

Vorlesung Telematik (Computer Networks)

WS2003/04

Transport Layer: Flow/Congestion Control and Others

Telematics group
University of Göttingen, Germany

Overview

- Introduction
- Flow Control
- Congestion Control
 - TCP Congestion Control
- Other functionalities:
 - Reliability
 - Multiplexing (socket)
- Homework
- Self-learning: TCP Delay Analysis

Credits:

➤ James Kurose & Keith Ross: Computer Networking(2nd Ed.), Addison-Wesley, 2002
WS 2003/04, fu@informatik.cs.uni-goettingen.de

Transport layer services

- Network layer: deliver data packets from a host to another host:
 - routing (control plane)
 - forwarding (data plane)
 - Call setup (e.g., ATM & IP QoS)
- Transport layer: provide *logical communications* between *application processes* running in different *hosts* (deliver user data from a process in one host to another process in another host)
 - multiplexing/demultiplexing
 - reliable data transfer
 - flow control
 - congestion control

Household analogy:

12 kids sending letters to 12 kids

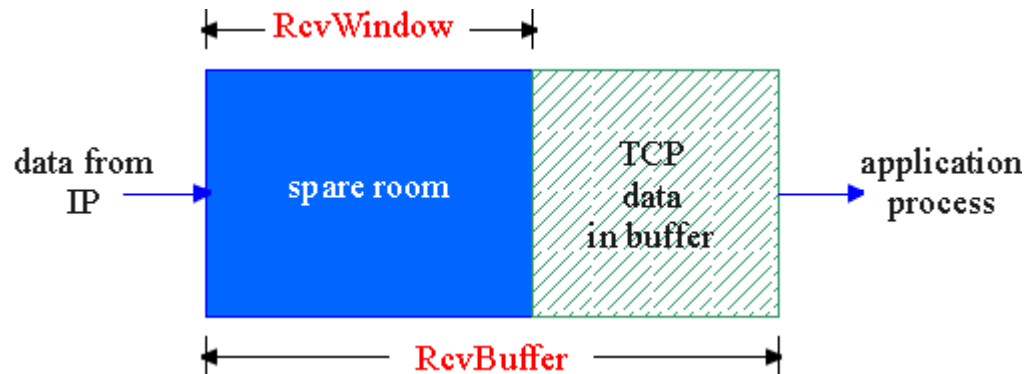
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Flow Control v.s. Congestion Control

- Flow control: end-to-end mechanism for regulating traffic between source and destination
- Congestion control: Mechanism used by the network to limit congestion
- The two are not really separable, but different
- In either case, both amount to mechanisms for limiting the amount of traffic entering the network
 - Sometimes the load is more than the network can handle

Flow Control

- receive side of end-to-end connection has a receive buffer:



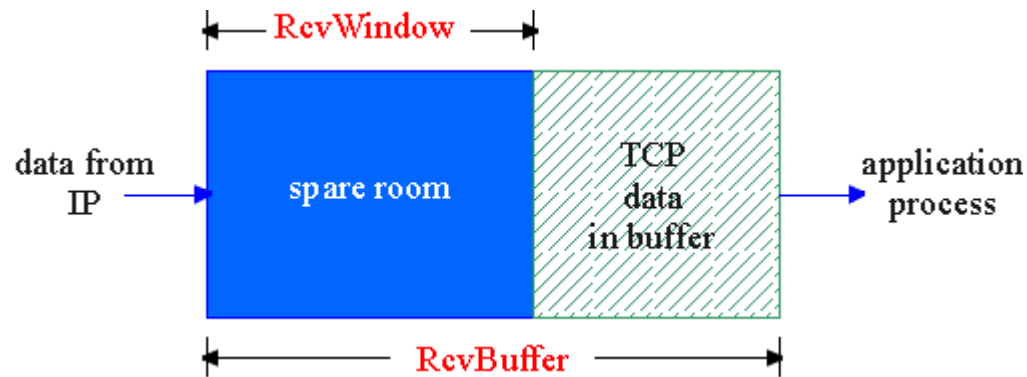
- app process may be slow at reading from buffer

flow control

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

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



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

- spare room in buffer
- = $RcvWindow$
- = $RcvBuffer - [LastByteRcvd - LastByteRead]$

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

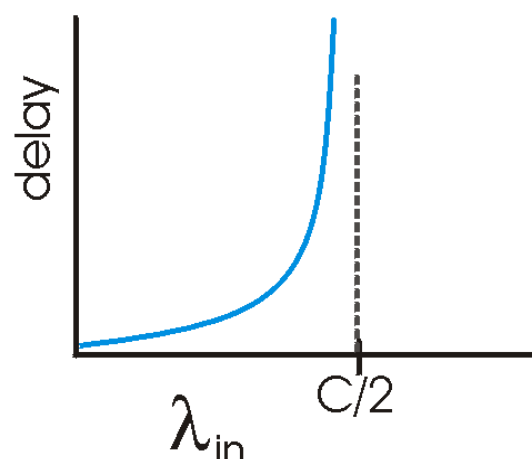
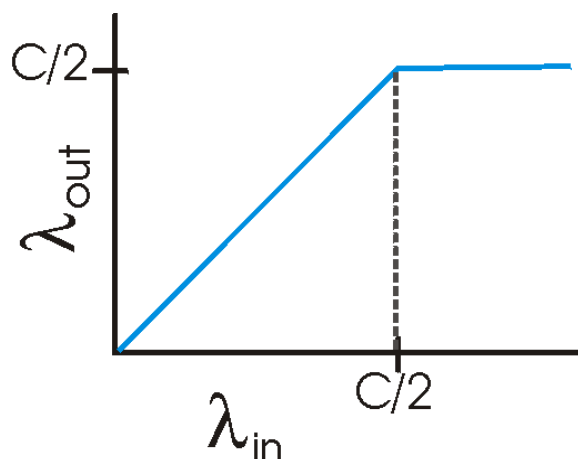
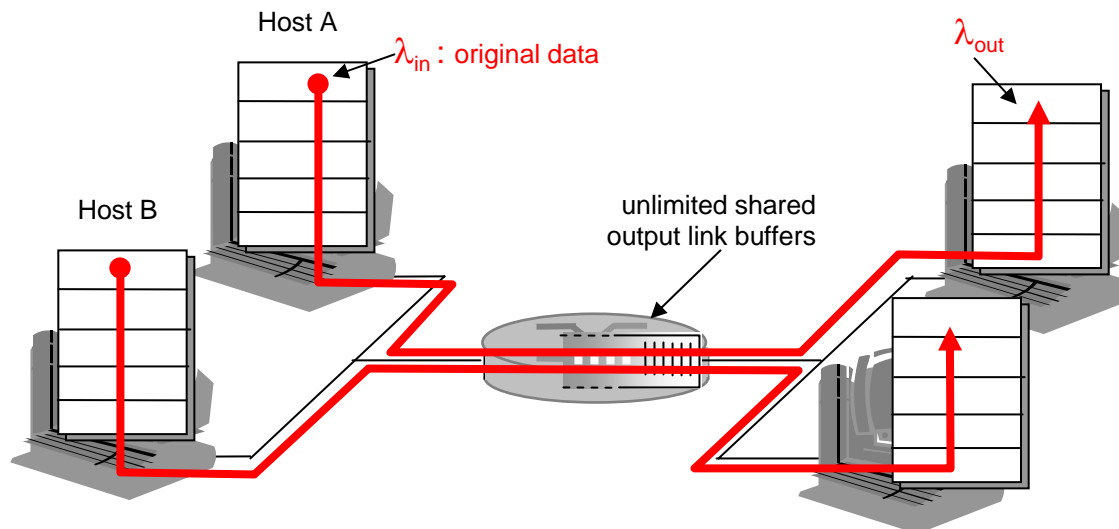
Principles of Congestion Control

Congestion:

