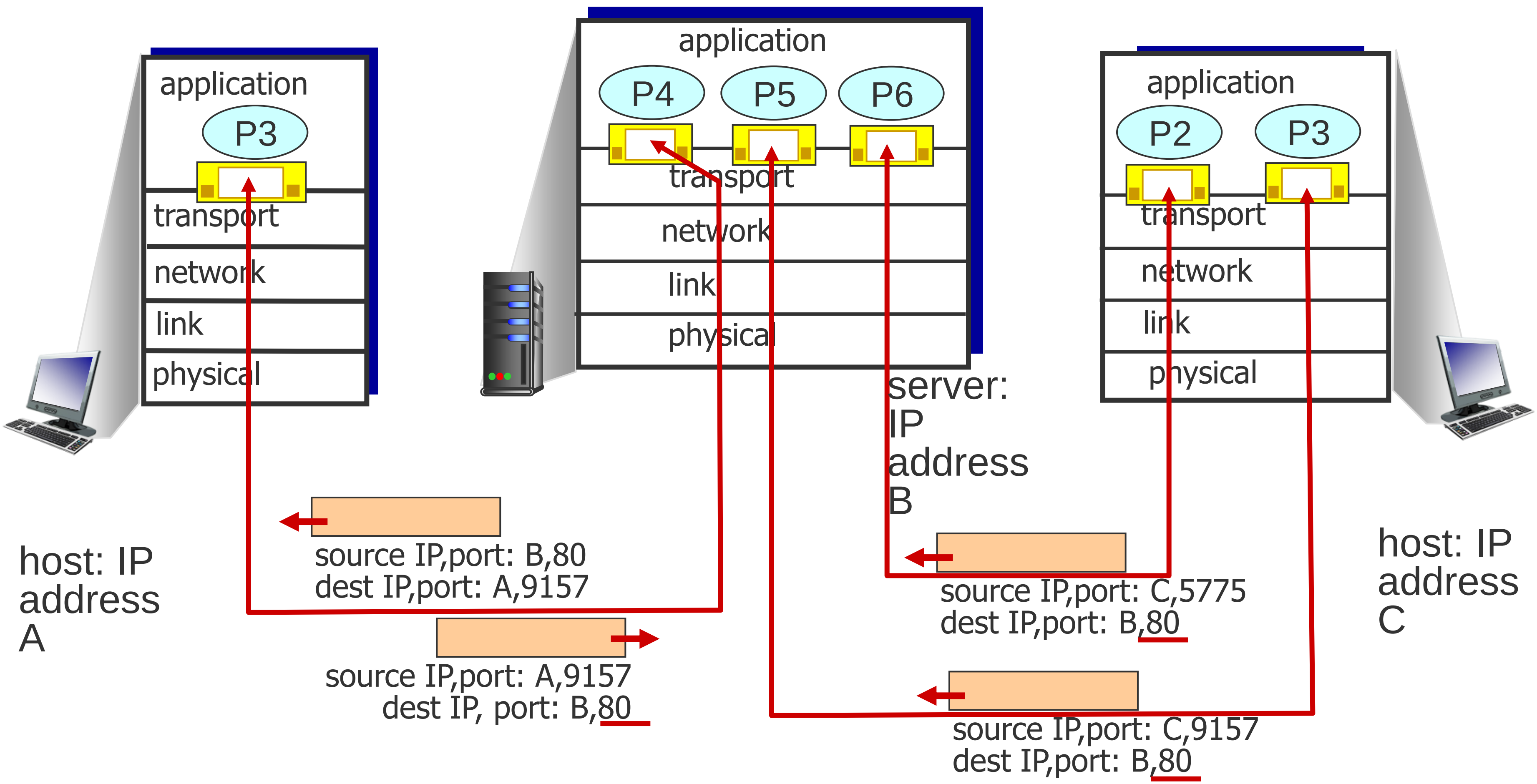
End-Point: Id (Connectionless)



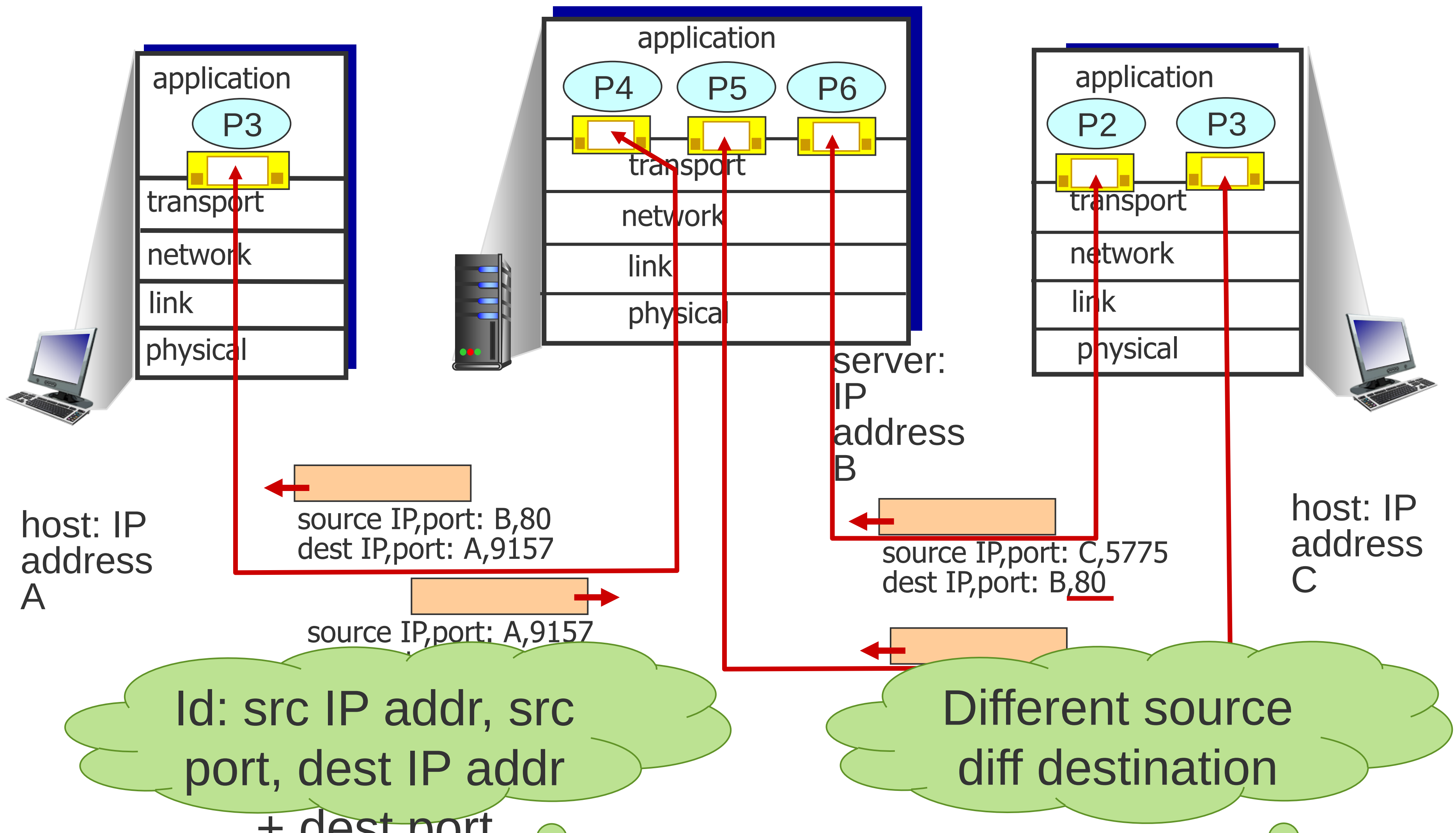
End-Point: Id (Connectionless)



End-Point: Id (Connection-Orient)



End-Point: Id (Connection-Orient)



User Datagram Protocol (UDP)

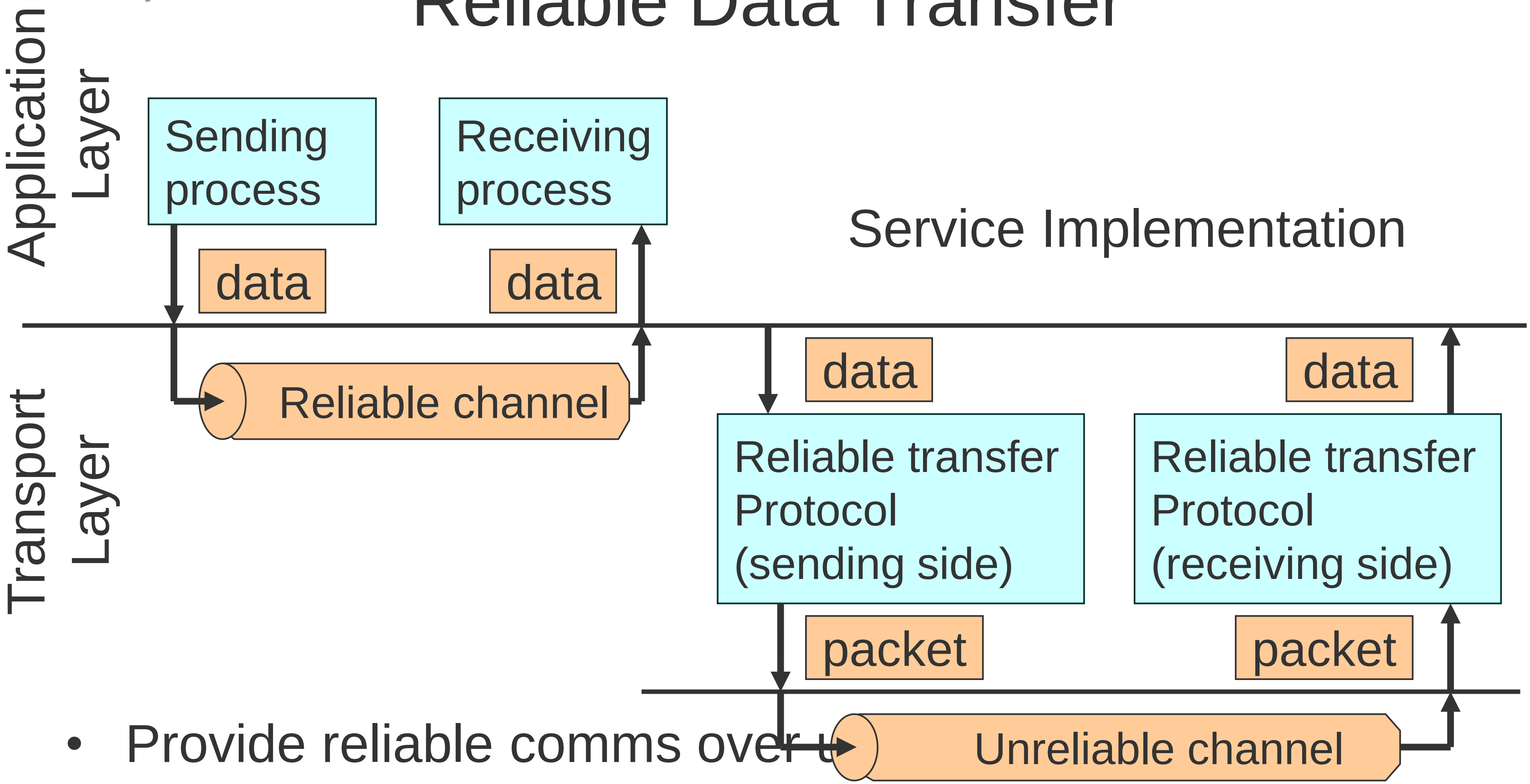
- “no frills”, “best effort” connectionless service, segments may be:
  - lost, reordered
- Segments independent
- Often used for streaming multimedia applications
  - loss tolerant
  - rate sensitive
- Other uses:
  - DNS, SNMP
  - NFS
  - o/s applications

Why is there a UDP?

- no connection setup (adds delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

[RFC 768]

Reliable Data Transfer

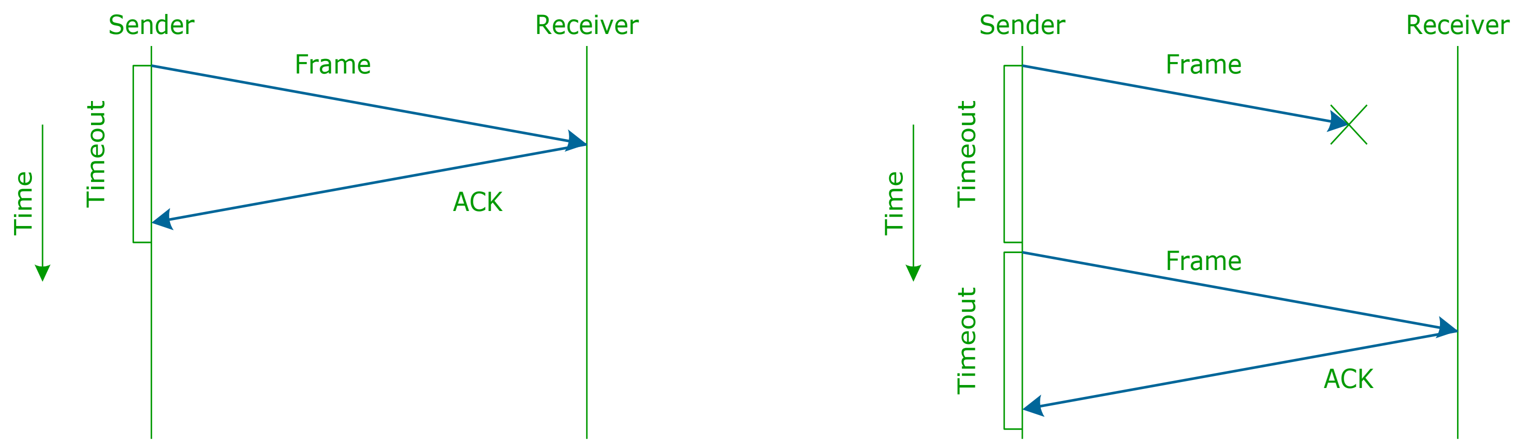


- Provide reliable comms over u
- Complexity depends on characteristics of unreliable channel

Recovering from Errors

- Recovery uses two mechanisms:
  - acknowledgements and timeouts
- Acknowledgement (**ACK**) is control packet from
  - receiver to transmitter of data packet being ACKed
- Receipt of ACK confirms delivery of data
- If ACK not receiver within **timeout**:
  - transmitter of data retransmits data; needs copy
- Process called automated repeat request (ARQ)
- ARQ mechanisms: **stop-and-wait**, **sliding window**

