Master Course
Computer Networks
IN2097

Prof. Dr.-Ing. Georg Carle
Christian Grothoff, Ph.D.
Stephan Günther

Chair for Network Architectures and Services
Department of Computer Science
Technische Universität München
http://www.net.in.tum.de

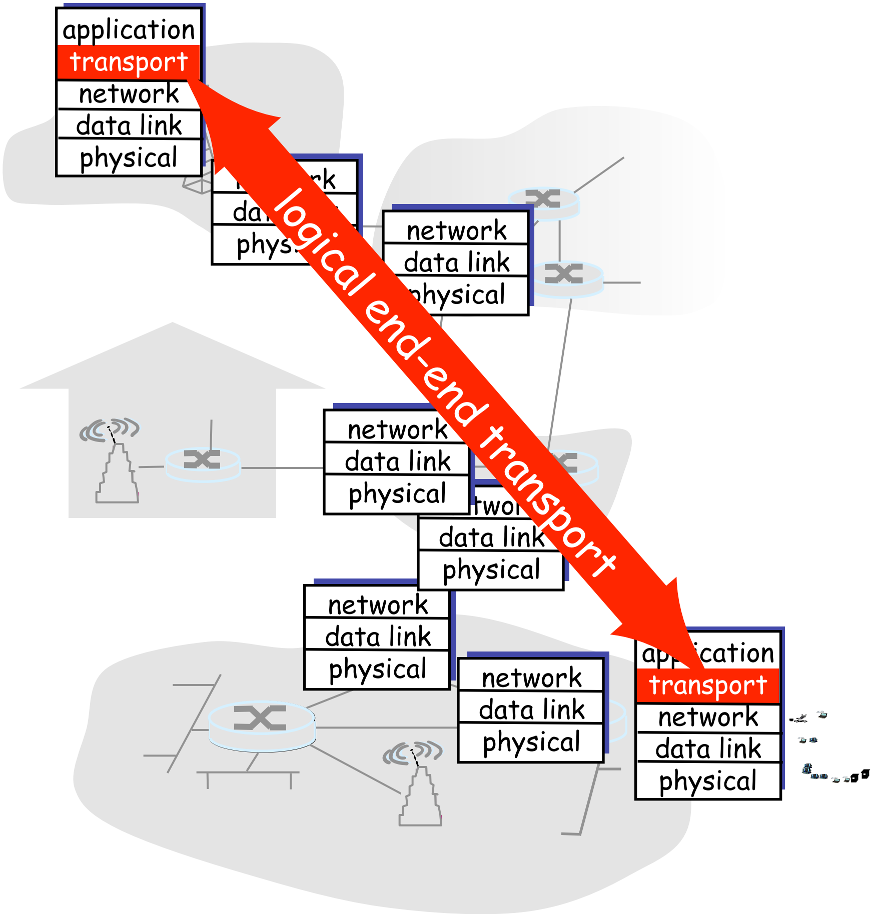
Roadmap

- Chapter: Transport Layer

  - Transport Layer Functions
  - UDP
  - TCP

Internet Transport-layer Protocols

- Unreliable, unordered delivery: UDP
  - simple extension of “best-effort” IP
- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup



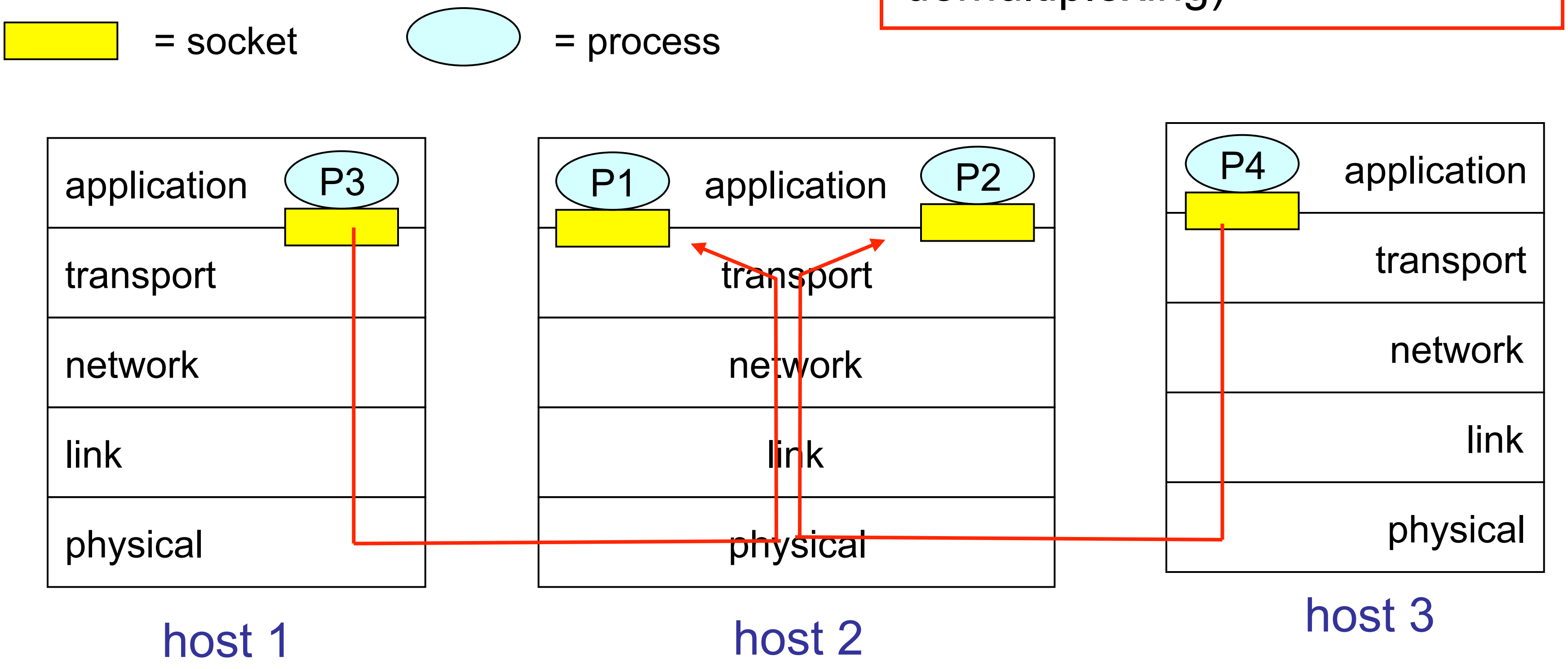
Multiplexing/Demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

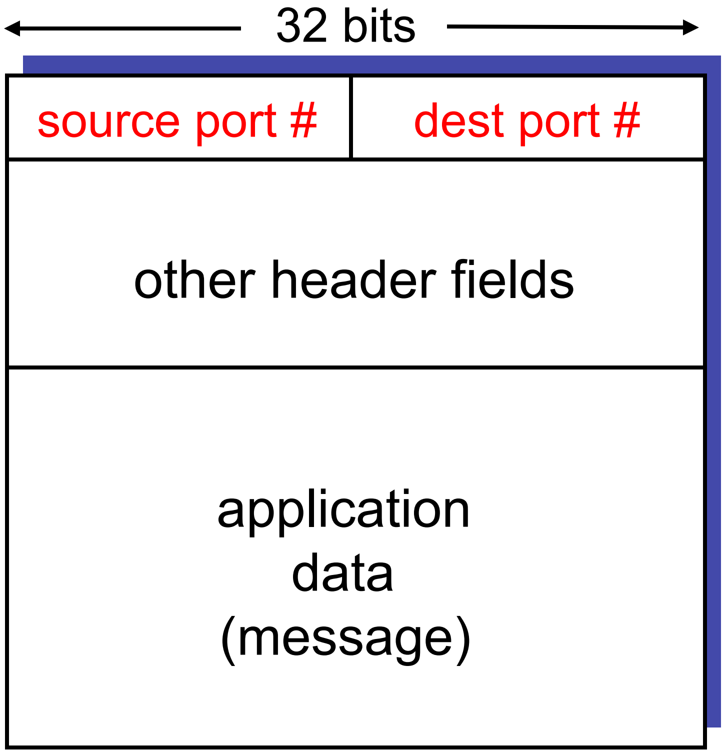
Multiplexing at send host:

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



Demultiplexing

- Host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source port number, destination port number
- Host uses IP five tuple <Src/Dst IP address, Protocol, Src/Dst Port number> to direct segment to appropriate socket



TCP/UDP segment format

Connectionless Demultiplexing

- Create sockets with specific port numbers:

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

- UDP socket identified by two-tuple:

  (dest IP address, dest port number)

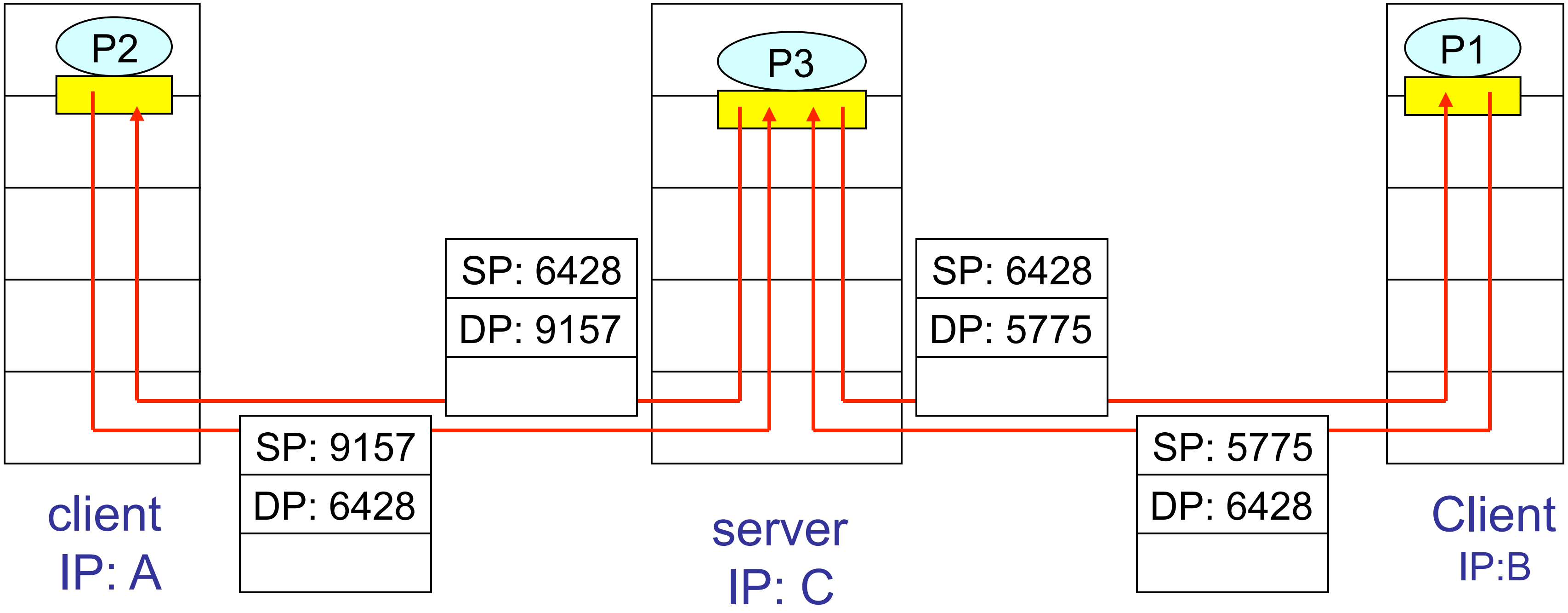
- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with

  different source IP addresses and/or
  different source port numbers

  directed to same socket

Connectionless Demultiplexing

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

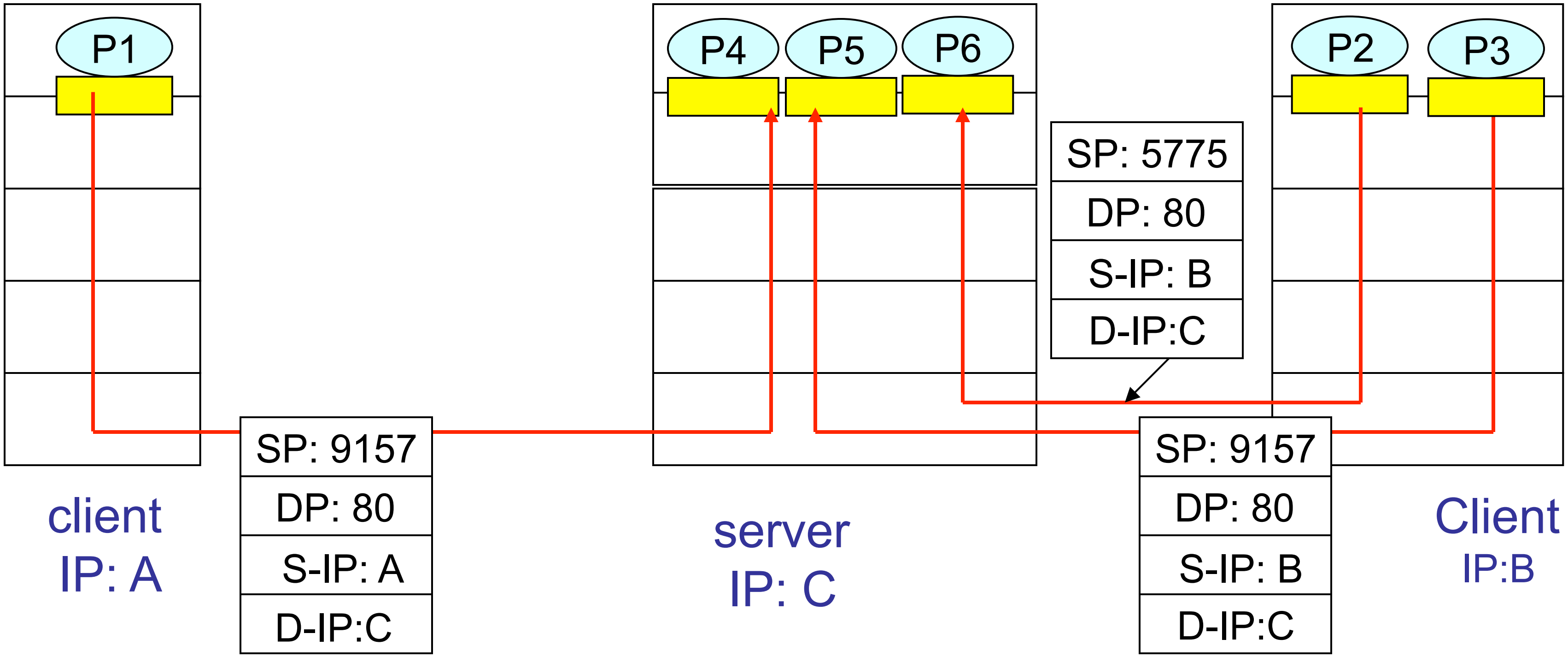


Source Port (SP) provides "return address"

