Master Course
Computer Networks
IN2097

Prof. Dr.-Ing. Georg Carle
Christian Grothoff, Ph.D.
Dr. Nils Kammenhuber

Chair for Network Architectures and Services
Institut für Informatik
Technische Universität München
http://www.net.in.tum.de

Chair for Network Architectures and Services – Prof. Carle
Department for Computer Science
TU München

Transport Layer

Chapter: Transport Layer

Our goals:

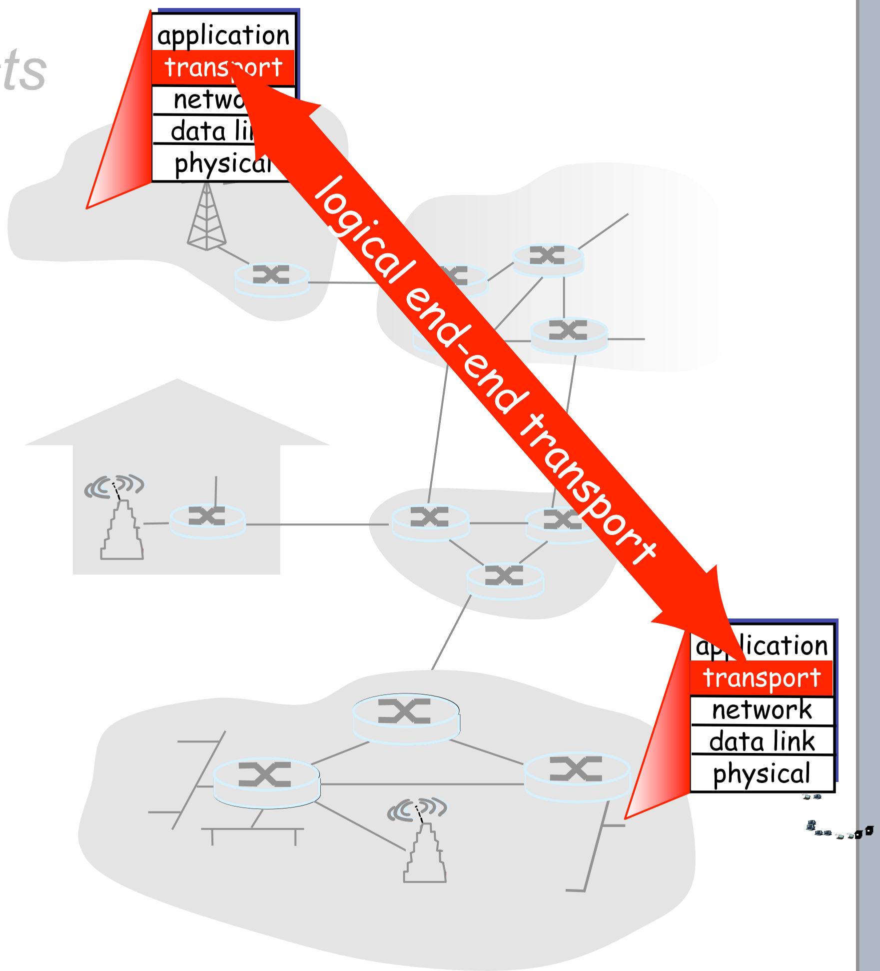
- Understand *principles* behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Learn about transport layer *protocols* in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
    - TCP congestion control
  - (Maybe: SCTP, if time permits)

Chapter 3 outline

- **Transport-layer services**
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- TCP congestion control

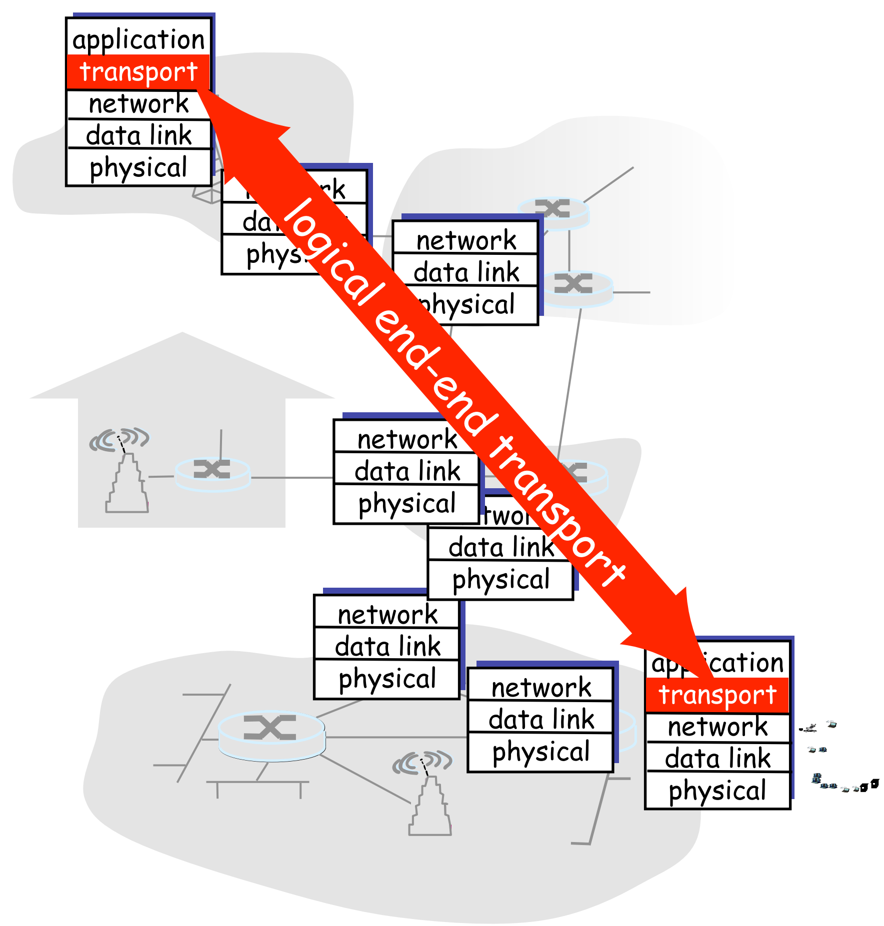
Transport services and protocols

- Provide logical communication *between application processes* running on different hosts
  - ↔Network layer: *between hosts*
- Transport protocols run in end systems
  - Sender side: breaks app messages into segments, passes to network layer
  - Rcver side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: mainly TCP, UDP



Internet transport-layer protocols

- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

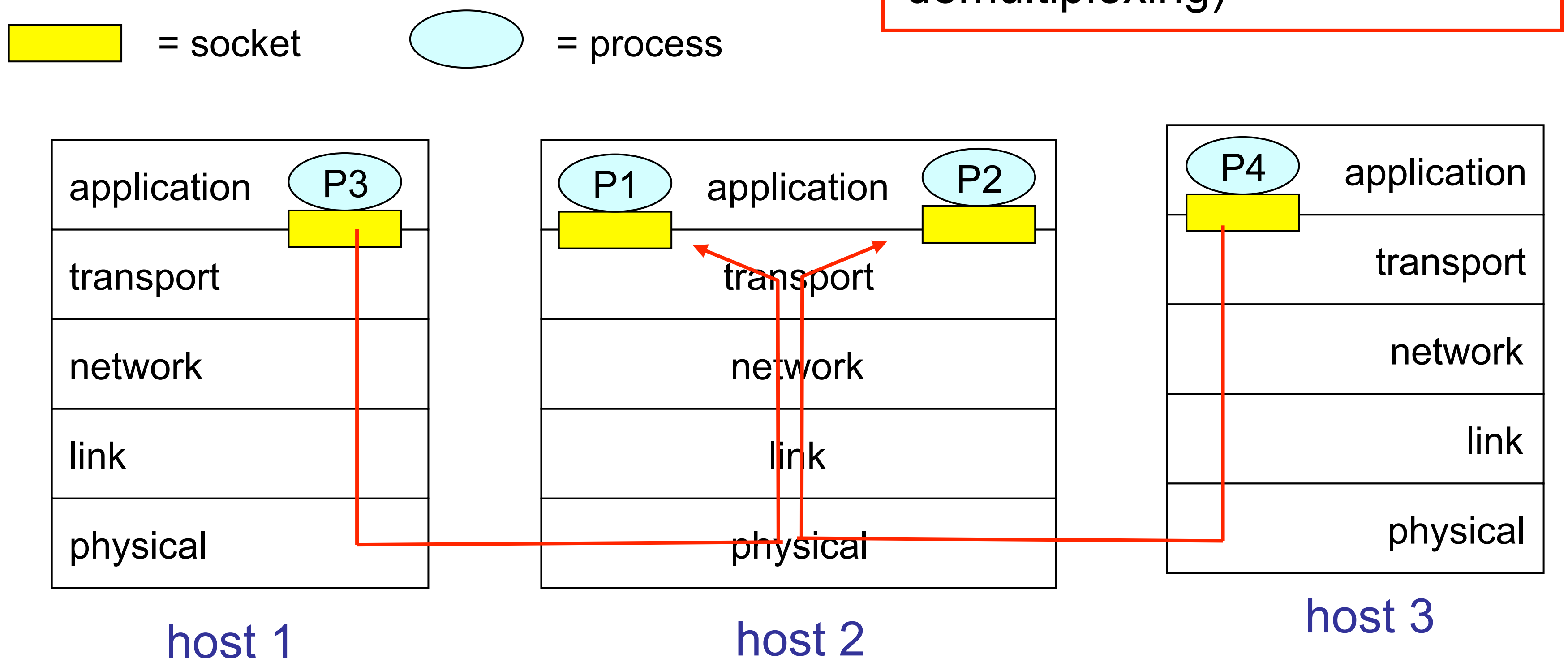
Socket: File handle that allows to send/receive network traffic

Demultiplexing at rcv host:

Delivering received segments to correct socket

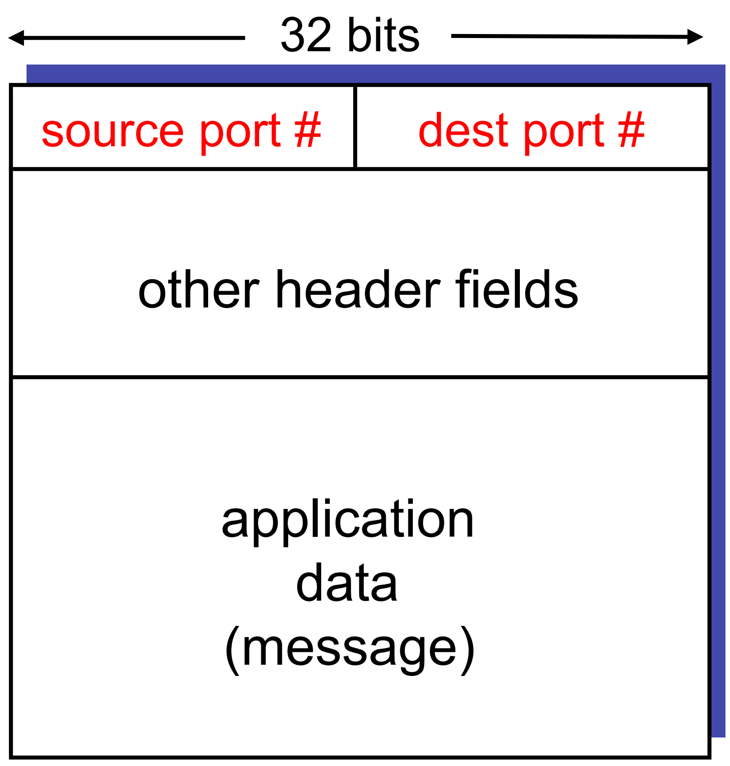
Multiplexing at send host:

Gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- Host receives IP datagrams
  - Each datagram has source IP address, destination IP address
  - Each datagram carries 1 transport-layer segment
  - Each segment has source, destination port number
- Host uses IP addresses *and* port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing (UDP)

- Create sockets with port numbers (in Java):

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

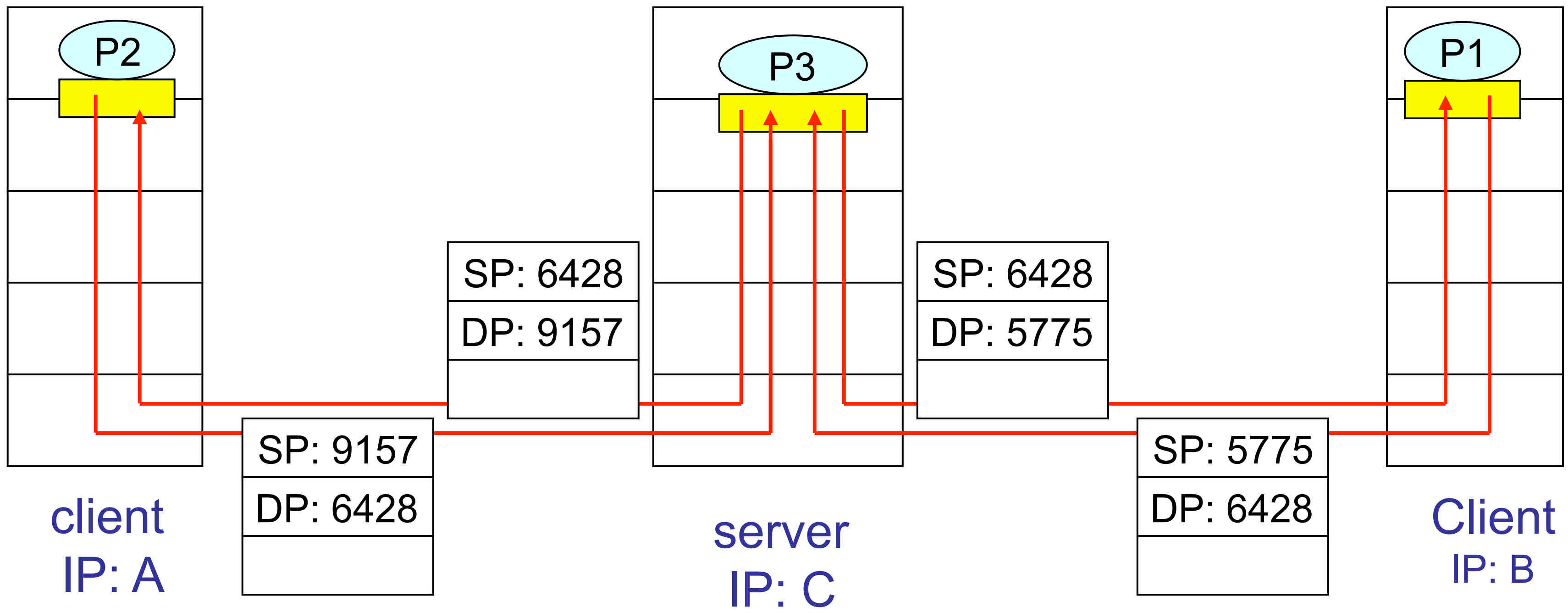
- UDP socket identified by two-tuple:

  (dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers: directed to *same* socket
  - Receiving process cannot easily distinguish differing communication partners on same socket

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

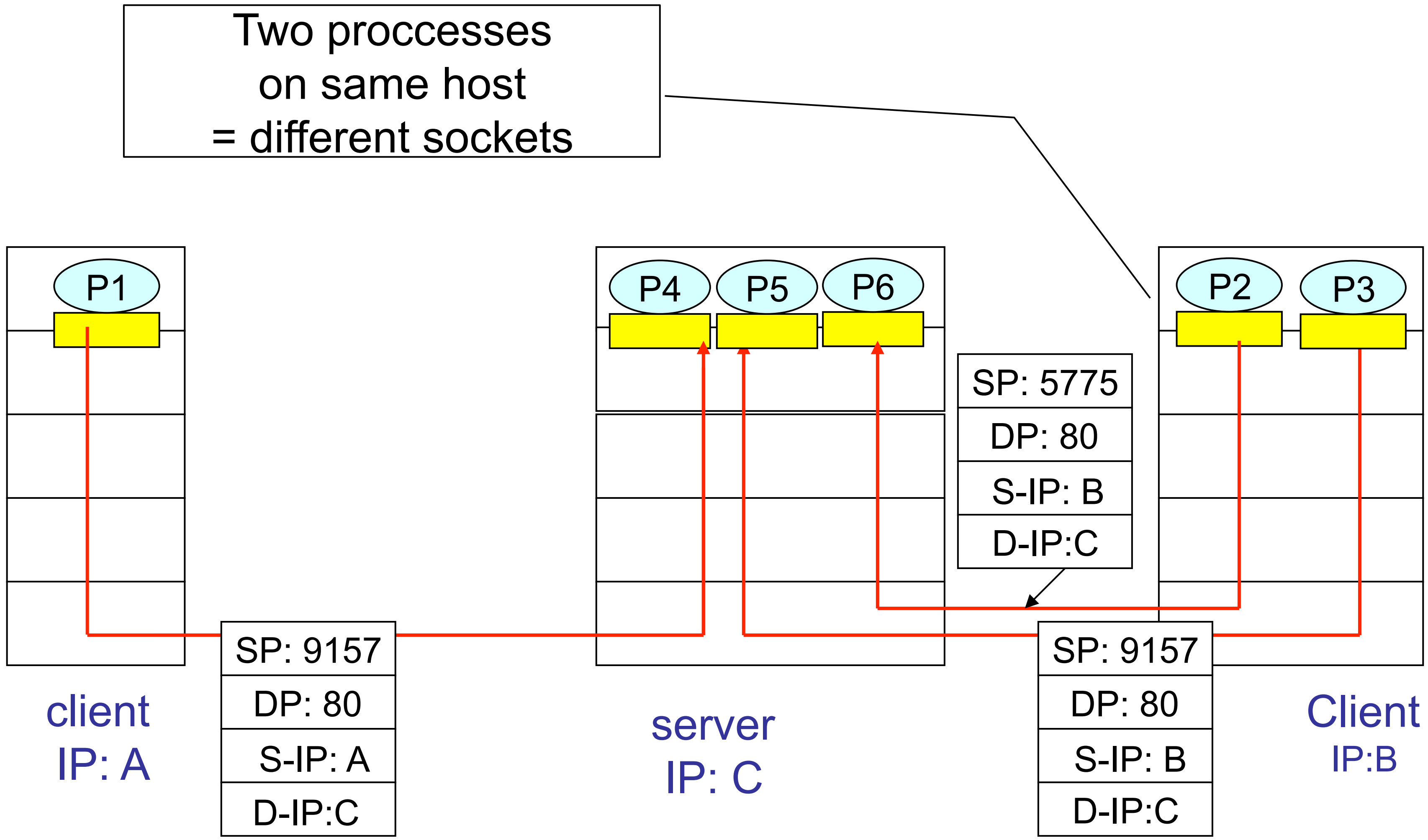


Source Port (SP) provides “return address”

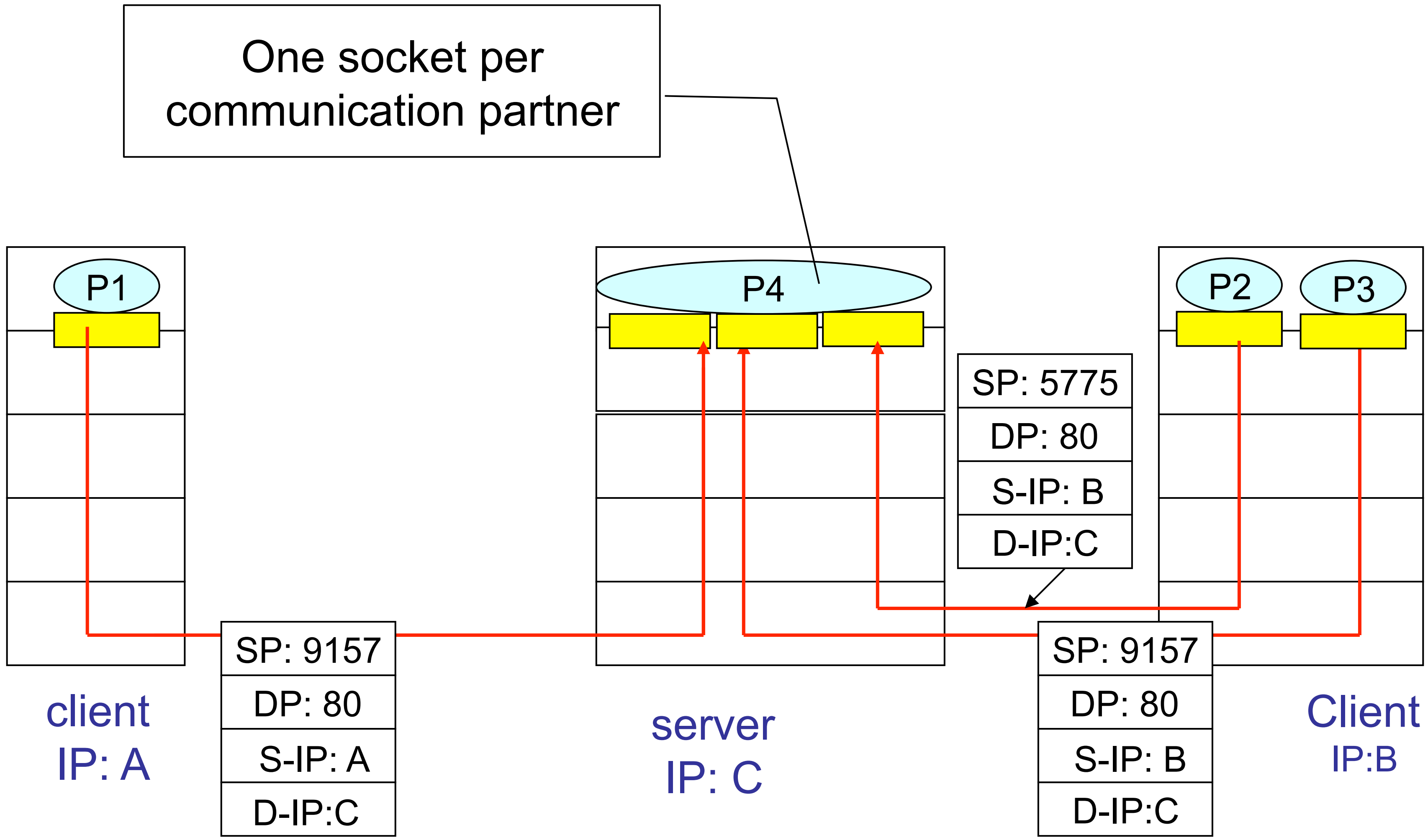
Connection-oriented demux (TCP)

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number
- Receiving host 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
- Example:
  Web servers have different sockets for each connecting client
  - Non-persistent HTTP will even have different socket for each request

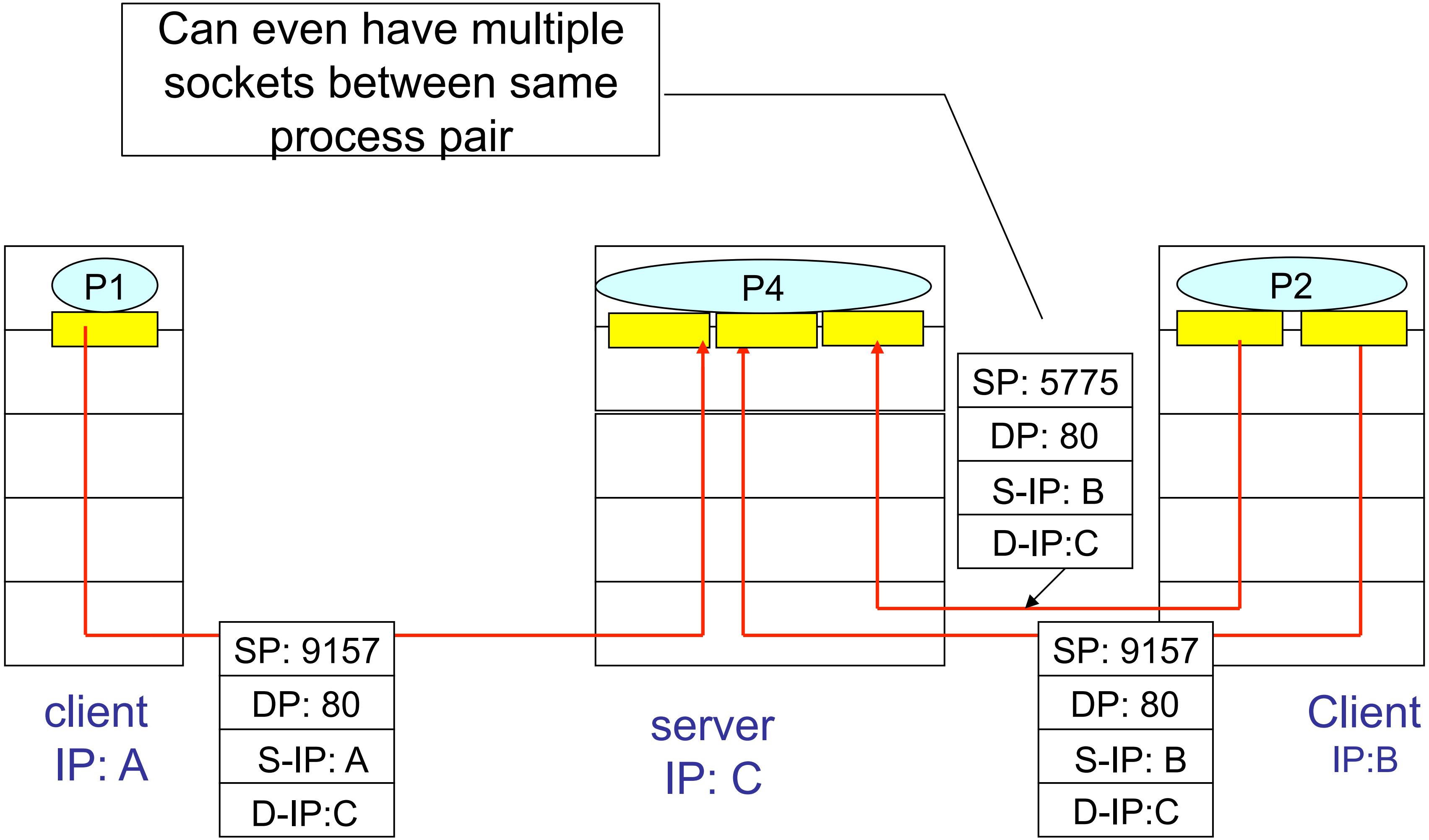
Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



Connection-oriented demux: Fast client



UDP: User Datagram Protocol [RFC 768]

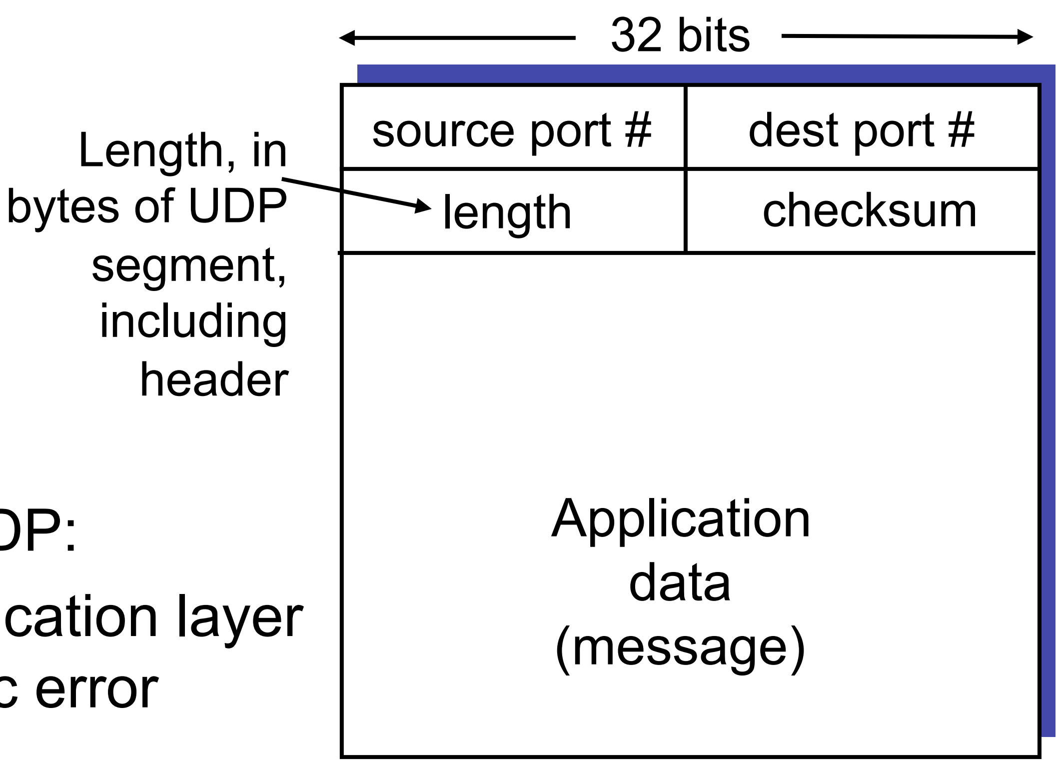
- "No frills," "bare bones" Internet transport protocol
- "Best effort" service; UDP segments may be:
  - lost
  - delivered out of order to app
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state at sender, at receiver
- Small segment header
- No congestion control: UDP can blast away as fast as desired

UDP: more

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses
  - DNS
  - SNMP
  - SIP
- Reliable transfer over UDP:
  - Add reliability at application layer → application-specific error recovery!