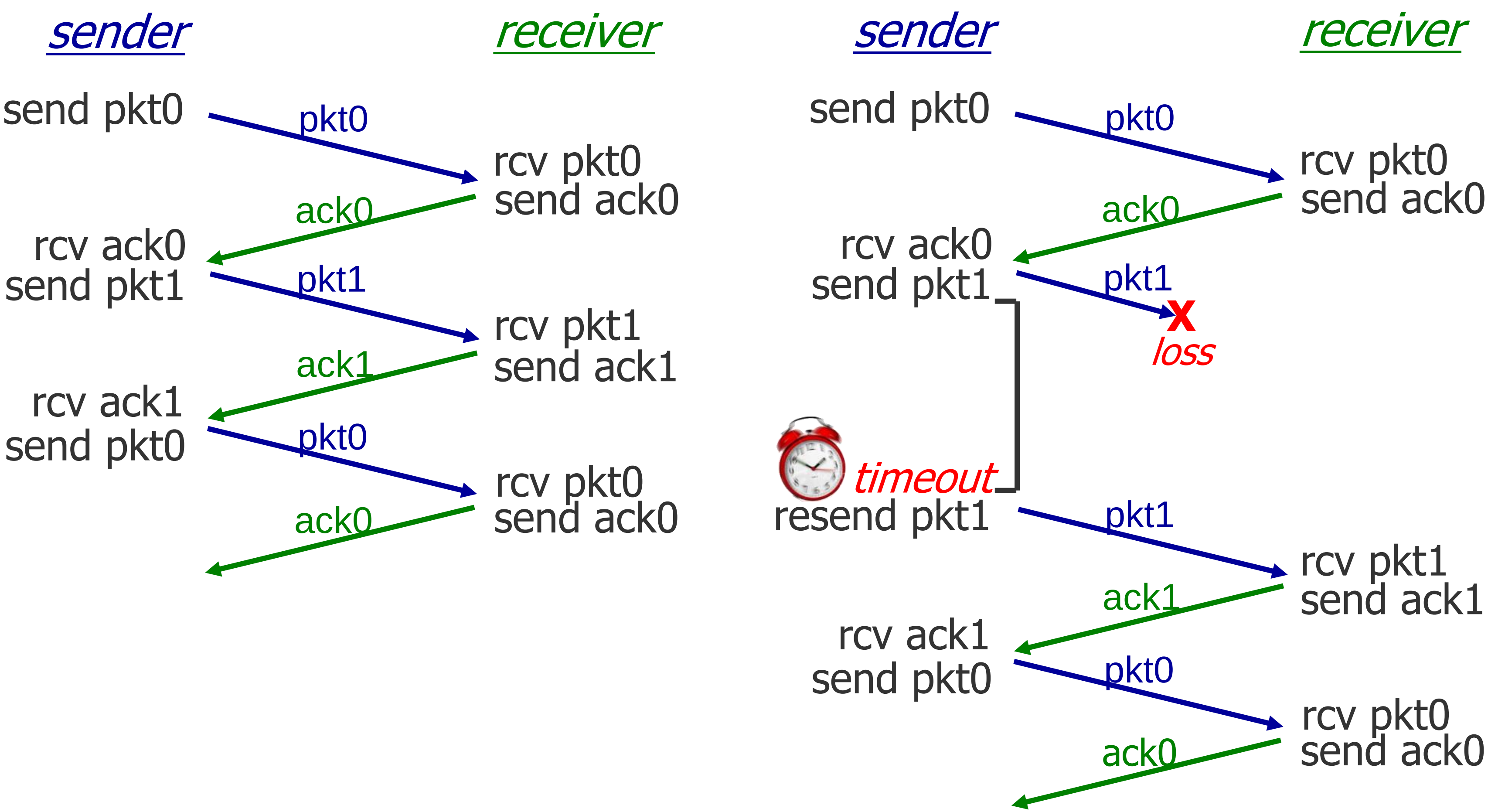
Reliability: Stop-and-Wait



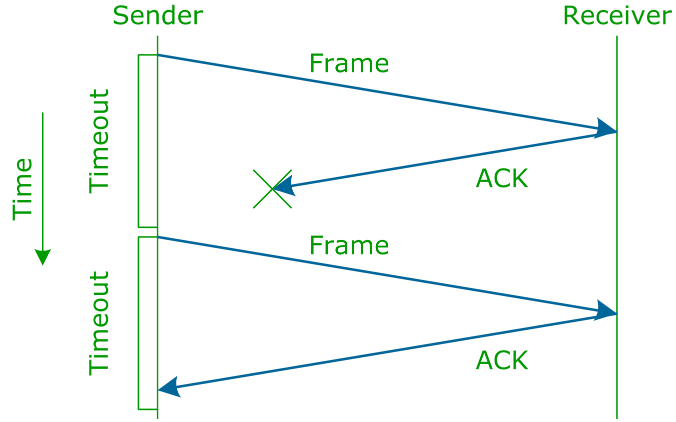
Peterson and Davie, Figure 2.19

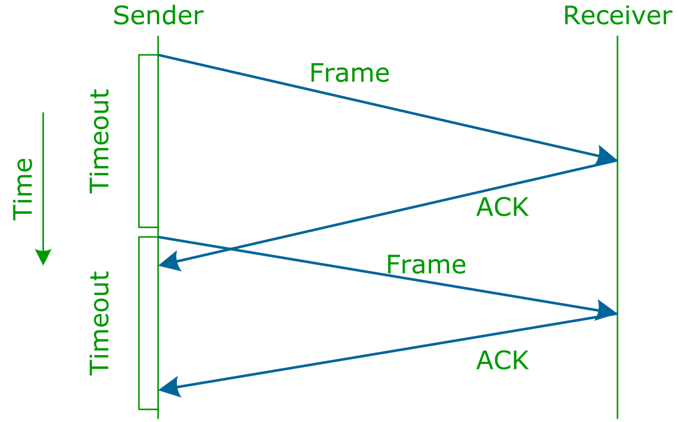
- Wait for acknowledgement before sending next packet
- Normal operation, ACK received before timeout expires
- If not, data packet is retransmitted and, hopefully, ACKed

Reliability: Stop-and-Wait



Reliability: Stop-and-Wait

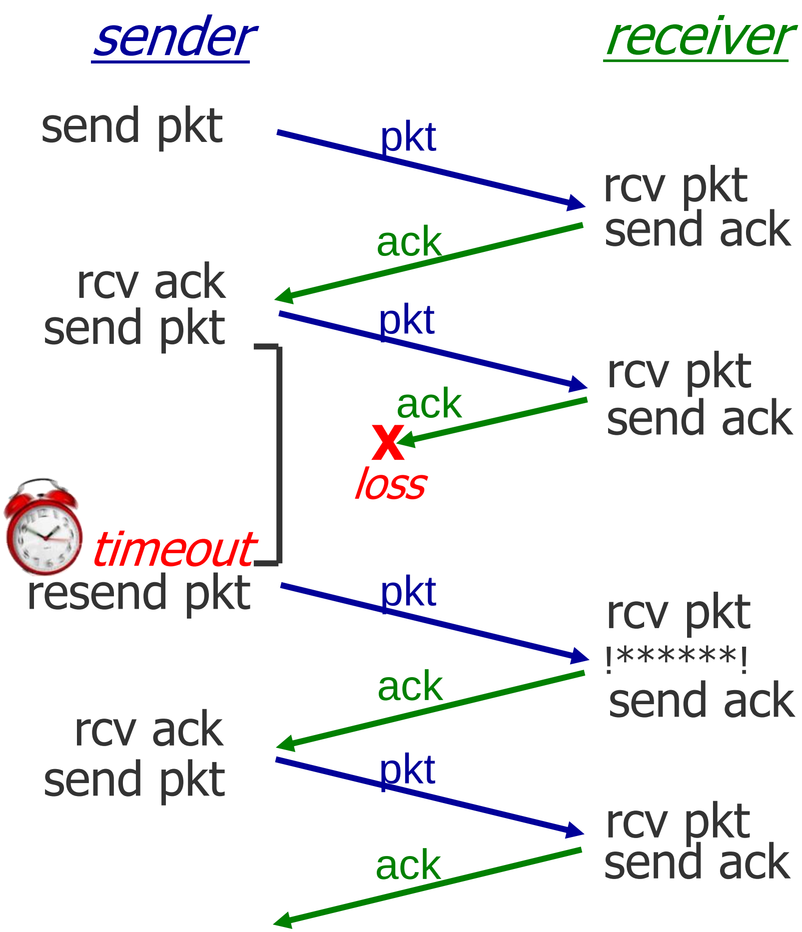


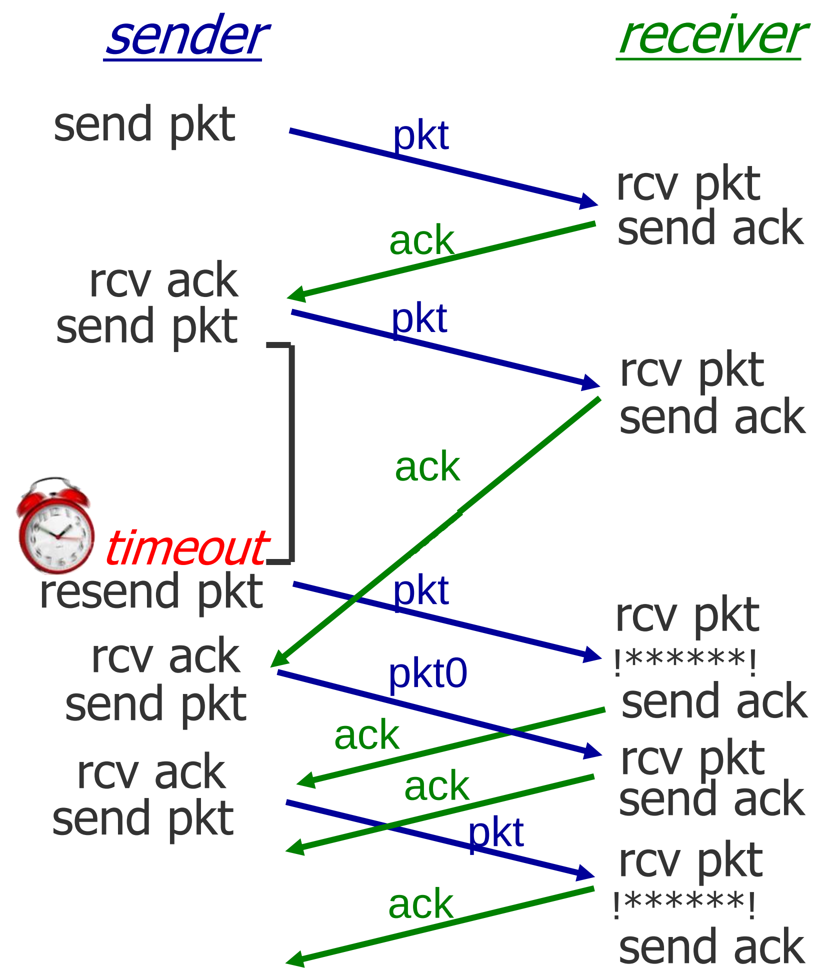


Peterson and Davie, Figure 2.19

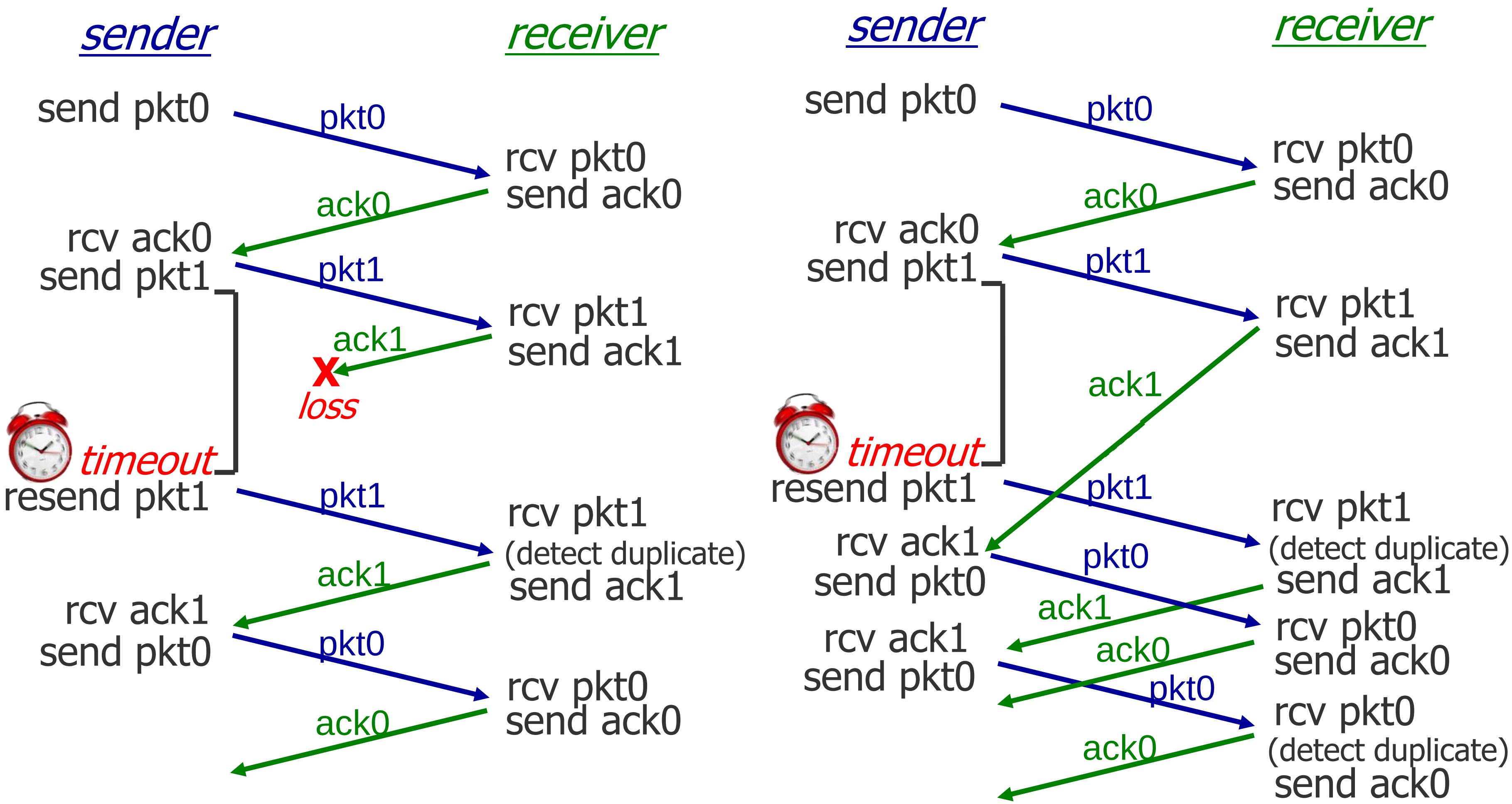
- Loss of ACK, or late arrival, causes retransmission
- Duplicate packet is received, but
- receiver believes that it is receiving a new packet
- Duplicates can be detected using sequence numbers
- One-bit sequence numbers used; 0 or 1

