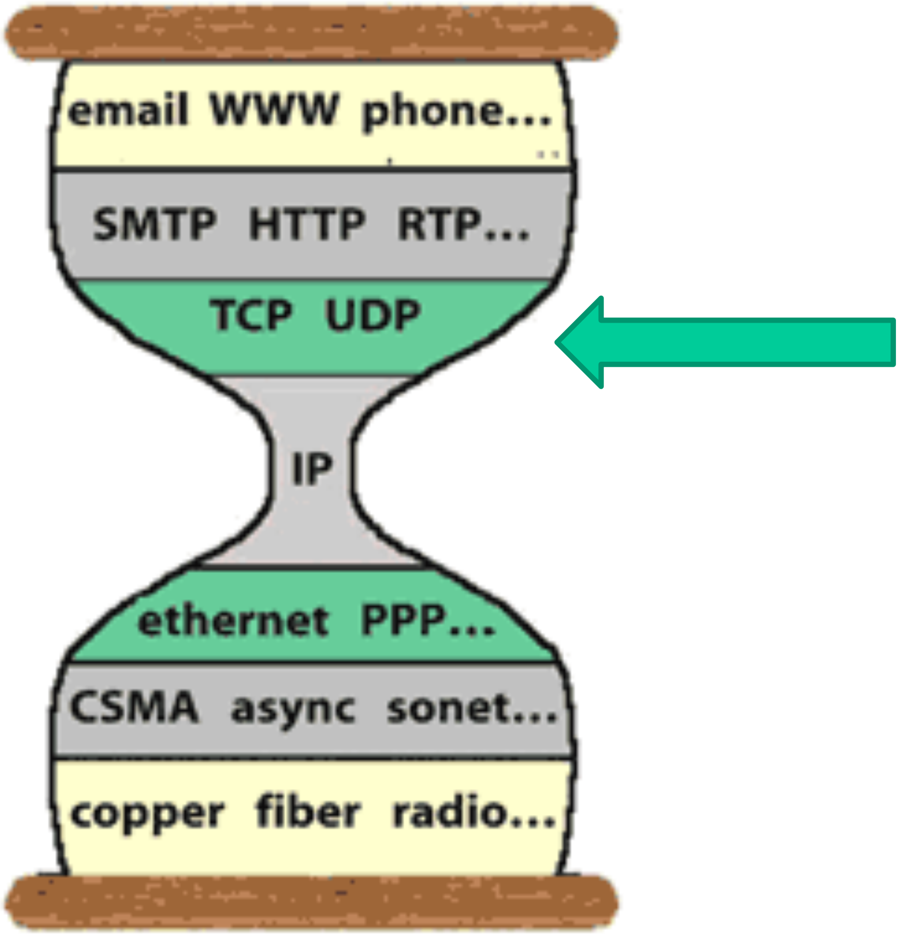
Continue with connection-oriented transport: TCP



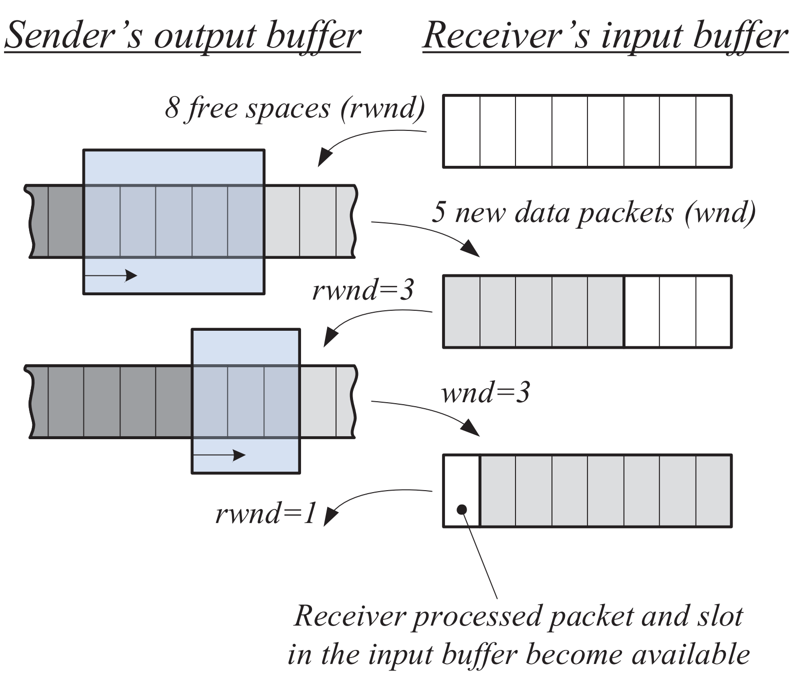
- Flow control
- Timeouts
- Congestion control