UDP segment format

UDP checksum

Goal: Detect TX errors (e.g., flipped bits) in transmitted segment

Sender:

- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1's complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

Receiver:

- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO → error detected. Drop segment.
  - YES → no error detected. *But maybe errors nonetheless?* More later ….

Internet Checksum Example

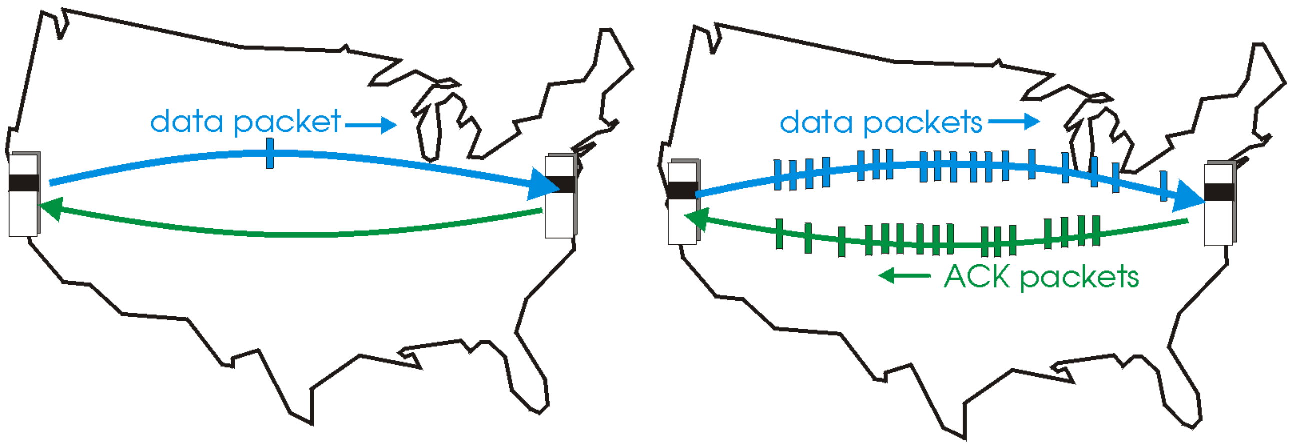
- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

		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
wrap around	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 (=inverse)		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Pipelined protocols

Pipelining: Sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

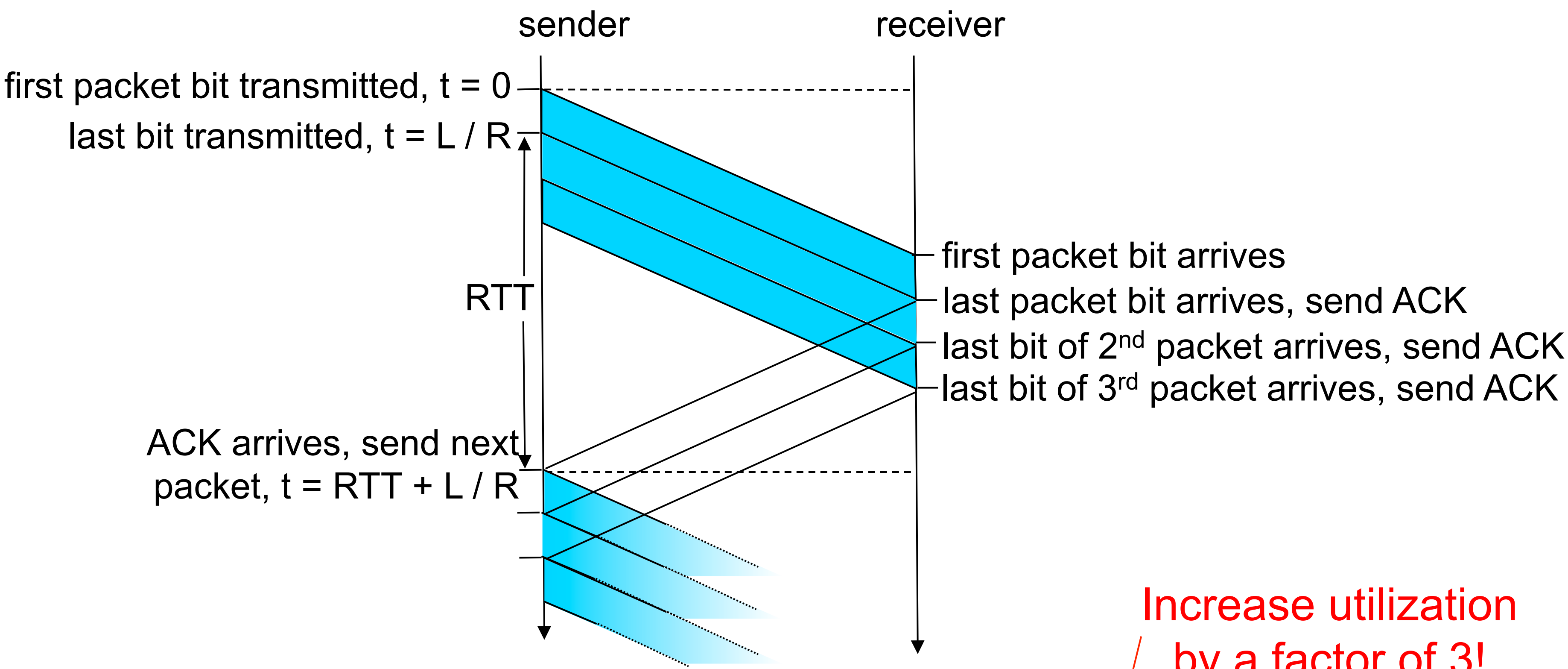
- Range of sequence numbers must be large enough
- Buffering at sender and/or receiver



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

- Two generic forms of pipelined protocols:
  - *Go-Back-N*
  - *Selective repeat*

Pipelining: increased utilization



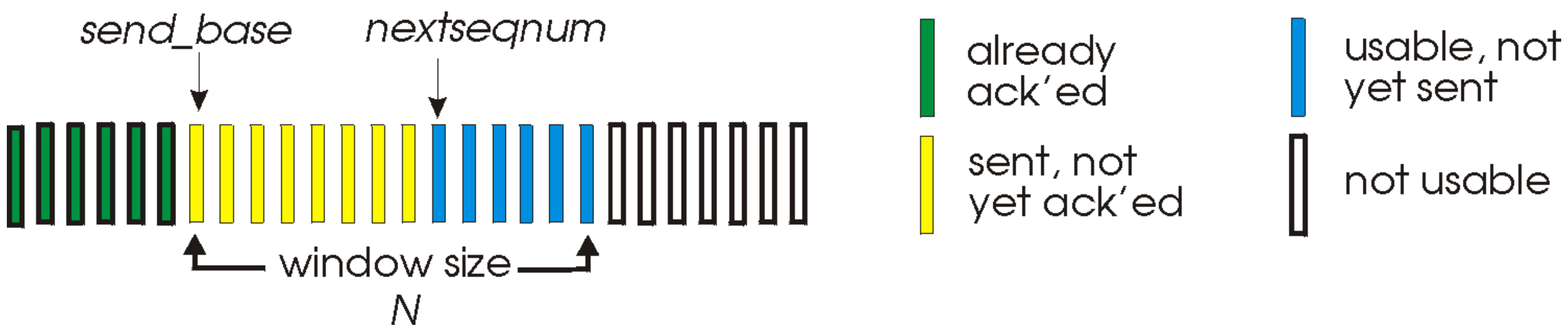
Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

Sender:

- k-bit sequence number in packet header
- "window" of up to N, consecutive unack'ed packets allowed

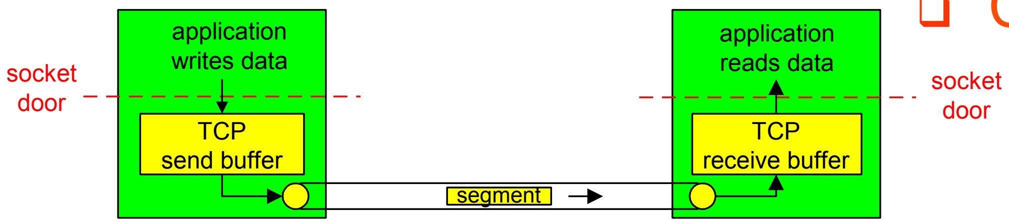


- ACK(n): acknowledges all packets up to and including packet seq# n – "cumulative ACK"
  - May receive duplicate ACKs (see receiver)
- Timer for each in-flight packet
- *Timeout(n):* retransmit pkt n and all higher seq # pkts in window

TCP: Overview

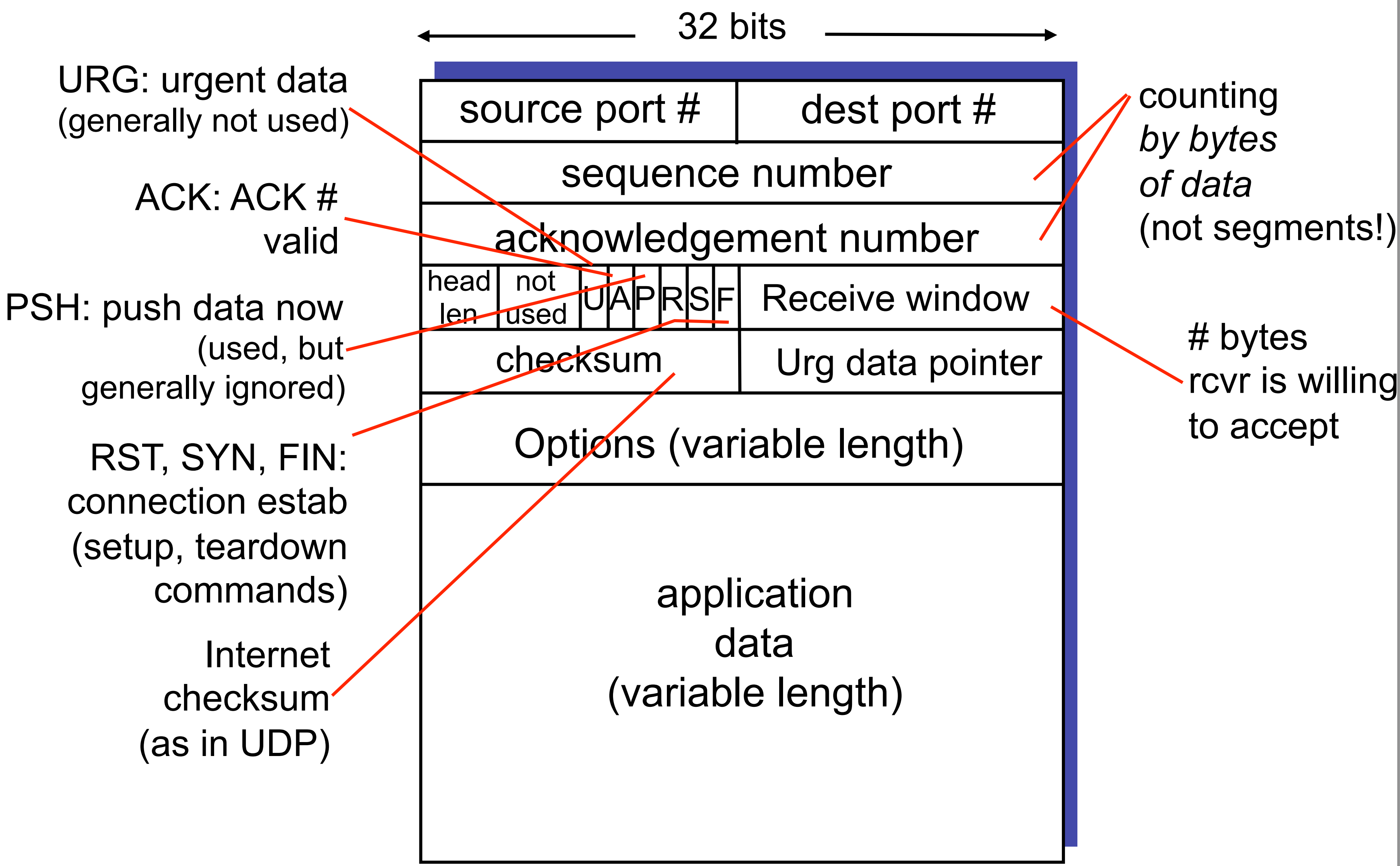
RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
  - one sender, one receiver
- Reliable, in-order *byte steam:*
  - no “message boundaries”
- Pipelined:
  - TCP congestion and flow control set window size
- *Send & receive buffers*



- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-oriented:
  - Handshaking (exchange of control msgs) initialises sender & receiver state before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver
- Congestion controlled:
  - Sender will not overwhelm network

TCP segment structure



TCP sequence numbers and ACKs

Sequence numbers:

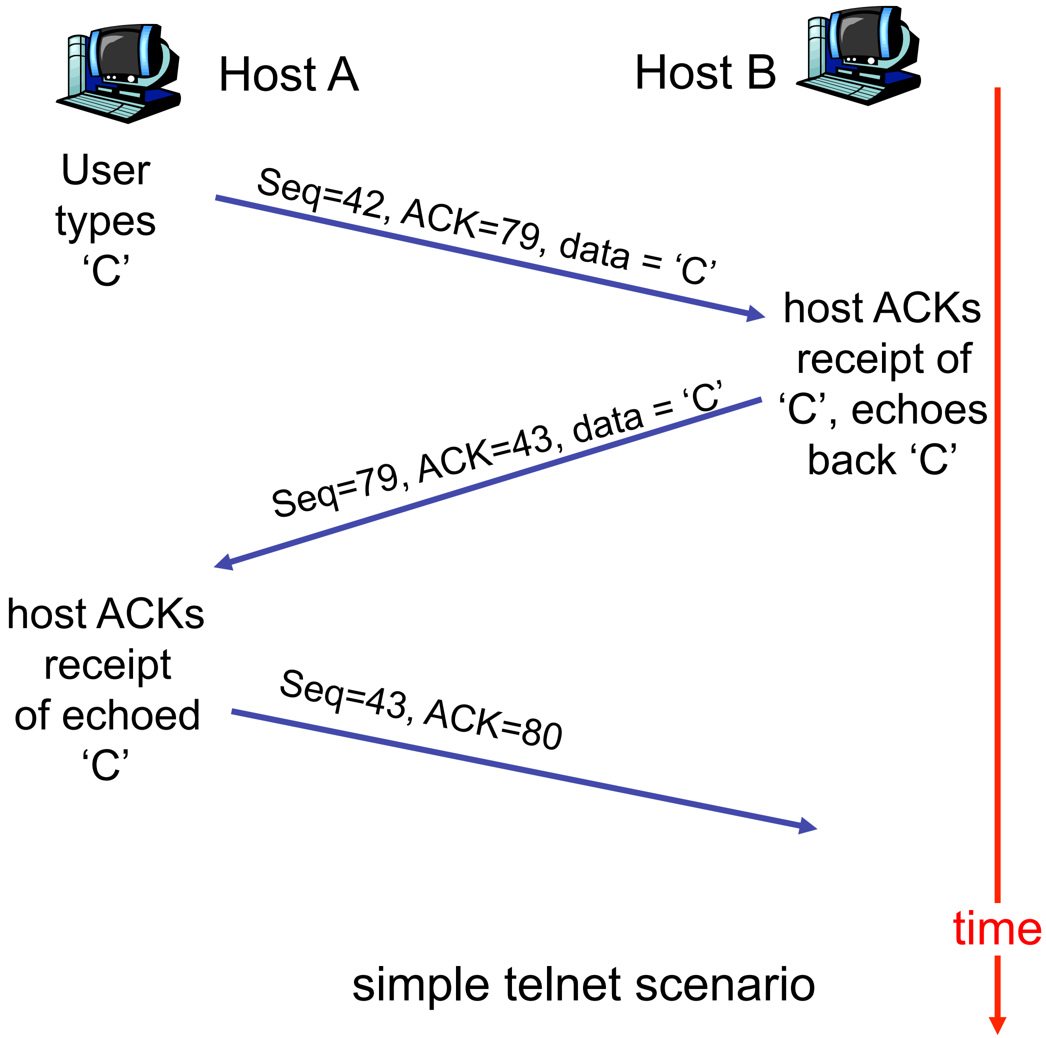
- Byte stream "number" of first byte in segment's data
- Start value not 0, but chosen arbitrarily

ACKs:

- Seq # of next byte expected from other side
- Cumulative ACK

Q: How should receiver handle out-of-order segments?

- TCP spec doesn't say → up to implementor



simple telnet scenario

TCP Round Trip Time (RTT) and Timeout

Q: How to set TCP timeout value for detecting lost packets?

- Obviously: Longer than RTT
  - but RTT varies
- Too short:
  - premature timeout
  - unnecessary retransmissions
- Too long:
  - slow reaction to segment loss

Q: How to estimate RTT?

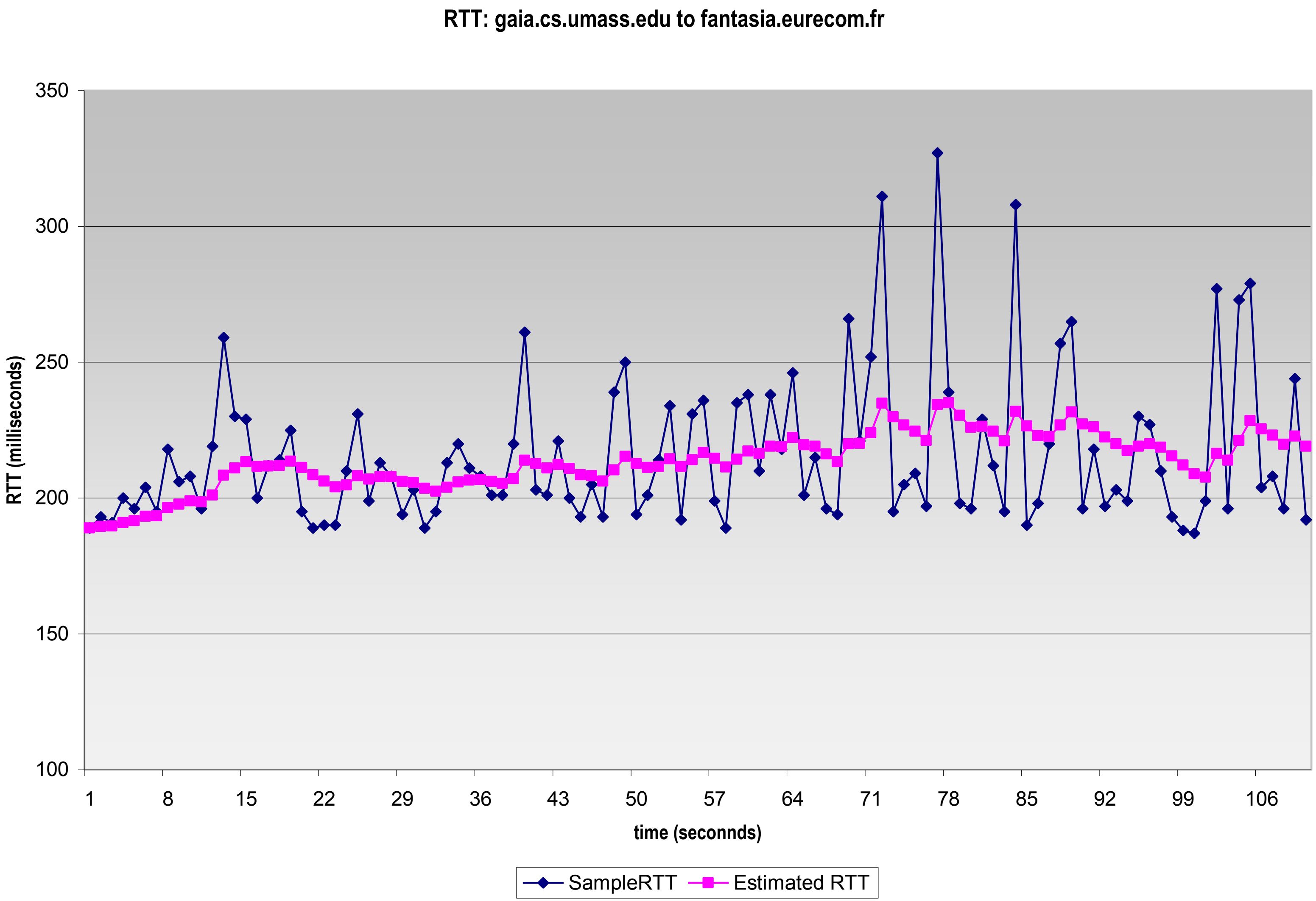
- `SampleRTT`: measured time from segment transmission until ACK receipt
  - Ignore retransmissions (why?)
- `SampleRTT` will vary, want estimated RTT "smoother"
  - Average several recent measurements, not just current `SampleRTT`
  - Exponential moving average (EMA)

TCP Round Trip Time and Timeout

$$\texttt{EstimatedRTT} = (1 - \alpha) * \texttt{EstimatedRTT} + \alpha * \texttt{SampleRTT}$$

- Exponential weighted moving average (EMA)
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimtedRTT`** plus "safety margin"
  - Small variation in **`EstimatedRTT`** → smaller safety margin
  - Large variation in **`EstimatedRTT`** → larger safety margin
- First estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β) * DevRTT +
             β * |SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP reliable data transfer

- TCP creates reliable data transfer service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate acks
- Initially, let's consider simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control

TCP sender events:

Data received from application:

- Create segment with seq #
- Seq # is byte-stream number of first data byte in segment
- Start timer if not already running (think of timer as for oldest unacked segment)
- Expiration interval: `TimeOutInterval`

When timeout occurs:

- Retransmit segment that caused timeout
- Restart timer

When ack received:

- *If* it acknowledges previously un-acked segments
  - Update what is known to be acked
  - Stop timer for this data
  - (Re)start timer if there are other outstanding segments

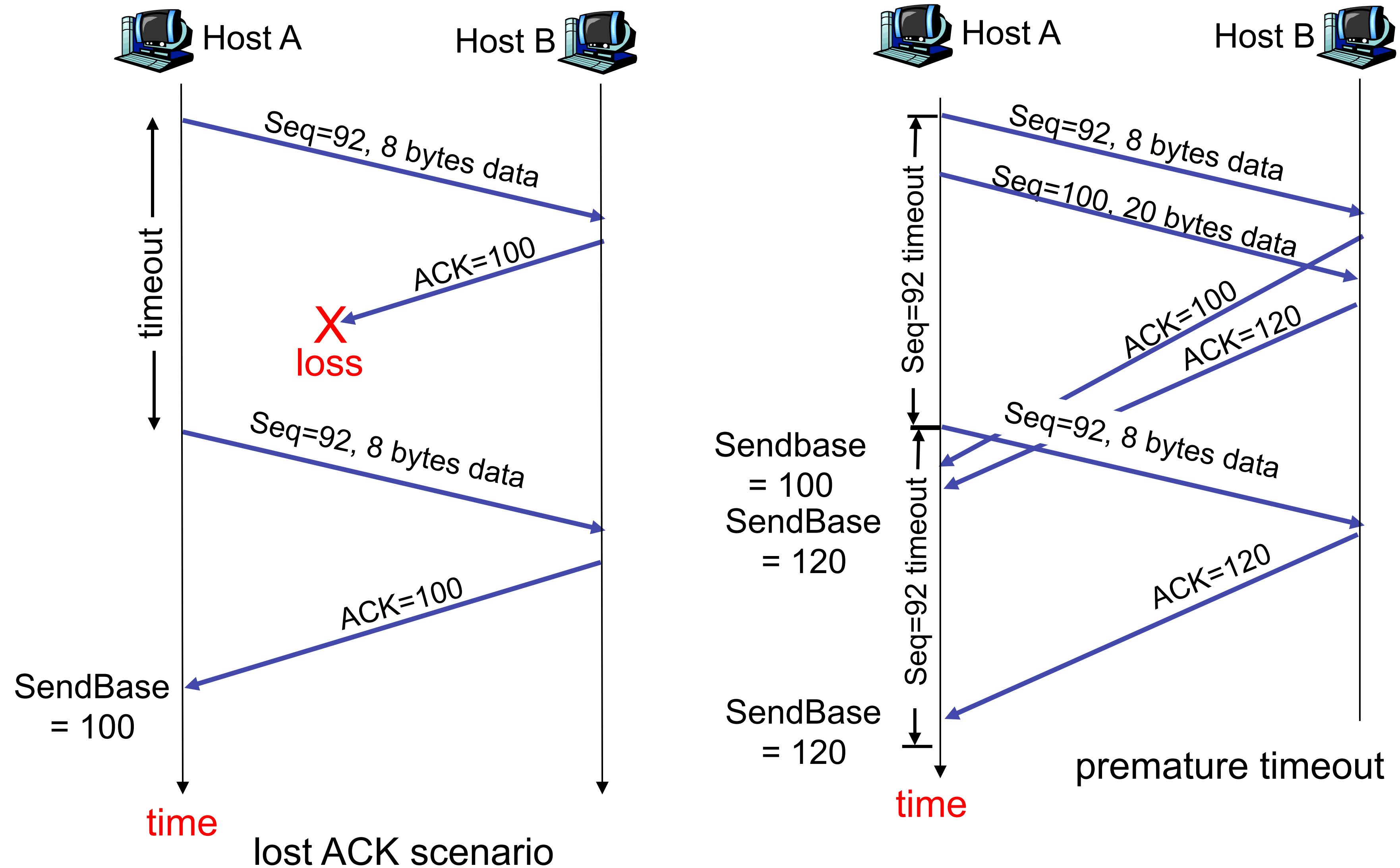
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
loop (forever) {
   switch(event)

   event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
             start timer
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
        retransmit not-yet-acknowledged segment with
               smallest sequence number
        start timer