Connection-Oriented Demultiplexing

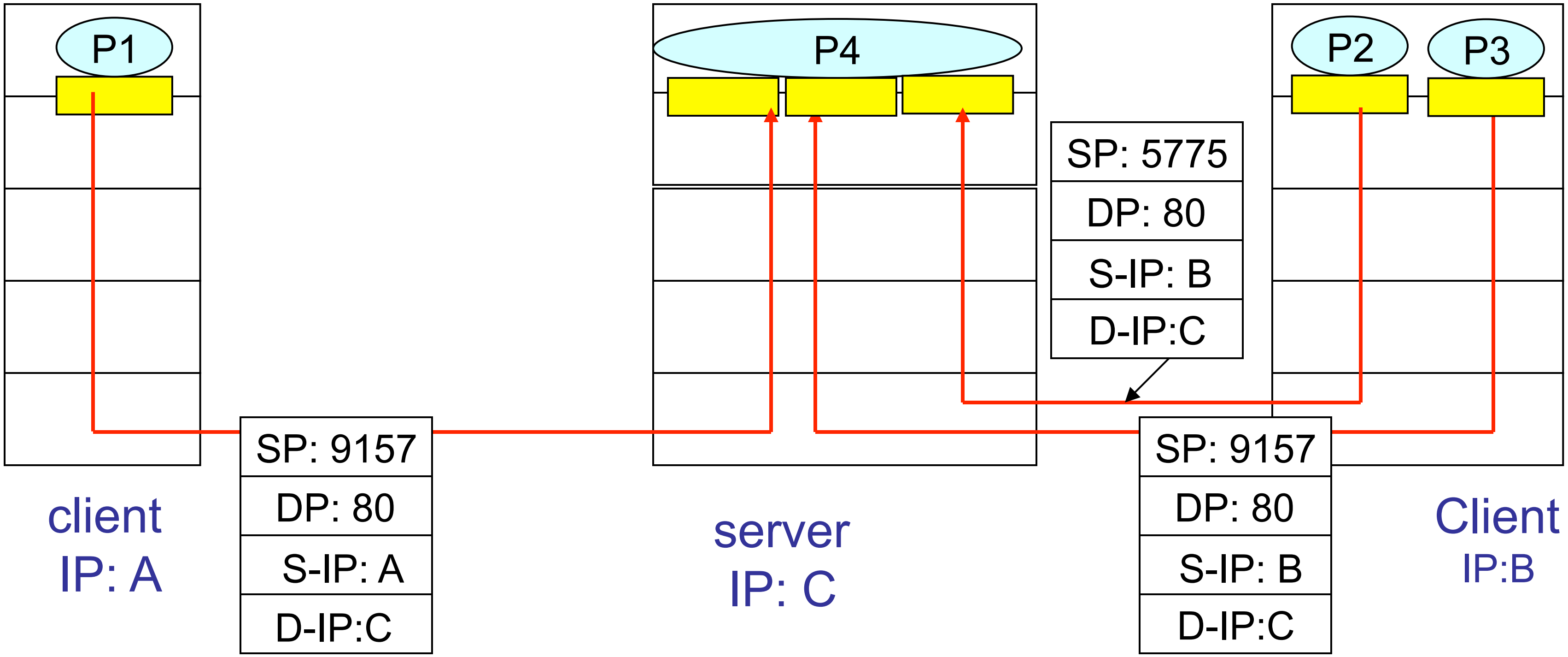
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Receiving host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP (TCP connection closed after transfer of requested object) have different socket for each request

Connection-Oriented Demultiplexing



Connection-Oriented Demultiplexing

- Threaded Web Server



UDP: User Datagram Protocol [RFC 768]

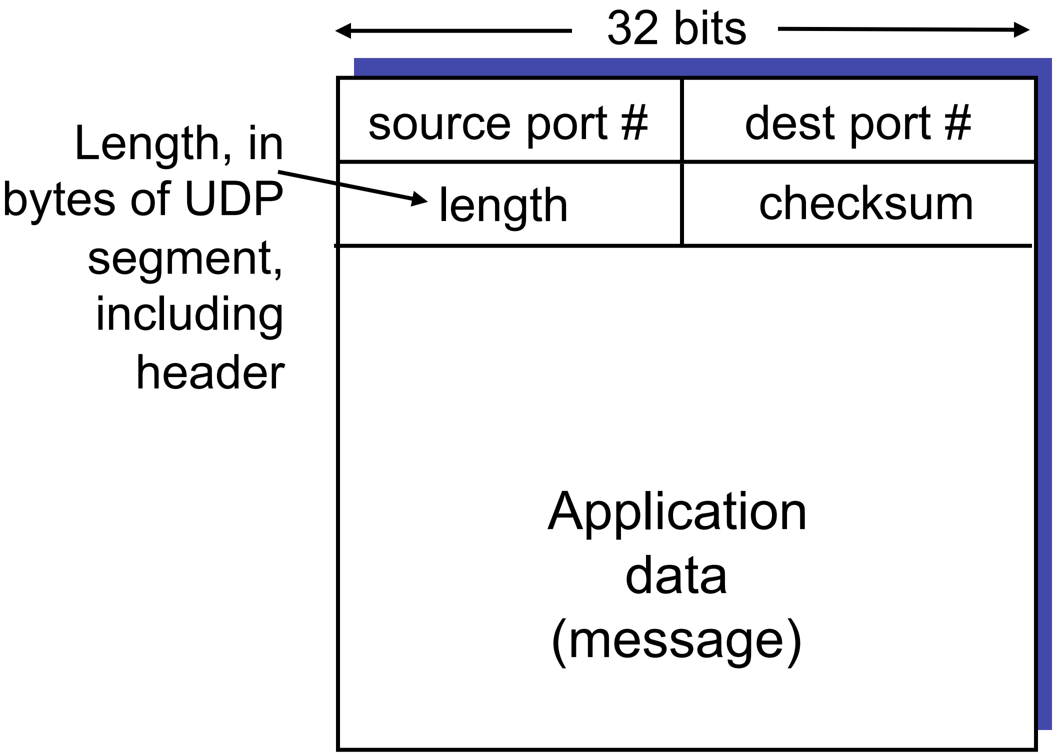
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

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

UDP

- often used for streaming multimedia applications
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - ⇨ application-specific error recovery


32 bits
source port #
dest port #
length
checksum
Application data (message)
Length, in bytes of UDP segment, including header

UDP segment format

- UDPlite: RFC 3828, Protocol Number 136
  - allows a potentially damaged data payload to be delivered to an application rather than being discarded
  - checksum coverage instead of length field

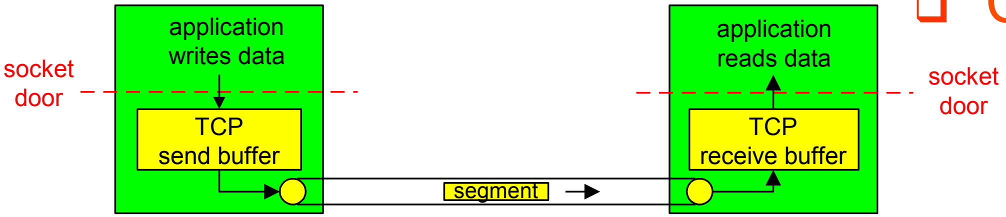
Section TCP - Outline

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

TCP: Overview

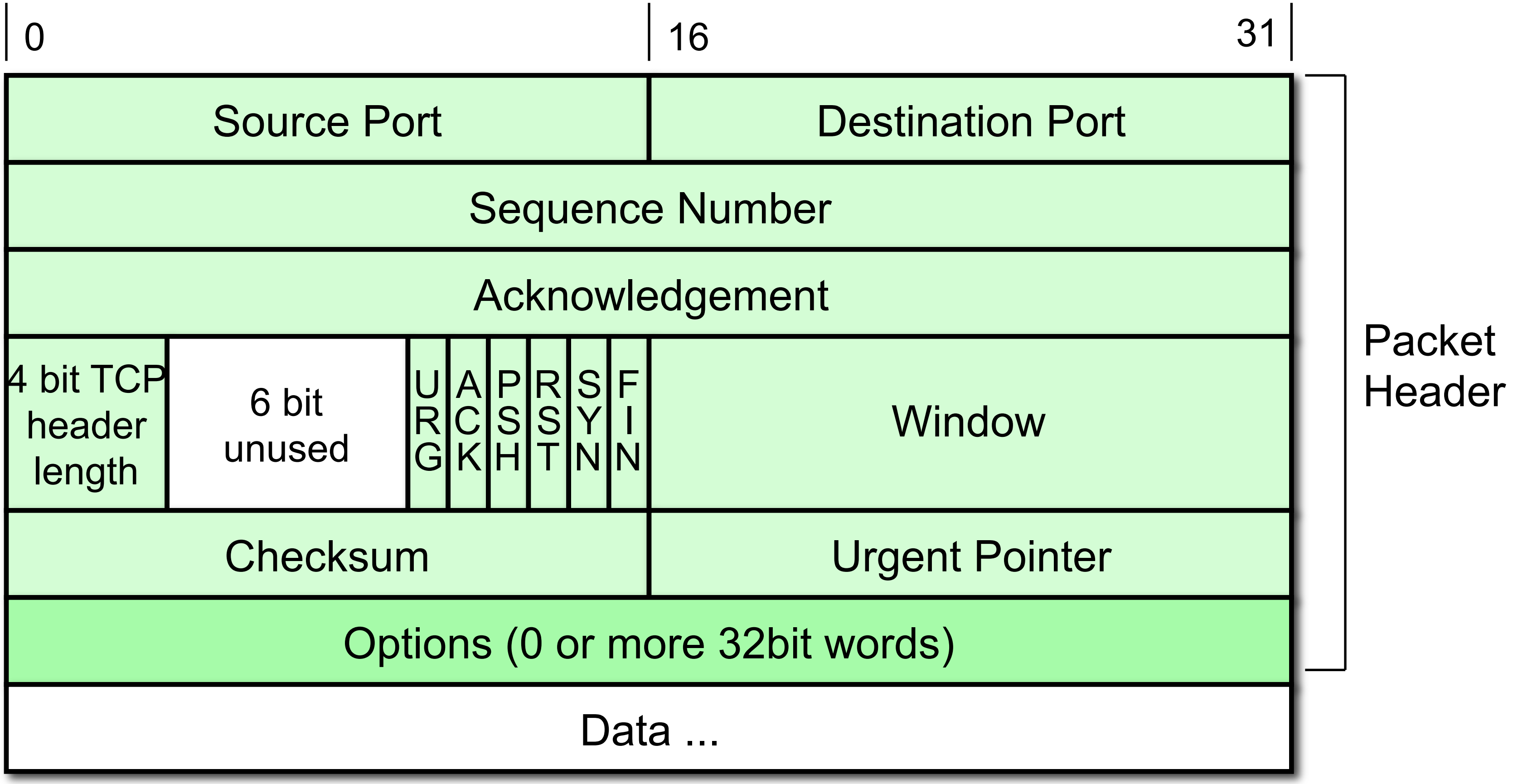
RFCs: 793, 1122, 1323, 2018, 2581

- 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) init’ s sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver
- Congestion controlled:
  - Will not overwhelm network

TCP Header



0

16

31

Source Port

Destination Port

Sequence Number

Acknowledgement

4 bit TCP header length

6 bit unused

URG

ACK

PSH

RST

SYN

FIN

Window

Checksum

Urgent Pointer

Options (0 or more 32bit words)

Data ...

Packet Header

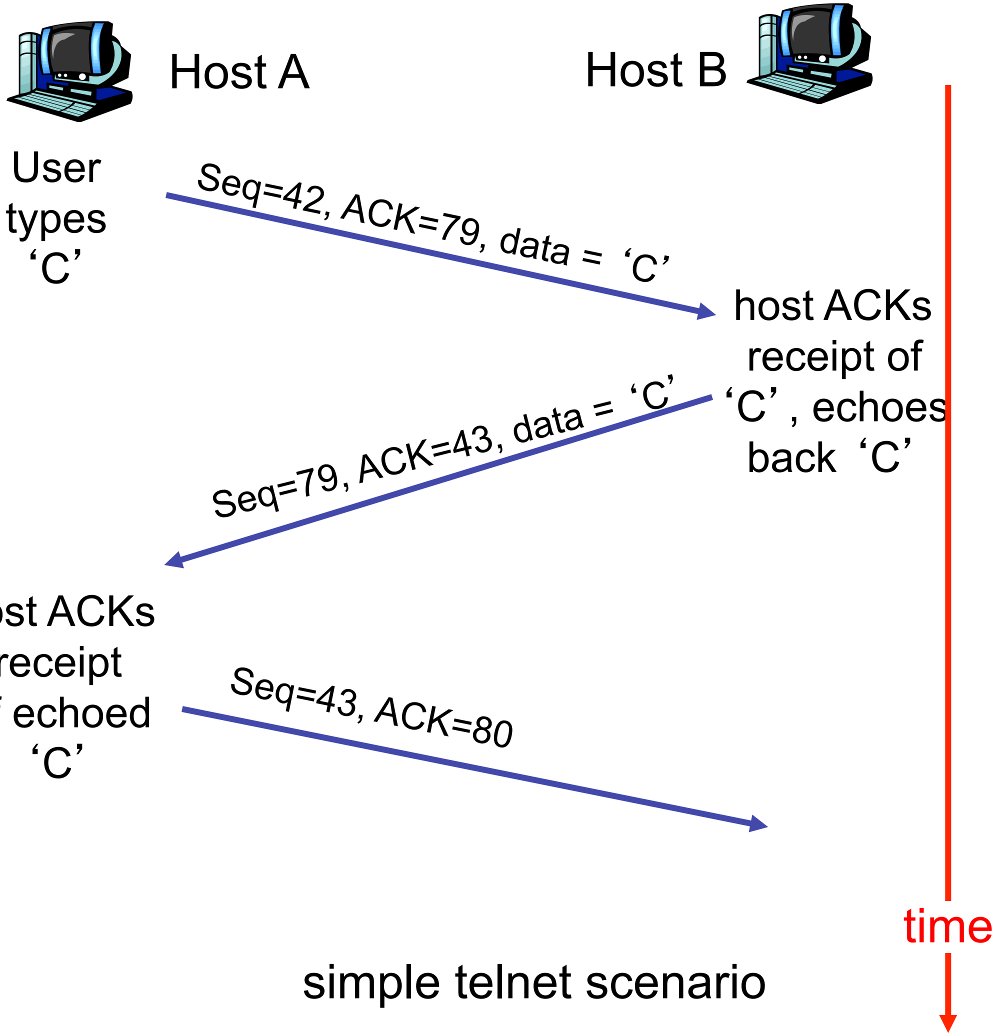
TCP seq. #’ s and ACKs

Seq. #’ s:

- byte stream “number” of first byte in segment’ s data

ACKs:

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


Host A
Host B
User types ‘C’
Seq=42, ACK=79, data = ‘C’
host ACKs receipt of ‘C’ , echoes back ‘C’
Seq=79, ACK=43, data = ‘C’
host ACKs receipt of echoed ‘C’
Seq=43, ACK=80
time

simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

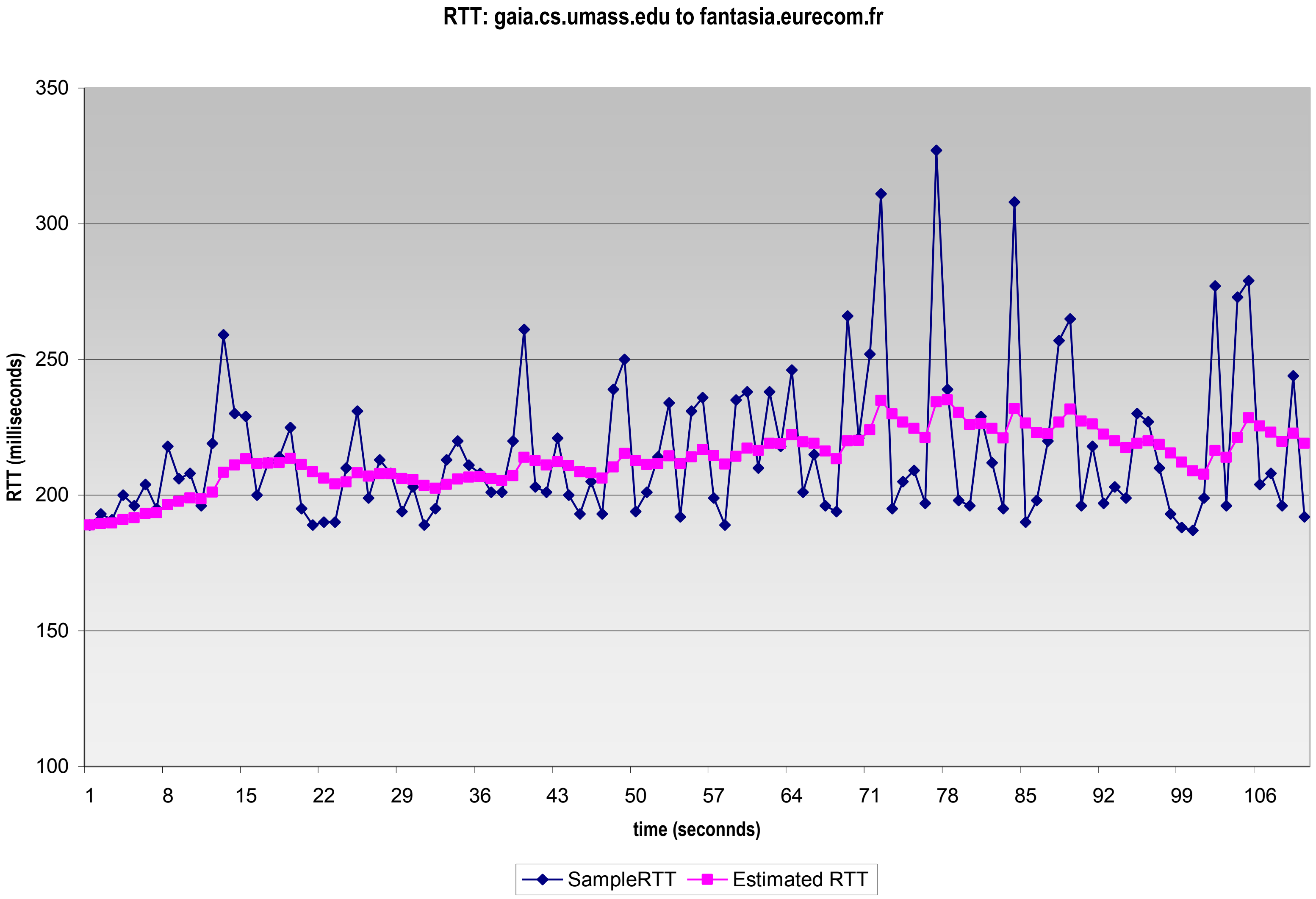
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

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

- exponential weighted moving average
- influence of past sample decreases
  (more weight on recent samples than on older samples)
- typical value: $\alpha = 0.125$

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimtedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** **->** larger safety margin
- Estimate of how much SampleRTT deviates from EstimatedRTT (DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT)

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

(typically, β = 0.25)
```

- Set timeout interval

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

Section TCP - Outline

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

TCP Reliable Data Transfer

- TCP creates reliable data transfer (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

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

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 */
```

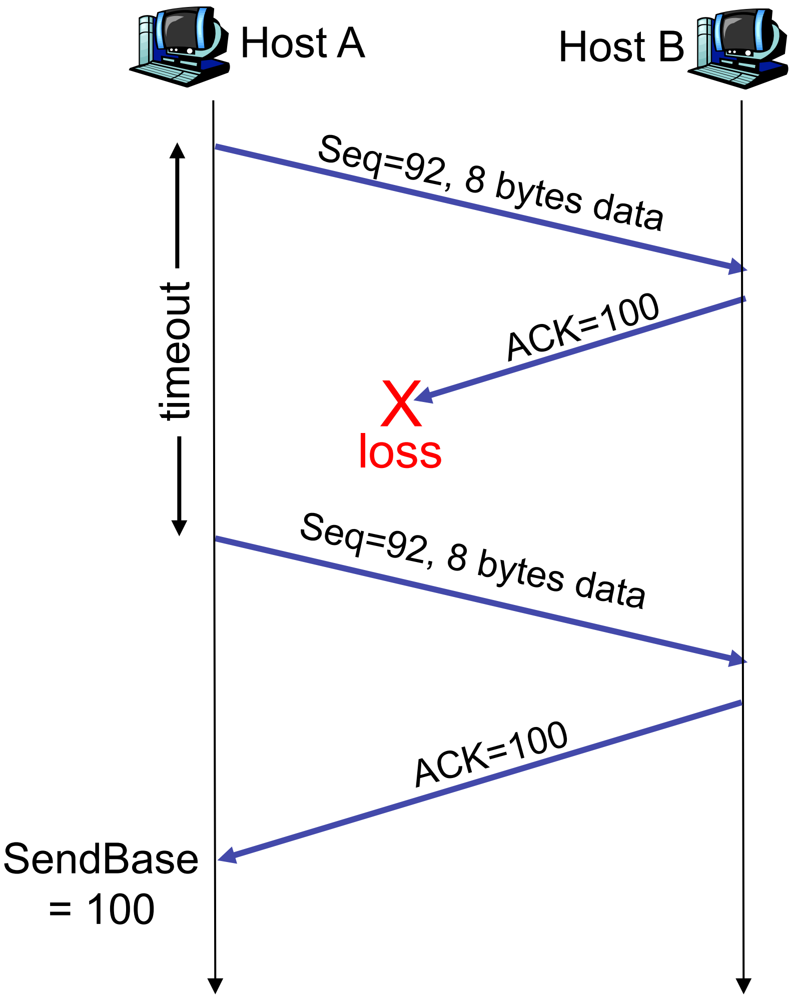
<u>Comment:</u>

- SendBase-1: last cumulatively ack’ed byte

<u>Example:</u>

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

TCP: Retransmission Scenarios



Host A

Host B

Seq=92, 8 bytes data

ACK=100

X

loss

timeout

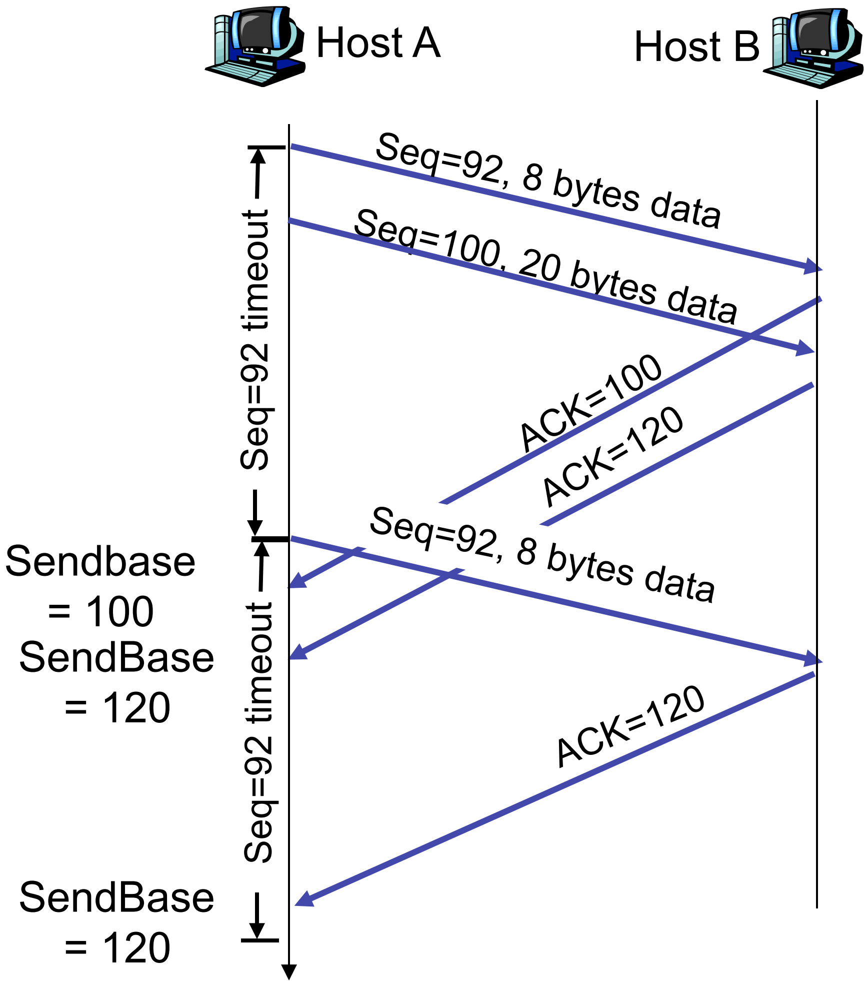
Seq=92, 8 bytes data

ACK=100

SendBase
= 100

time

lost ACK scenario



Host A

Host B

Seq=92, 8 bytes data

Seq=100, 20 bytes data

ACK=100

ACK=120

Seq=92 timeout

Seq=92, 8 bytes data

Sendbase
= 100

SendBase
= 120

Seq=92 timeout

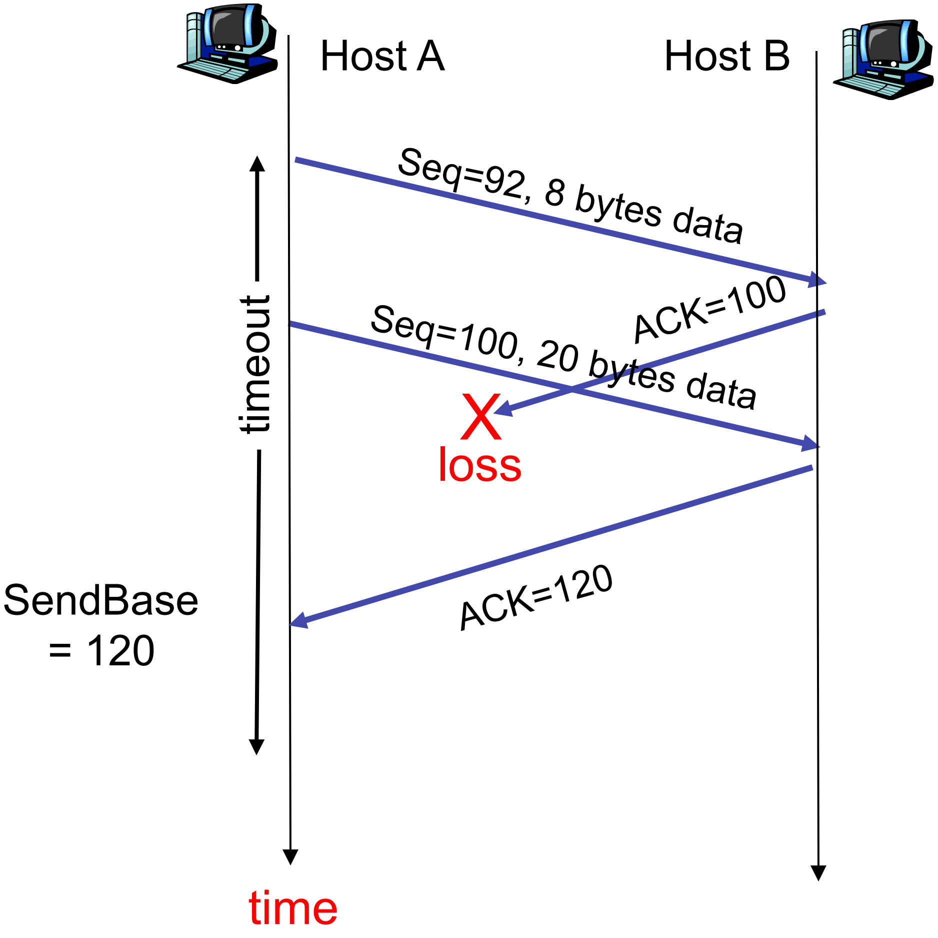
ACK=120

SendBase
= 120

time

premature timeout

TCP Retransmission Scenarios (more)



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

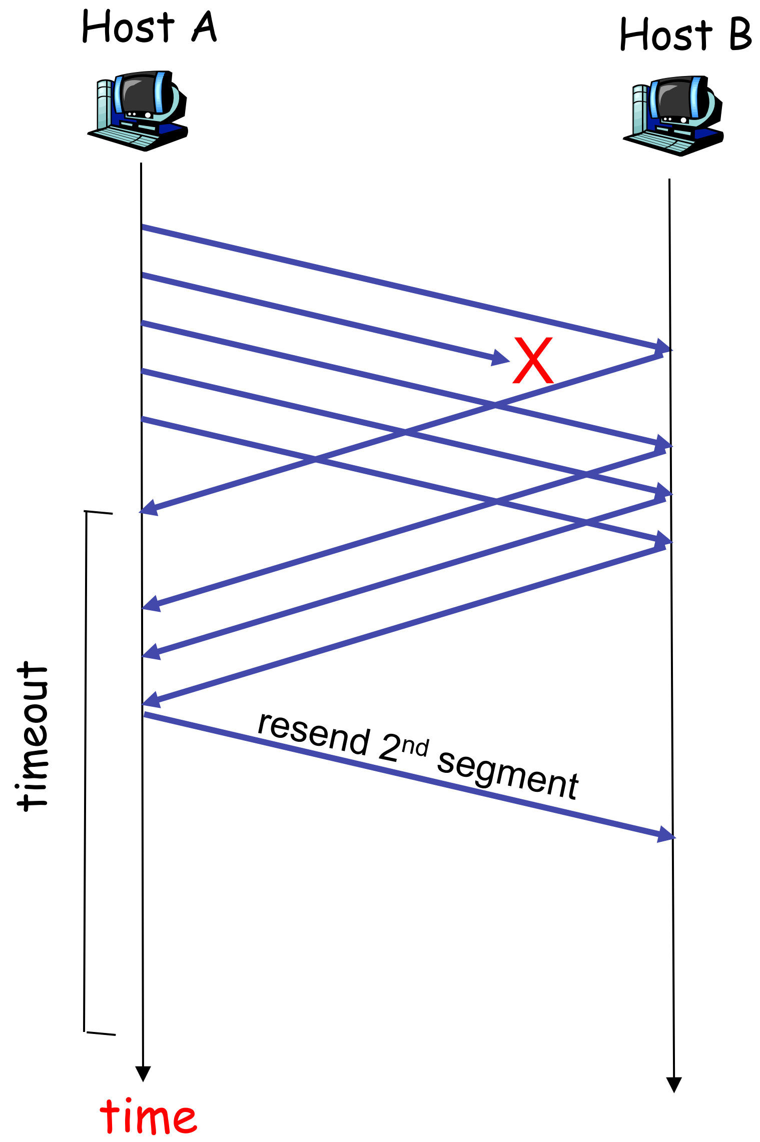
Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately 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 (which is lower end of gap)
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 likely will 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