Reliability: Stop-and-Wait

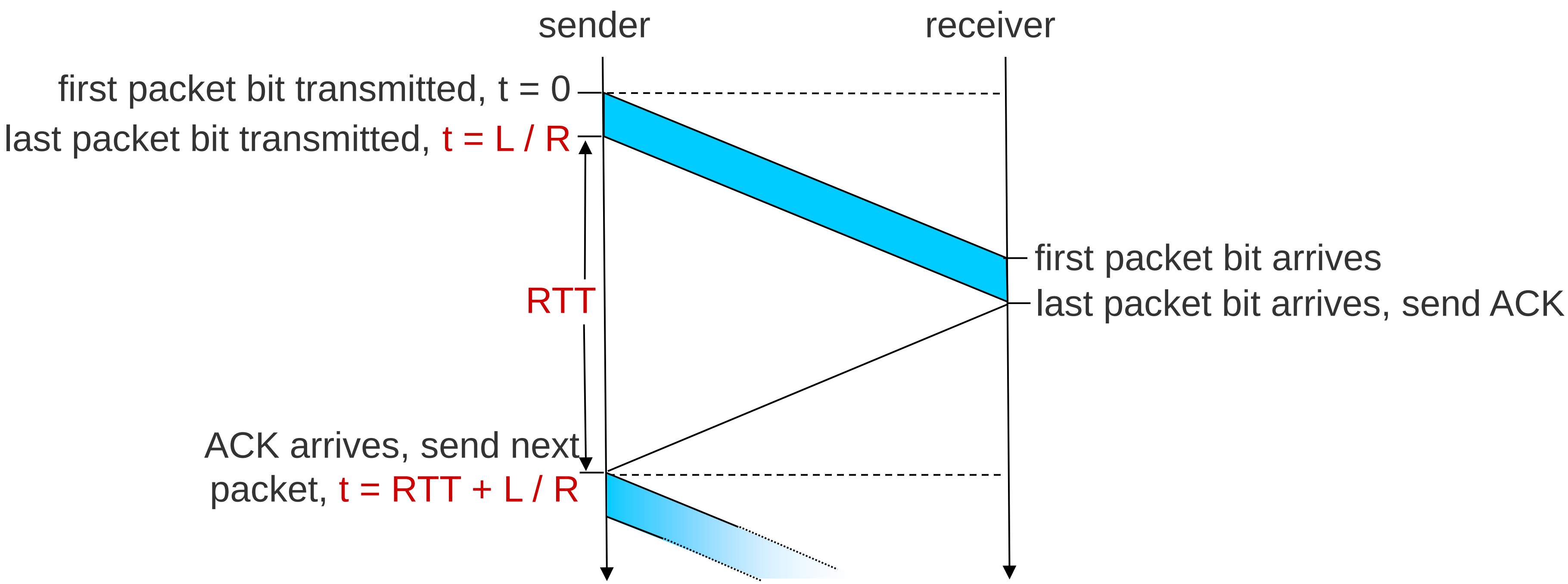




Reliability: Stop-and-Wait



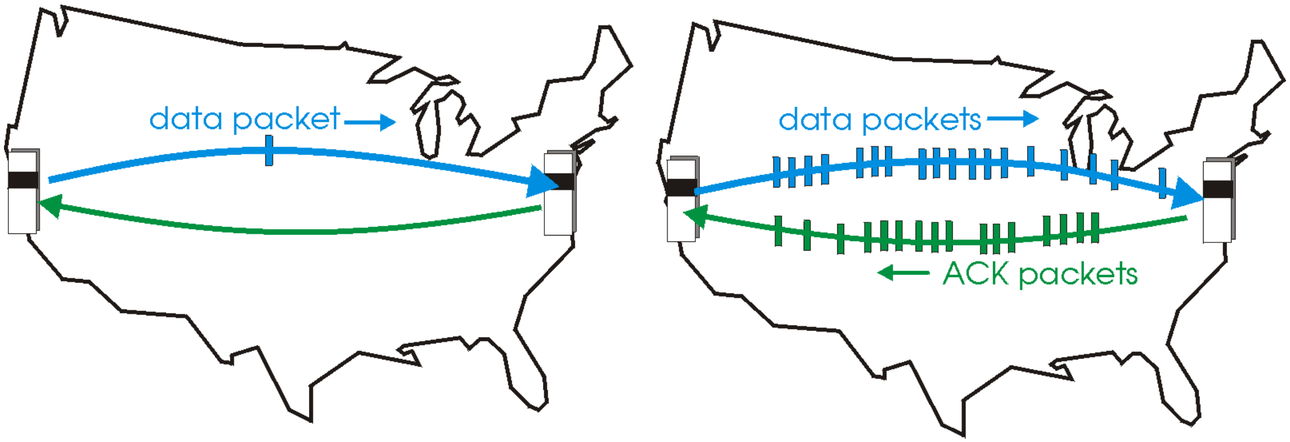
Reliability: S-a-Wait - Utilisation



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

Network protocols limit use of physical resources

Reliability: Pipelined Protocols

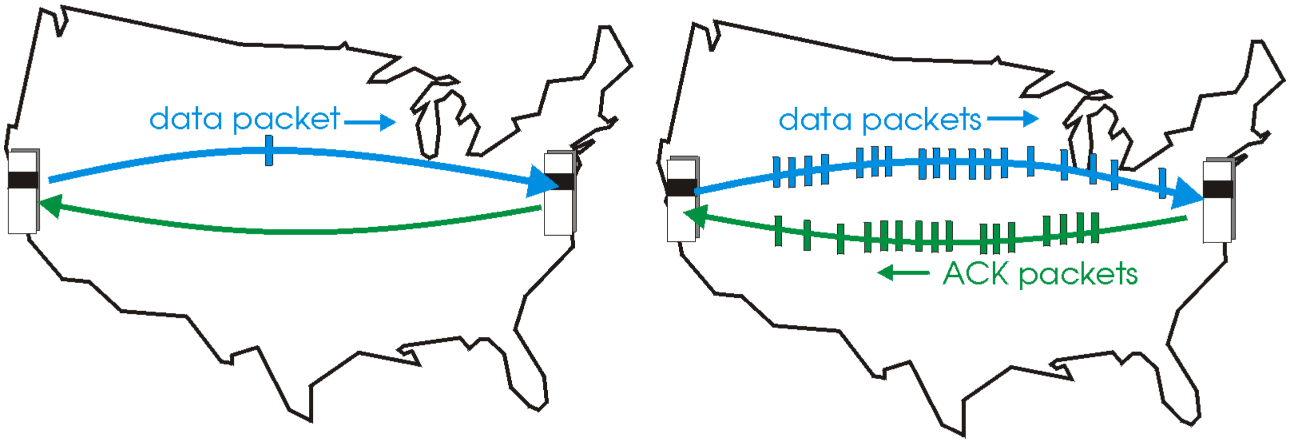


- Allow multiple, “in-flight”, yet-to-be-acknowledged pkts
- Example: received ACK 5, window 6

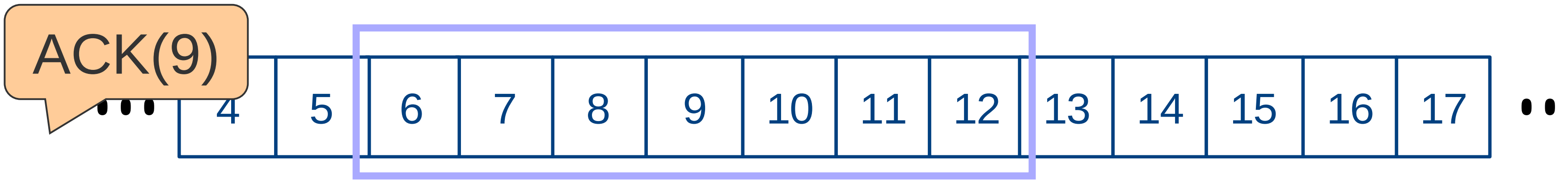
…	4	5	6	7	8	9	10	11	12	13	14	15	16	17	…

- Often referred to as sliding window

Reliability: Pipelined Protocols

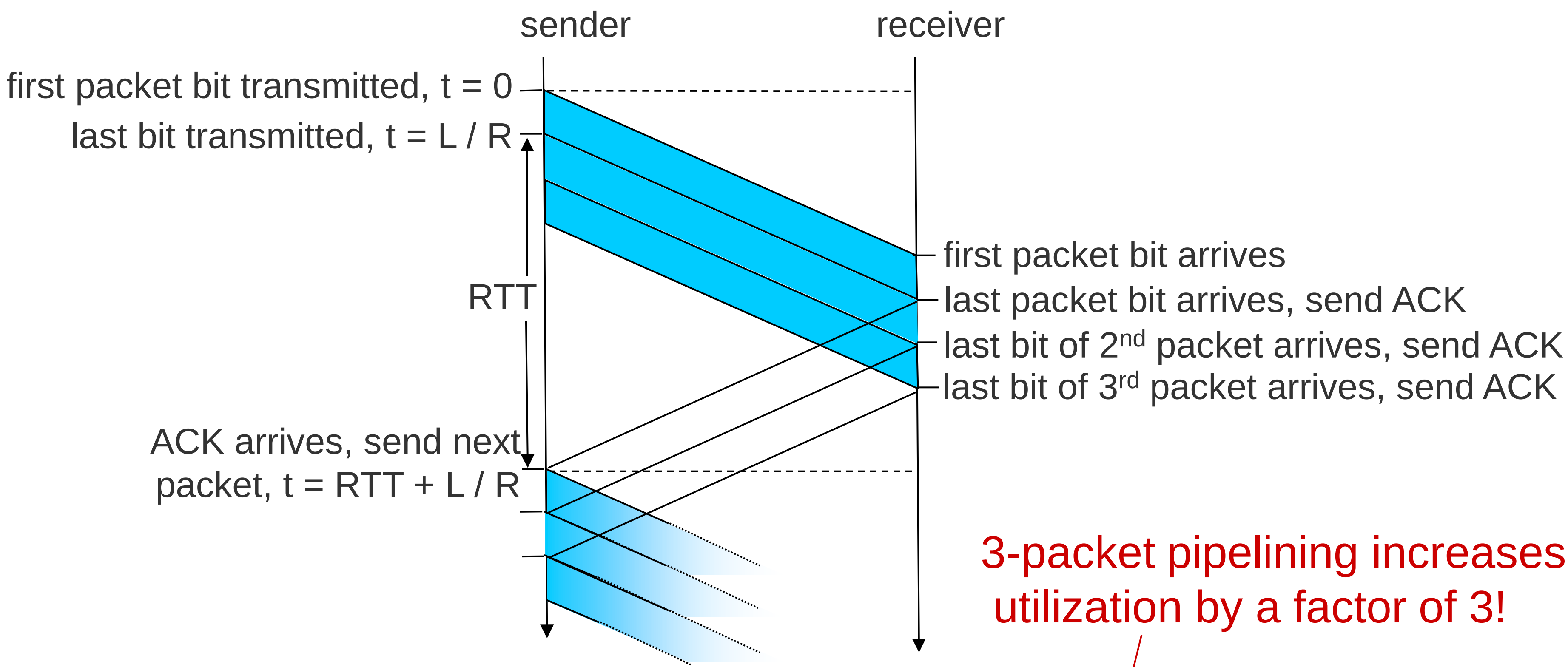


- Allow multiple, “in-flight”, yet-to-be-acknowledged pkts
- Example: received ACK 5, window 6



- Often referred to as sliding window

Reliability: Pipelined Utilisation



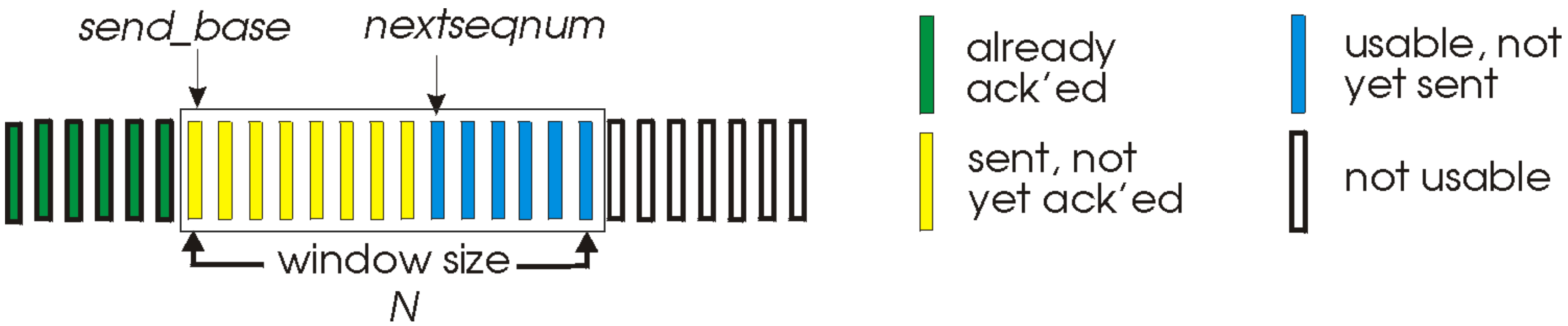
3-packet pipelining increases utilization by a factor of 3!

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

Reliability: Pipelined Approaches

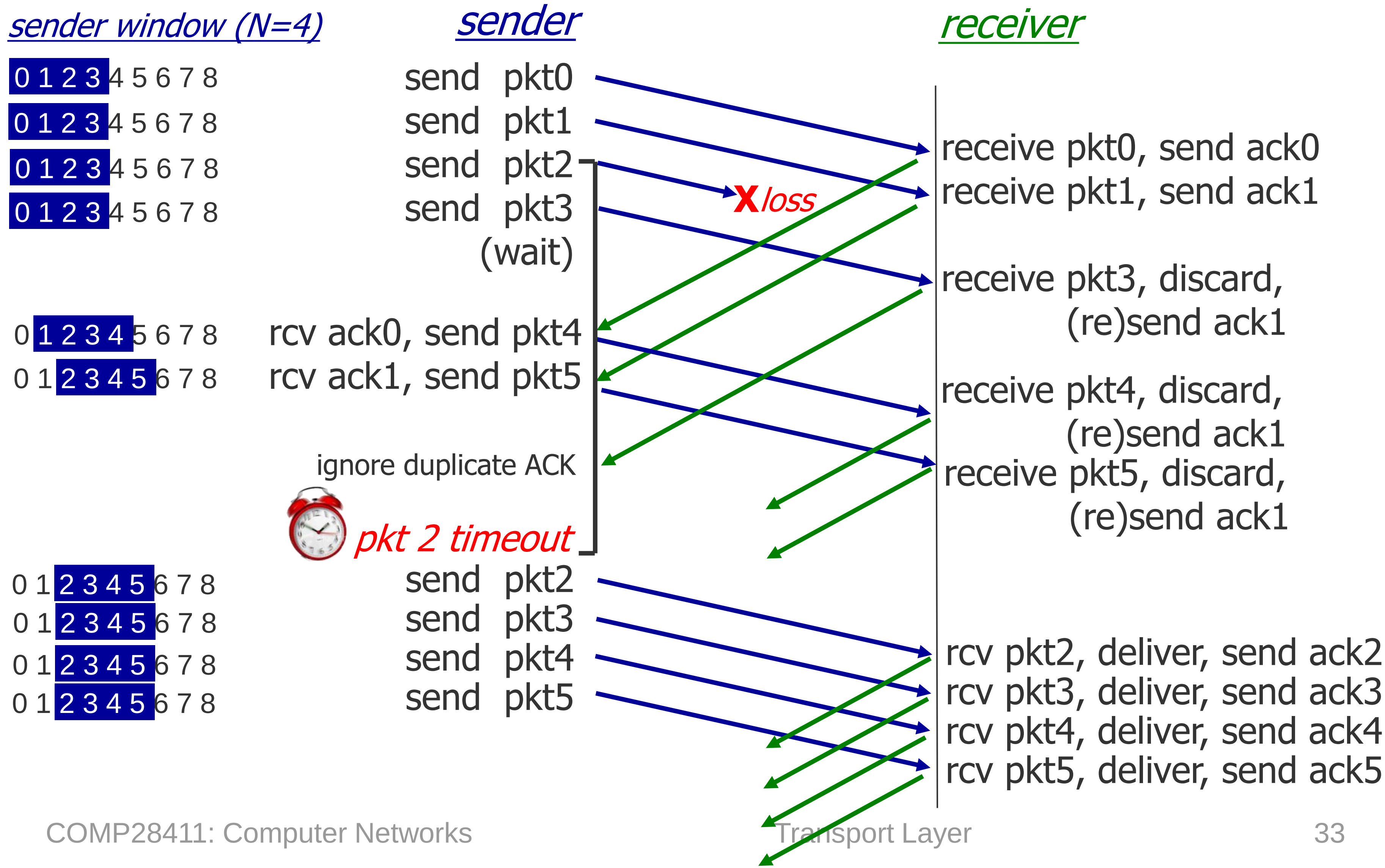
- What if expecting packet 8 and get packet 9?
- If packet 8 delayed, when arrives can send ACK(9)
- If lost, timeout will expire and will be resent
- **Go-Back-N**, send cumulative ACKs:
  - ACK(n) acknowledges all packets upto n
  - likely that will also get packets 9, 10, … resent
- **Selective repeat**:
  - explicitly acknowledges all packet
  - only unsuccessfully received packets are resent
- Could **negatively acknowledge** (NACK) packet 8,
  - requests retransmission without timeout expiring