TCP Flow Control

Flow control: Prevent sender from overrunning receiver by transmitting too much too fast

receiver: informs sender of (dynamically changeable) amount of free buffer space (**`RcvWindow`** field in TCP header)

sender: keeps the amount of transmitted, unACKed data no more than most recently received **`RcvWindow`** value



How to Set TCP Retransmission Timer

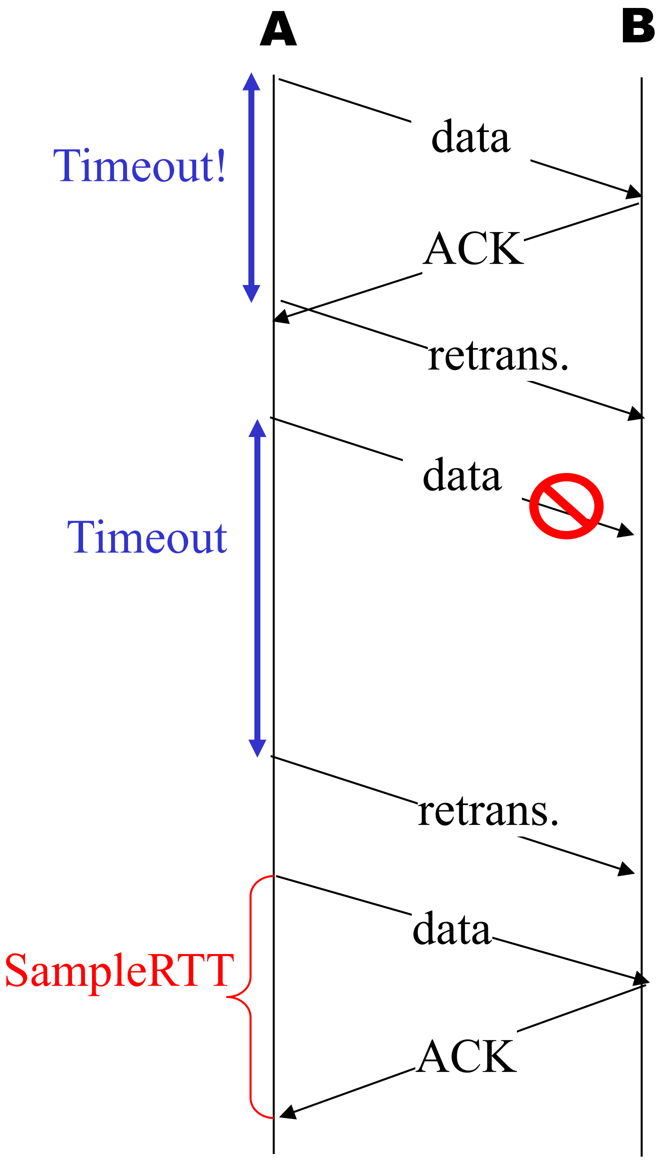
- TCP sets rxt timer based on measured RTT

SRTT: EstimatedRTT

SRTT= (1-α) x SRTT + α x SampleRTT

- Setting retransmission timer:
  - SRTT plus “safety margin”