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

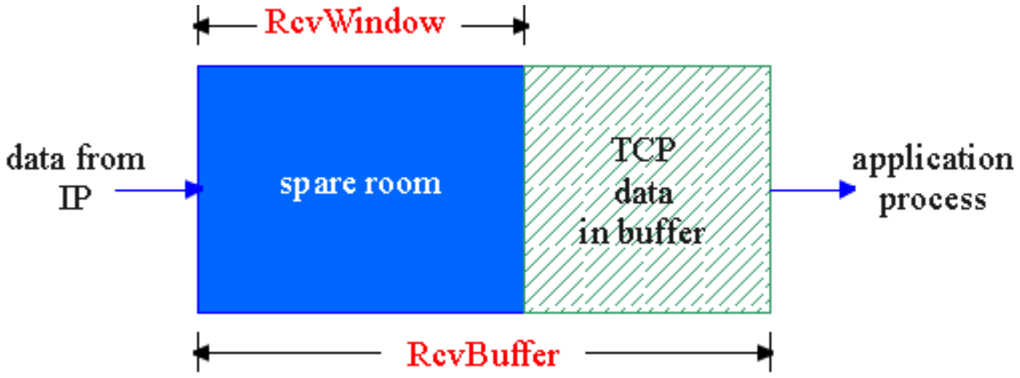
fast retransmit

Chapter Outline

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

TCP Flow Control

- Receive side of TCP connection has a receive buffer:

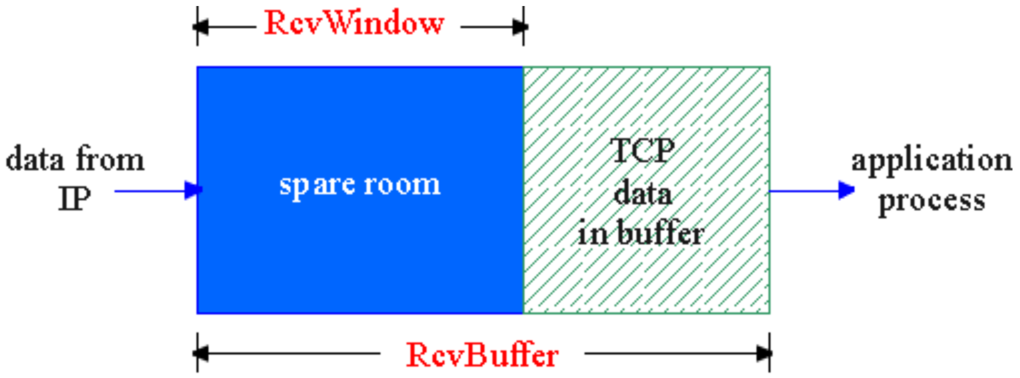


flow control

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

- Application process may be slow at reading from buffer
- Speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control



(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

Chapter Outline

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

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.)

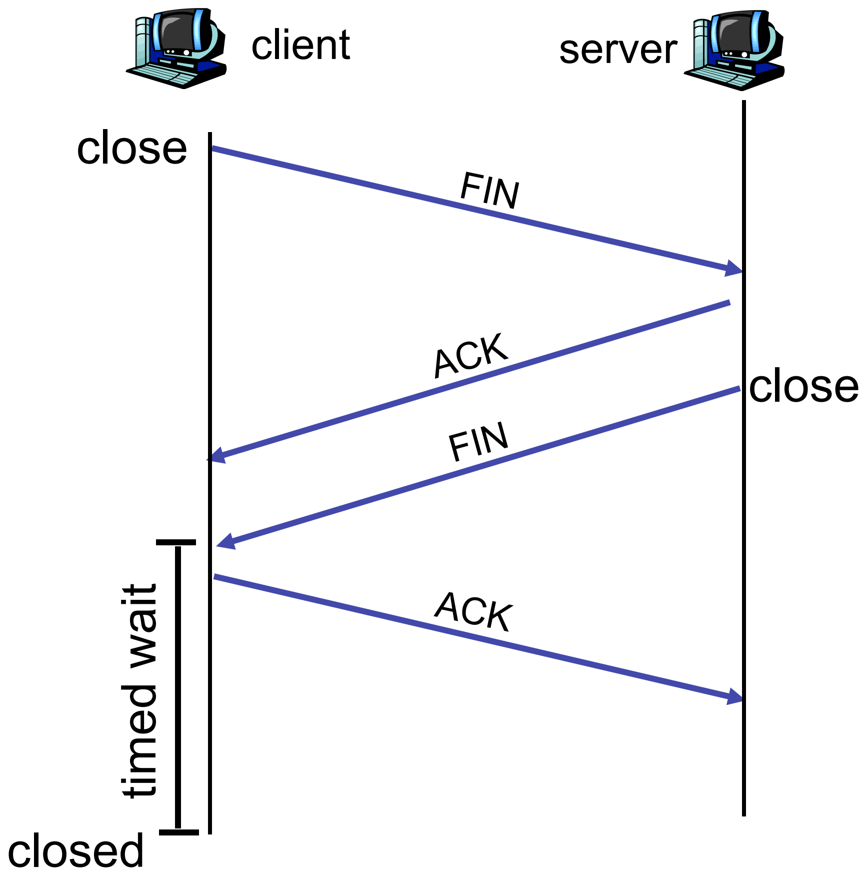
Closing a connection:

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.

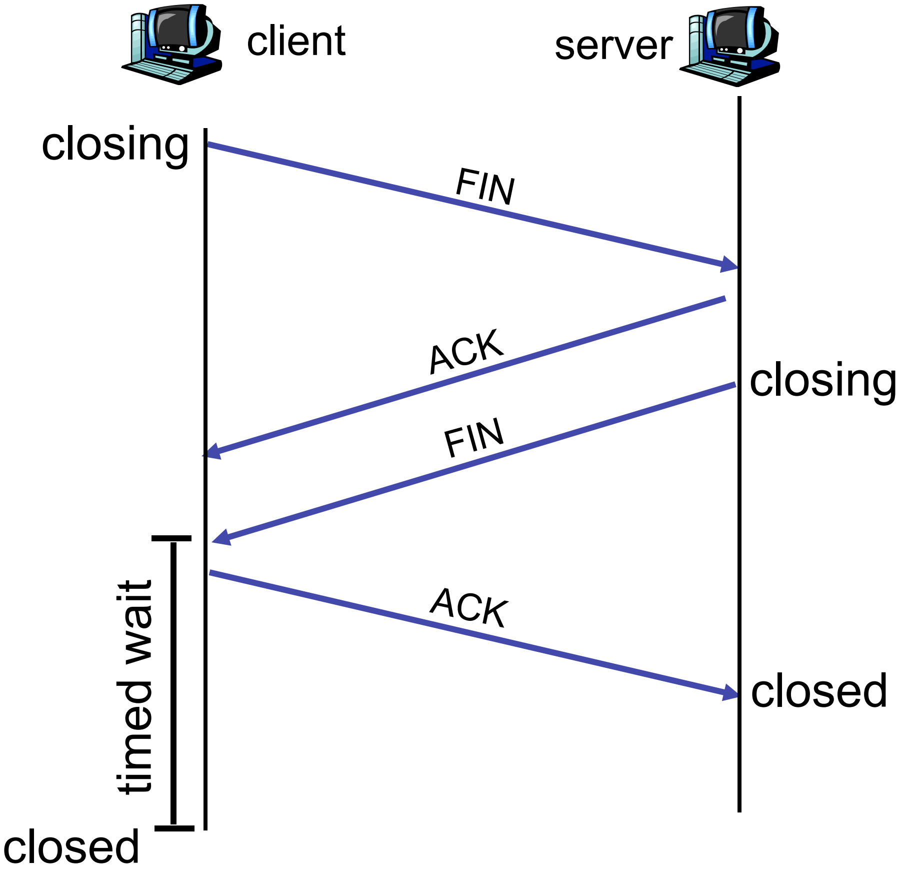


TCP Connection Management (cont.)

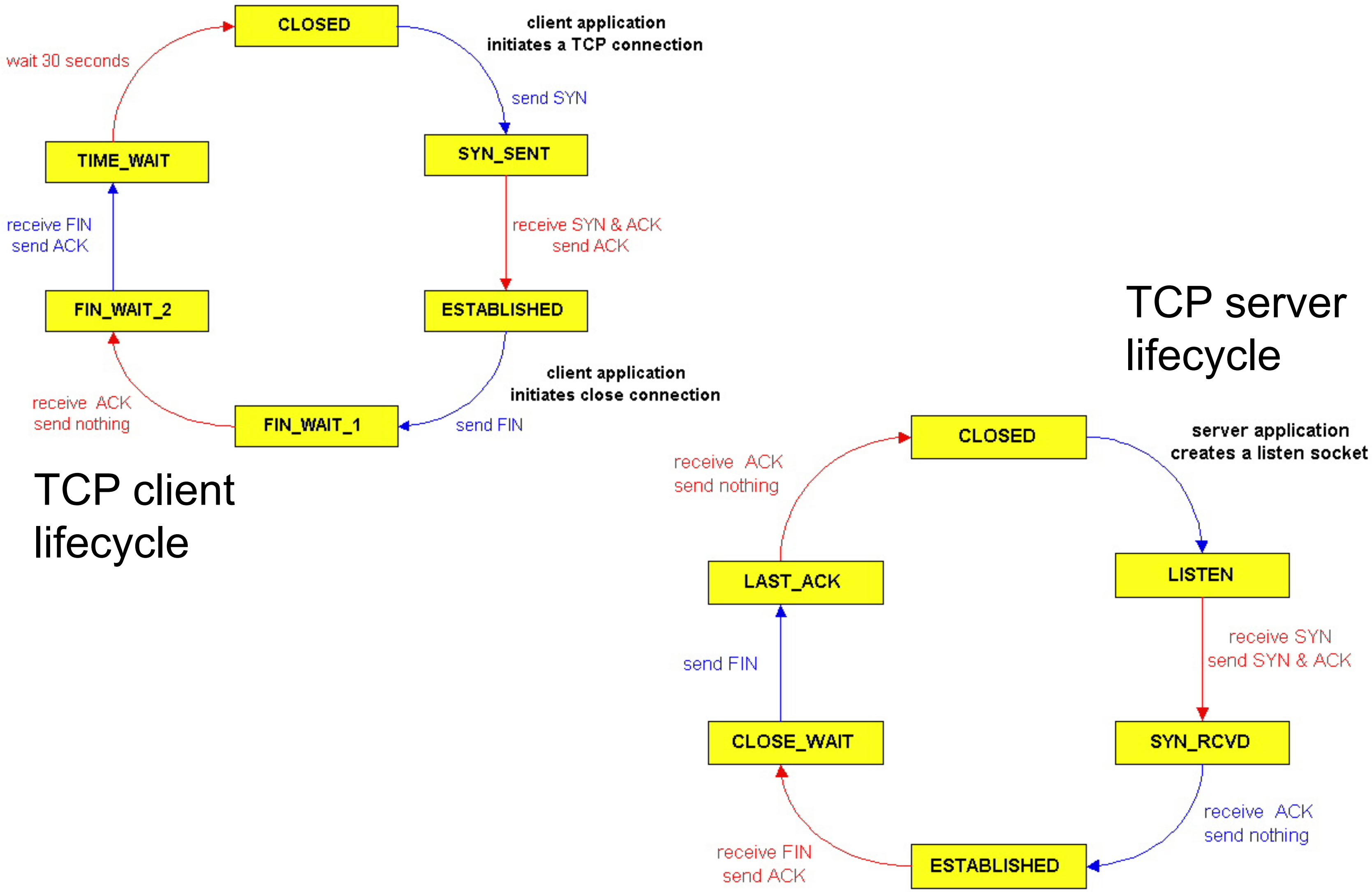
Step 3: client receives FIN, replies with ACK.

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

Step 4: server, receives ACK. Connection closed.



TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle

Chapter outline

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

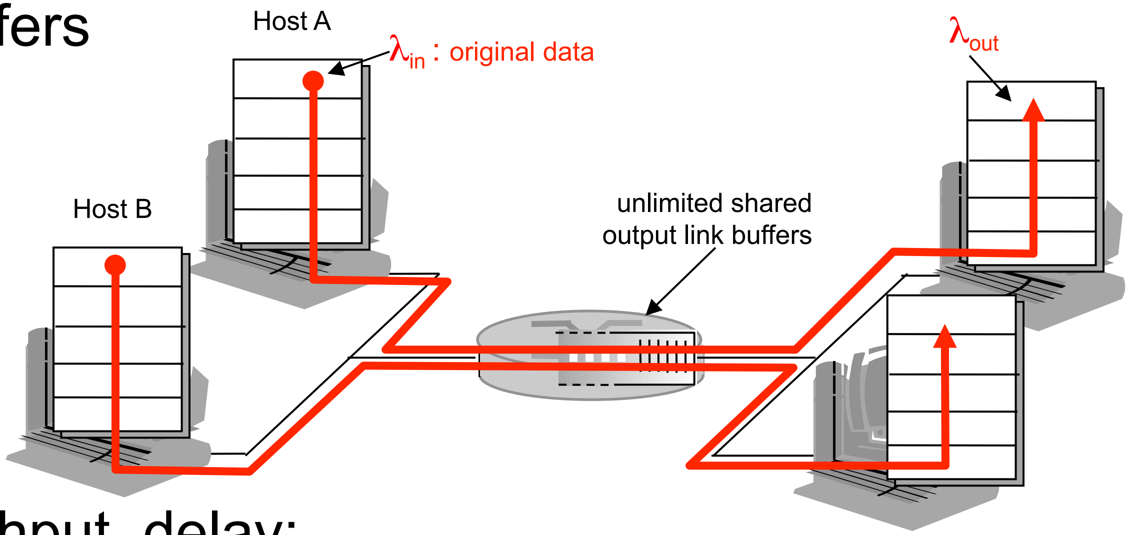
Principles of Congestion Control

Congestion:

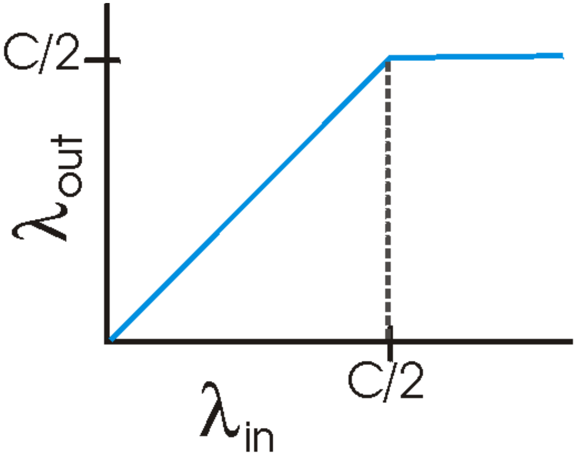
- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

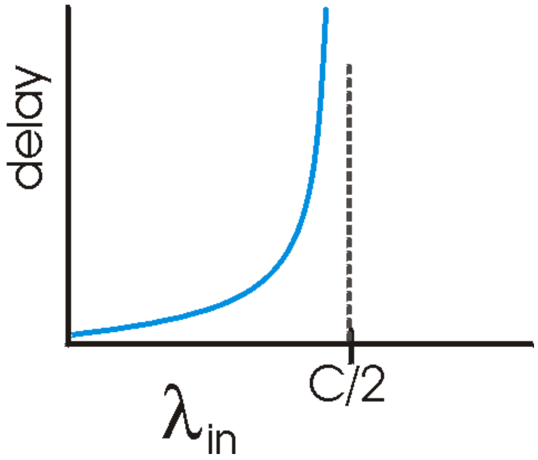
Causes/Costs of Congestion: Scenario 1

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


Host A
λ_{in} : original data
λ_{out}
Host B
unlimited shared output link buffers

- per-connection throughput, delay:

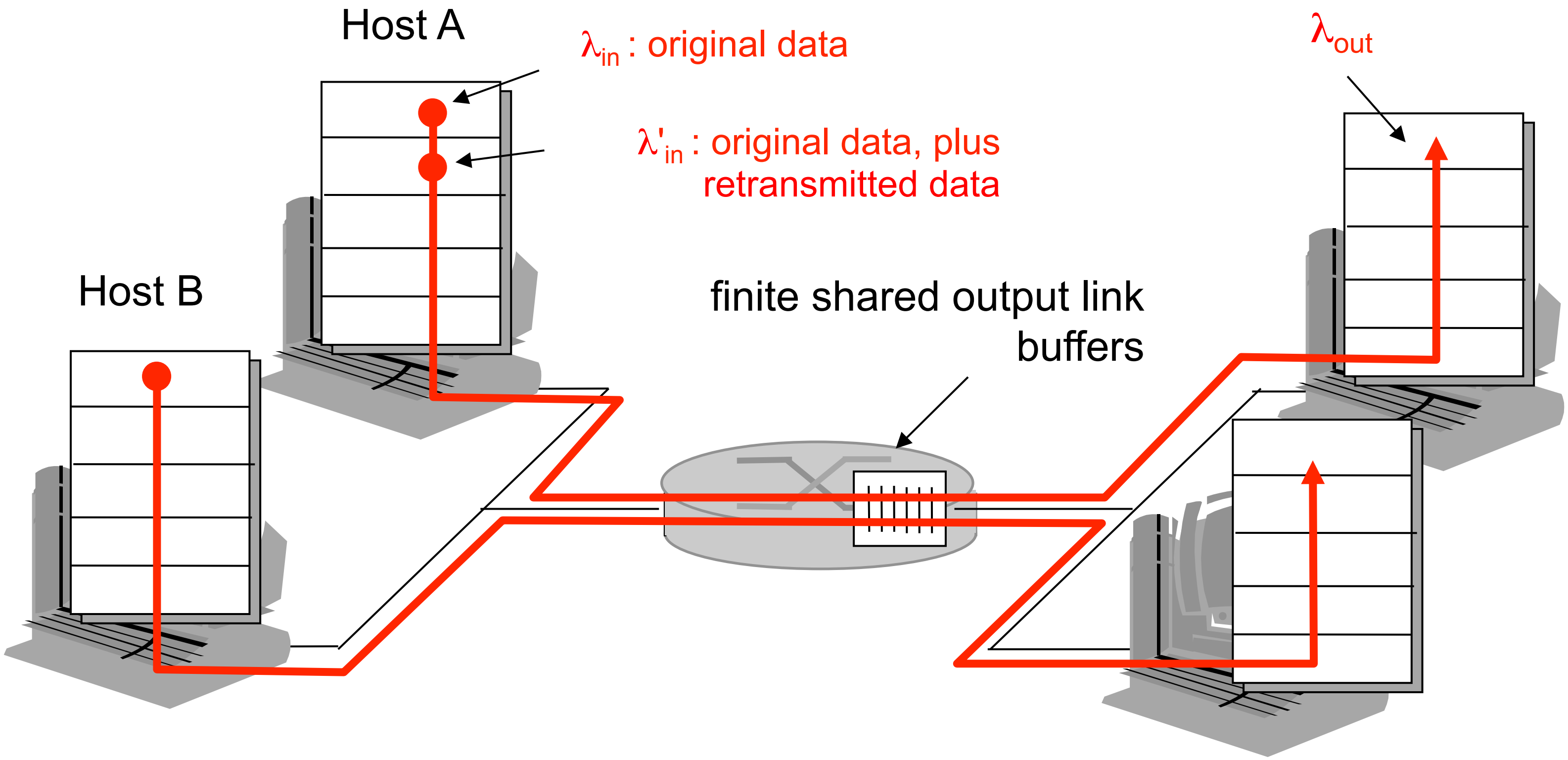




- large delays when congested
- maximum achievable throughput: C/2

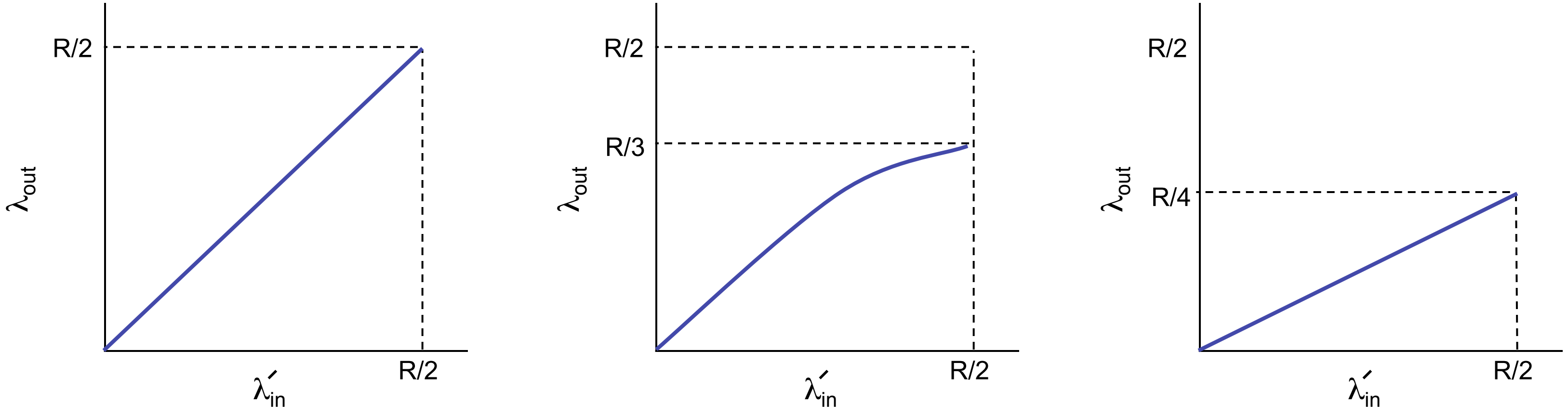
Causes/Costs of Congestion: Scenario 2

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


Host A
λ_{in} : original data
λ_{out}
λ'_{in} : original data, plus retransmitted data
Host B
finite shared output link buffers

Causes/Costs of Congestion: Scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a. b. c.

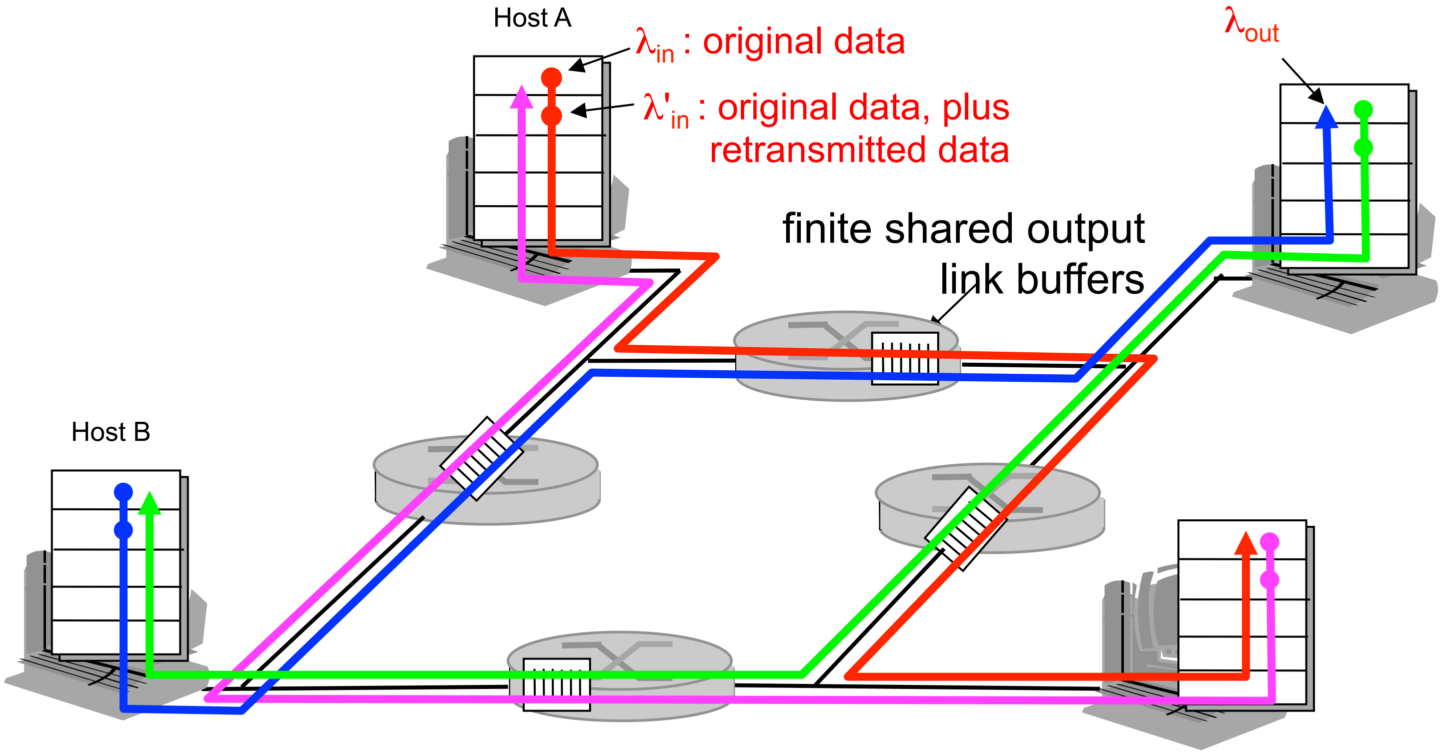
"costs" of congestion:

- more work (retransmissions) for given "goodput"
- unneeded retransmissions: link carries multiple copies of packet

Causes/Costs of Congestion: Scenario 3

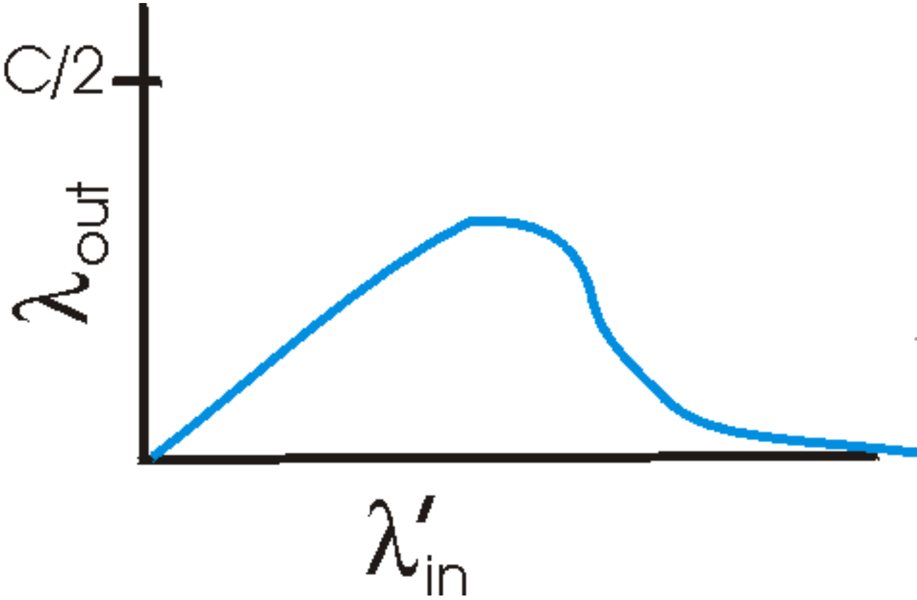
- four senders
- multihop paths
- timeout/retransmit

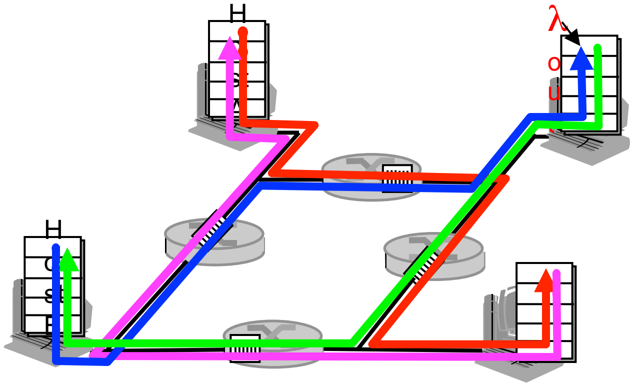
Q: what happens as λ_{in} and λ'_{in} increase ?



Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
λ_{out}
finite shared output link buffers
Host B

Causes/Costs of Congestion: Scenario 3



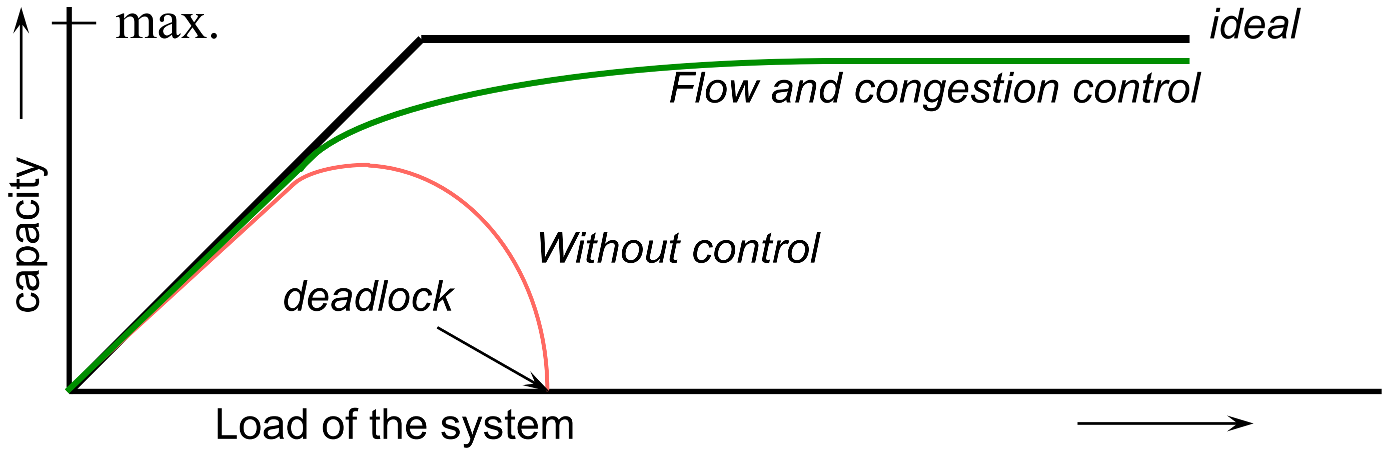


Another "cost" of congestion:

- ❑ when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Congestion Control

- Goals and problems hereby
  - Reasonable behavior in case of network (over)load
  - Without controlling the outgoing amount of data, the capacity may drop to zero because of deadlocks
  - Fair ressource sharing
  - Criteria: effective, simple, robust, end-host driven



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

Chapter Outline

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

Congestion Control (Van Jacobson)

- Problem: the end host does not know a lot about the network.
  - It only knows if a packet has been delivered successfully or not