    event: ACK received, with ACK field value of y
        if (y > SendBase) {
             SendBase = y
            if (there are currently not-yet-acknowledged segments)
                   start timer  }
  }  /* end of loop forever */
```

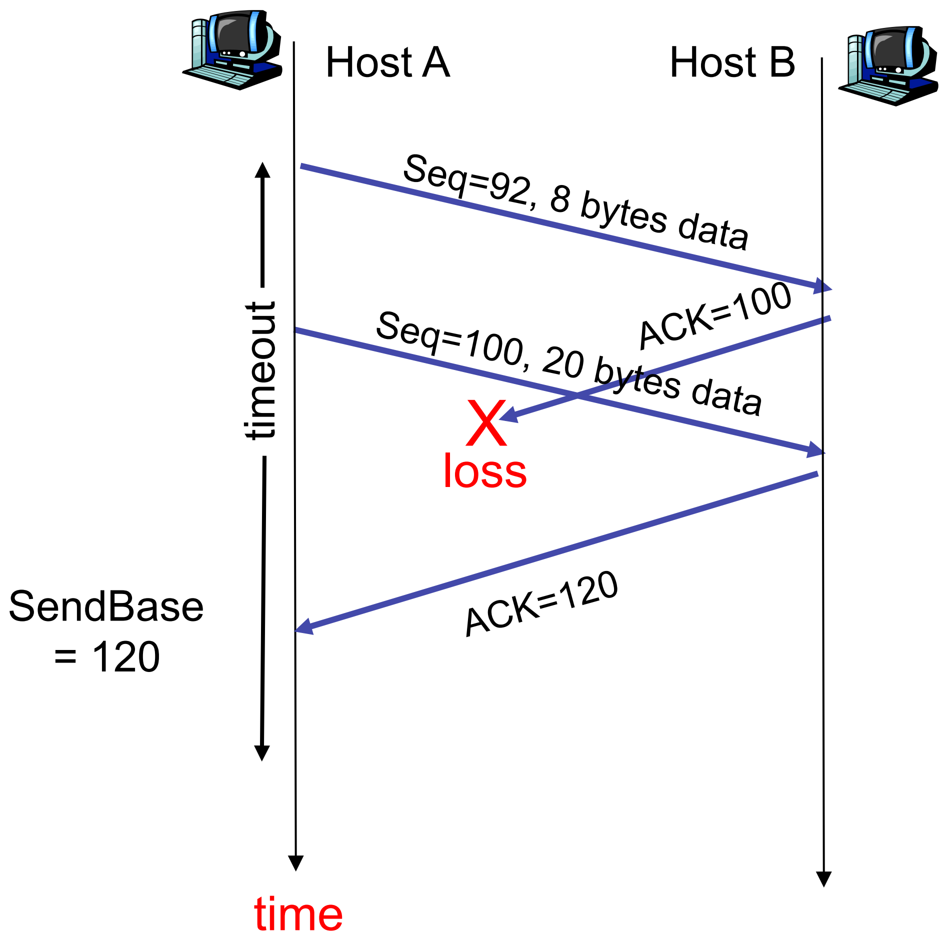
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: Retransmission scenarios



TCP retransmission scenarios (more)



Cumulative ACK scenario

Retransmit of Seq# 92?
Or no retransmit?

No retransmit: We have cumulative ACKs!

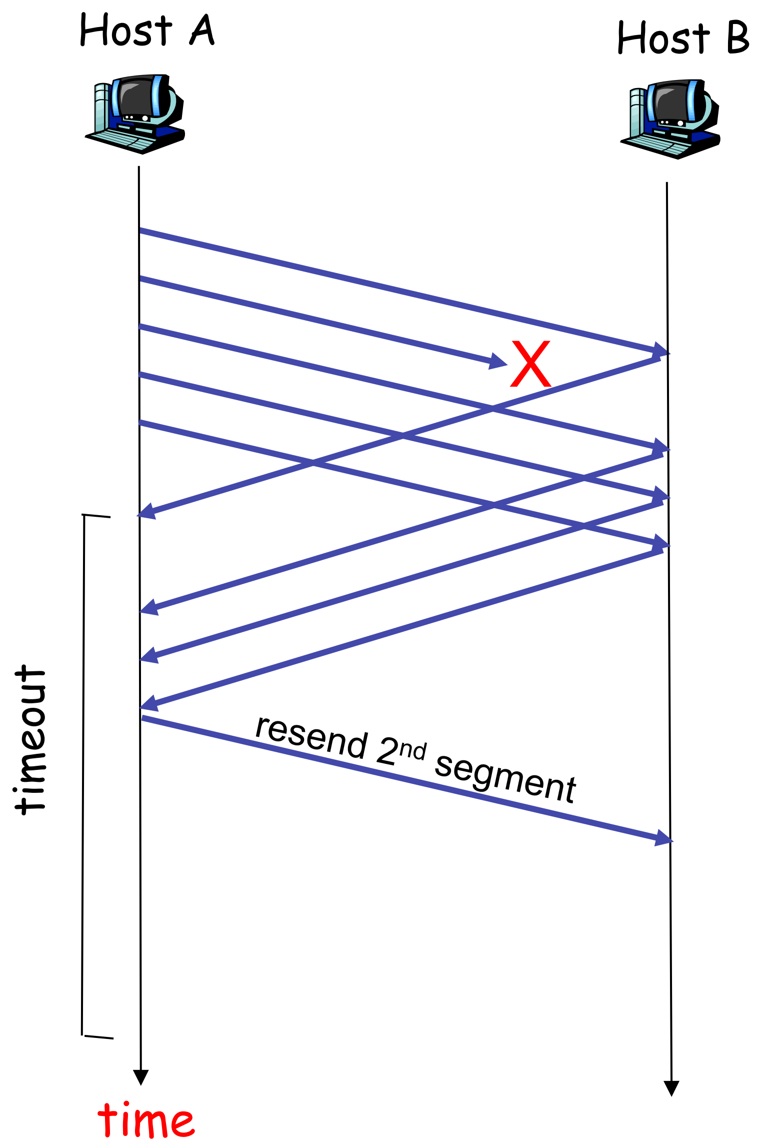
TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

A small TCP optimisation: Fast Retransmit

- Time-out period often relatively long:
  - Long delay before resending lost packet
- Can detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - Fast retransmit:
    - Resend segment before timer expires
    - Assume that only one segment was lost

Resending a segment after triple duplicate ACK



Fast retransmit algorithm:

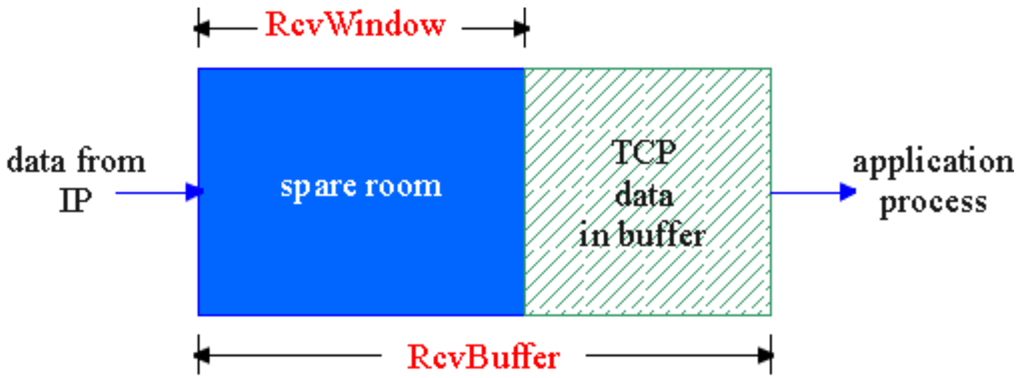
```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
             }
          else {
               increment count of dup ACKs received for y
   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

a duplicate ACK for
already ACKed segment

fast retransmit

TCP Flow Control

- Receive side of TCP connection has a receive buffer:

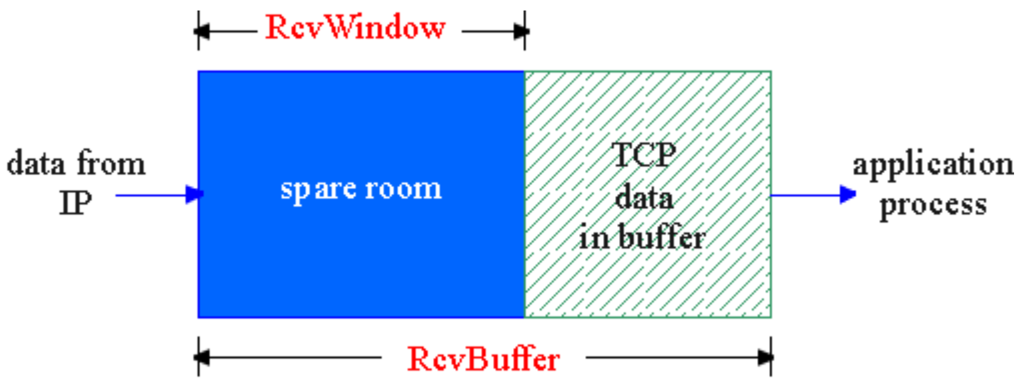


flow control

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

- Application process may be slow at reading from buffer (e.g., mobile phone)
- Speed-matching service: matching the send rate to the receiving application's drain rate

TCP Flow control: How it works



(Suppose TCP receiver discards out-of-order segments)

- Spare room in buffer

= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- Receiver advertises spare room by including value of `RcvWindow` in segments
- Sender limits unACKed data to `RcvWindow`
  - guarantees receive buffer doesn't overflow

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- Initialize TCP variables:
  - Sequence numbers
  - Buffers, flow control info (e.g. **`RcvWindow`**)
- *Client:* connection initiator

```
Socket clientSocket = new
 Socket("hostname","port number");
```

- *Server:* contacted by client

```
Socket connectionSocket =
 welcomeSocket.accept();
```

Note: Cannot distinguish client and server after connection establishment

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- i.e., SYN bit is set
- Specifies initial seq #
- No data

Step 2: server host receives SYN, replies with SYNACK segment

- i.e., SYN and ACK bits set
- Server allocates buffers
- Specifies server initial seq.#

Step 3: client receives SYNACK, replies with ACK segment, which *may* contain data

TCP Connection Management (cont.)

Closing a connection:

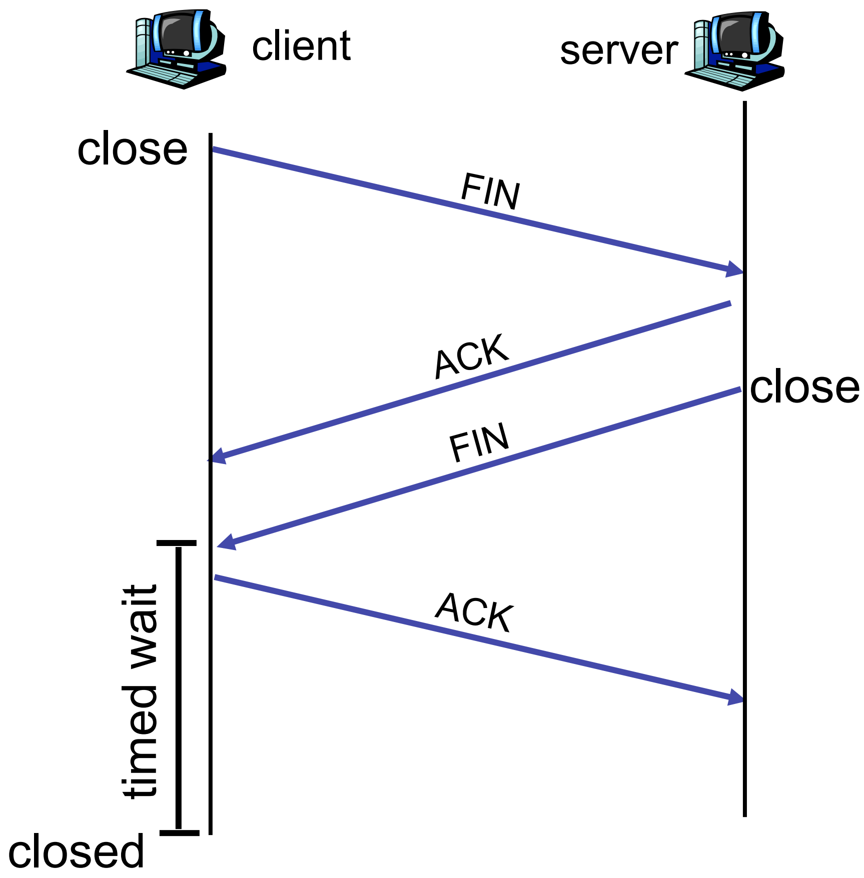
"Client" closes socket:
clientSocket.close();

Step 1: Client end system sends TCP FIN control segment to server

- Promise: "I won't transmit any further data to you": Half-closed connection

Step 2: Server receives FIN, replies with ACK. Informs application. Application closes connection, TCP sends FIN.

Note: Server can continue sending data between step 1 and Step 2!



TCP Connection Management (cont.)