Timer= SRTT + 4 X DevRTT



After obtain a new RTT sample:

important

- difference = SampleRTT - SRTT
- SRTT' = (1-α) x SRTT + α x SampleRTT

  = SRTT + α x difference
- DevRTT' = (1-β) x DevRTT + β x |difference|

  = DevRTT + β (|difference| - DevRTT)

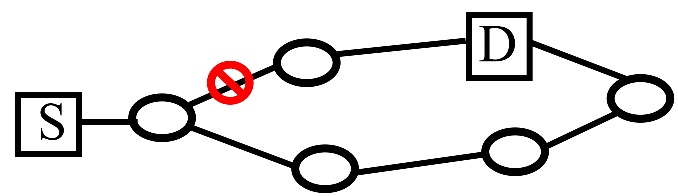
- Retransmission Timer (RTO) = SRTT + 4 x DevRTT

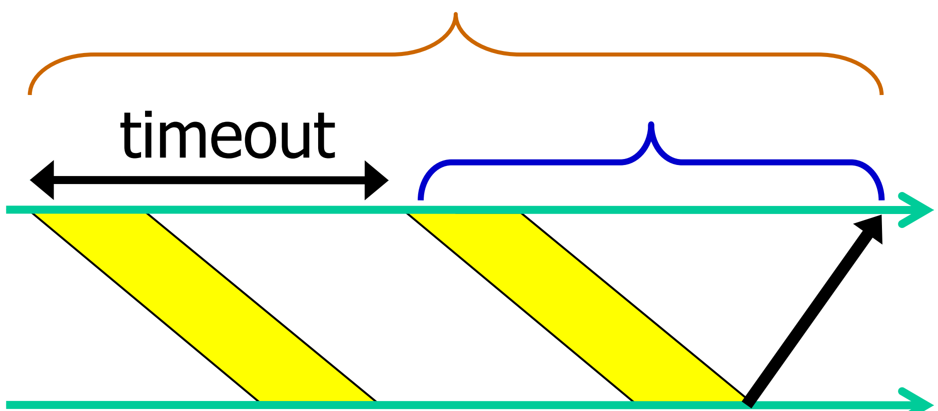
Typically: α = 1/8, β = 1/4

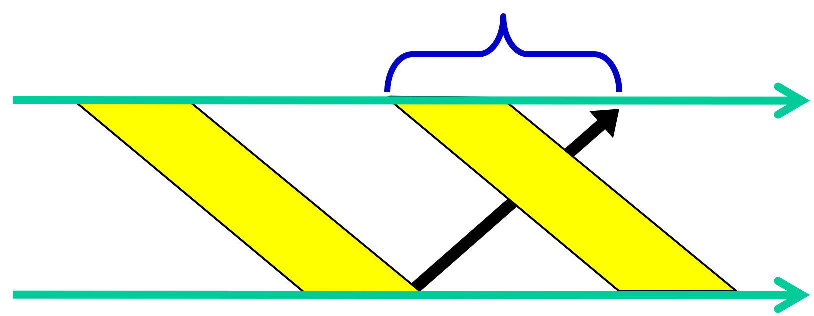
How to measure RTT in cases of *retransmissions*?

Options

- take the delay between first transmission and final ACK?
- take the delay between last retransmission of segment(n) and ACK(n)?
- Don't measure?







Karn's algorithm

in case of retransmission

- do not take the RTT sample (i.e. do not update SRTT or DevRTT)
- double the retransmission timer value (RTO) after each timeout
- Take RTT measure again upon next data transmission (that did not get retransmitted)

One more question

What initial **SRTT**, **DevRTT** values to start with?