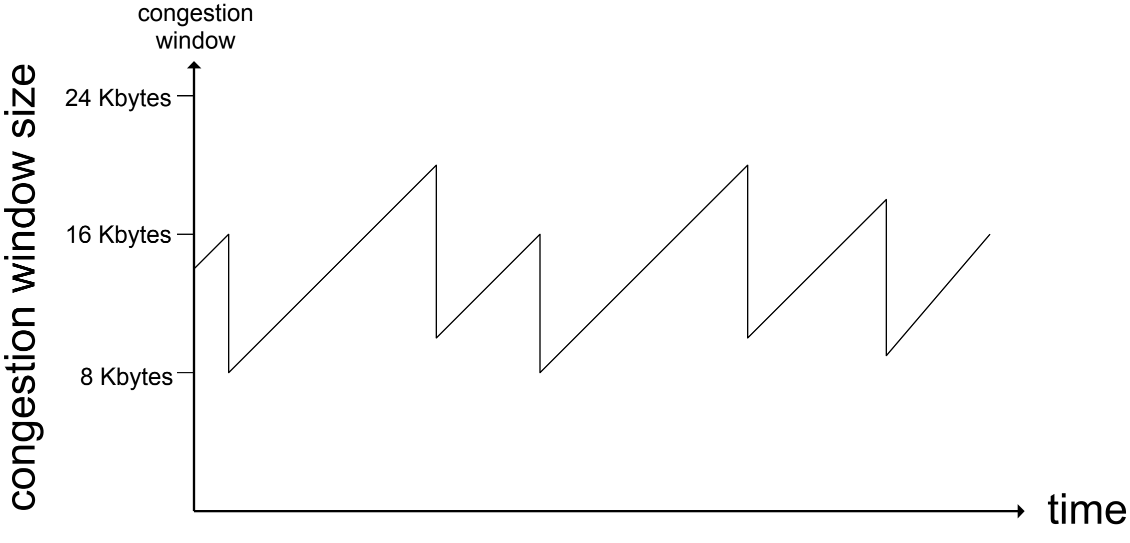
- Self clocking:
  - for every segment that leaves the network we can send a new one

- Assumption:
  - packet loss only because of congestion
  - Not true for wireless networks

TCP congestion control: additive increase, multiplicative decrease

- *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:

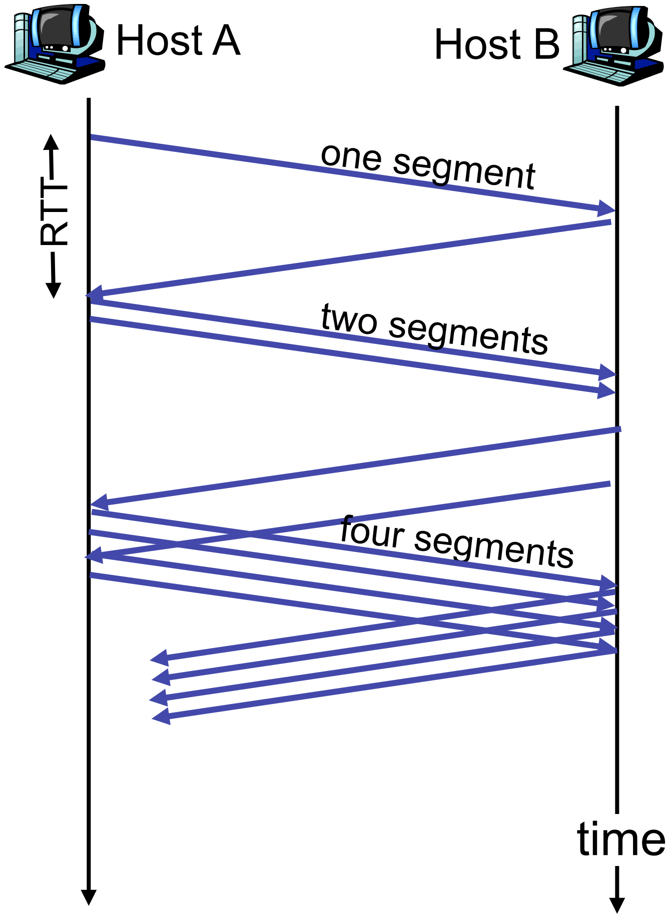
  - AIMD
  - slow start
  - conservative after timeout events

TCP Slow Start

- When connection begins, `CongWin` = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- 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
- Summary: initial rate is slow but ramps up exponentially fast



Refinement: Inferring Loss

- 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:

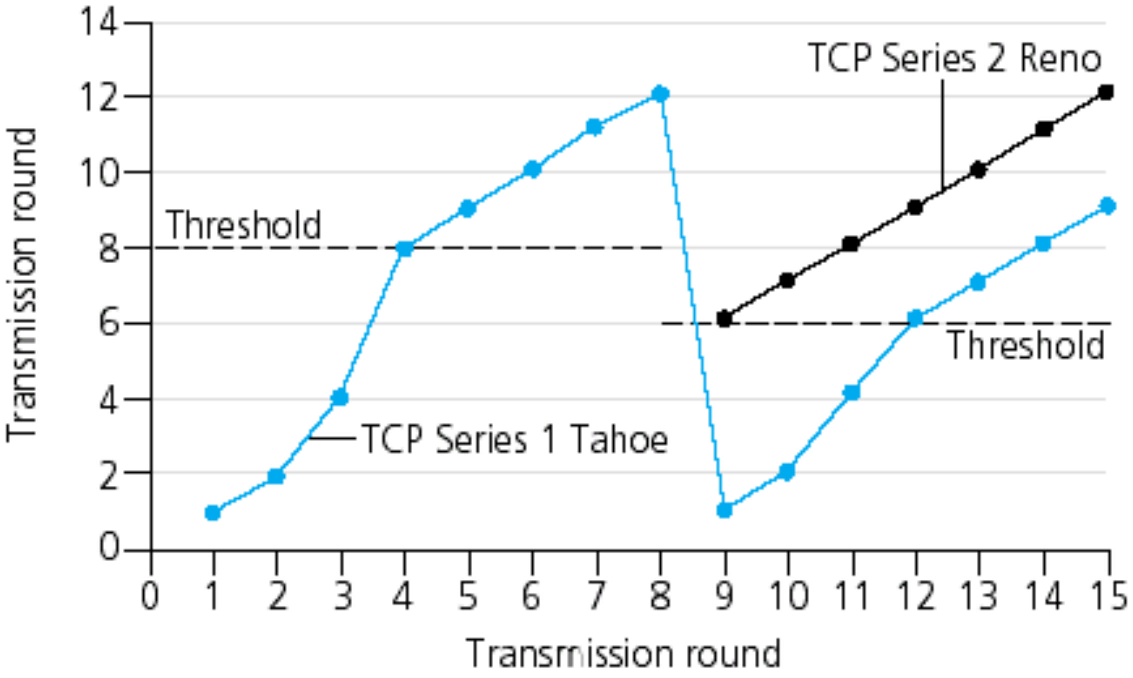
- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a “more alarming” congestion scenario

Refinement

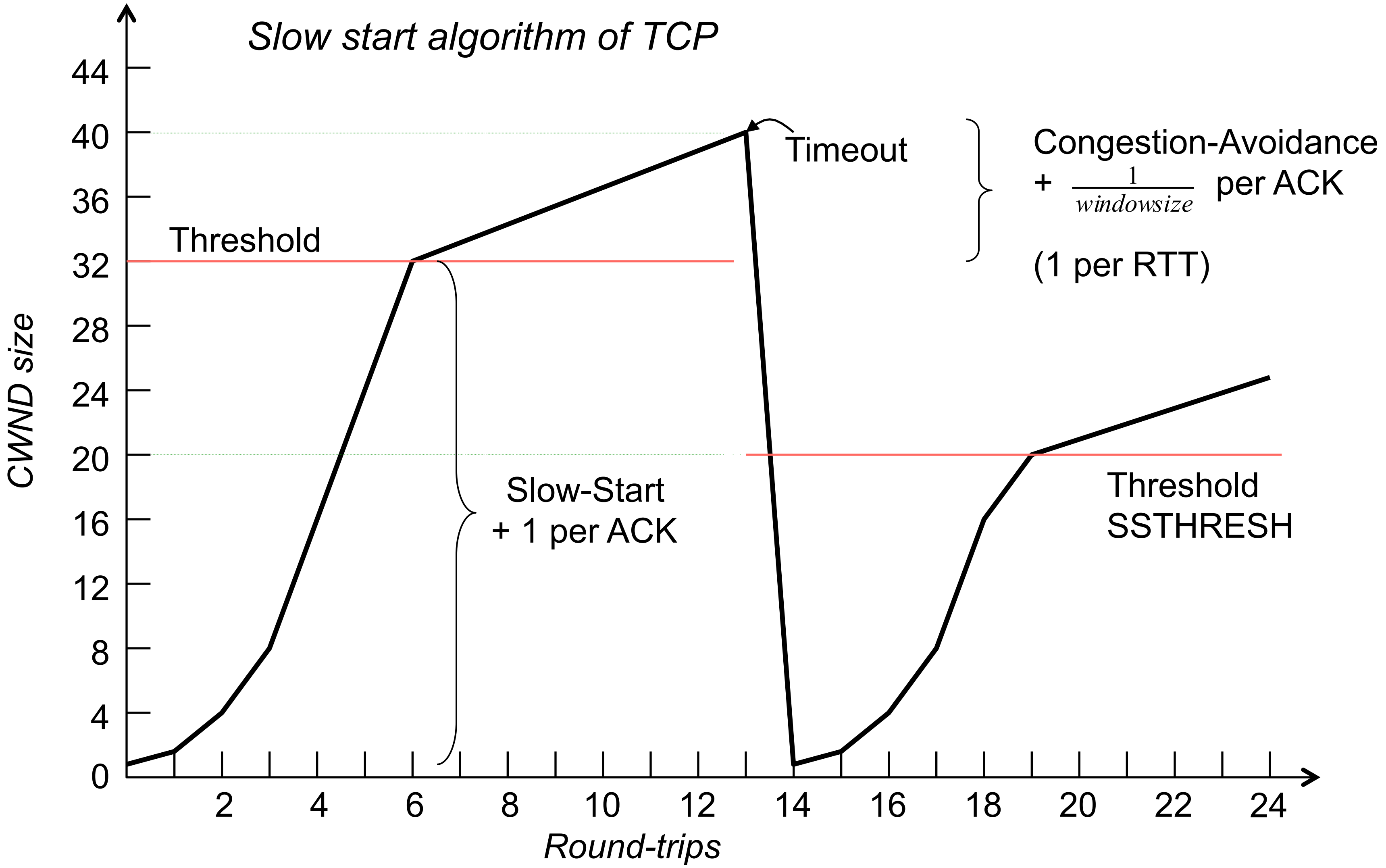
- Q: When should the exponential increase switch to linear?
- A: When CongWin gets to 1/2 of its value before timeout.

Implementation:

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

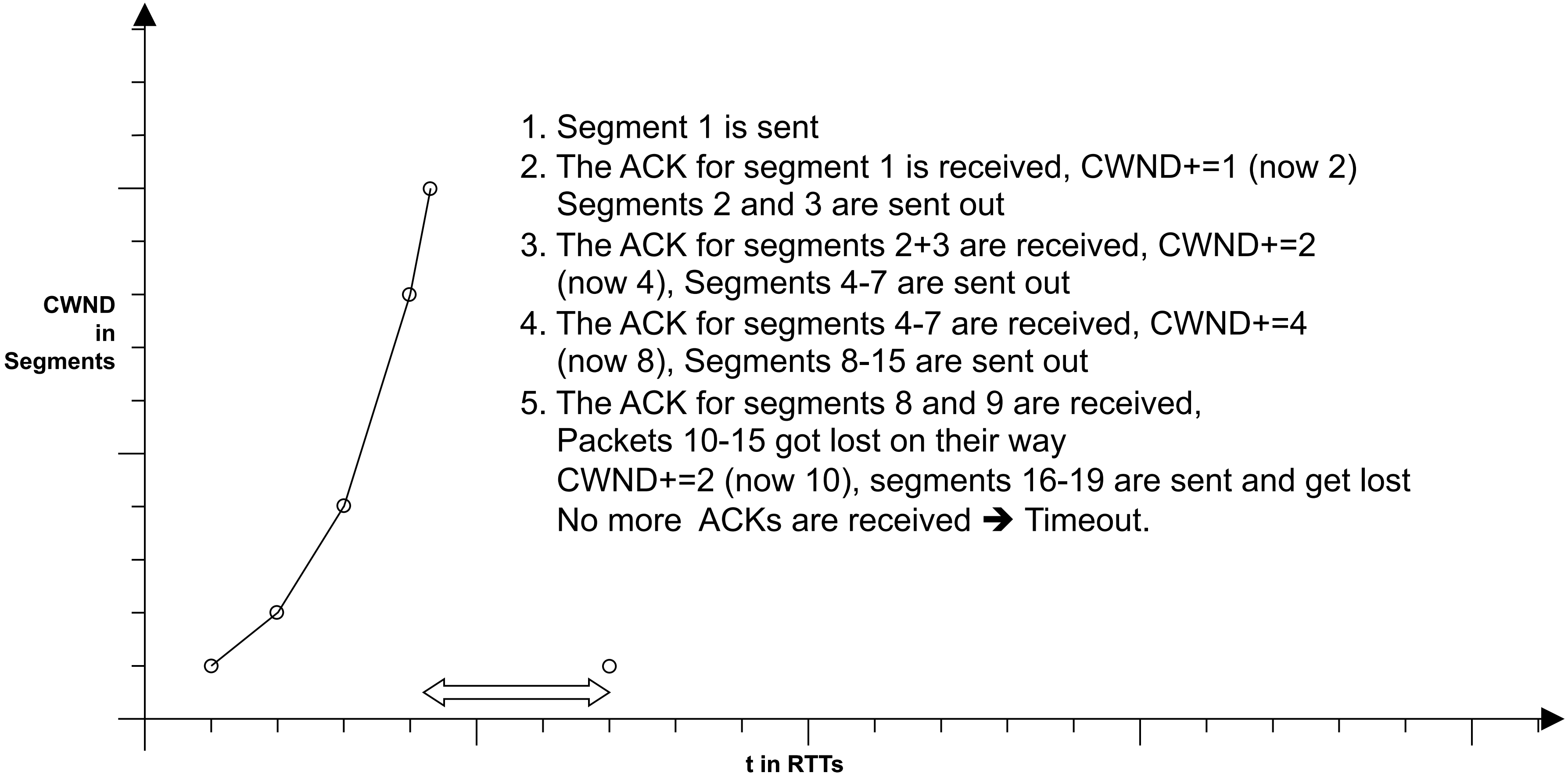


TCP Congestion Control Without Fast Retransmit



Here the window size is measured in number of packets
Real TCP uses bytes.