Reliability: Go-Back-N

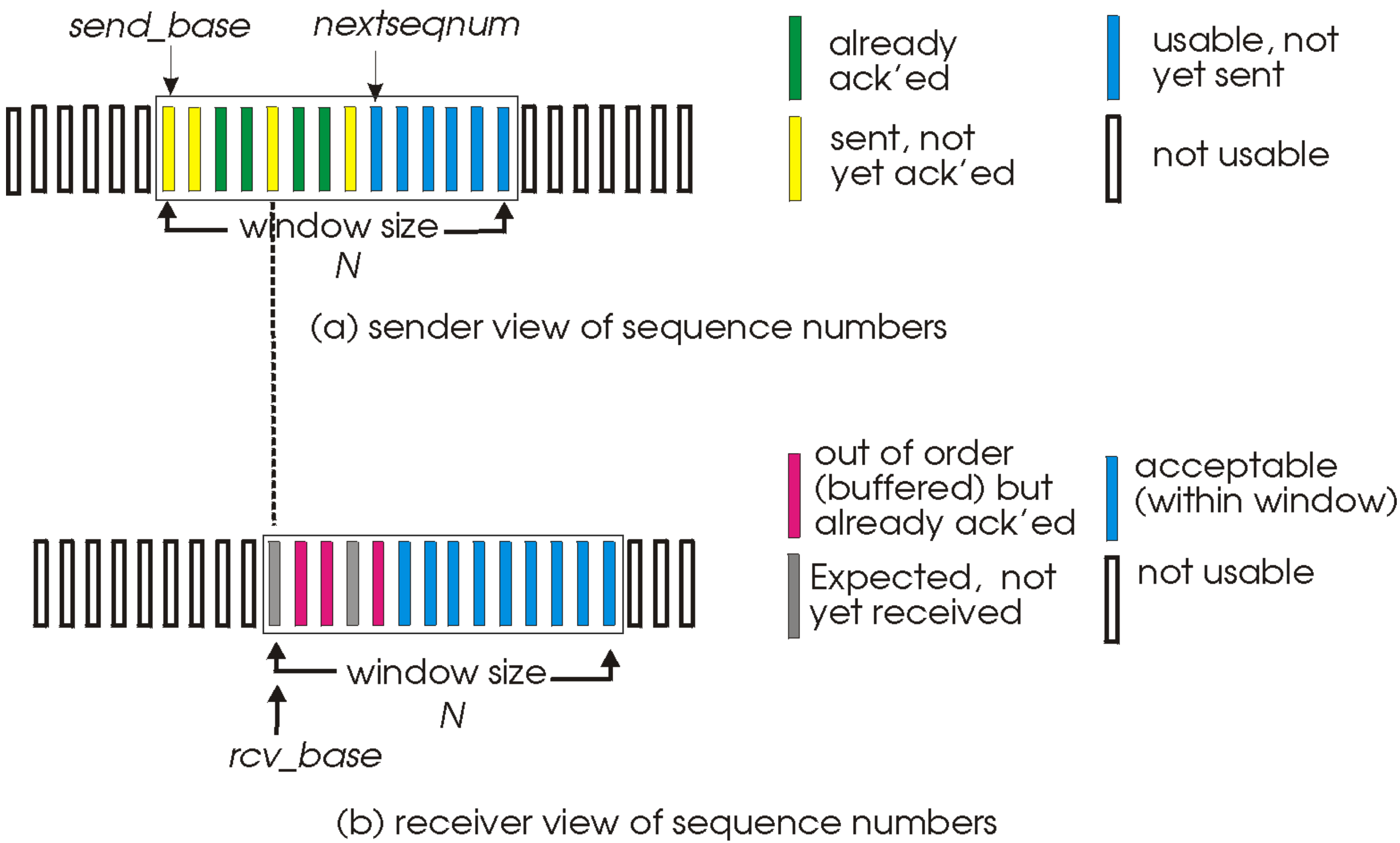


- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed
- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
- may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window

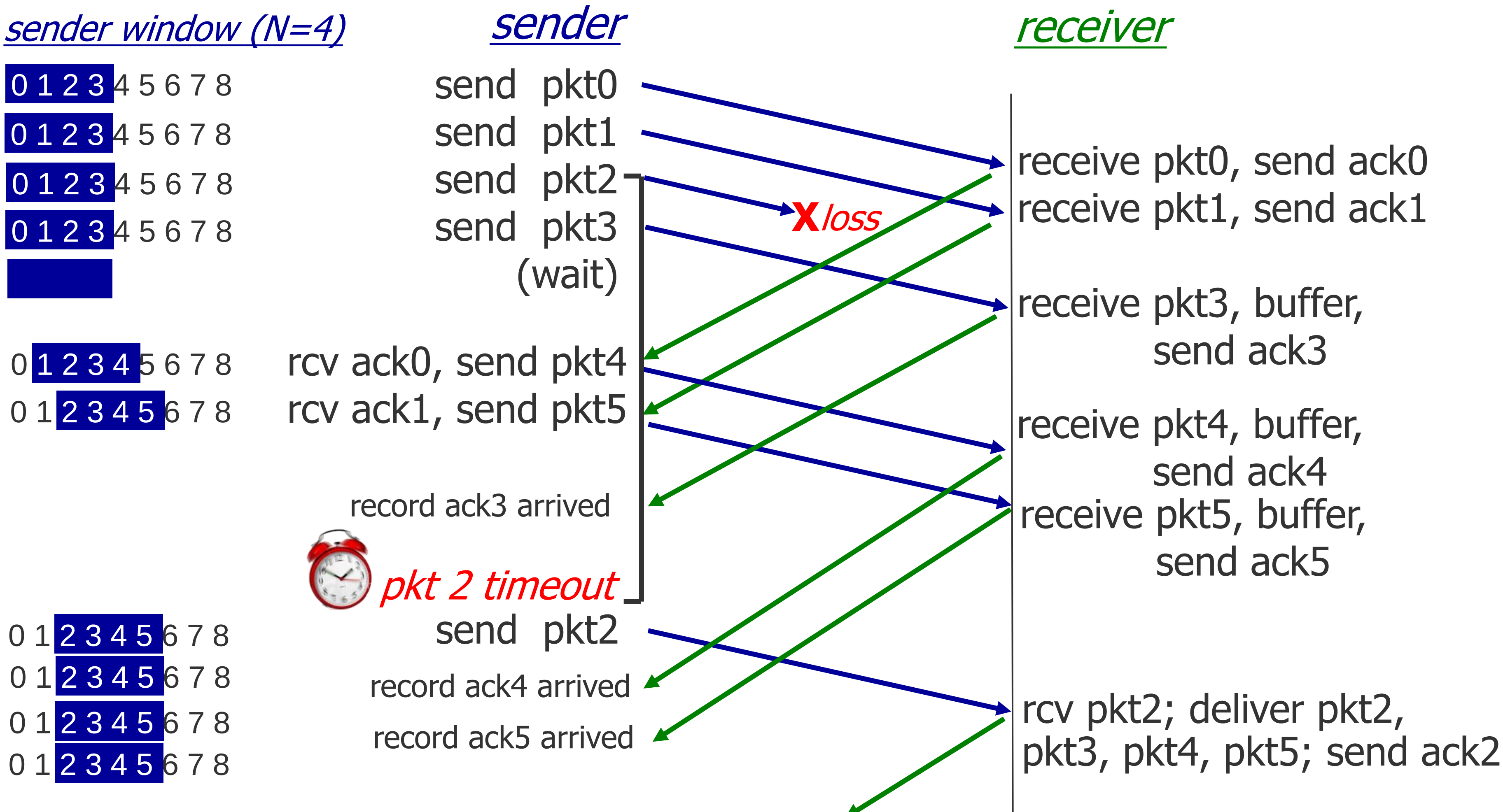
Reliability: Go-Back-N Example (H



Reliability: Selective Repeat



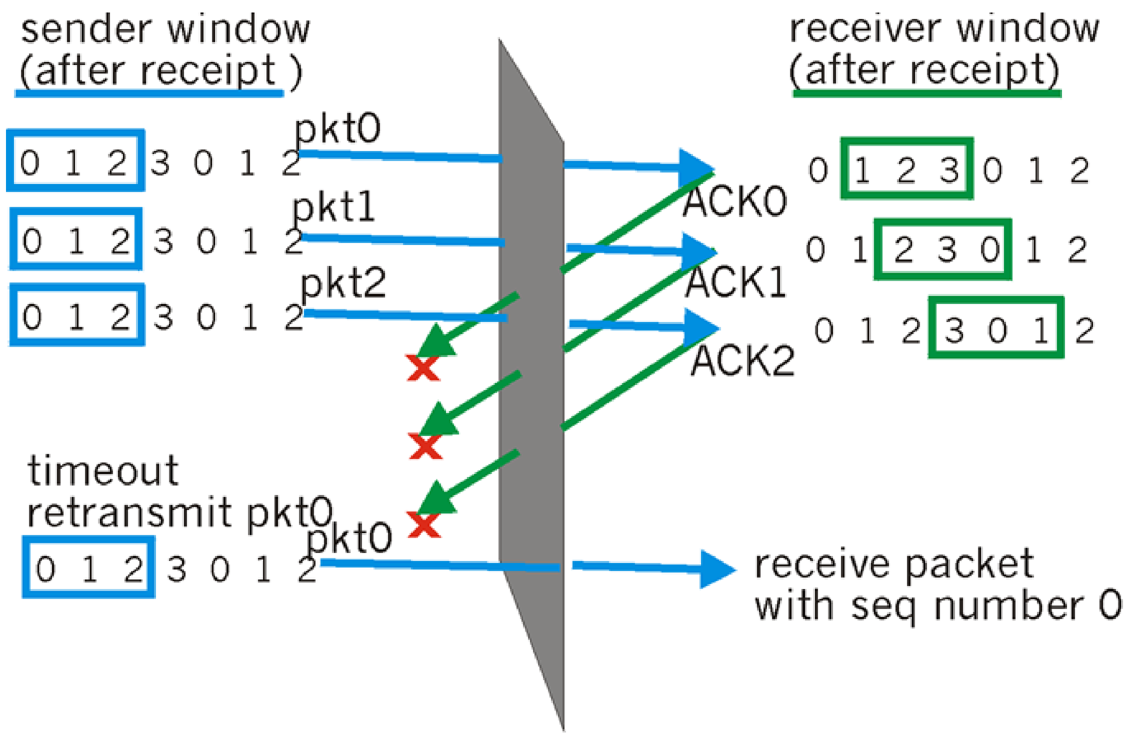
Reliability: Selective Repeat



Q: what happens when ack2 arrives?

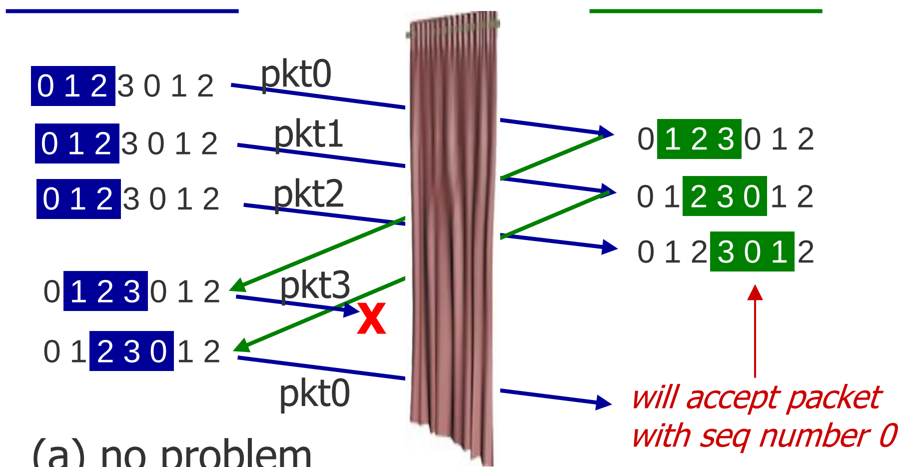
Sliding Windows – Finite Seq. Numbers

- Sequence numbers wrap, must interpret correctly
- For example,

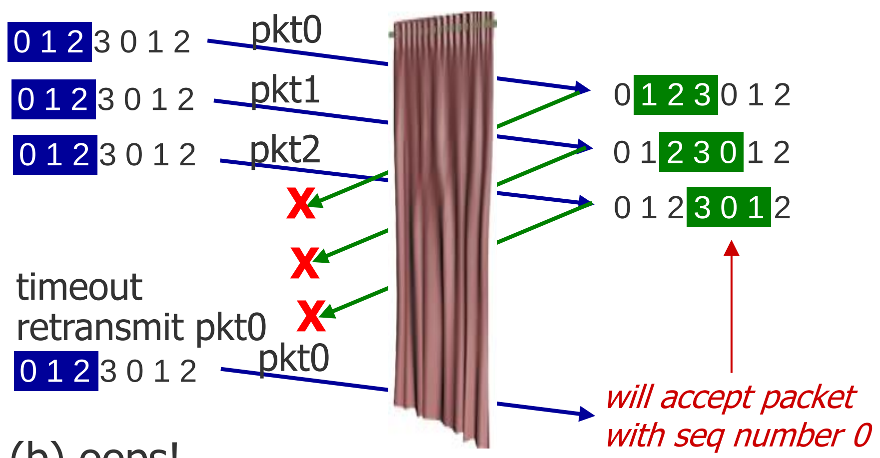


- Max window size = (max sequence number + 1)/2

Reliability: Window Issue



(a) no problem



(b) oops!