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

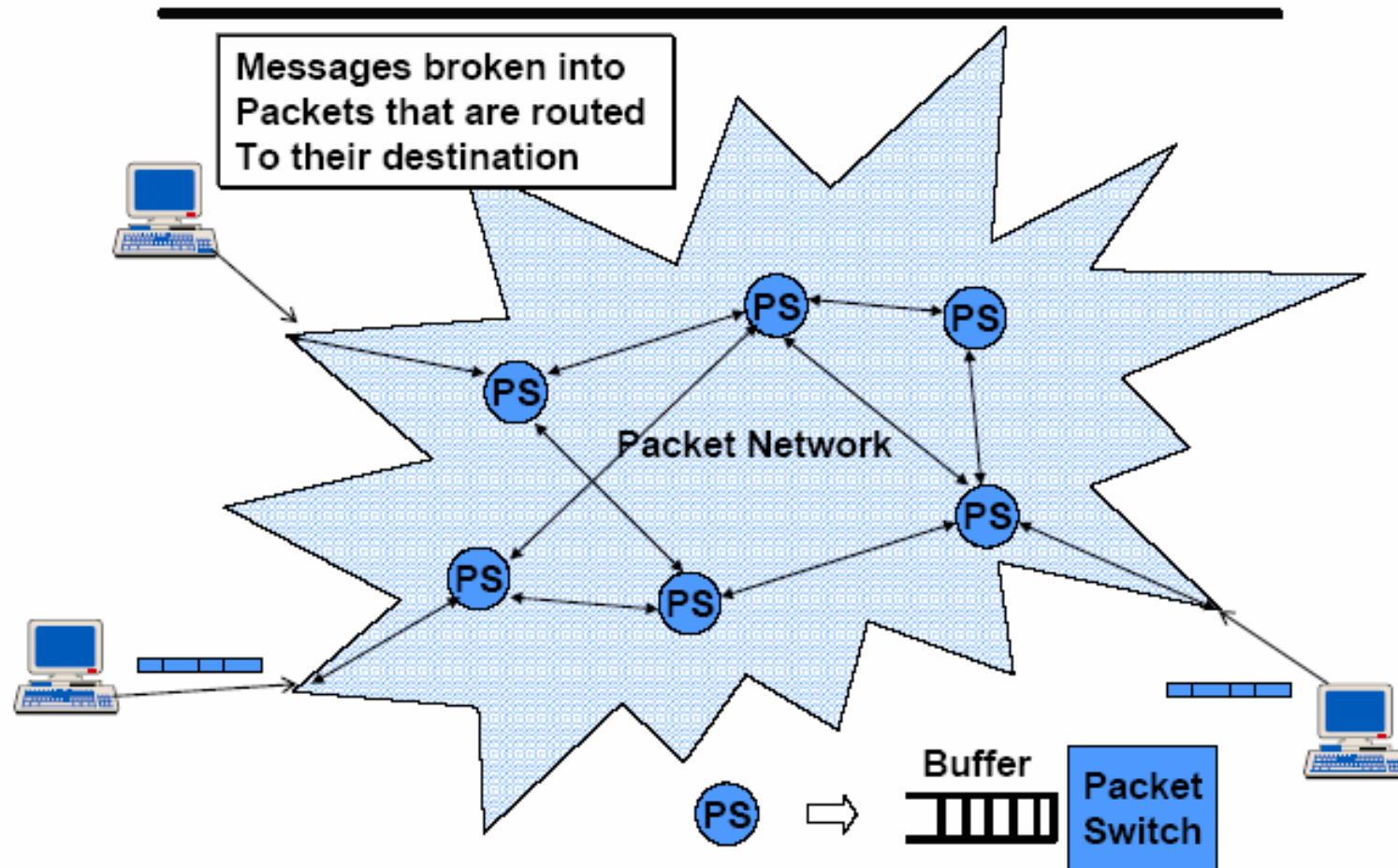
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

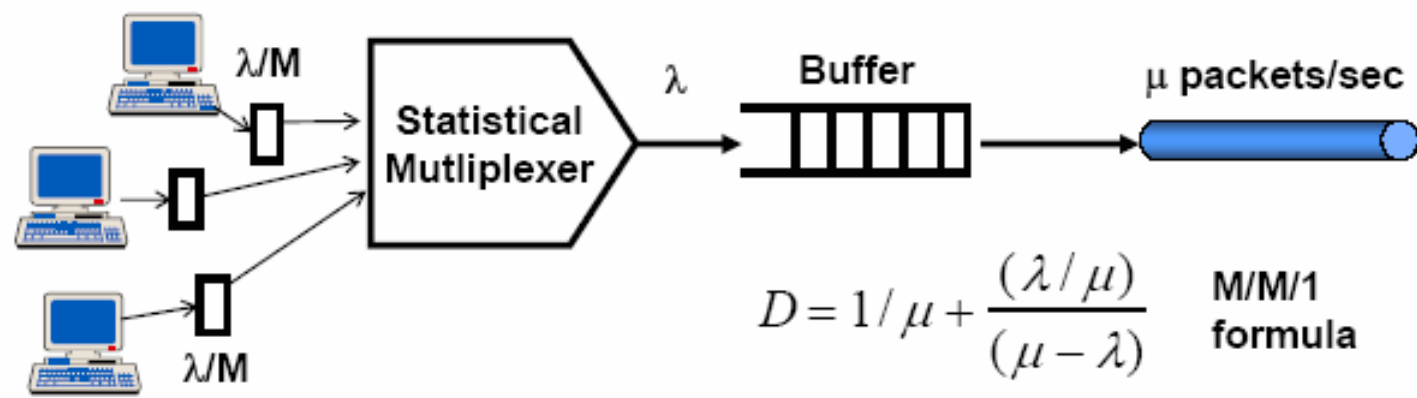
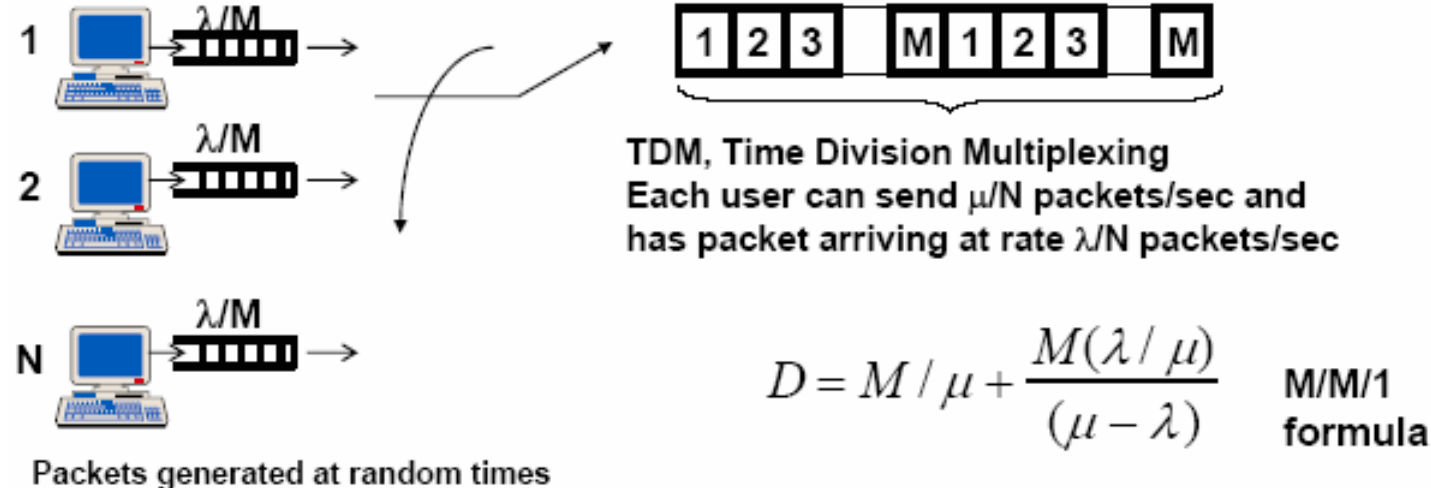


- large delays when congested
- maximum achievable throughput

Packet Switched Networks

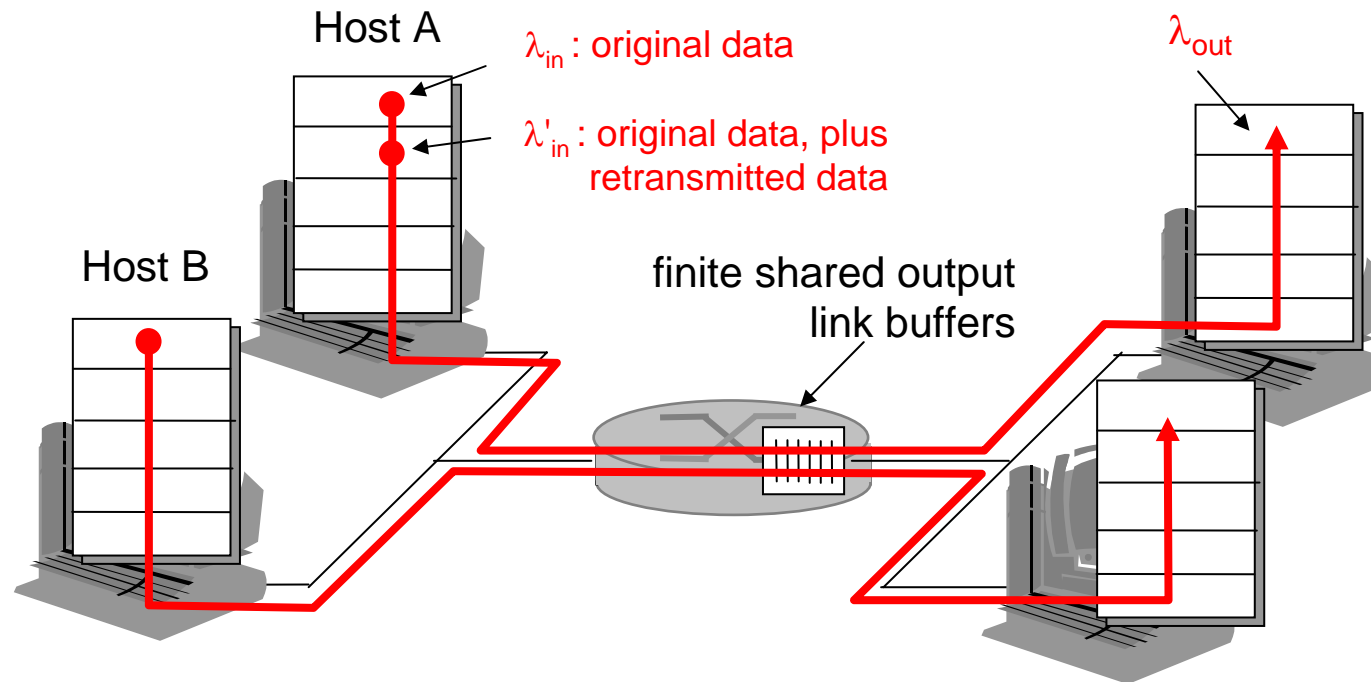


Packet-switching v.s. Circuit Switching (ref. Queueing theory)