Slow Start

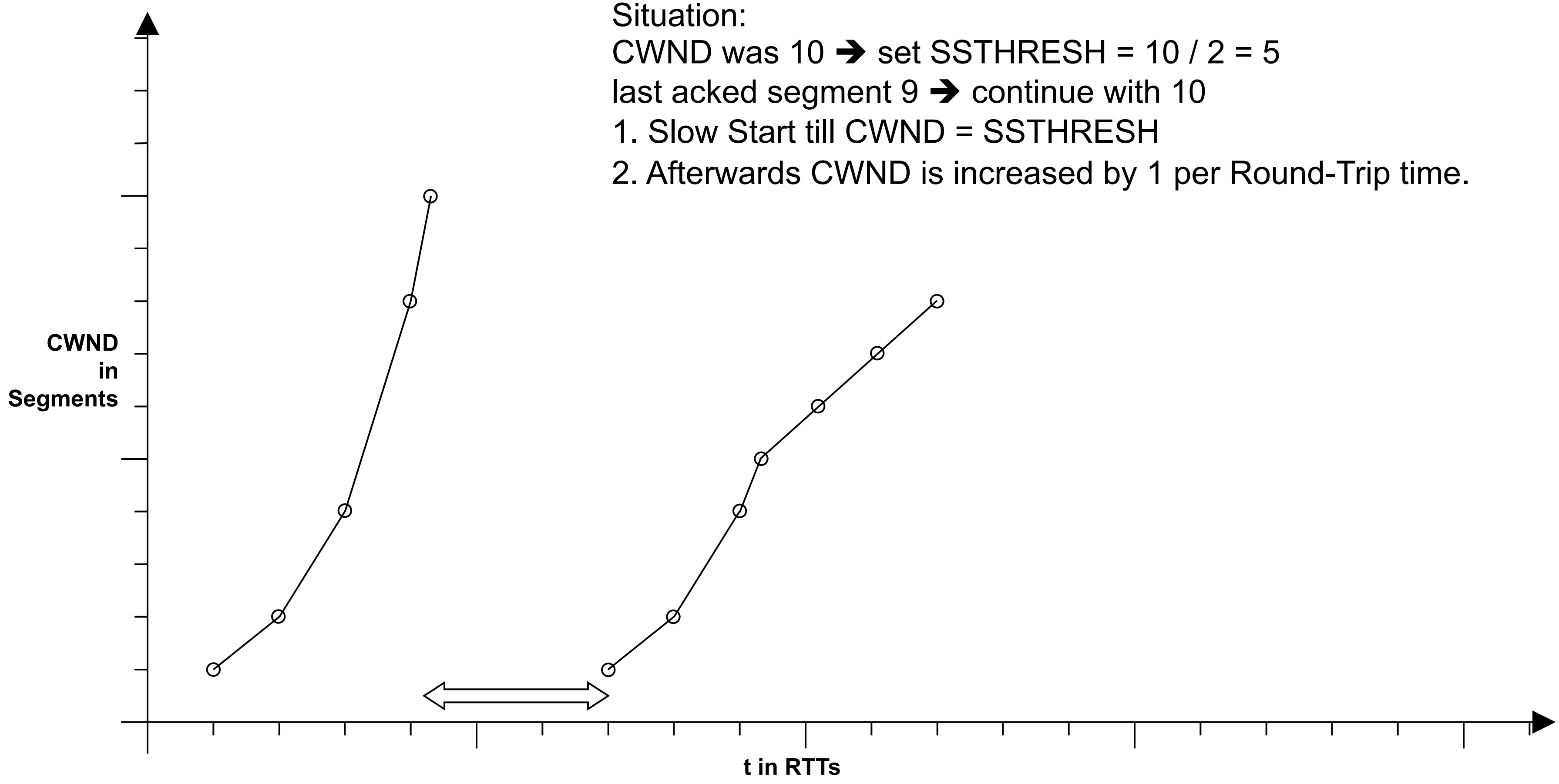


1. Segment 1 is sent
2. The ACK for segment 1 is received, CWND+=1 (now 2)
   Segments 2 and 3 are sent out
3. The ACK for segments 2+3 are received, CWND+=2
   (now 4), Segments 4-7 are sent out
4. The ACK for segments 4-7 are received, CWND+=4
   (now 8), Segments 8-15 are sent out
5. The ACK for segments 8 and 9 are received,
   Packets 10-15 got lost on their way
   CWND+=2 (now 10), segments 16-19 are sent and get lost
   No more ACKs are received ➔ Timeout.

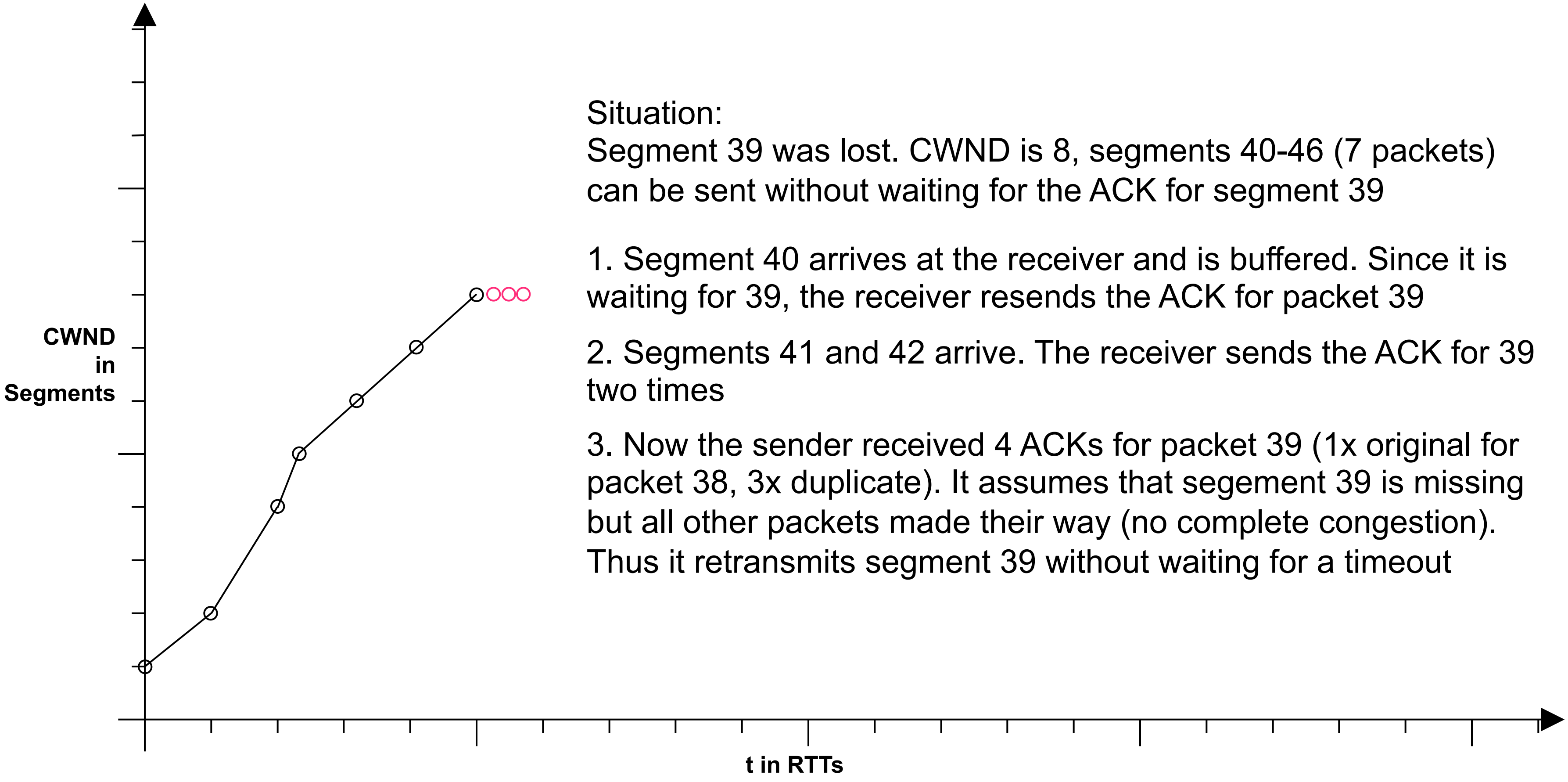
- Basic idea: packet loss indicates congestion
- Algorithm slowly approaches the limit

Congestion Avoidance

Situation:
CWND was 10 ➔ set SSTHRESH = 10 / 2 = 5
last acked segment 9 ➔ continue with 10
1. Slow Start till CWND = SSTHRESH
2. Afterwards CWND is increased by 1 per Round-Trip time.



Fast Retransmit



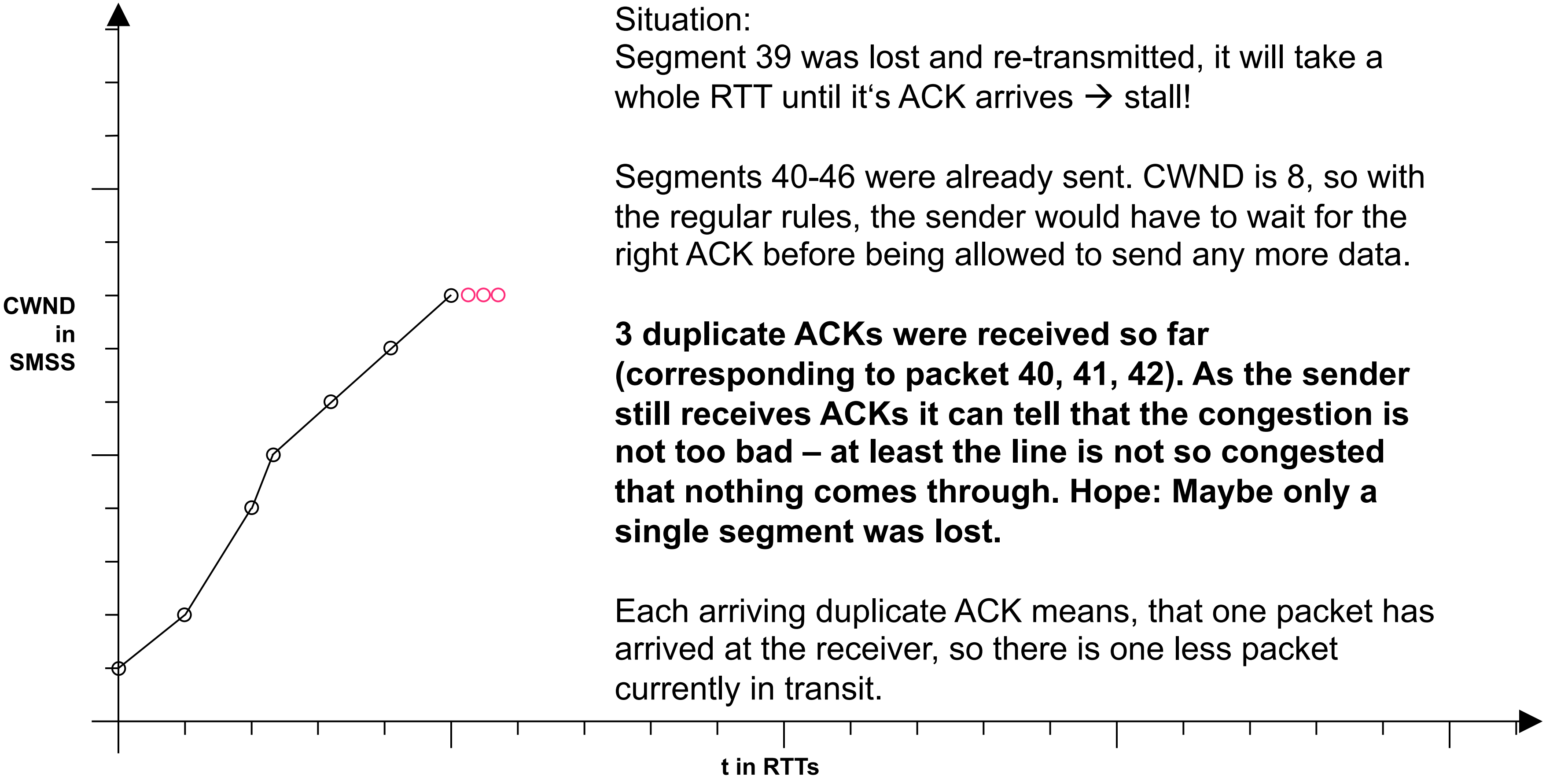
Situation:
Segment 39 was lost. CWND is 8, segments 40-46 (7 packets) can be sent without waiting for the ACK for segment 39

1. Segment 40 arrives at the receiver and is buffered. Since it is waiting for 39, the receiver resends the ACK for packet 39

2. Segments 41 and 42 arrive. The receiver sends the ACK for 39 two times

3. Now the sender received 4 ACKs for packet 39 (1x original for packet 38, 3x duplicate). It assumes that segement 39 is missing but all other packets made their way (no complete congestion). Thus it retransmits segment 39 without waiting for a timeout

Fast Recovery I



Situation:
Segment 39 was lost and re-transmitted, it will take a whole RTT until it‘s ACK arrives → stall!

Segments 40-46 were already sent. CWND is 8, so with the regular rules, the sender would have to wait for the right ACK before being allowed to send any more data.

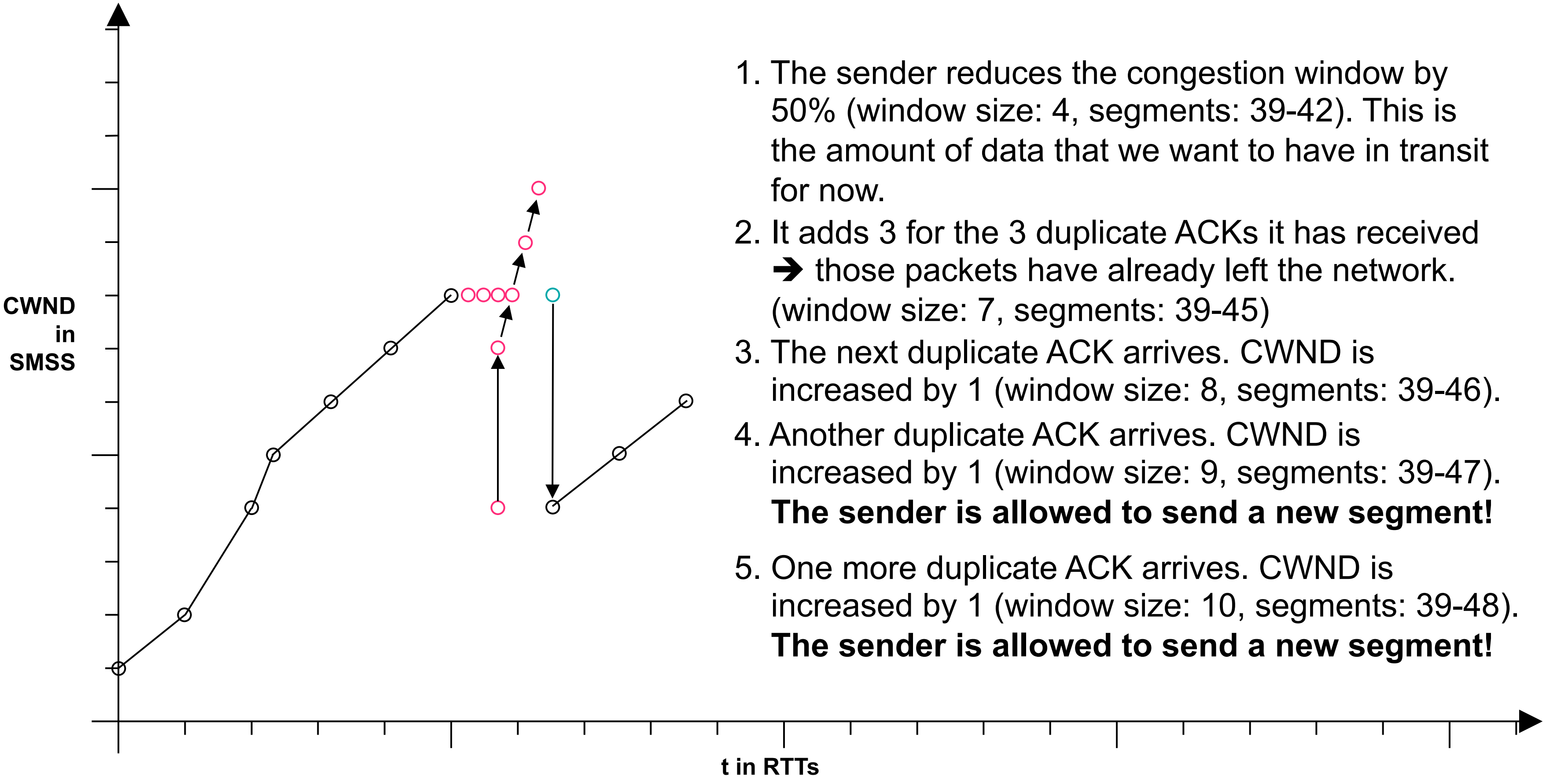
3 duplicate ACKs were received so far (corresponding to packet 40, 41, 42). As the sender still receives ACKs it can tell that the congestion is not too bad – at least the line is not so congested that nothing comes through. Hope: Maybe only a single segment was lost.