Transport Services and Protocols (2)

Andy Carpenter

(Andy.Carpenter@manchester.ac.uk)

Transmission Control Protocol

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

- 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 Format

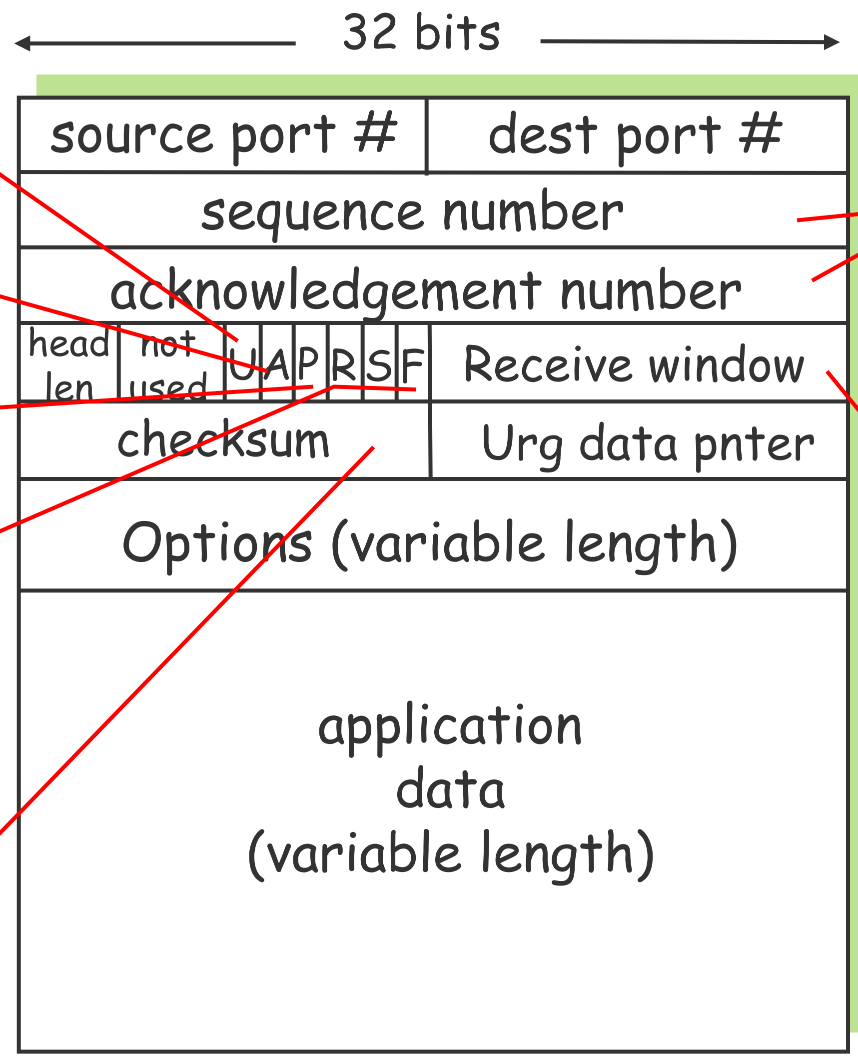
URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection establishment (setup, teardown commands)

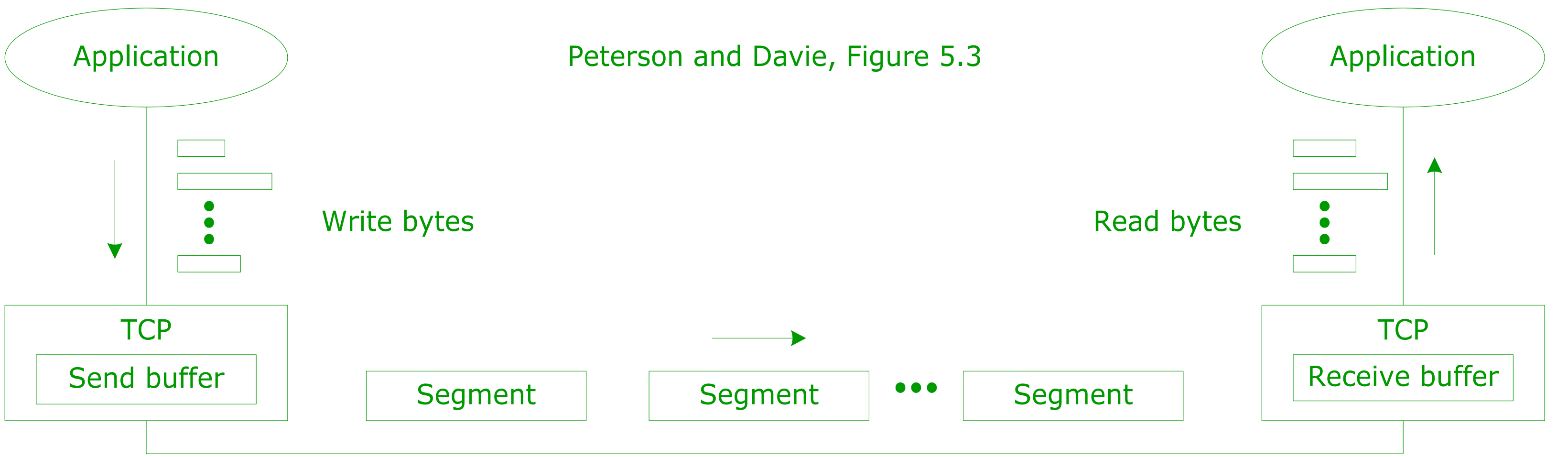
Internet checksum (as in UDP)



counting by bytes of data (not segments!)

bytes receiver willing to accept

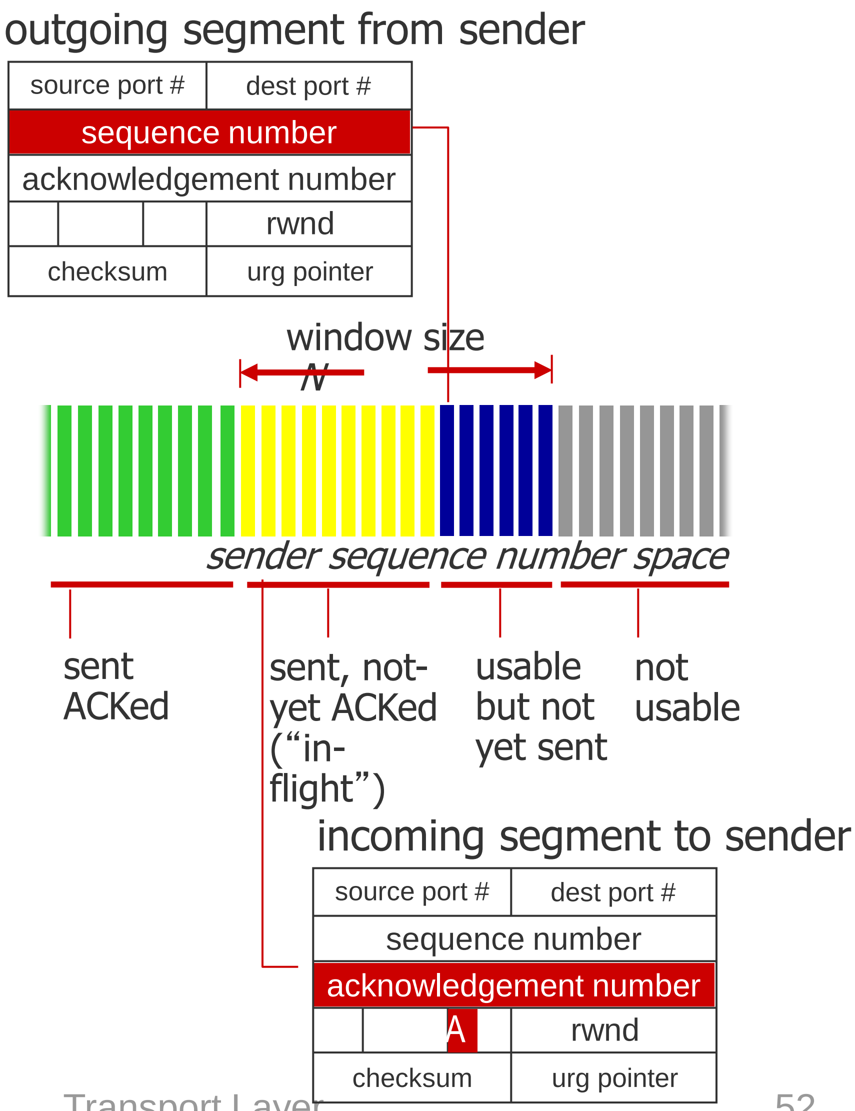
TCP: Segmentation



- Application writes bytes to connection as needs
- TCP collects (buffers) bytes before sending
- Sends set of bytes as a TCP segment in IP datagram
- Receiving TCP also buffers
- Application reads bytes from connection as required

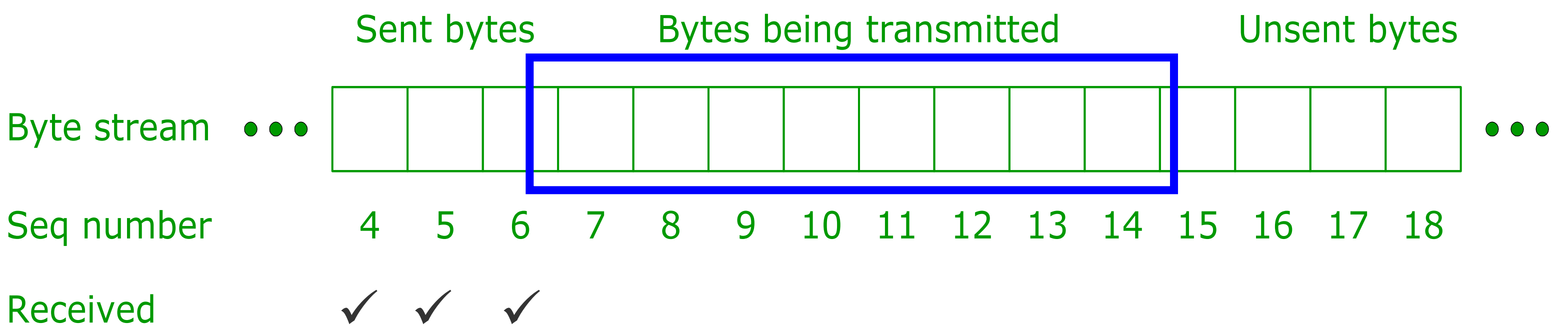
TCP seq. numbers, ACKs

- sequence numbers:
  - byte stream "number" of first byte in segment's data
- acknowledgements:
  - seq # of next byte expected from other side
- cumulative ACK



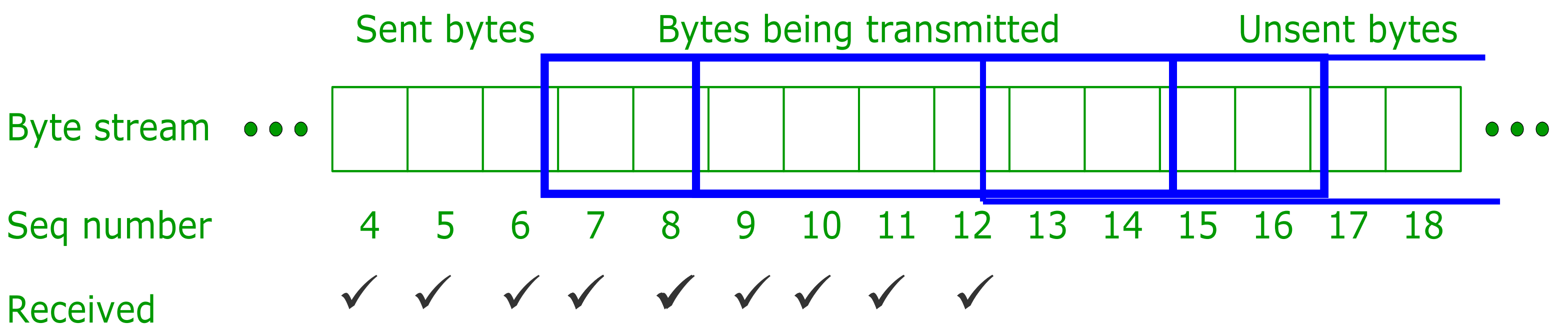
TCP: Reliability

- Example, window 8, last ACK 7



TCP: Reliability

- Example, window 8, last ACK 7



Receive segment containing bytes with sequence numbers 7 and 8

Acknowledge byte with sequence number 9 and slide window along byte stream

Receive segment containing bytes with sequence numbers 10, 11 and 12

Note bytes received, but do not acknowledge or slide window

Receive segment containing byte with sequence number 9

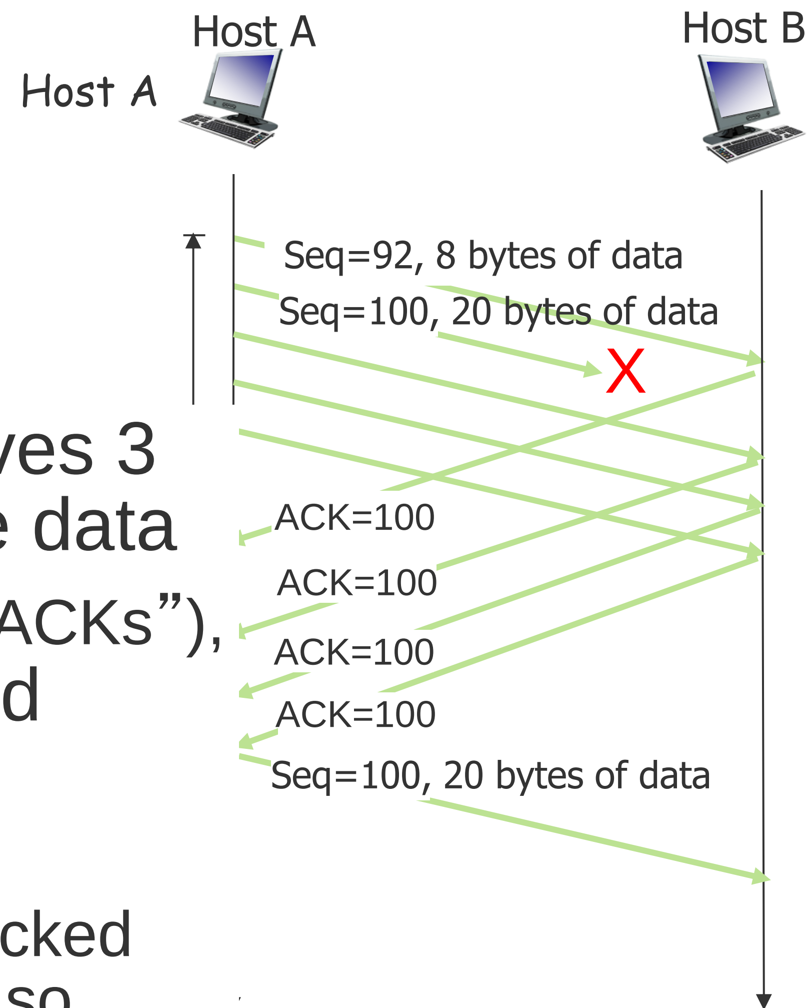
Acknowledge byte with sequence number 13 and slide window along byte stream