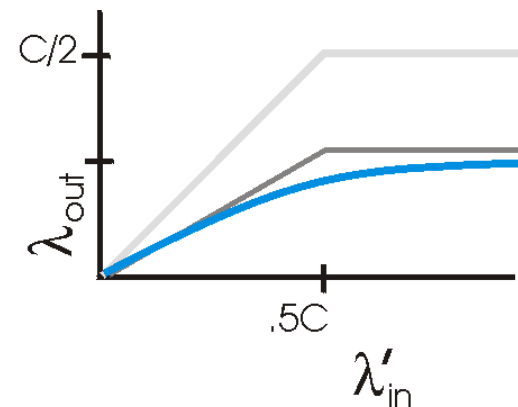
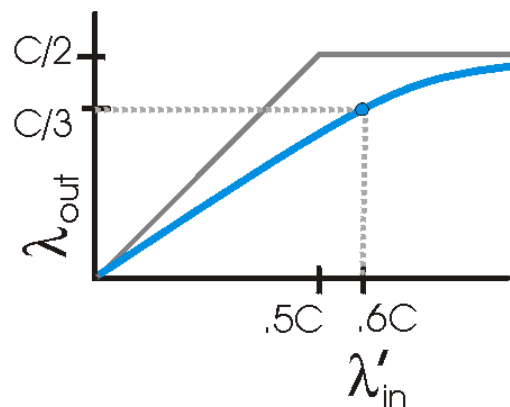
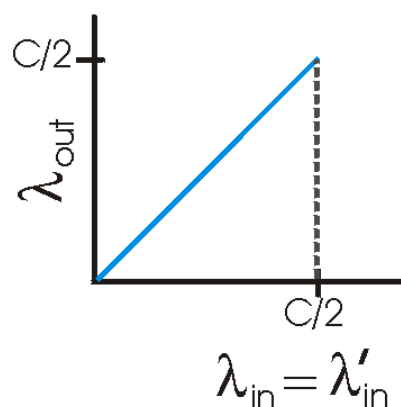
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (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}



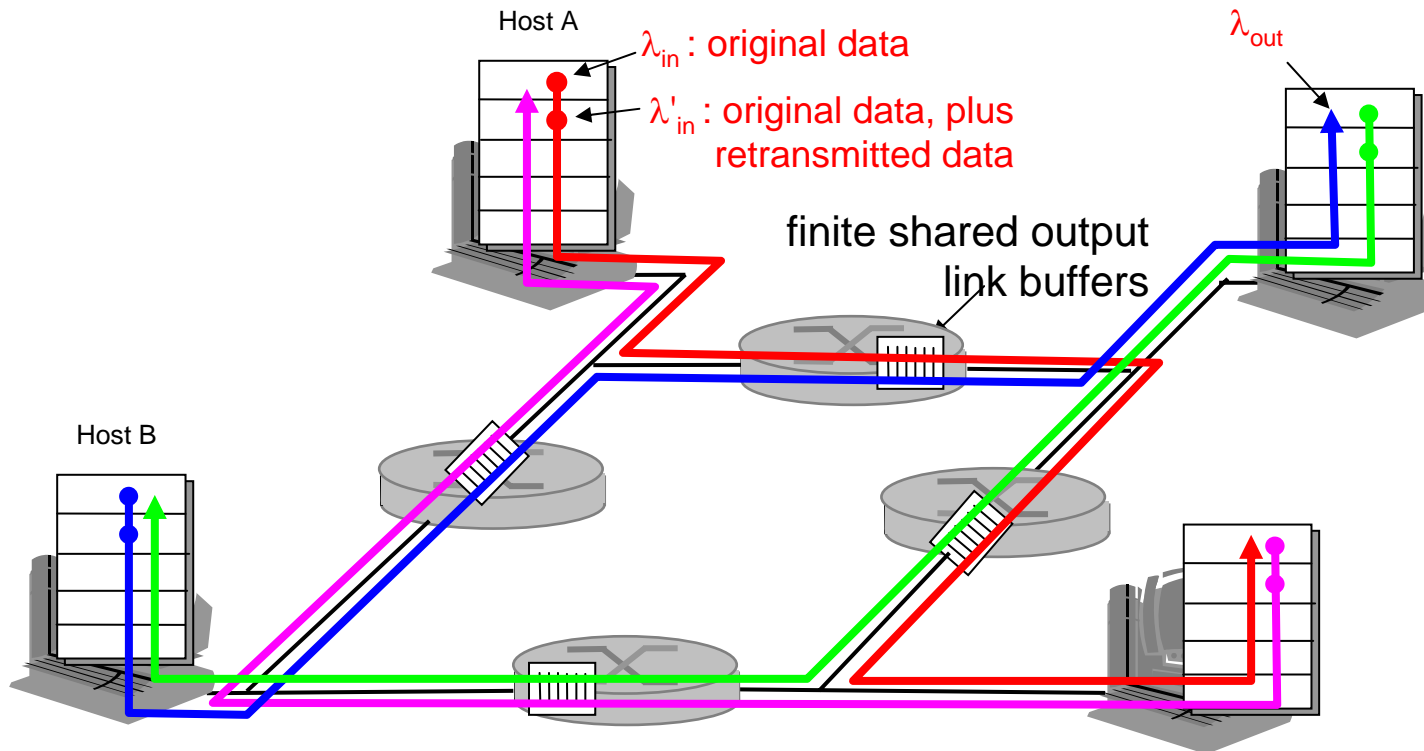
“costs” of congestion:

- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt

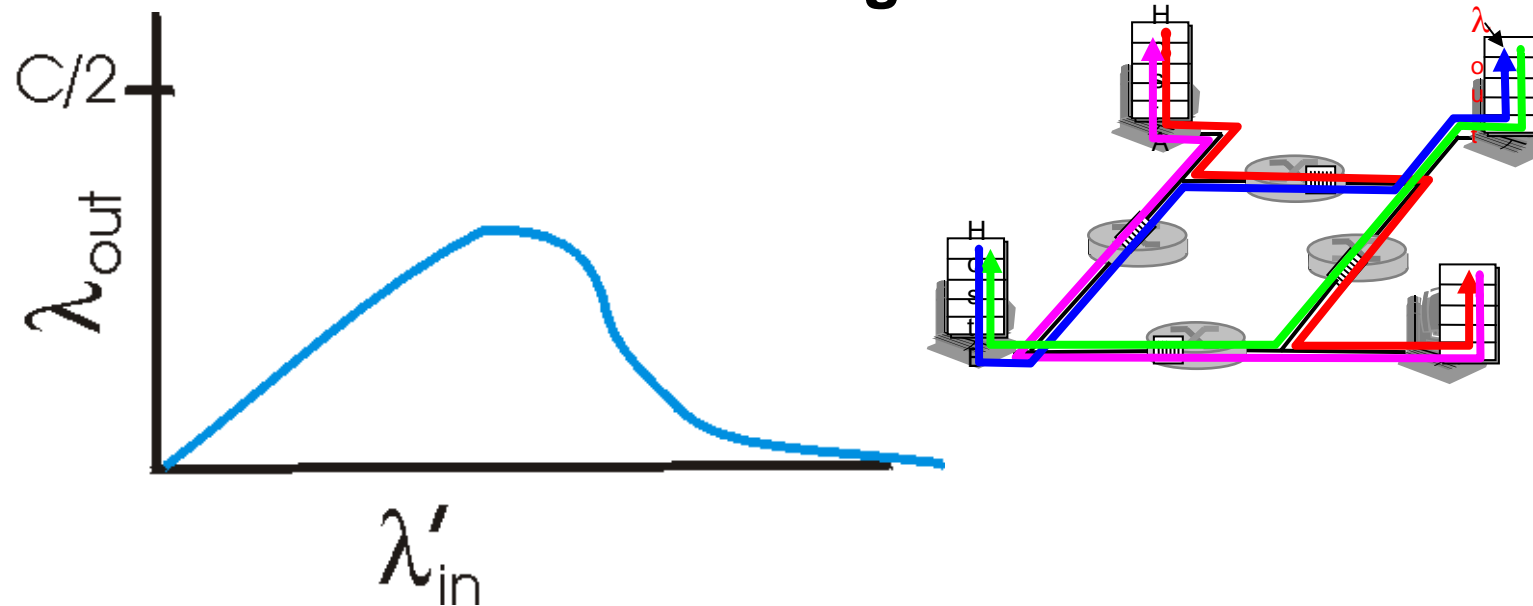
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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



Causes/costs of congestion: scenario 3



Another “cost” of congestion:

- when packet 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, ATM)
 - explicit rate sender should send at

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

- end-end control (no network assistance)
- sender limits transmission:
 $\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$
- Roughly,
$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$
- **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

TCP congestion: three parts:

- AIMD
- slow start
- conservative after timeout events

TCP AIMD

multiplicative decrease:

cut CongWin in half
after loss event

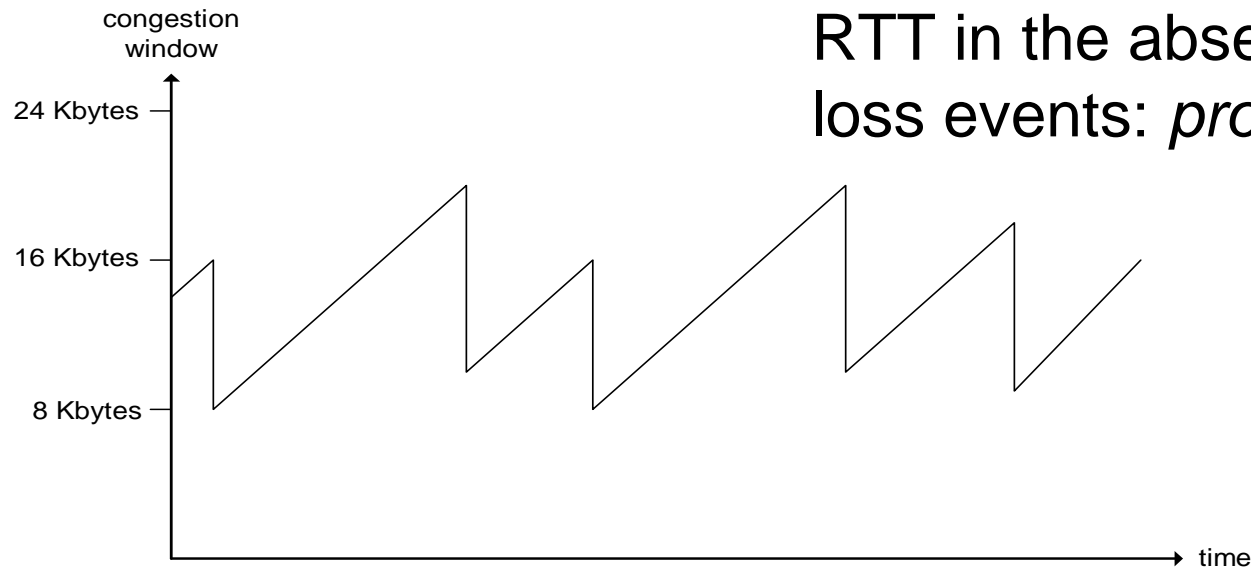
additive increase: increase

CongWin by 1 MSS

(TCP Maximum Segment Size,

≤ 65495 , Ethernet: $1500 - 40 = 1460$) every

RTT in the absence of
loss events: *probing*



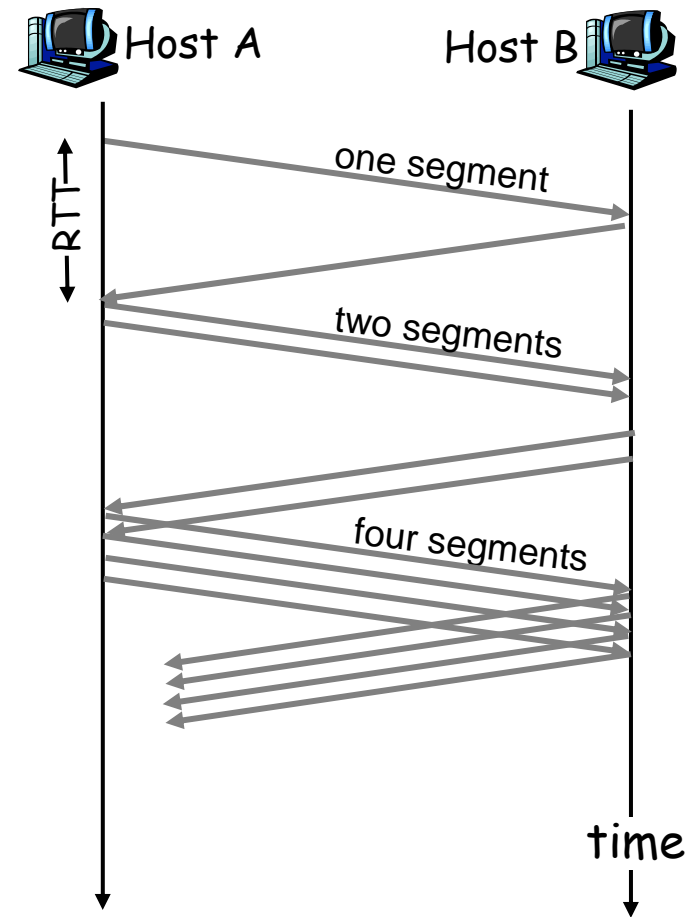
Long-lived TCP connection

TCP Slow Start

- When connection begins, **CongWin = 1 MSS**
 - Example: MSS = 500 bytes & RTT = 200 msec
 - initial rate = 20 kbps
- available bandwidth may be \gg 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
- Summary: initial rate is slow but ramps up exponentially fast



Refinement

- After 3 dup ACKs:
 - CongWin is cut in half
 - window then grows linearly
- But after timeout event:
 - CongWin instead set to 1 MSS;
 - window then grows exponentially
 - to a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is "more alarming"

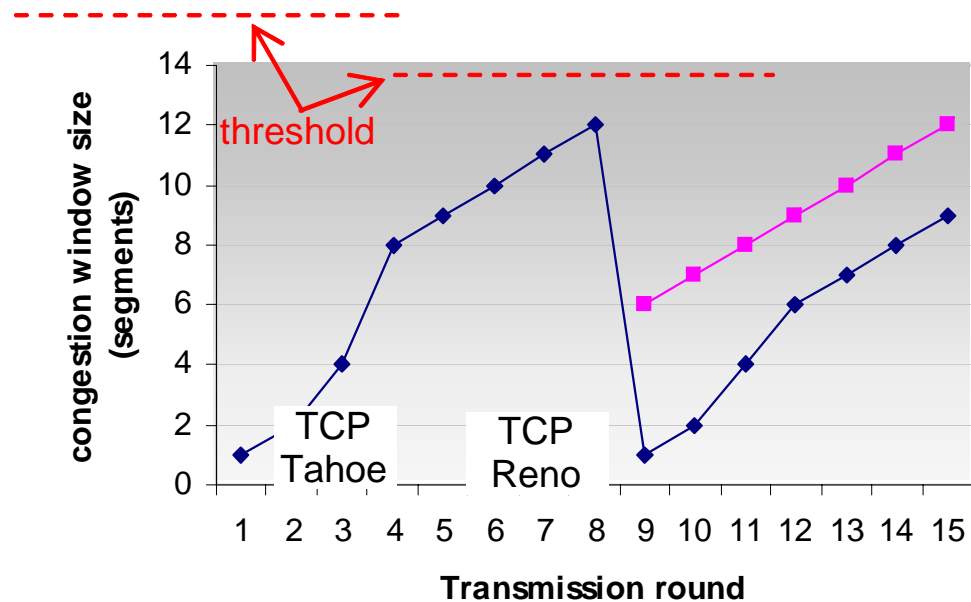
Refinement (more)

Q: When should the exponential increase switch to linear?

A: When **CongWin** gets to 1/2 of its value before timeout.

Implementation:

- 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.

Multiplexing/demultiplexing

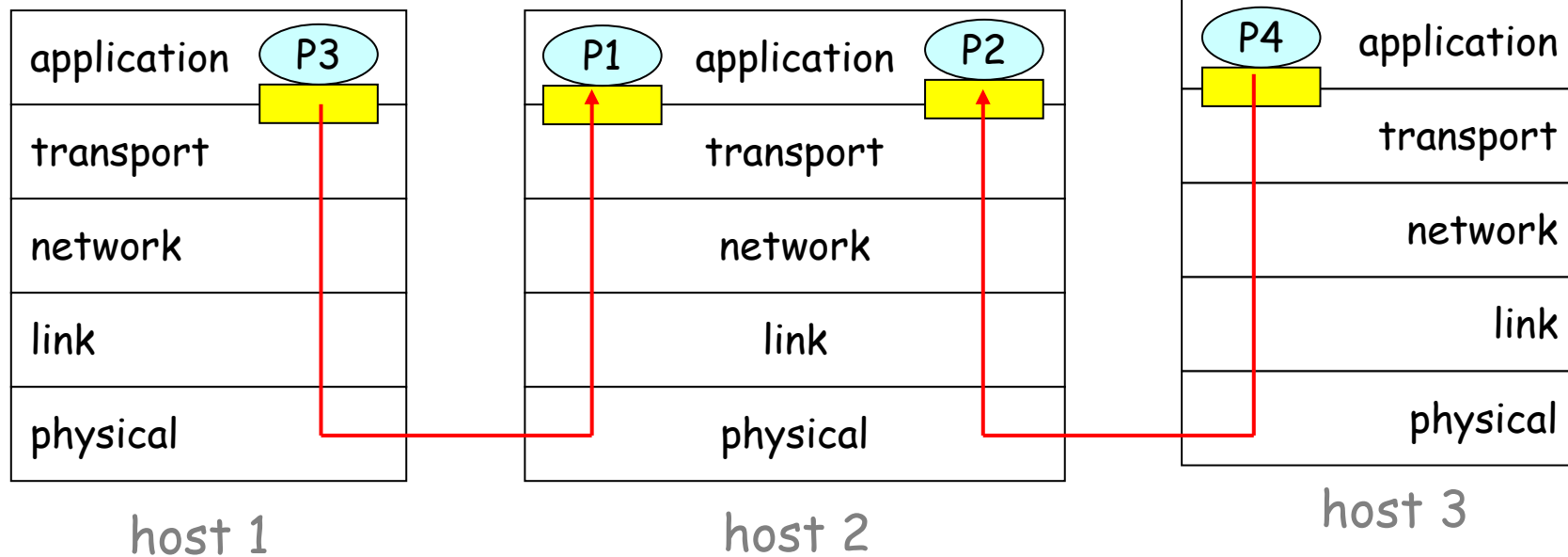
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