Each arriving duplicate ACK means, that one packet has arrived at the receiver, so there is one less packet currently in transit.

Idea of fast Recovery:

- Congestion is there but not too bad → Reduce the Window by 50%
- Then keep the number of segments in transit equal to the new window size even though there are no new ACKs (Compare Jacobson’s Self-Clocking).

Fast Recovery II



1. The sender reduces the congestion window by 50% (window size: 4, segments: 39-42). This is the amount of data that we want to have in transit for now.
2. It adds 3 for the 3 duplicate ACKs it has received ➔ those packets have already left the network. (window size: 7, segments: 39-45)
3. The next duplicate ACK arrives. CWND is increased by 1 (window size: 8, segments: 39-46).
4. Another duplicate ACK arrives. CWND is increased by 1 (window size: 9, segments: 39-47). **The sender is allowed to send a new segment!**
5. One more duplicate ACK arrives. CWND is increased by 1 (window size: 10, segments: 39-48). **The sender is allowed to send a new segment!**

7. The ACK for segment 39 arrives (actually it is a cummulative ACK for segment 46). The sender remembers the value of CWND before starting fast retransmit. CWND is set to half of the old value.
8. **The sender continues with Congestion Avoidance. (window size: 4, segments 47-50)**

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

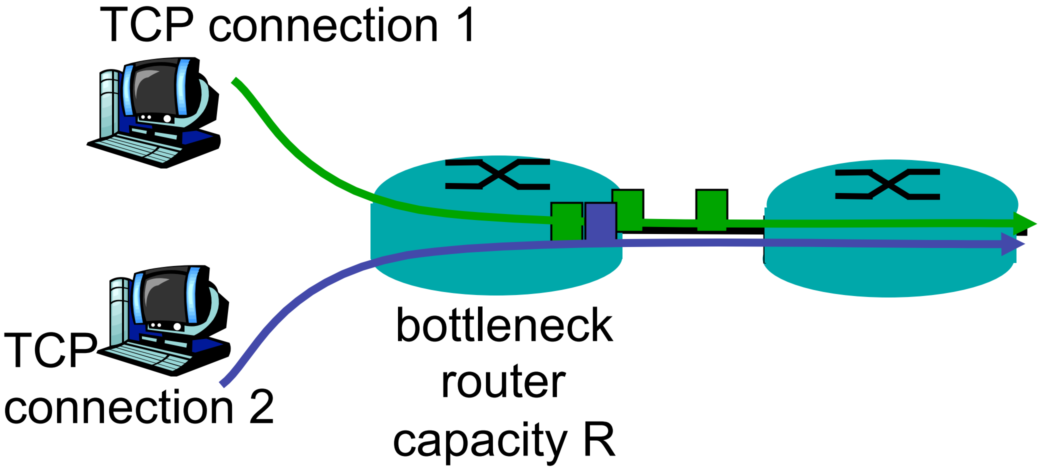
TCP throughput

- 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

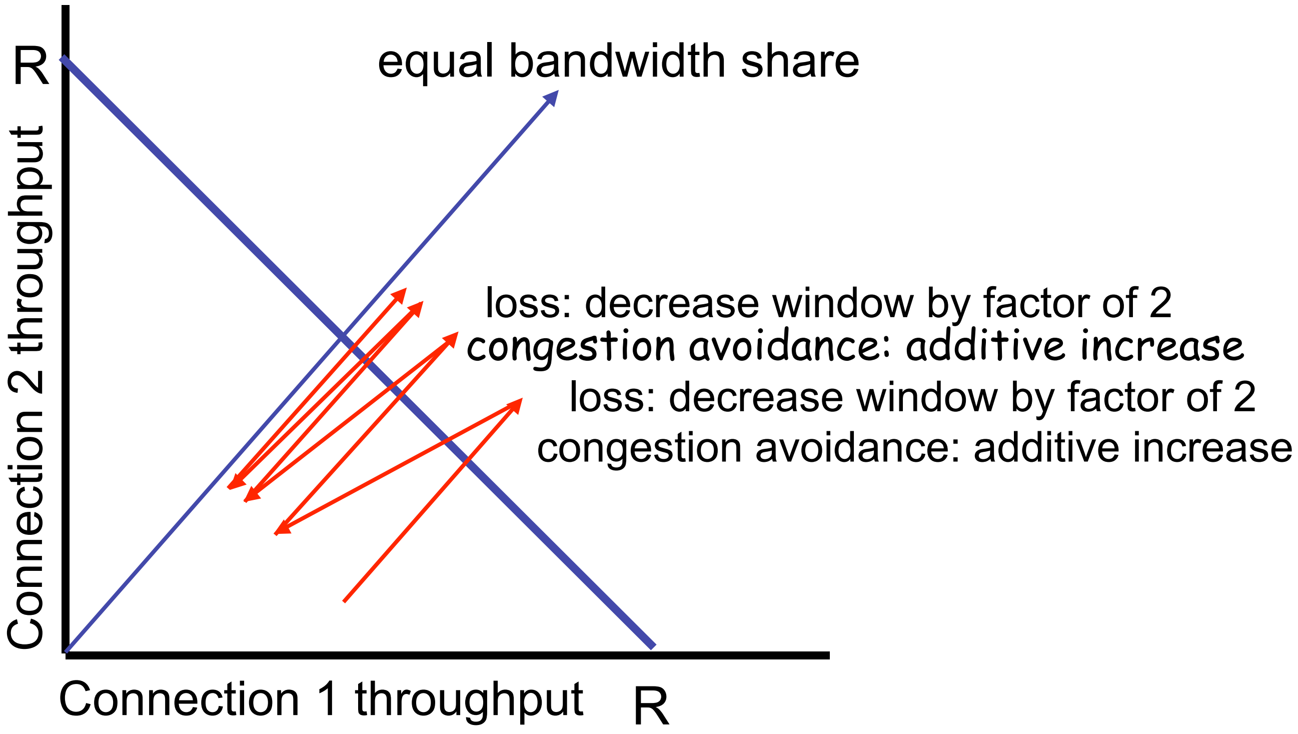
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

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


R
equal bandwidth share
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 2 throughput
Connection 1 throughput
R

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: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !

Advanced Topics

- TCP for high bandwidth long distance connections

- TCP Throughput Formula
- Overview of Deployment of TCP variants
- Detection of TCP-unfriendly Flows

TCP for High Bandwidth Long Distance Connections

- Several transport protocol variants for high bandwidth long distance connections (LFNs - Long Fat Networks) exist
- Frequent property
  - Effectively use available bandwidth
  - Unfriendly – “doesn’t play nicely with others”
  - Unfair to different RTT flows
  - achieves better performance than standard TCP
  - is not fair to standard TCP
- General approaches for congestion control
  - loss-based: NewReno, CUBIC
  - delay-based: Vegas, CAIA Delay Gradiant (CDG)

TCP Reno

- TCP Fast Recovery algorithm described in RFC 2581
- Implementation introduced 1990 in BSD Reno release
- Behaviour
  - sender only retransmits a packet
    - after a retransmit timeout has occurred
    - or after three duplicate acknowledgements have arrived

    triggering the Fast Retransmit algorithm.
  - a single retransmit timeout might result in the retransmission of several data packets
  - each invocation of the Fast Retransmit algorithm leads to retransmission of only a single data packet
  - problems may arrive when multiple packets are dropped from a single window

TCP NewReno

- c.f. RFC 3782 - April 2004, Proposed Standard
- „careful“ variant of Experimental RFC 2582 NewReno as default
- Properties
  - addresses problems that may arrive when multiple packets are dropped from a single window
  - with multiple packet drops, acknowledgement for retransmitted packet acks some but not all packets transmitted before the Fast Retransmit
    ⇨ „partial acknowledgment“

TCP Vegas

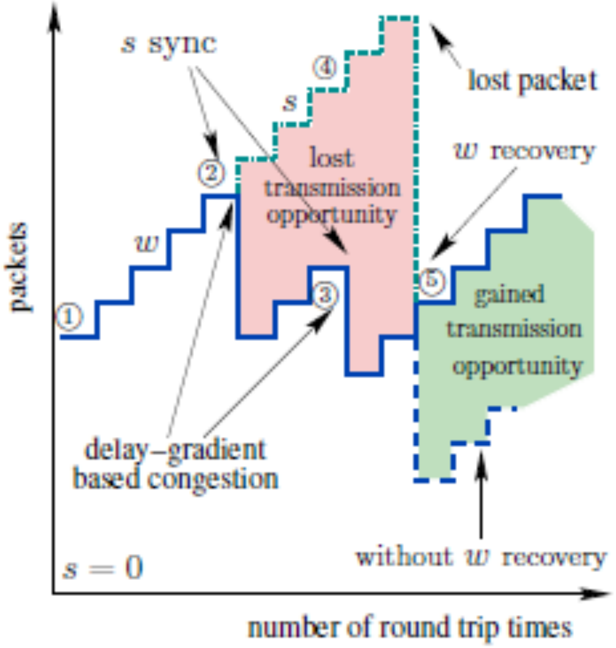
- TCP Vegas
  - by Lawrence Brakmo, Sean W. O'Malley, Larry L. Peterson at University of Arizona
  - published at SIGCOMM 1994
- Properties
  - delay-based congestion control
  - uses i^{th} RTT > min RTT + delay threshold, delay measured every RTT
  - Additive Increase Additive Decrease (AIAD) to adjust cwnd
- Properties
  - implementations available for Linux and BSD

TCP CUBIC

- CUBIC
  - Loss-based congestion control
    optimised for high bandwidth, high latency
- Properties
  - modified window-growth-control algorithm
  - window grows slowly around W_{max}
  - fast “probing” growth away from W_{max}
  - Standard TCP outperforms CUBIC’s window growth function in short RTT networks.
  - CUBIC emulates standard (time-independent) TCP window adjustment algorithm, select the greater of the two windows (emulated versus cubic)
- Implemenation:
  - in Linux since kernel 2.6.19, in FreeBSD 8-STABLE

Delay Gradient TCP

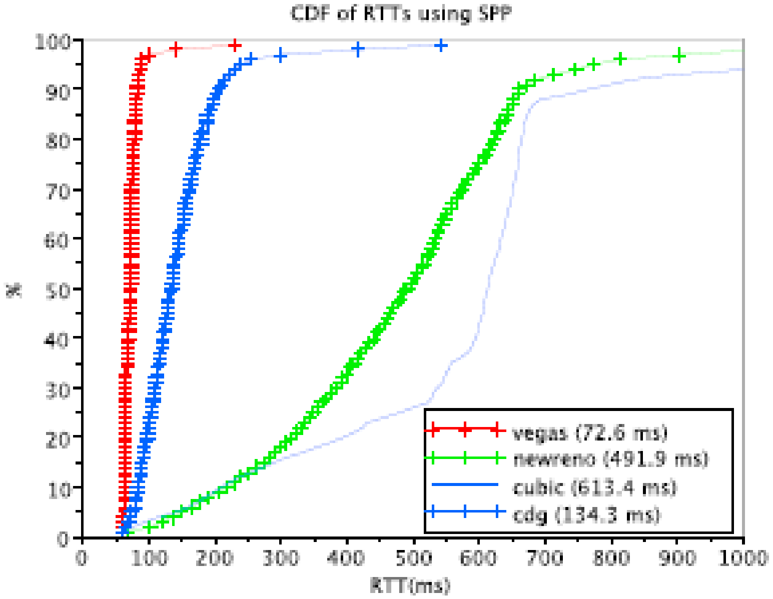
- D. Hayes, G. Armitage, "Revisiting TCP Congestion Control using Delay Gradients," IFIP/TC6 NETWORKING 2011, Valencia, Spain, 9-13 May 2011 http://caia.swin.edu.au/cv/dahayes/content/networking2011-cdg-preprint.pdf
- CDG ("CAIA Delay-Gradient") modified TCP sender behaviour:
  - uses delay gradient as a congestion indicator
  - has average probability of back off independent of RTT
  - works with loss-based congestion control flows, eg NewReno
  - tolerates non-congestion packet loss, and backoff for congestion related packet loss

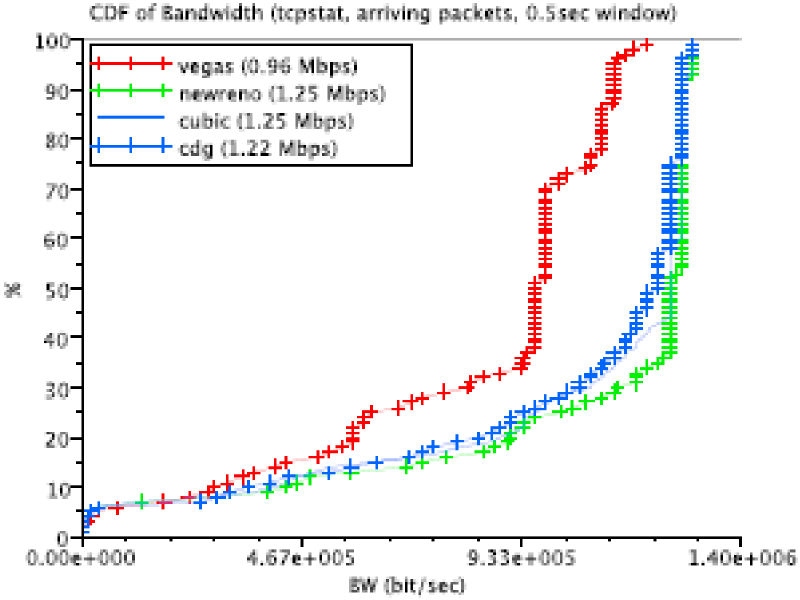


s sync
④
s
lost packet
②
lost transmission opportunity
w recovery
w
③
⑤
①
gained transmission opportunity
packets
delay–gradient based congestion
s = 0
without w recovery
number of round trip times

Comparison of TCP Variants

- Grenville Armitage: A rough comparison of NewReno, CUBIC, Vegas and ‘CAIA Delay Gradient’ TCP (v0.1), CAIA Technical report 110729A, 29 July 2011 http://caia.swin.edu.au/reports/110729A/CAIA-TR-110729A.pdf


CDF of RTTs using SPP
RTT(ms)
vegas (72.6 ms)
newreno (491.9 ms)
cubic (613.4 ms)
cdg (134.3 ms)


CDF of Bandwidth (tcpstat, arriving packets, 0.5sec window)
vegas (0.96 Mbps)
newreno (1.25 Mbps)
cubic (1.25 Mbps)
cdg (1.22 Mbps)
0.00e+000
4.67e+005
9.33e+005
1.40e+006
BW (bit/sec)