Reliability: TCP – Fast Retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost duplicate A
  - sender (many se to-back
  - if segme there wi many du

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

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



st retransmit after sender ceipt of triple duplicate ACK

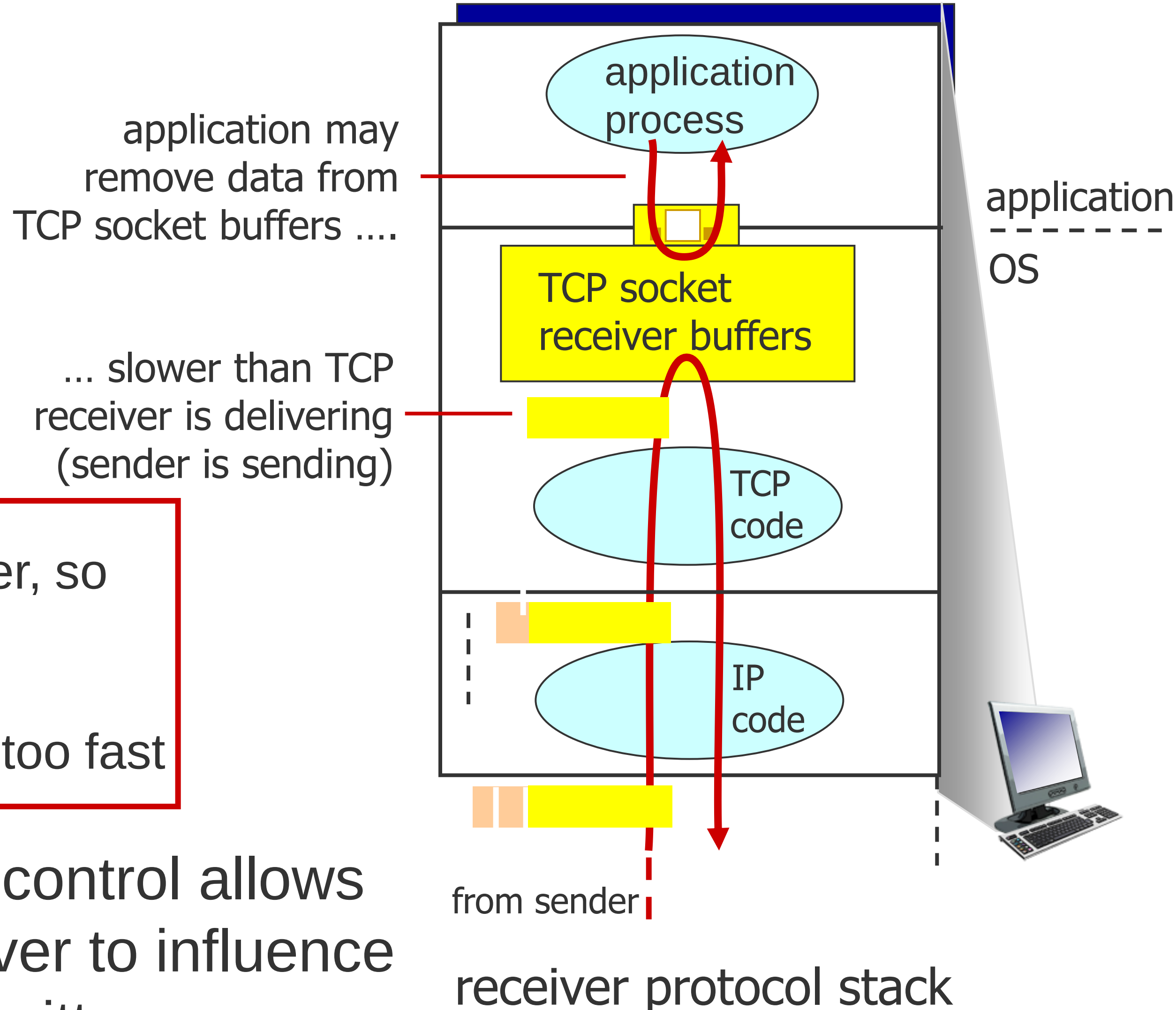
Reliability: Retransmit Timeout

- If value too small, unnecessarily retransmit segments
- If value too large, get excessive delays before retransmit
- Appropriate value related to RTT, but that depends on:
  - pair of hosts involved, time, congestion in network
- For each connection:
  - use adaptable algorithm to determine RTT
- Basic algorithm sets timeout to twice estimated RTT
- Karn/Partridge algorithm reduces miscalculations
- Jacobson/Karels algorithm
  - copes with significant variance in real RTT

Reliability: Estimation of RTT

- Estimate RTT as average of measured RTTs:
- When re-sending, to which transmit does ACK relate?
- Karn/Partridge algorithm:
  - only measures RTT for non-retransmitted segments
  - doubles timeout value for each retransmission
- Jacobson/Karels Algorithm:
  - if RTT variation small, average is good approx.
  - if RTT variation large, average is poor approx.
  - algorithm takes account of variation

Flow Control



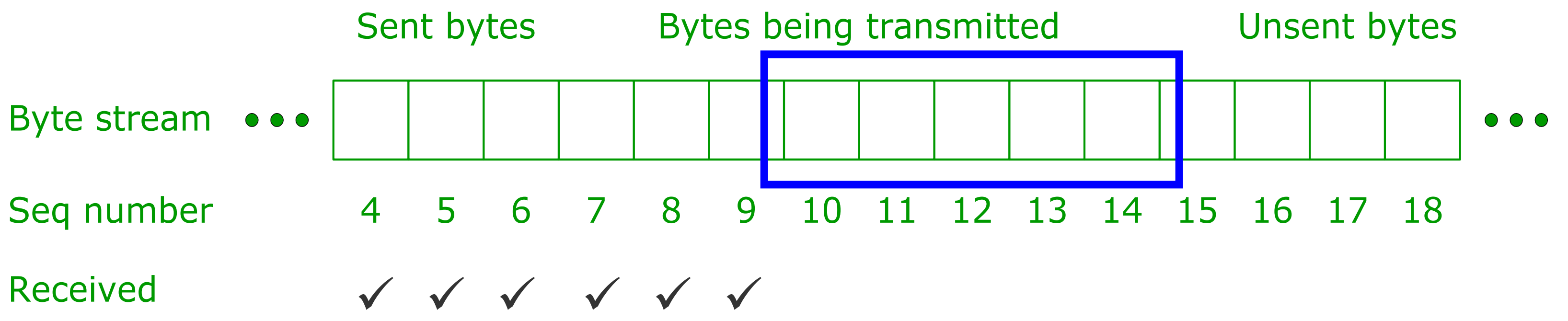
flow control

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

Flow control allows receiver to influence transmitter

Flow Control: Example

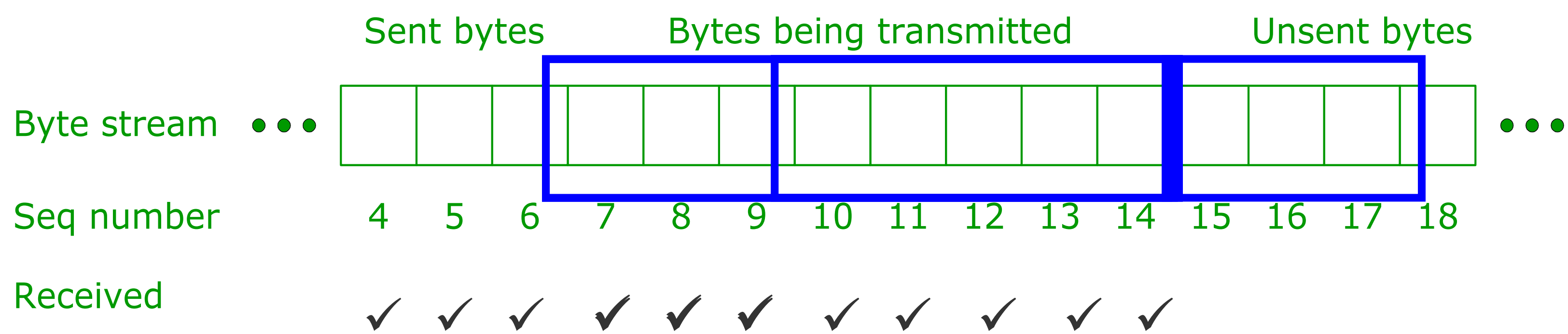
- Uses variant of sliding windows algorithm
  - window size can be changed



- Can get deadlock

Flow Control: Example

- Uses variant of sliding windows algorithm
  - window size can be changed



Receive segment containing bytes with sequence numbers 7, 8 and 9

edge byte with sequence 0 and reduce window size

Receive segment containing bytes with sequence numb 10-14

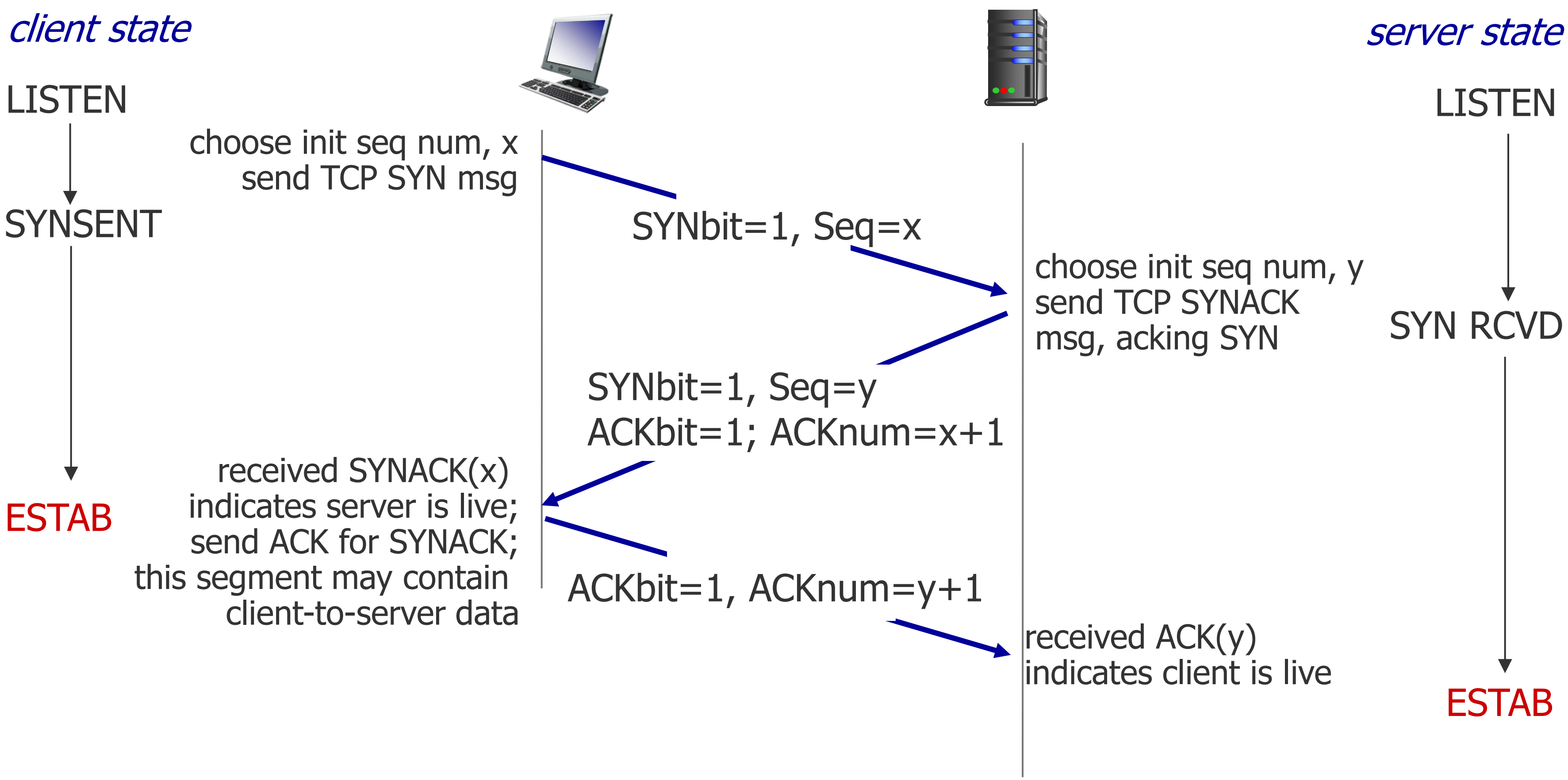
edge byte with sequence

to

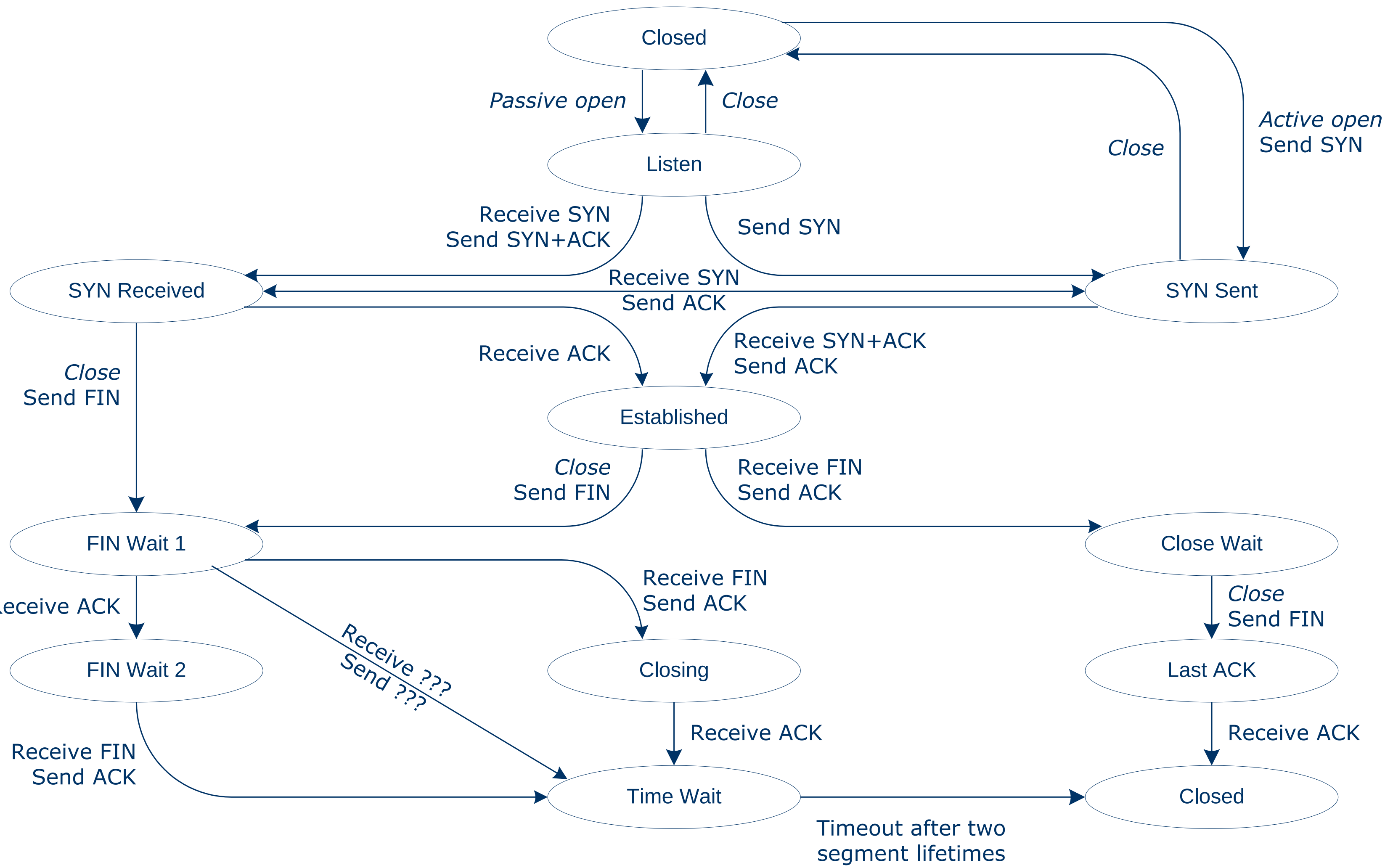
Re-acknowledge byte with sequence number 15 and increase window size