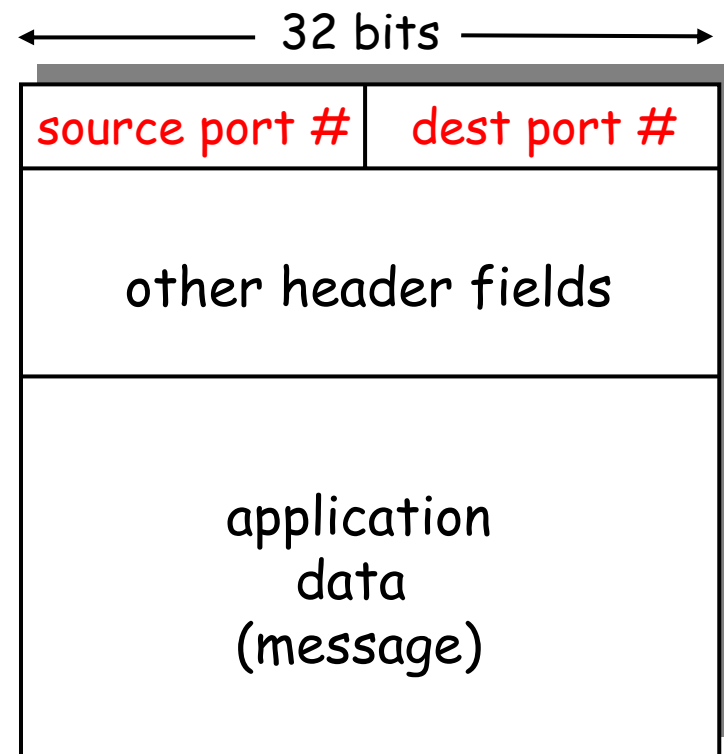
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

■ = socket ○ = process



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 (recall: well-known port numbers for specific applications)
- **host uses IP addresses & port numbers to direct segment to appropriate socket**



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new  
    DatagramSocket(99111);
```

```
DatagramSocket mySocket2 = new  
    DatagramSocket(99222);
```

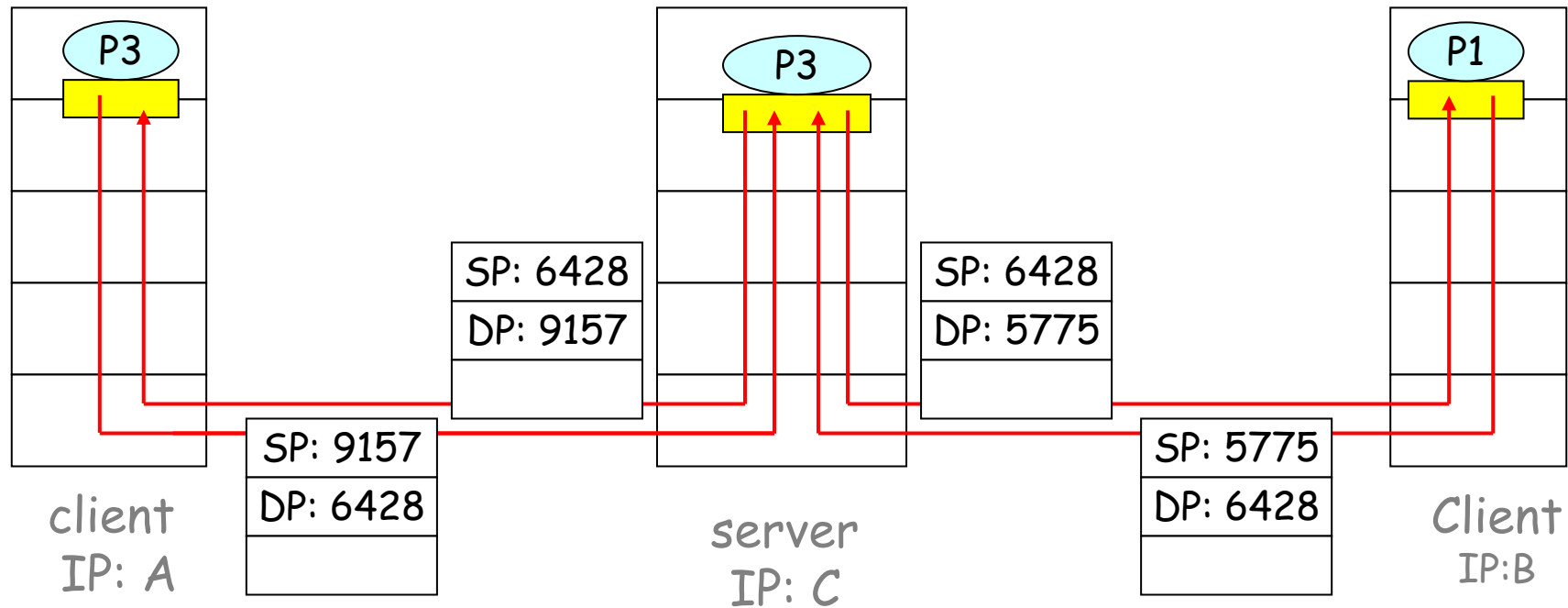
- UDP socket identified by two-tuple:

(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

Connectionless demux (cont)

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

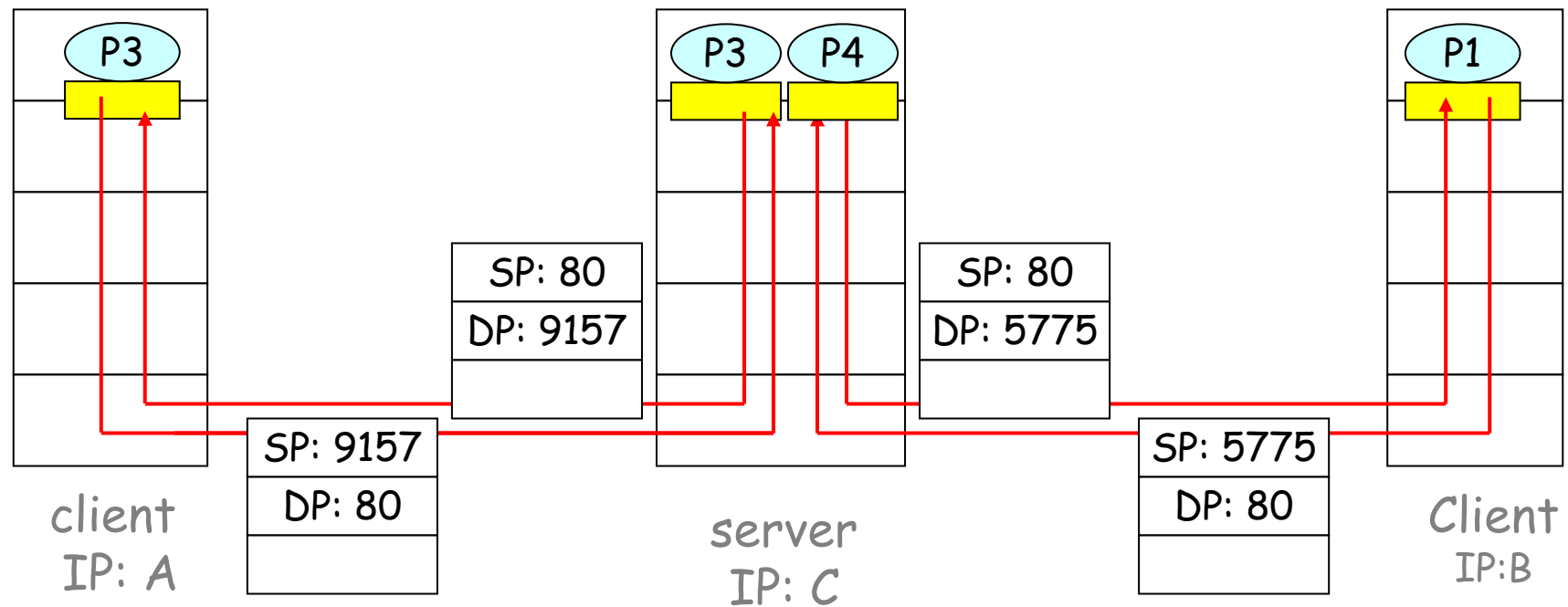


SP provides "return address"

Connection-oriented demux

- TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- recv 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
- Web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



TCP reliable data transfer

- TCP creates rdt 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 consider simplified TCP sender:
 - ignore duplicate acks
 - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- 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`

timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
 - update what is known to be acked
 - start timer if there are outstanding segments