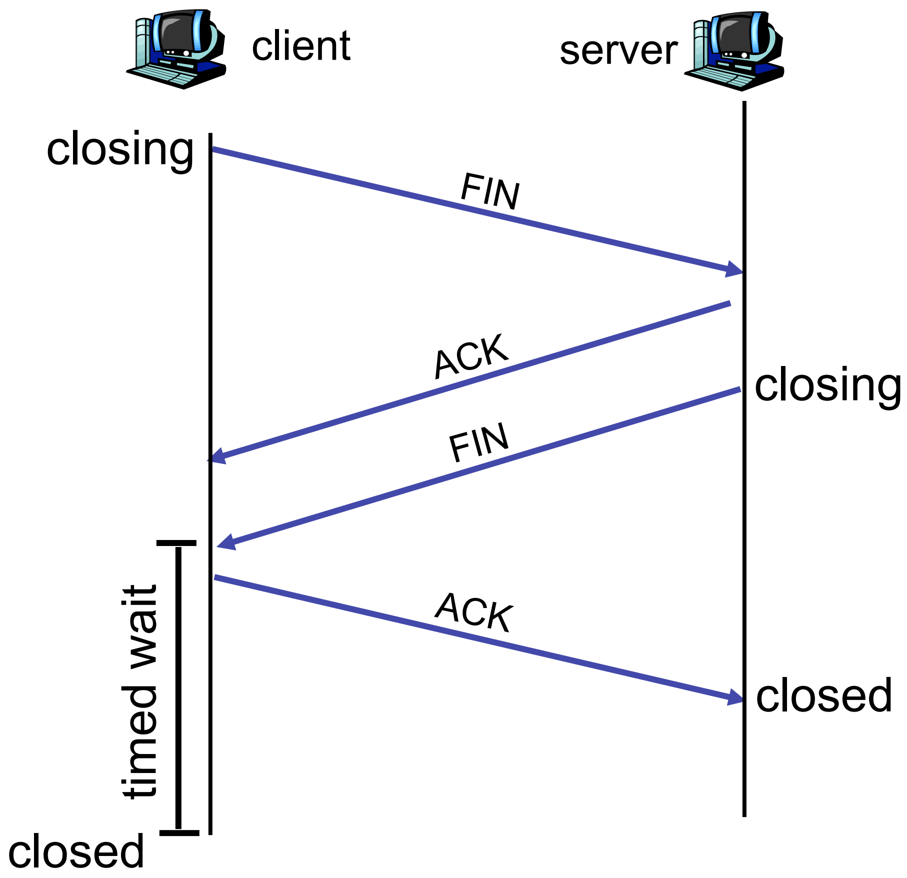
Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" – will respond with ACK to received FINs

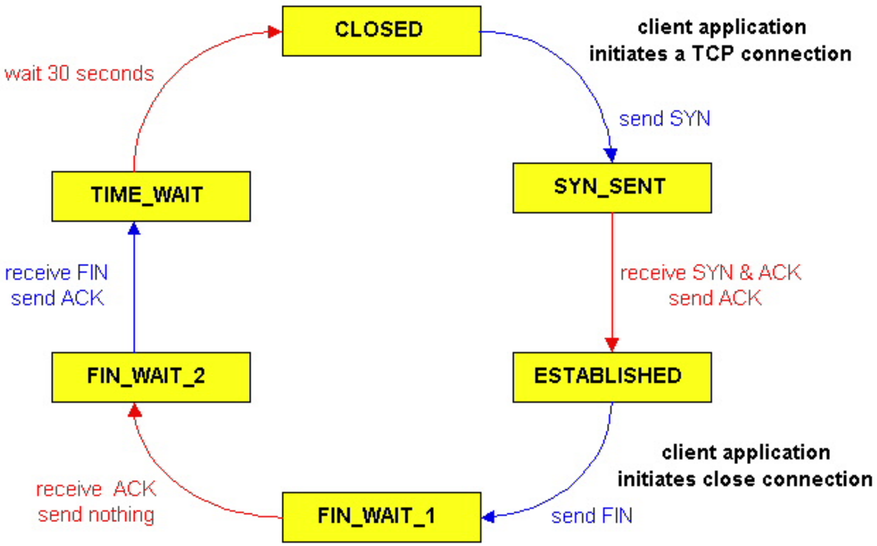
Step 4: server, receives ACK. Connection closed.

Notes:

- With small modification, can handle simultaneous FINs
- Any partner in connection can send the first FIN

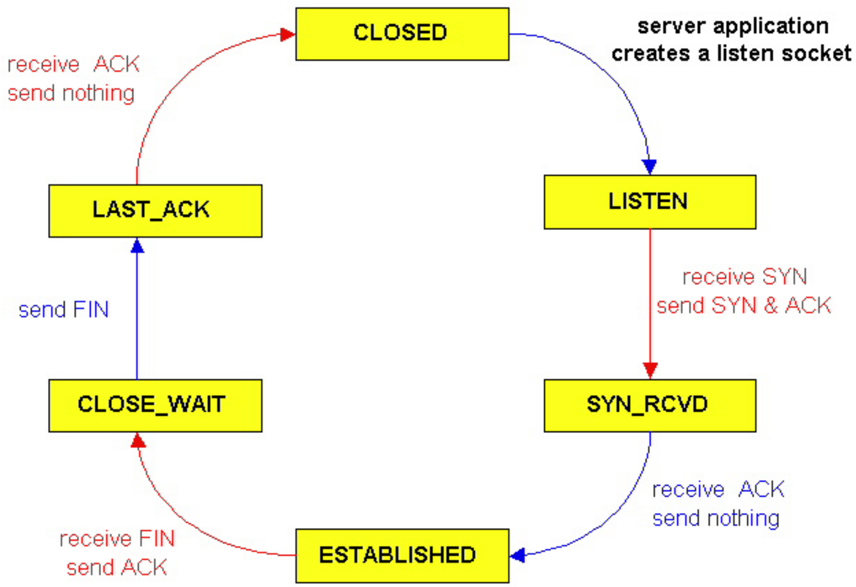


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



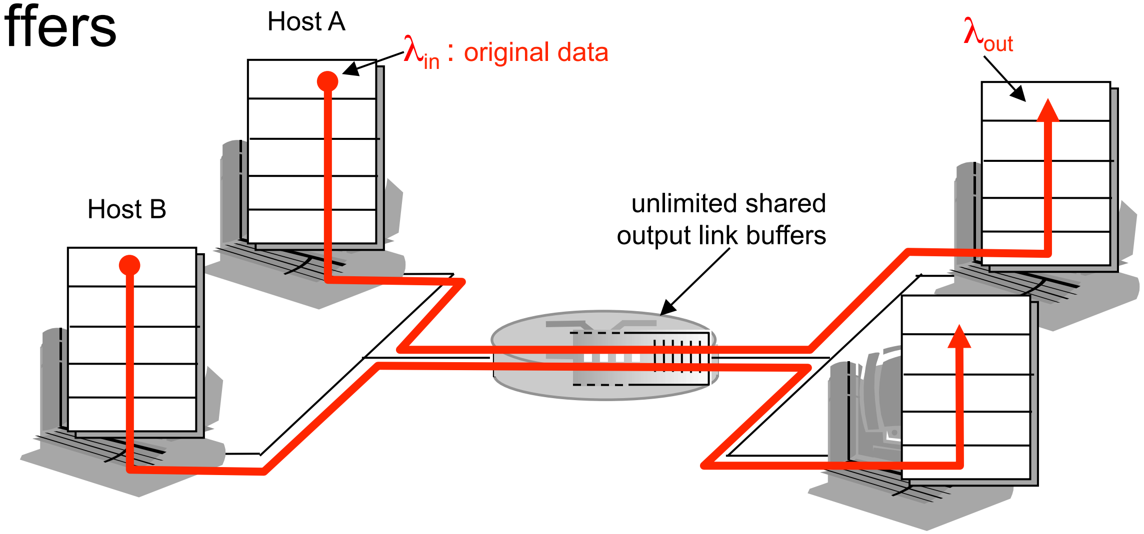
Principles of Congestion Control

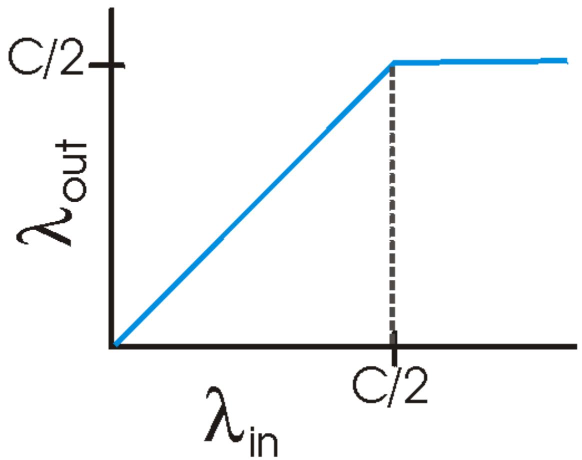
Congestion:

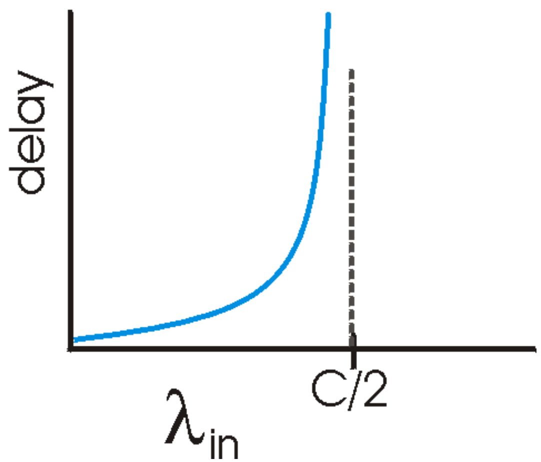
- Informally: "Too many sources sending too much data too fast for the *network* to handle"
- What's the difference to flow control?
  - Flow control: "One source sending too much data too fast for the *other application* to handle"
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)
- A top-10 problem!

Causes/costs of congestion: scenario 1

- Two senders, two receivers
- One router, infinite buffers
- No retransmission



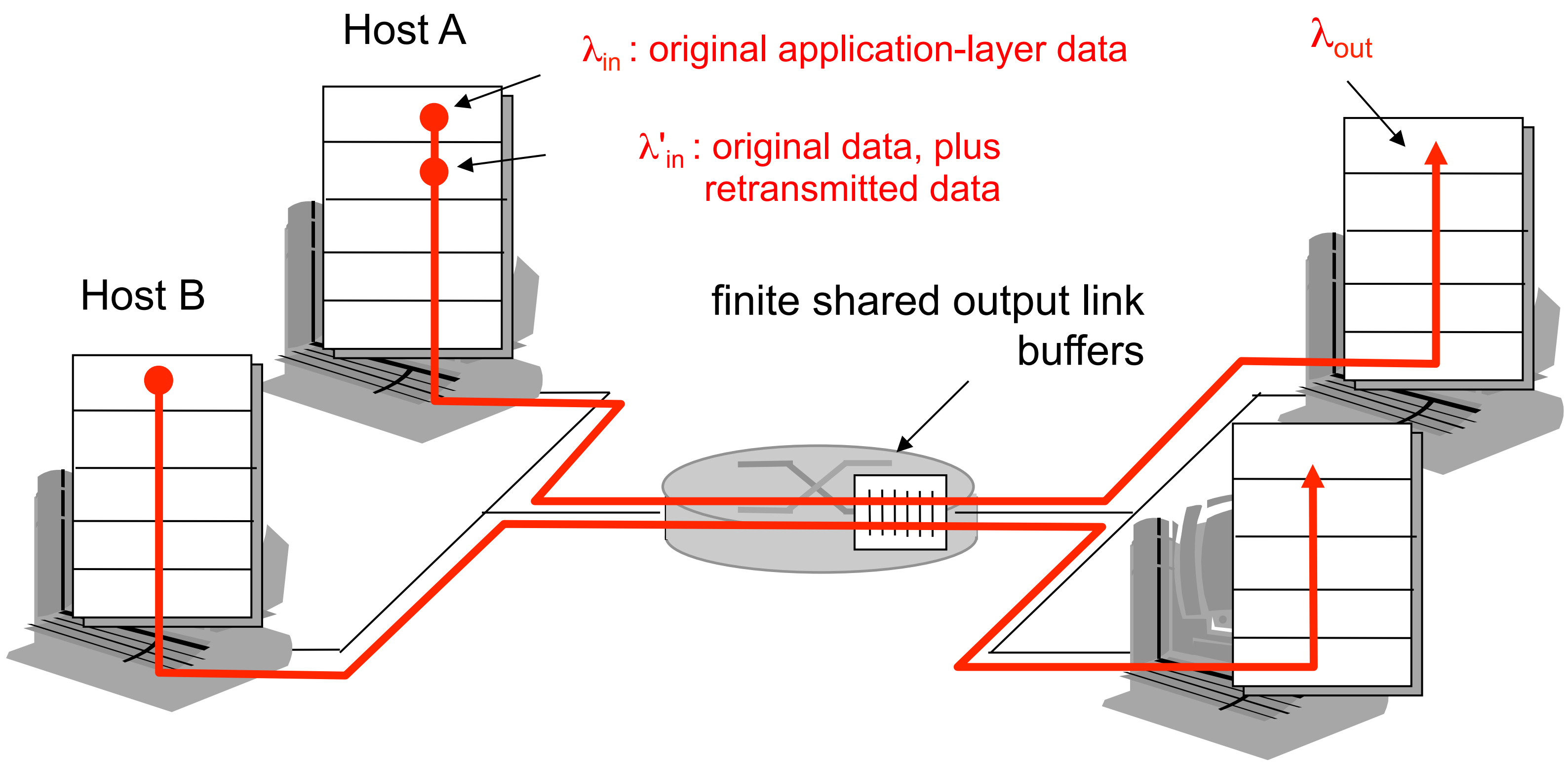




- Large delays when congested
- Maximum achievable throughput

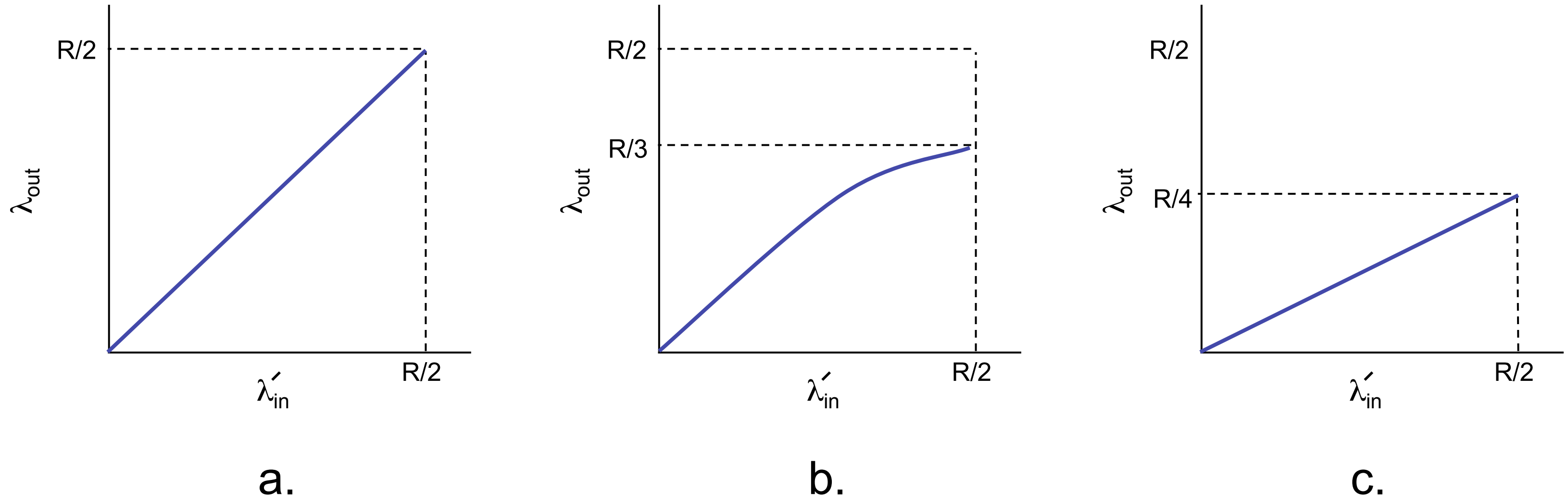
Causes/costs of congestion: scenario 2

- One router, *finite* buffers
- Sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- Always: $\lambda_{in} = \lambda_{out}$ for application-layer data (called "goodput")
- "Perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- Retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