- Can get deadlock

TCP 3-way handshake



TCP: State Transition Diagram



TCP: Keeping the Pipe Full

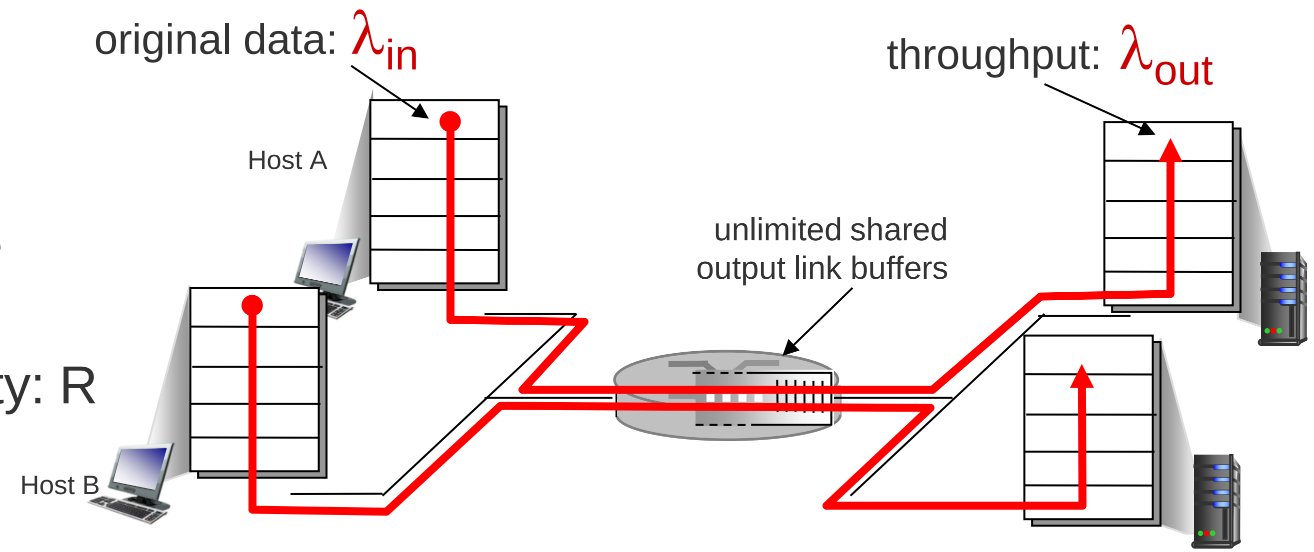
- Capacity of physical connection (100ms RTT)

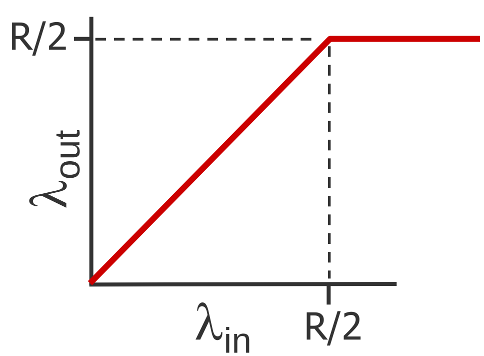
T1 (1.5Mbps)	18KB	Ethernet (10Mbps)	122KB
T3 (45Mbps)	549KB	FDDI (100Mbps)	1.2MB
STS-3 (155Mbps)	1.8MB	STS-12 (622Mbps)	7.4MB
STS-24 (1.2Gbps)	14.8MB		

- Maximum window of single TCP connection: 216 = 64KB
- Maximising use of cables relies on shared use
- Approaching point where:
  - capacity on cable exists but no acknowledgement
- Proposed extension allows:
  - multiplier factor for sequence number and window size

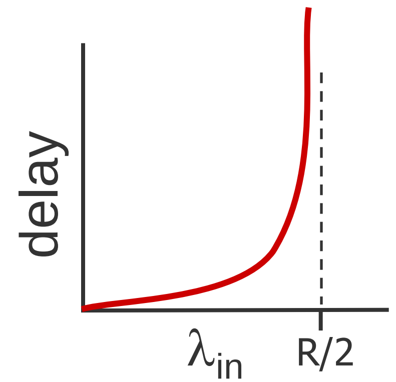
Congestion: Causes/Costs 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission



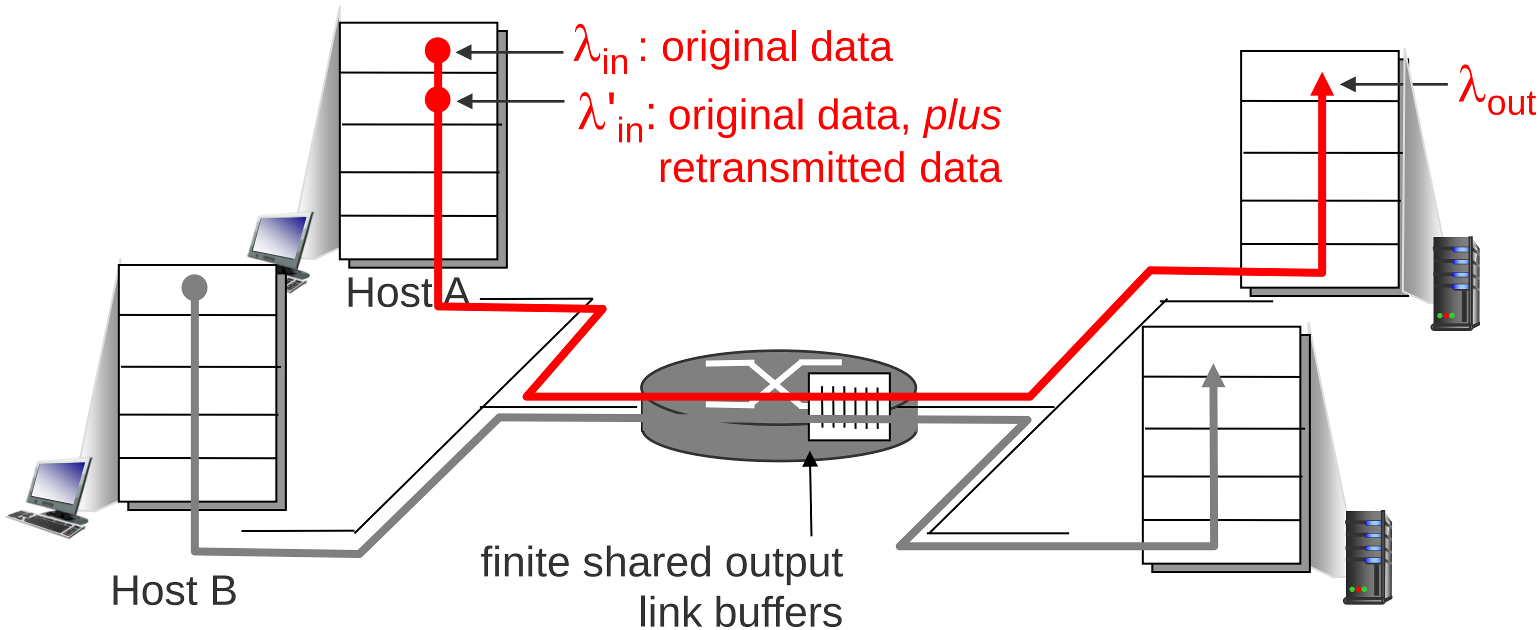


- maximum per-connection throughput: R/2



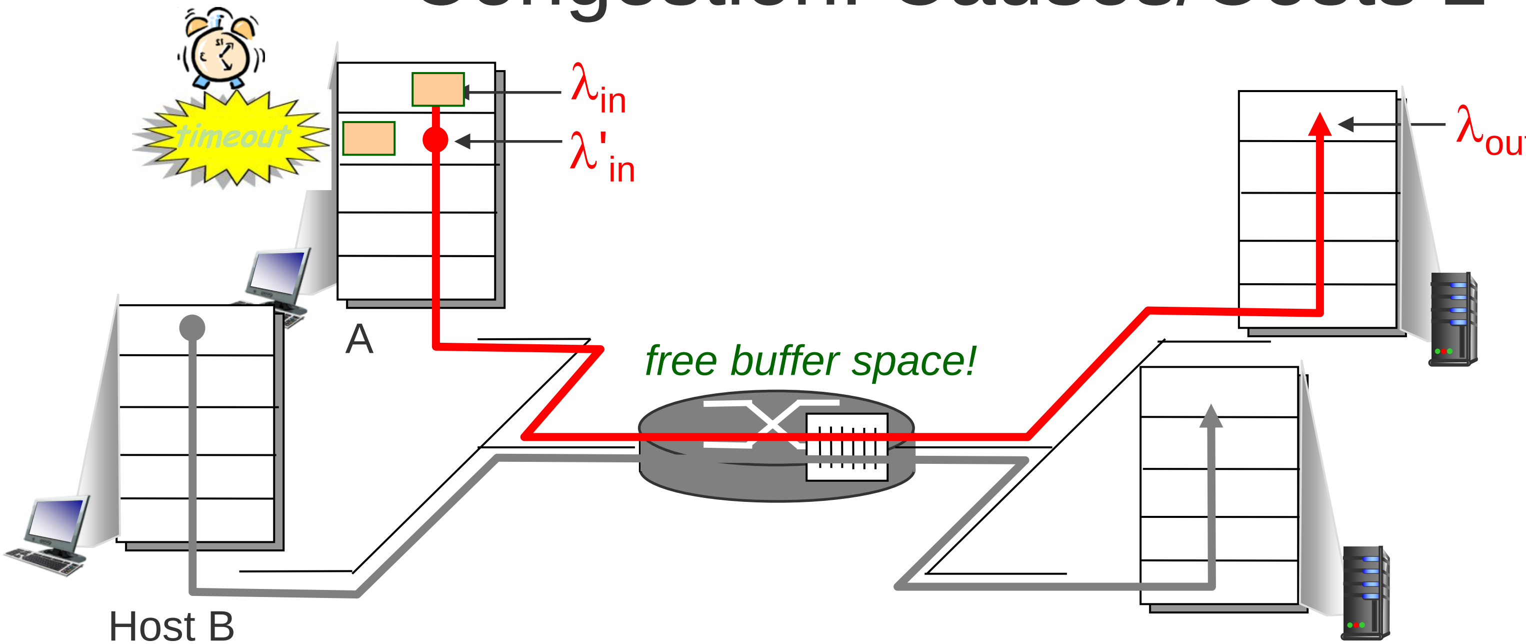
- large delays as arrival rate, λ_{in}, approaches capacity

Congestion: Causes/Costs 2



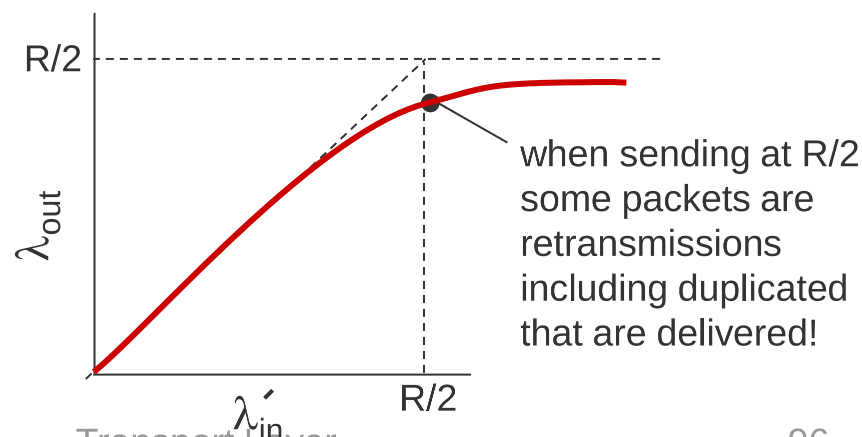
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : λ_{in} λ_{in}