```
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 */
```

WS 2003/04, fu@informatik.cs.uni-goettingen.de

TCP sender (simplified)

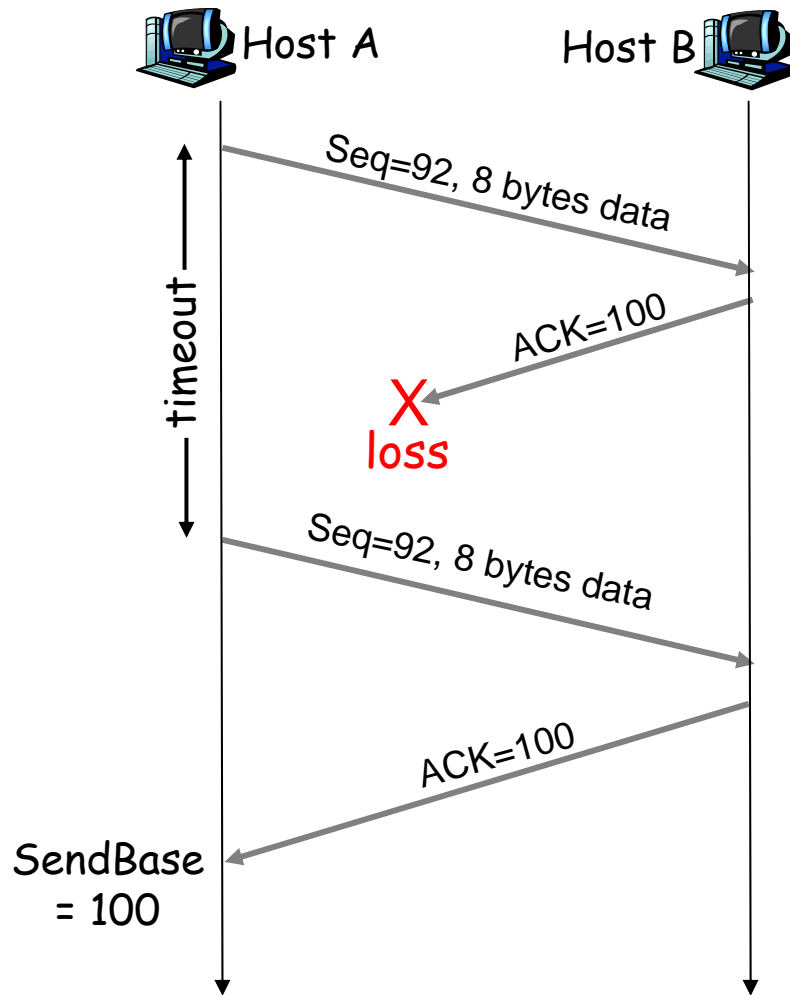
Comment:

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

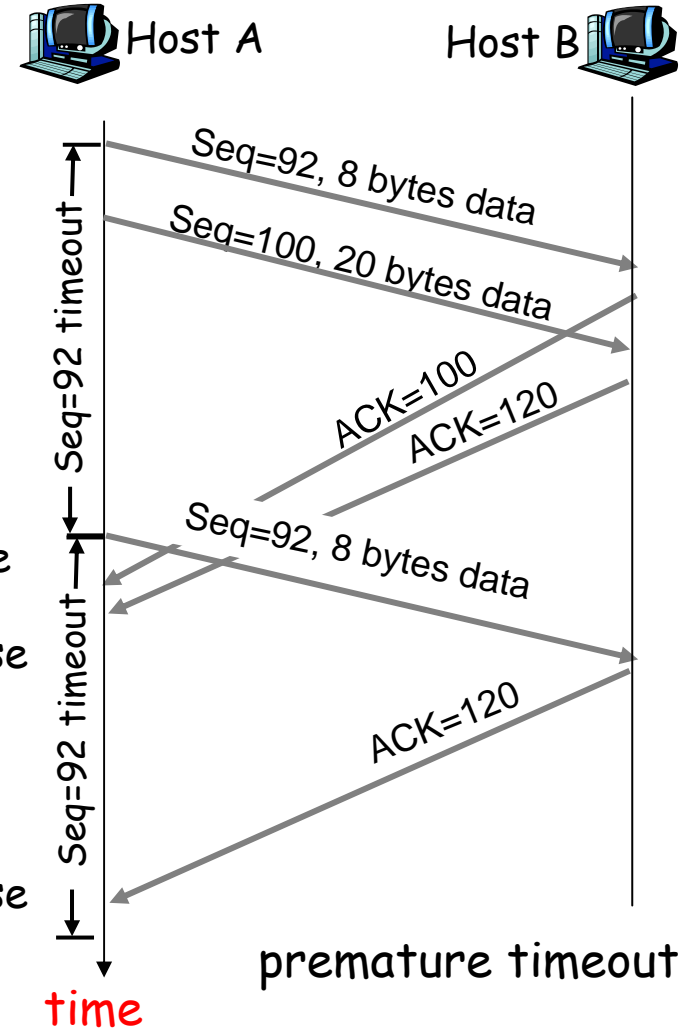
Example:

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

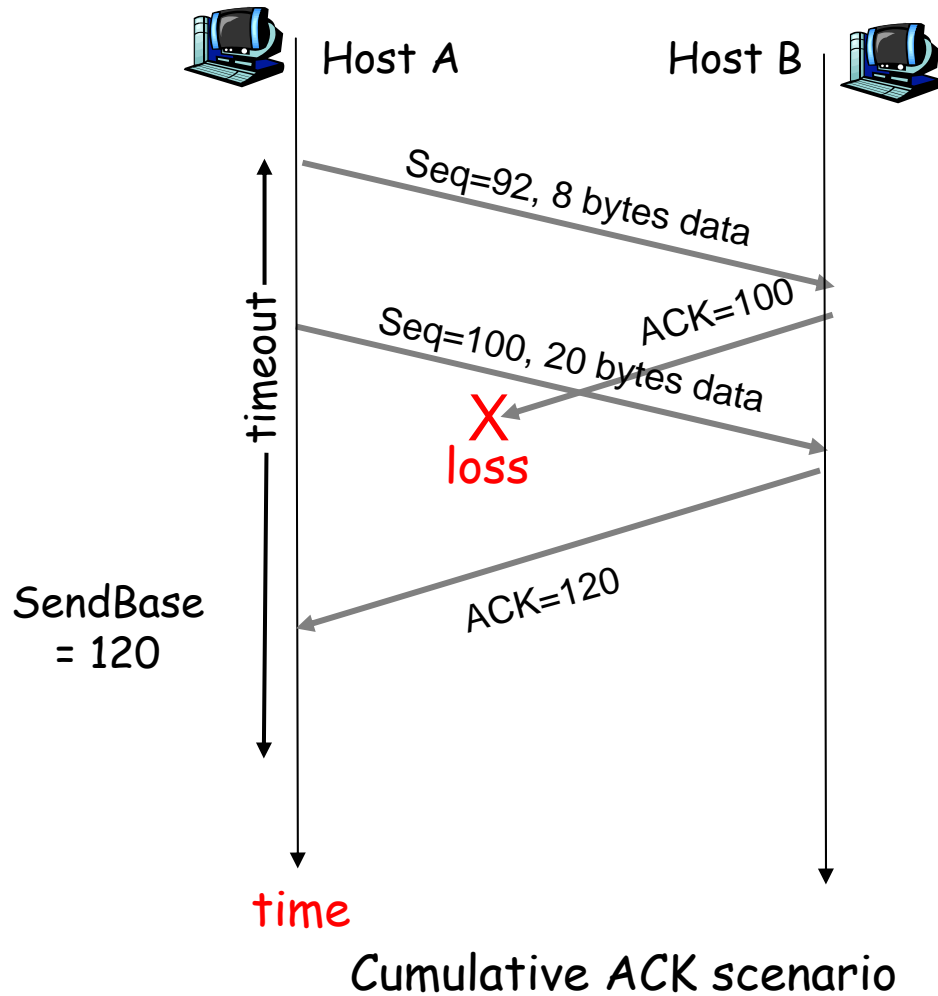
TCP: retransmission scenarios



lost ACK scenario



TCP retransmission scenarios (more)



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-expected 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

Fast Retransmit

- Time-out period often relatively long:
 - long delay before resending lost packet
- 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

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 Connection Management

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

- initialize TCP variables:
 - seq. #s
 - 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();`

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

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

- server allocates buffers
- specifies server initial seq. #

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

TCP Connection Management (cont.)

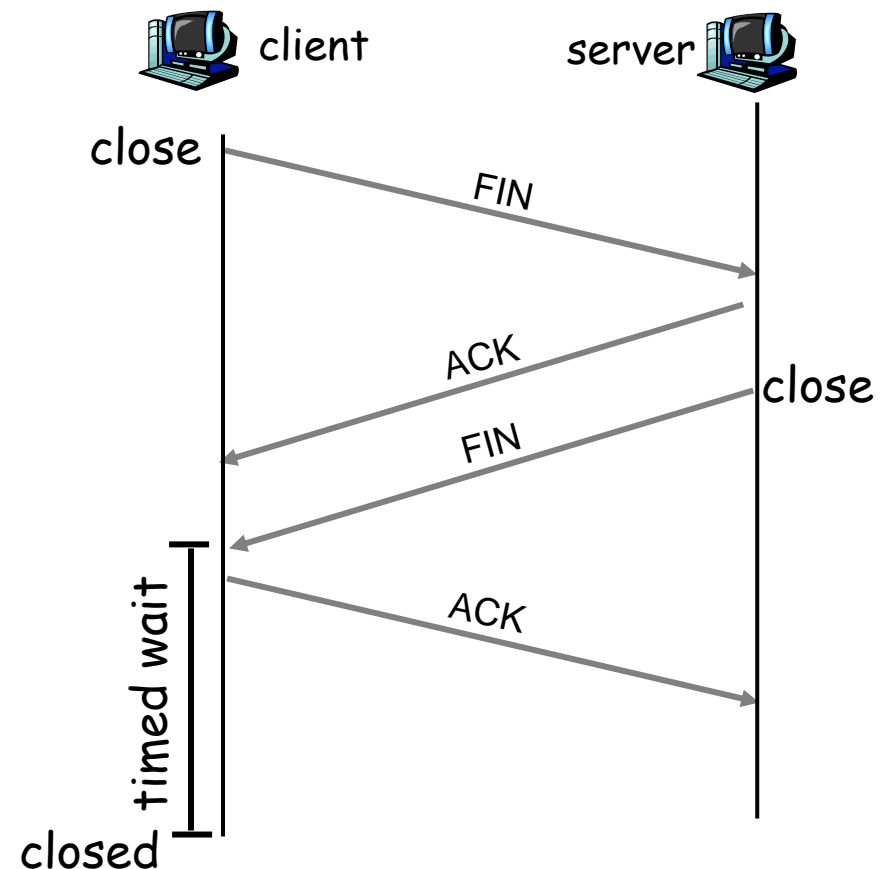
Closing a connection:

client closes socket:

```
clientSocket.close();
```

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

Step 2: server receives FIN,
replies with ACK. Closes
connection, sends FIN.



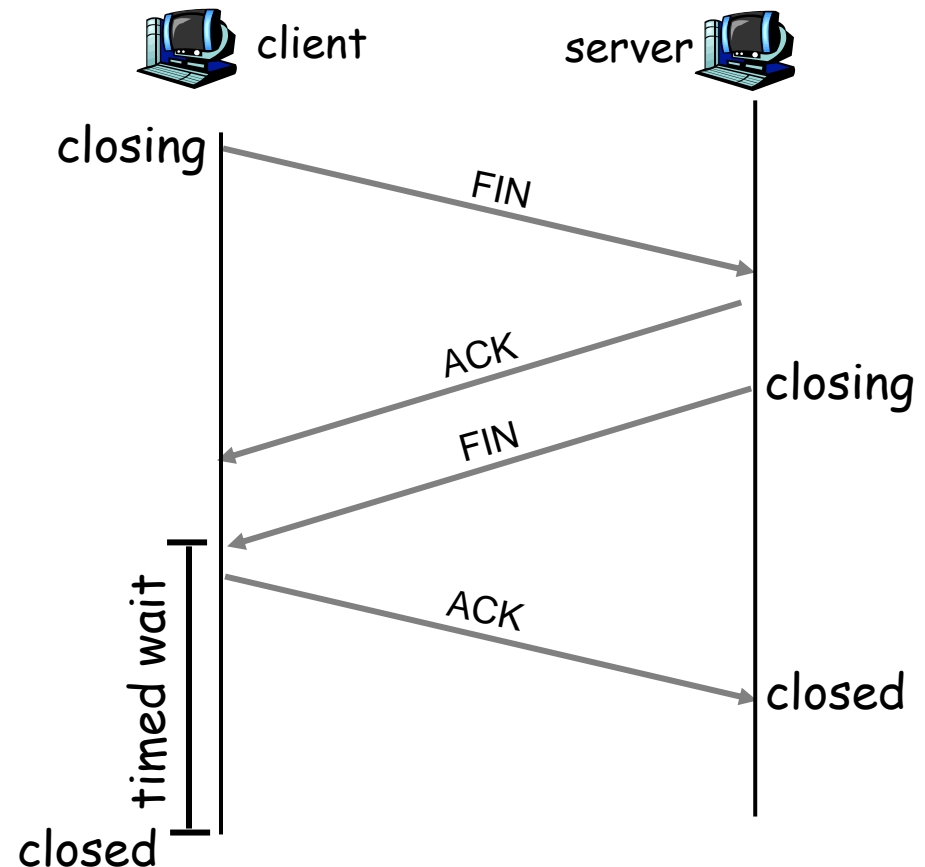
TCP Connection Management (cont.)

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.

Note: with small modification,
can handle simultaneous
FINs.



Transport Layer: Summary

- principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
 - Connection management (for connection-oriented protocol)

Connectless v.s. connection-oriented:

- Reliability, fragmentation
- Different overhead
- Others?

Next:

- Learn about protocol “stacks”: TCP/IP, ATM families
- Specific issues with networks: QoS, security, multimedia (VoIP)...

Homework

- What are transport layer services? Identify the difference between connection-oriented and connectless services.
- Elaborate TCP's operational phases
- Kurose/Ross: Chapter 3 Review questions 4& 8; Problems 2, 16, 26

Delay modeling

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:

- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:

- Assume one link between client and server of rate R
- S : MSS (bits)
- O : object size (bits)
- no retransmissions (no loss, no corruption)

Window size:

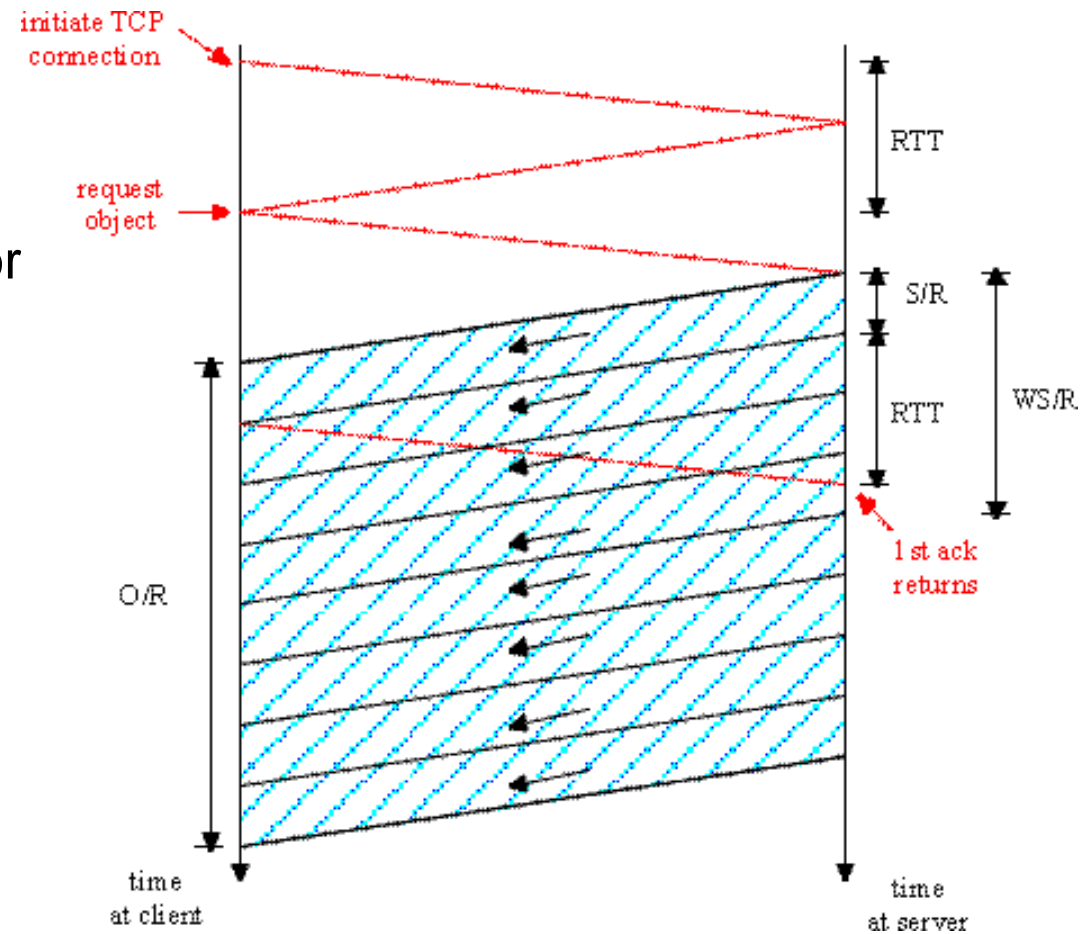
- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start

Fixed congestion window (1)

First case:

$WS/R > RTT + S/R$: ACK for first segment in window returns before window's worth of data sent

$$\text{delay} = 2RTT + O/R$$

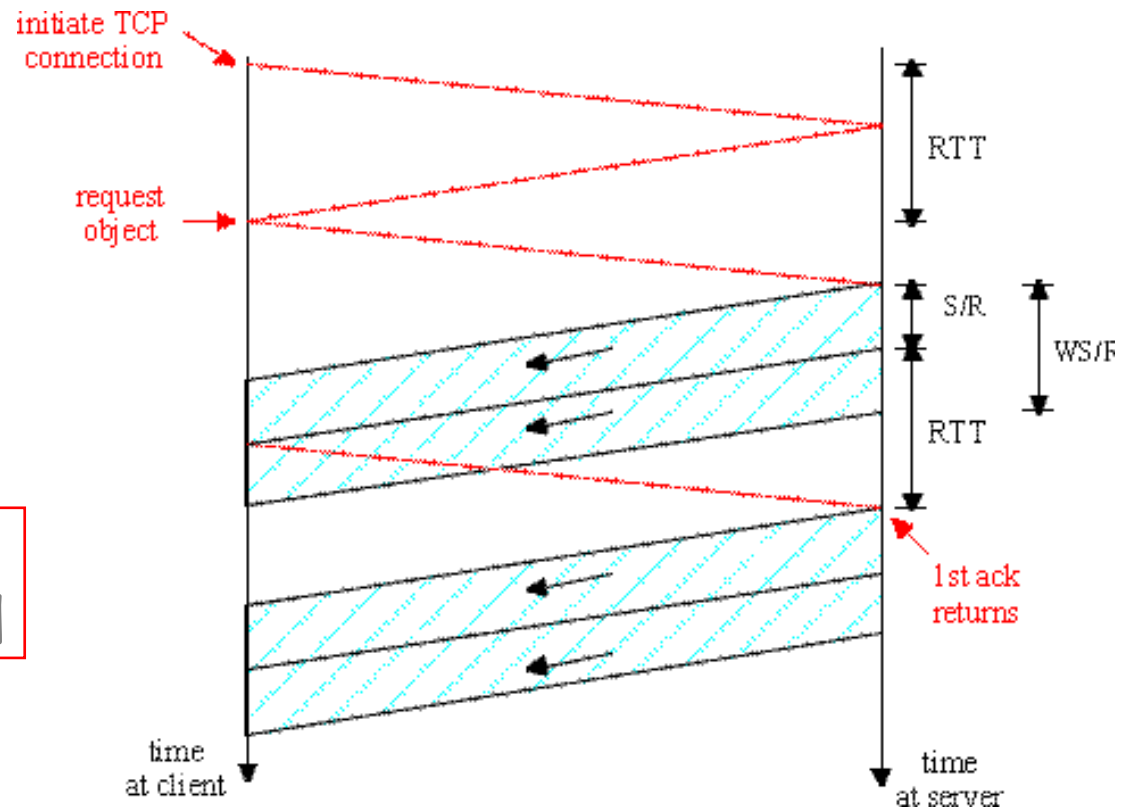


Fixed congestion window (2)

Second case:

- $WS/R < RTT + S/R$: wait for ACK after sending window's worth of data sent

$$\text{delay} = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]$$



TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

$$Latency = 2RTT + \frac{O}{R} + P \left[RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R}$$

where P is the number of times TCP idles at server:

$$P = \min\{Q, K - 1\}$$

- where Q is the number of times the server idles if the object were of infinite size.
- and K is the number of windows that cover the object.

TCP Delay Modeling: Slow Start (2)

Delay components:

- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

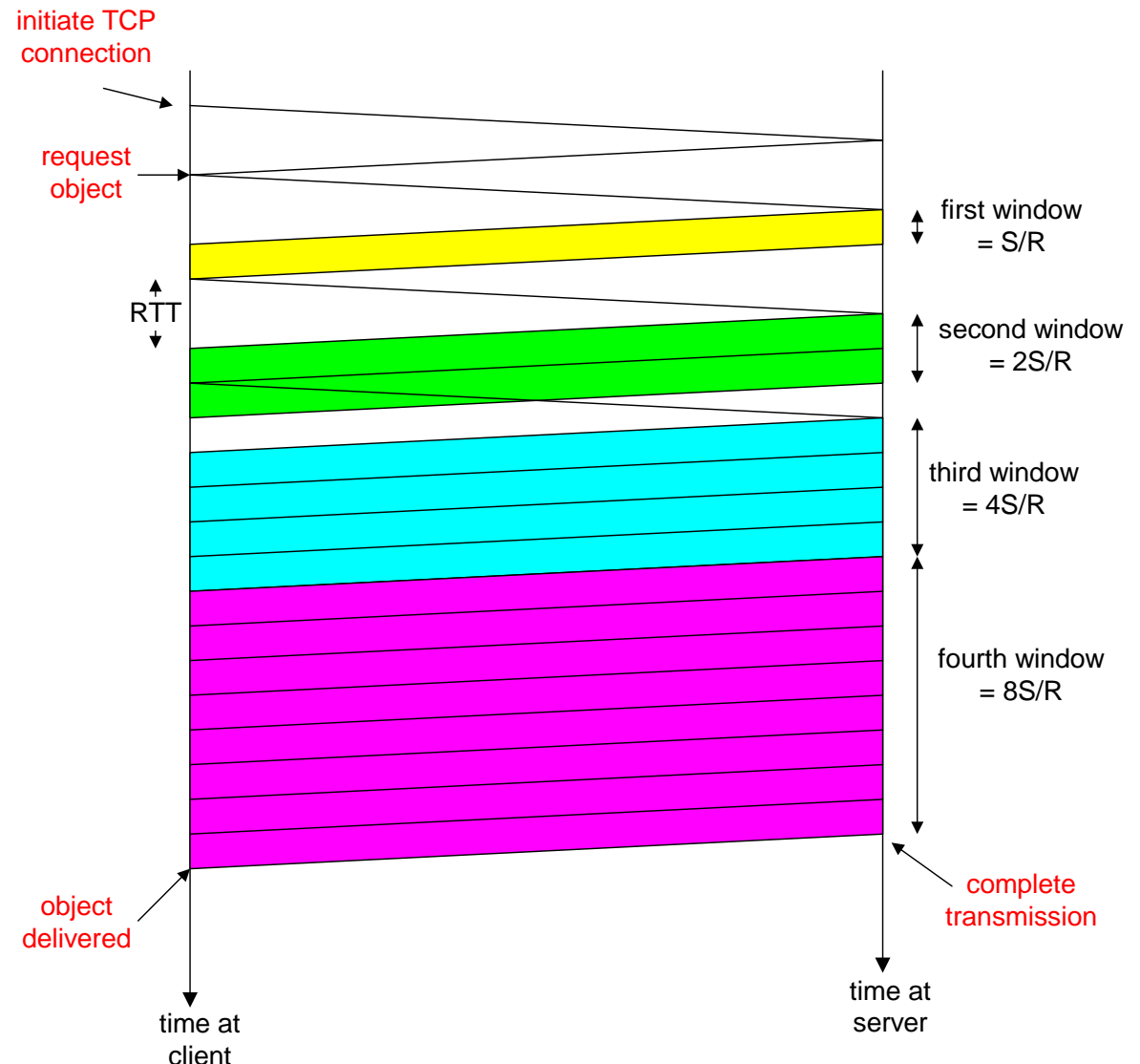
Server idles:

$$P = \min\{K-1, Q\} \text{ times}$$

Example:

- $O/S = 15$ segments
- $K = 4$ windows
- $Q = 2$
- $P = \min\{K-1, Q\} = 2$

Server idles $P=2$ times



TCP Delay Modeling (3)

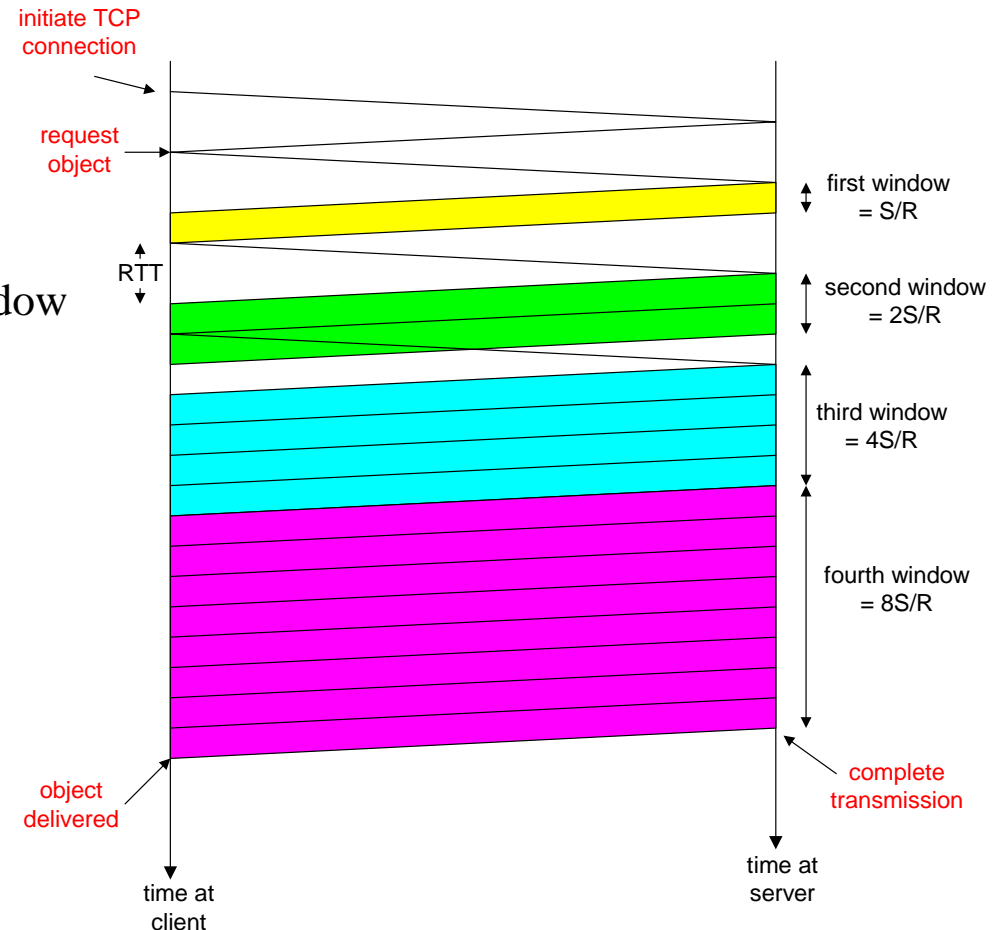
$\frac{S}{R} + RTT =$ time from when server starts to send segment

until server receives acknowledgement

$2^{k-1} \frac{S}{R} =$ time to transmit the k th window

$\left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]^+ =$ idle time after the k th window

$$\begin{aligned} \text{delay} &= \frac{O}{R} + 2RTT + \sum_{p=1}^P \text{idleTime}_p \\ &= \frac{O}{R} + 2RTT + \sum_{k=1}^P \left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right] \\ &= \frac{O}{R} + 2RTT + P \left[RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R} \end{aligned}$$



TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K ?

$$\begin{aligned} K &= \min\{k : 2^0 S + 2^1 S + \dots + 2^{k-1} S \geq O\} \\ &= \min\{k : 2^0 + 2^1 + \dots + 2^{k-1} \geq O/S\} \\ &= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\ &= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\ &= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil \end{aligned}$$

Calculation of Q , number of idles for infinite-size object, is similar (see HW).