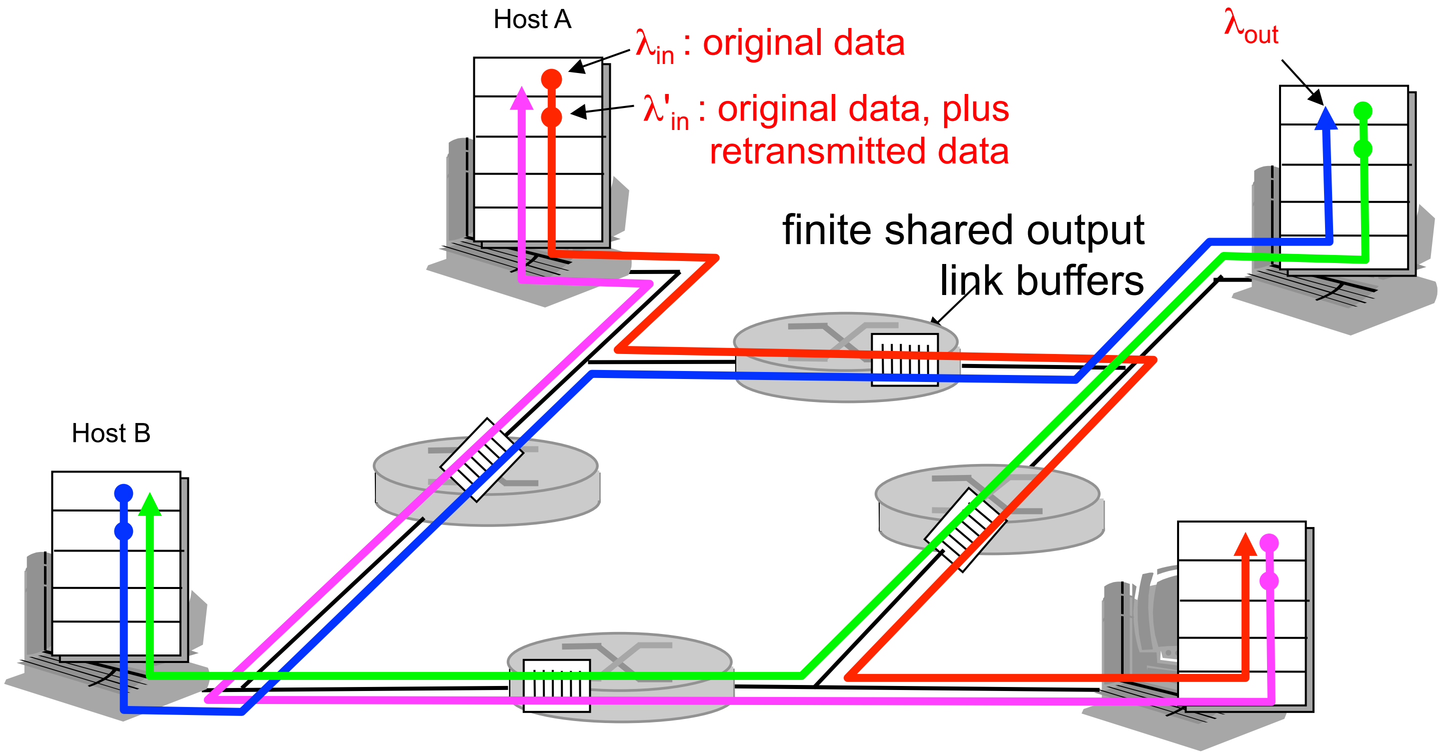
"Costs" of congestion:

- More work (retransmissions) for given "goodput"
- Unnecessary retransmissions: Link carries multiple copies of same packet

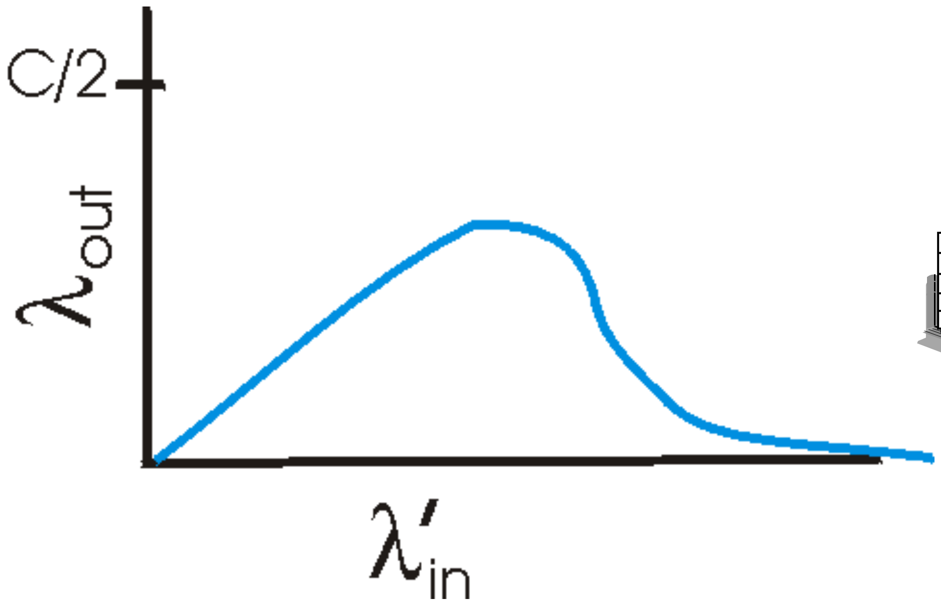
Causes/costs of congestion: scenario 3

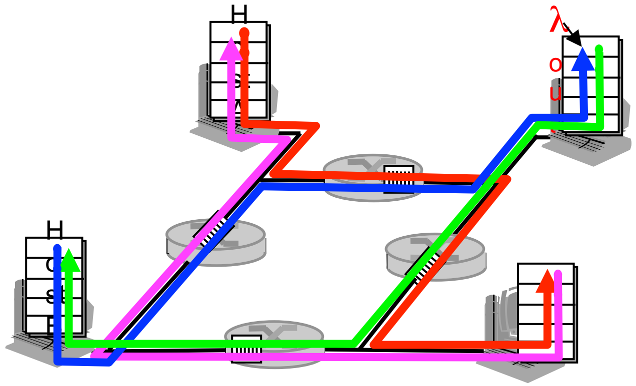
- Four senders
- Multihop paths
- Timeout/retransmit

<u>Q:</u> What happens as λ_{in} and λ'_{in} increase ?



Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- When packet is dropped, any upstream transmission capacity used for that packet was wasted

Approaches towards congestion control

Two broad approaches towards congestion control:

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 bit, ICMP source quench ATM)
  - Explicit rate sender should send at
- TCP/IP has support for ECN, but almost never used
- ICMP source quench: dito

Case study: ATM ABR congestion control

ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

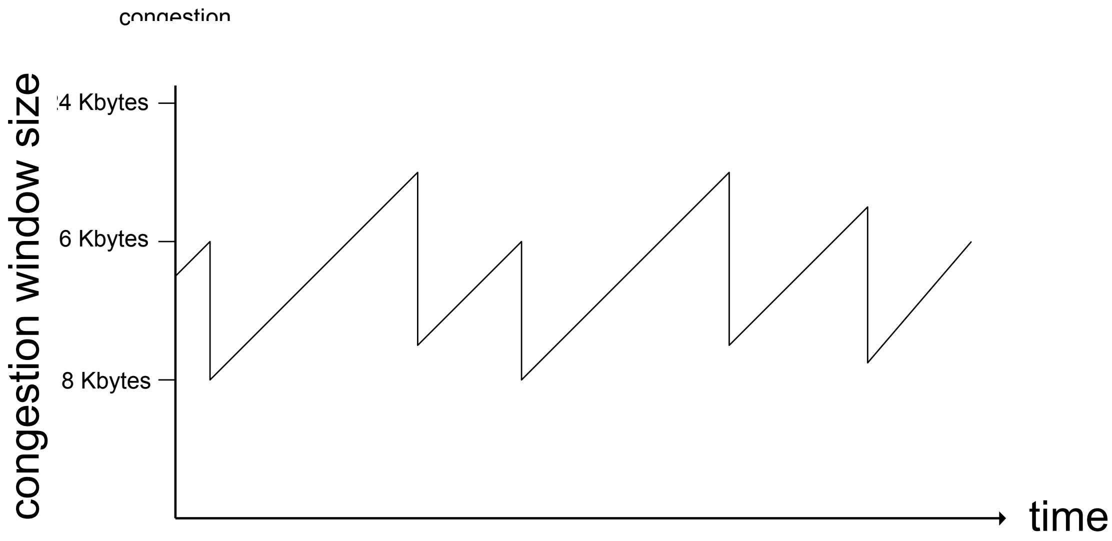
RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

TCP congestion control: Additive increase, Multiplicative decrease (AIMD)

- *Approach:* Increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *Additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - *Multiplicative decrease*: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



TCP Congestion Control: details

- Sender limits transmission:

  LastByteSent – LastByteAcked

  ≤ CongWin

- Roughly,

$$\boxed{\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}}$$

- **CongWin** is dynamic: Function of perceived network congestion

How does sender perceive congestion?

- Loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**CongWin**) after loss event

Three mechanisms:

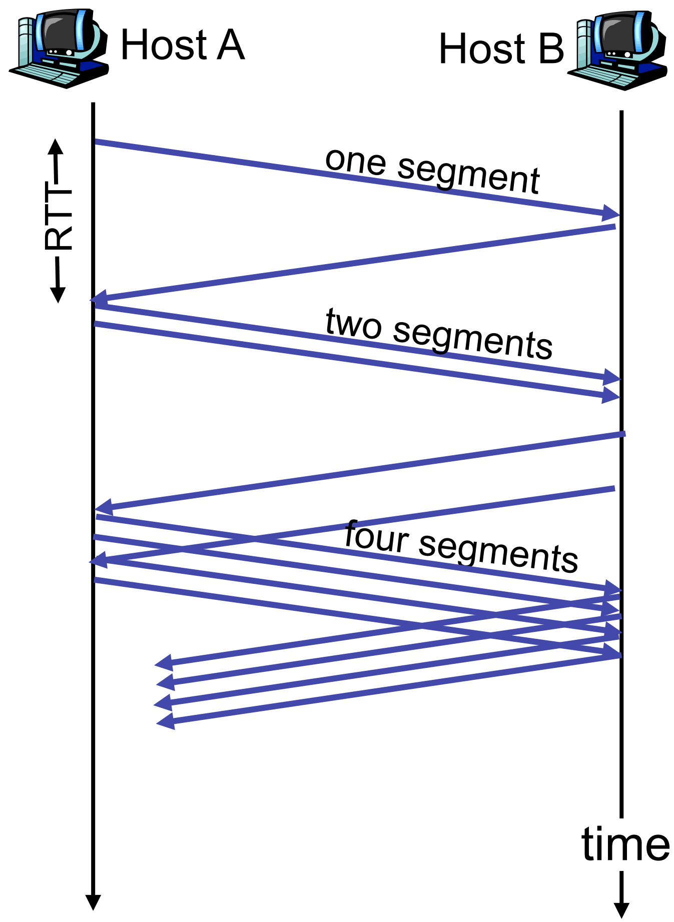
- AIMD
- Slow start
- conservative after timeout events

TCP Slow Start

- When connection begins, **`CongWin`** = 1 MSS
  - Example: MSS = 500 bytes; RTT = 200 msec
  - Initial rate = 20 kbps
- But: Available bandwidth may be >> MSS/RTT
  - Desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - Double **`CongWin`** every RTT
  - Done by incrementing **`CongWin`** for every ACK received
  - N.B.: Exponential growth caused by additions, not multiplications or exponentiations!
- Summary: Initial rate is slow but ramps up exponentially fast



Refinement: Inferring loss

- After 3 duplicate ACKs:
  - **`CongWin`** is cut in half
  - Window then grows linearly
- <u>But:</u> after timeout event:
  - **`CongWin`** instead set to 1 MSS;
  - Window then grows exponentially
  - to a *threshold*, then grows linearly

Philosophy:

Why this distincion?