Congestion: Causes/Costs 2

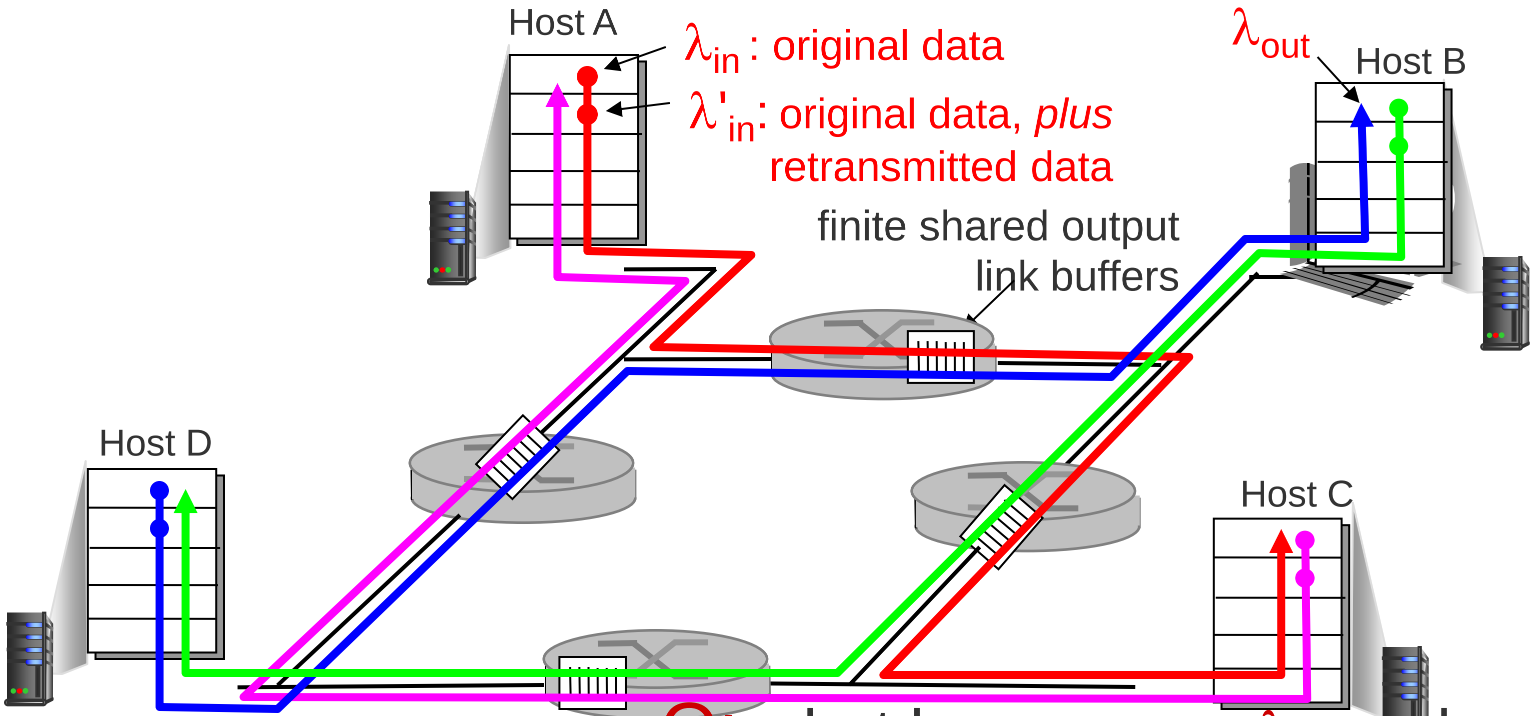


Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



Congestion: Causes/Costs 3

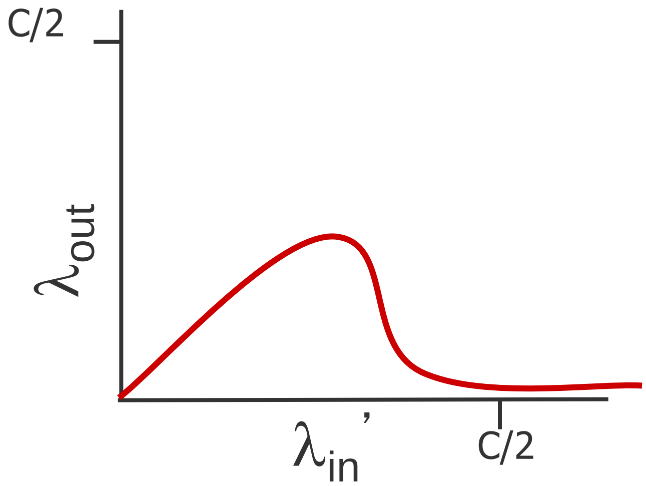


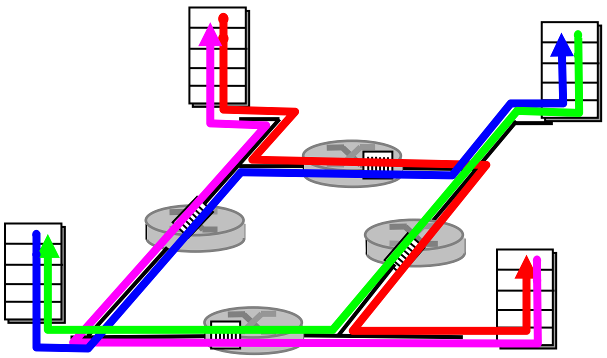
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow$ 0

Congestion: Causes/Costs 3





- another "cost" of congestion:
- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Congestion: Control Approaches

end-end congestion control:

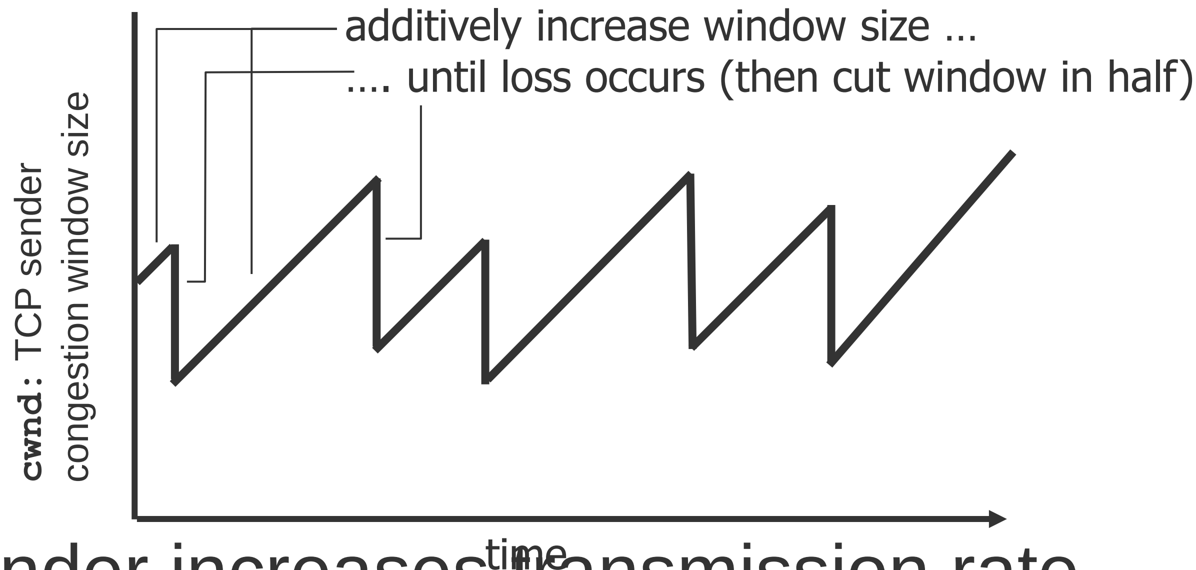
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

Congestion: TCP Control

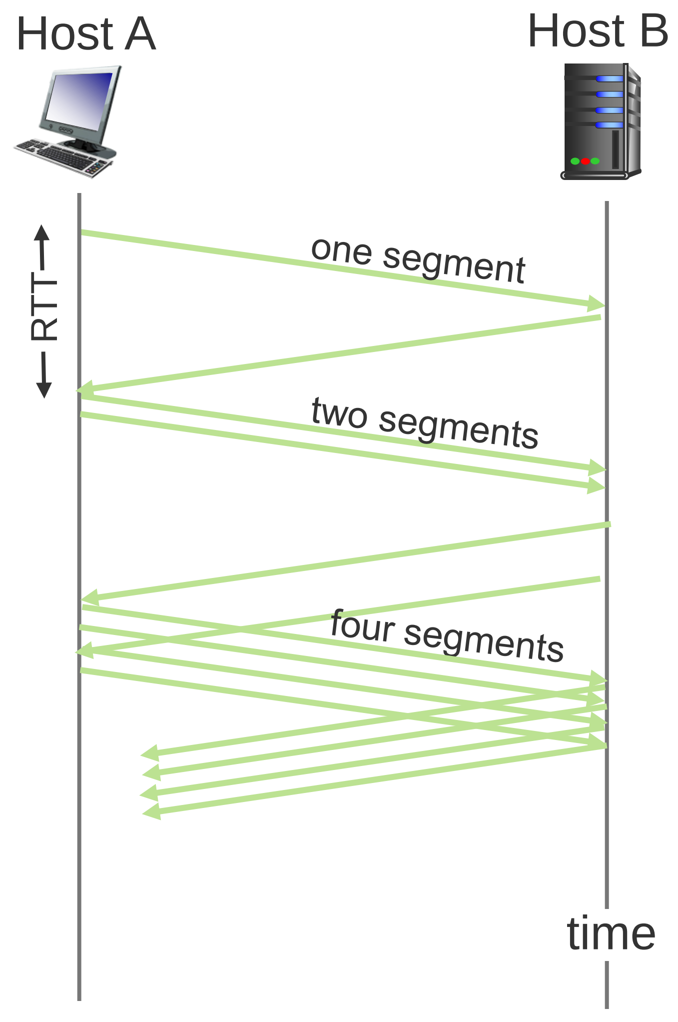
AIMD saw tooth behavior: probing for bandwidth



- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **cwnd** in half after loss

TCP Slow Start

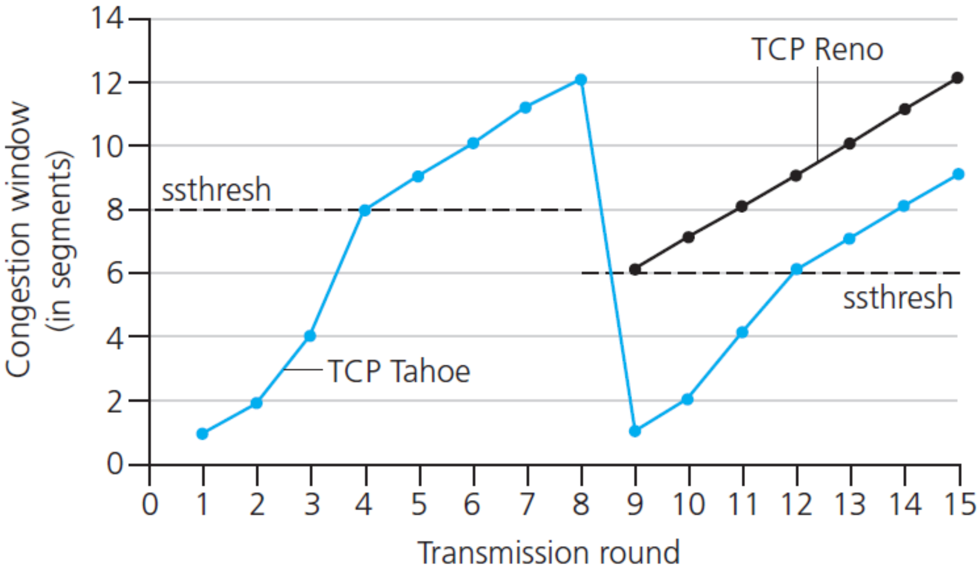
- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- <u>*summary:*</u> initial rate is slow but ramps up exponentially fast



TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

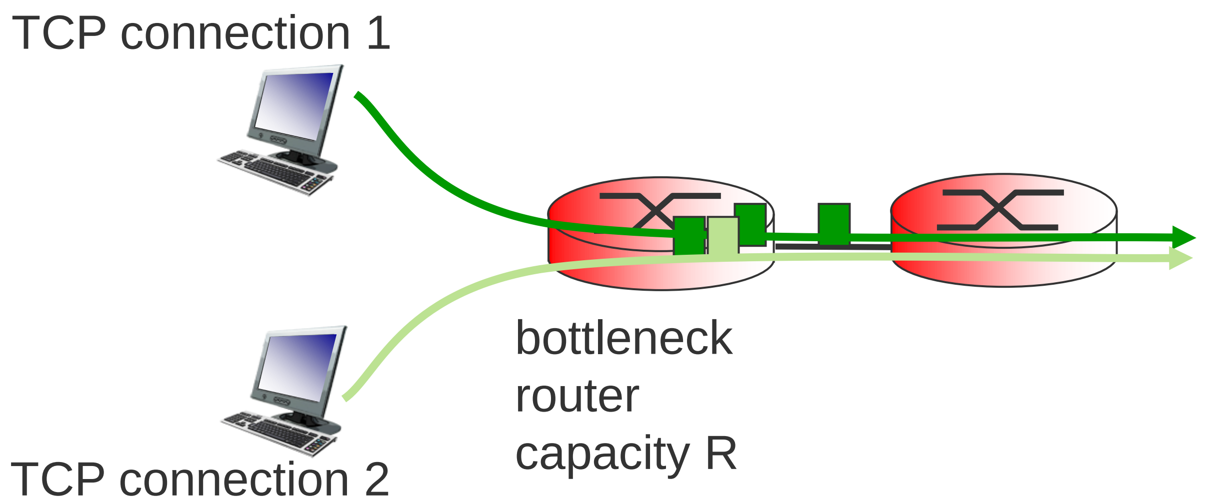
A: when **`cwnd`** gets to 1/2 of its value before timeout.



Implementation:

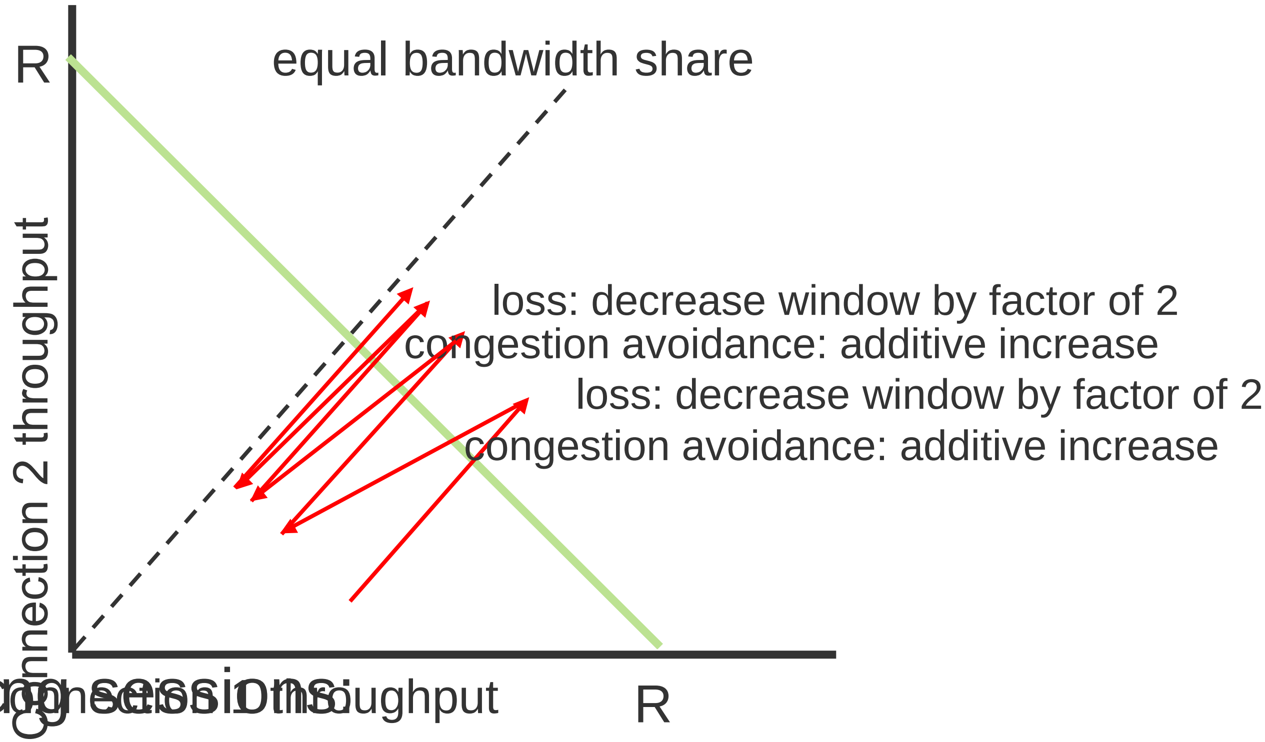
- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

Fairness: TCP



- *fairness goal:* if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

Fairness: TCP



two competing sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally

Summary

- Examined end-to-end issues
- End-point identification, multiplexing/demultiplexing
  – all that UDP does
- Reliable data transfer
  – one important aspect is adaptive timeout
- Flow control
- Congestion control
- TCP use all to simplify applications