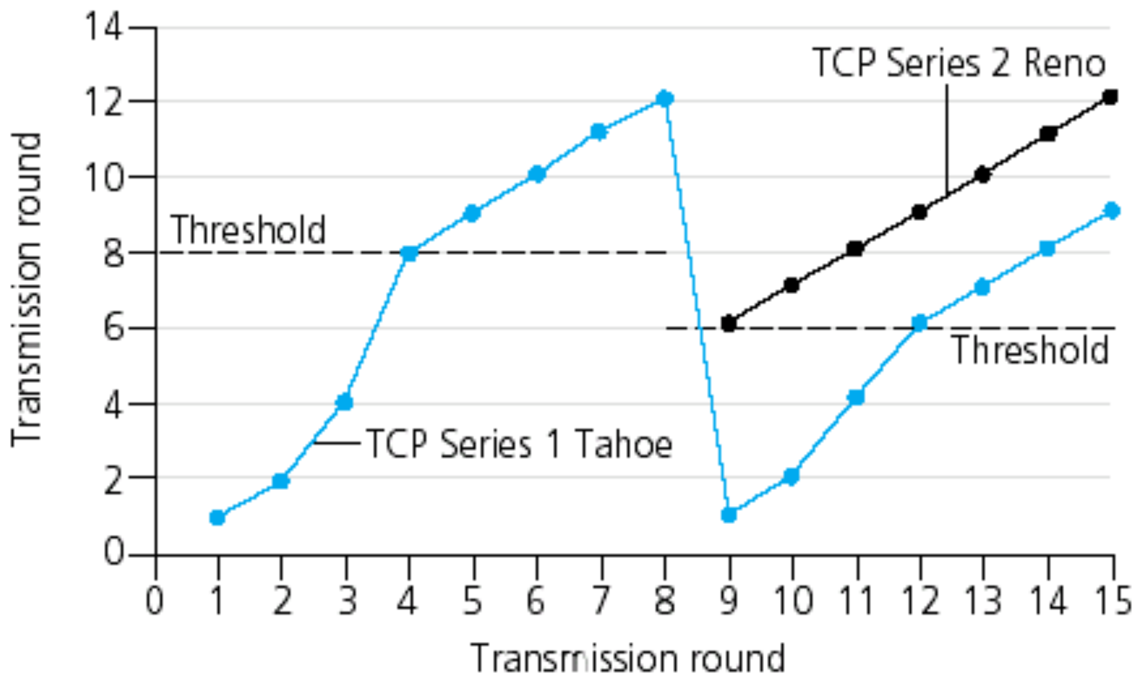
- 3 duplicate ACKs indicates: Network still capable of delivering some (actually, most) segments
- Timeout indicates a more alarming congestion scenario: (Almost) no segments got through!

Refinement

- Q: When should the exponential increase switch to linear?
- A: When CongWin gets to 1/2 of its value before timeout.

<u>Implementation:</u>

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event



Summary: TCP Congestion Control

- When `CongWin` is below `Threshold`, sender in slow-start phase, window grows exponentially.
- When `CongWin` is above `Threshold`, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, `Threshold` set to `CongWin/2` and `CongWin` set to `Threshold`.
- When timeout occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS.

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP summary

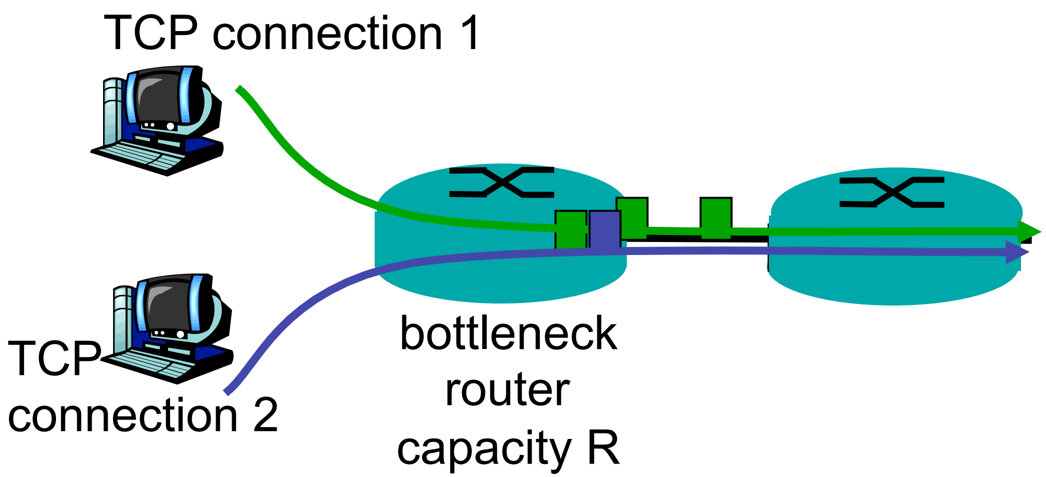
- Connection-oriented: SYN, SYNACK; FIN
- Retransmit lost packets; in-order data: sequence no., ACK no.
- ACKs: either piggybacked, or no-data pure ACK packets if no data travelling in other direction
- Don’t overload receiver: rwin
  - rwin advertised by receiver
- Don’t overload network: cwin
  - cwin affected by receiving ACKs
- Sender buffer = min { rwin, cwin }
- Congestion control:
  - Slow start: exponential growth of cwin
  - Congestion avoidance: linear groth of cwin
  - Timeout; duplicate ACK: shrink cwin
- Continuously adjust RTT estimation

TCP throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: 0.75 W/RTT

TCP Fairness

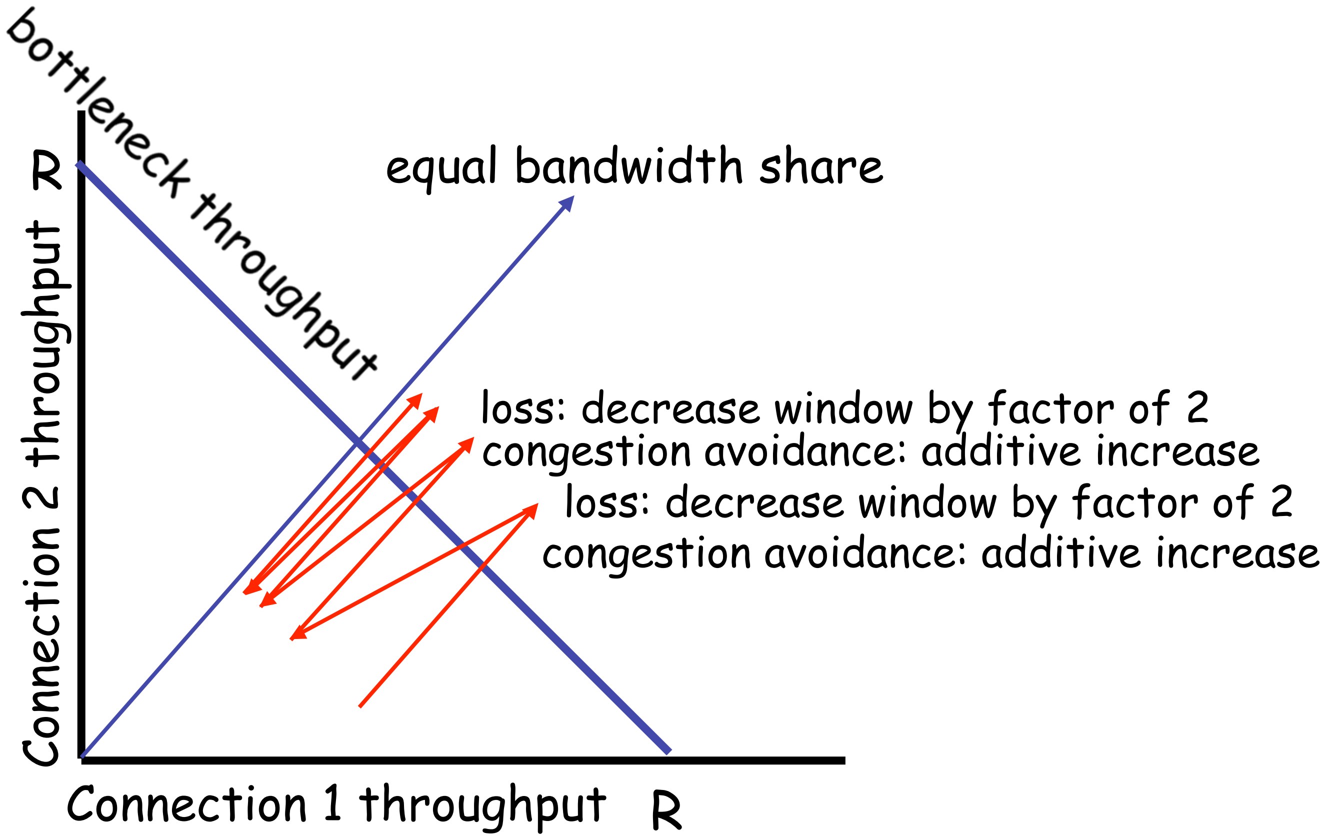
Fairness goal: If K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - Do not want rate throttled by congestion control
- Instead use UDP:
  - Pump audio/video at constant rate, tolerate packet loss
- Research area: Make these protocols TCP friendly
- One approach: DCCP (Datagram Congestion Control Protocol)
  - "UDP with congestion control"
  - Not very popular (as yet)

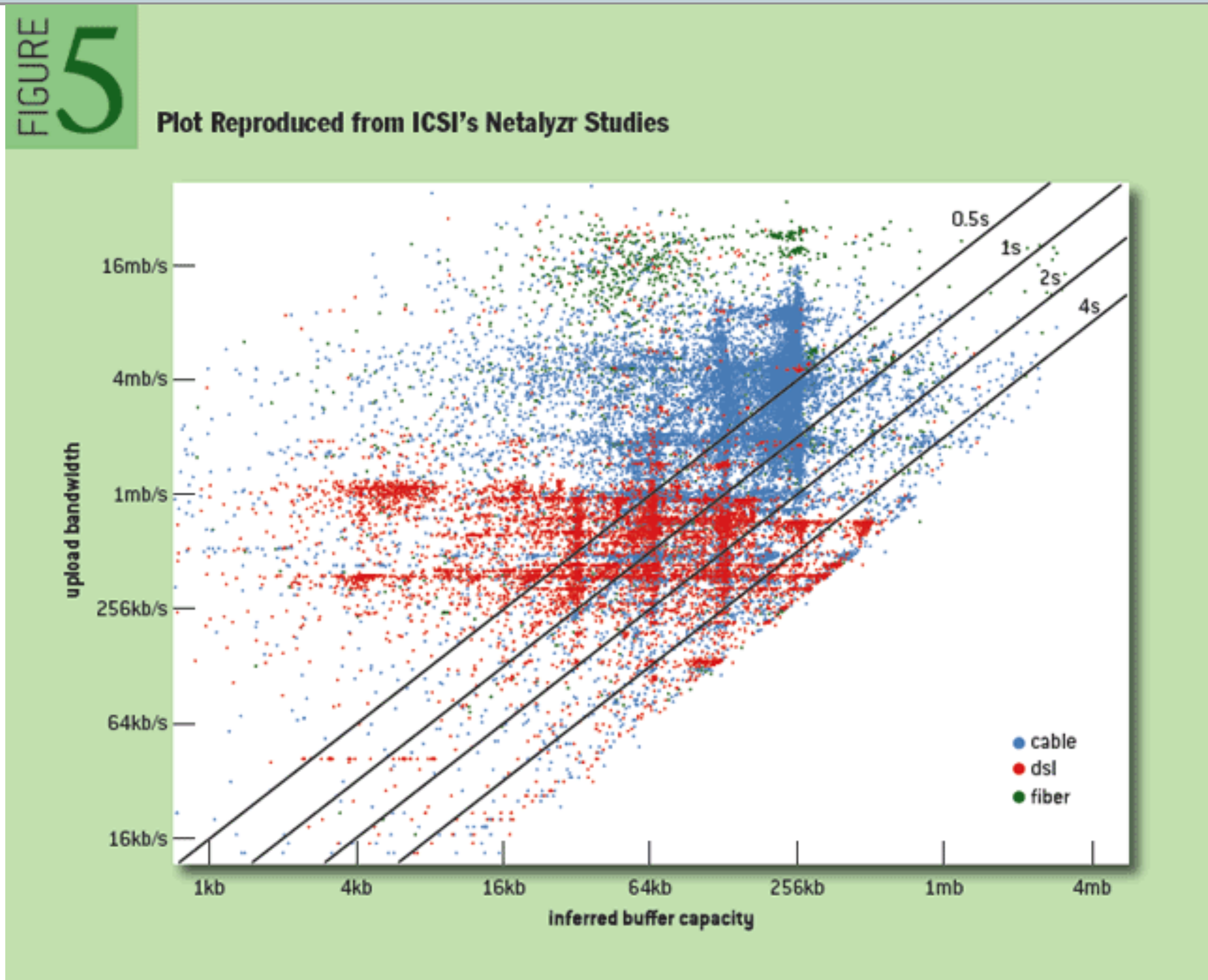
Fairness and parallel TCP connections

- Nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: Bottleneck link of rate R that is already supporting 9 connections
  - New application opens 1 TCP conn → gets rate R/10
  - New application opens 11 TCP conns → gets rate R/2 !

TCP and buffer bloat

- Capacities of router queues
  - "Large queue = good: Less packet losses at bottlenecks"
  - Do you agree? What would happen to TCP?
- Effects of large Buffers at bottleneck on TCP connections
  - Once queues are full: Queueing delays increase dramatically
  - TCP congestion control gets no early warning
    - No duplicate ACKS ➔ no Fast Retransmit
    - Instead: Sudden timeouts
  - Congestion windows way too large
  - Many parallel TCP connections over same link get warning way too late
    - Synchronisation: Oscillation between "All send way too much" and "all get frightened by timeouts and send way too little"
    - Huge variations in queueing delays ➔ DevRTT becomes very large ➔ Timeout value becomes very large

Buffer bloat is a real-world problem



Chapter: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network “edge” (application, transport layers)
- into the network “core”