- Set the default values by some engineered guessing
- what if the guessed value too small?
  - Unnecessary retransmissions
- what if the guessed value too large?
  - In case of first or first few packets being lost, wait longer than necessary before retransmission
- Current practice
  - **initial SRTT value: 3 sec, DevRTT 3 sec**
  - **Once get first sample RTT, SRTT←sample RTT, DevRTT=SRTT/2**

An example

difference = SampleRTT – SRTT

SRTT = SRTT + 1/8 x difference

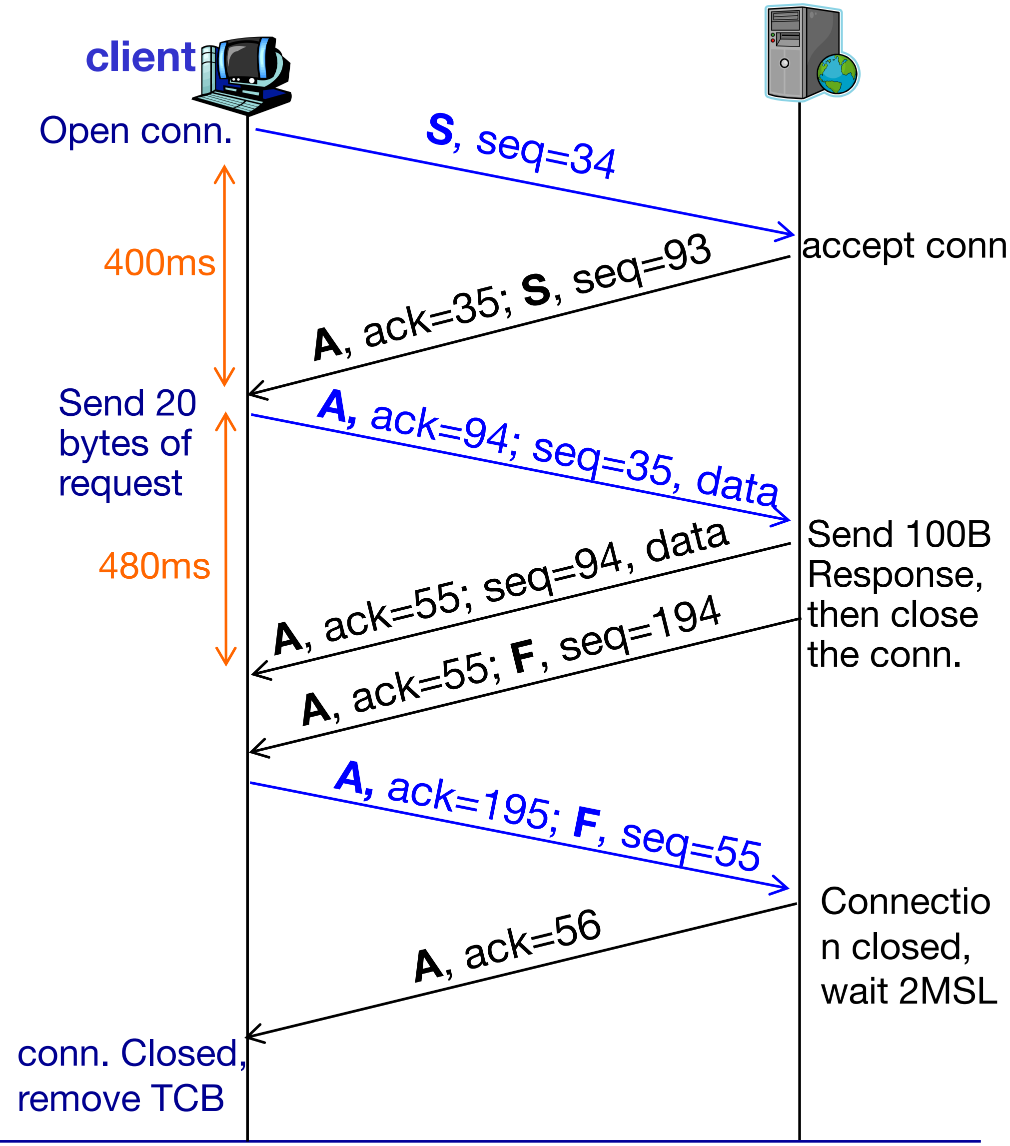
DevRTT = DevRTT + 1/4 (|difference| - DevRTT)

RTO = SRTT + 4 x DevRTT

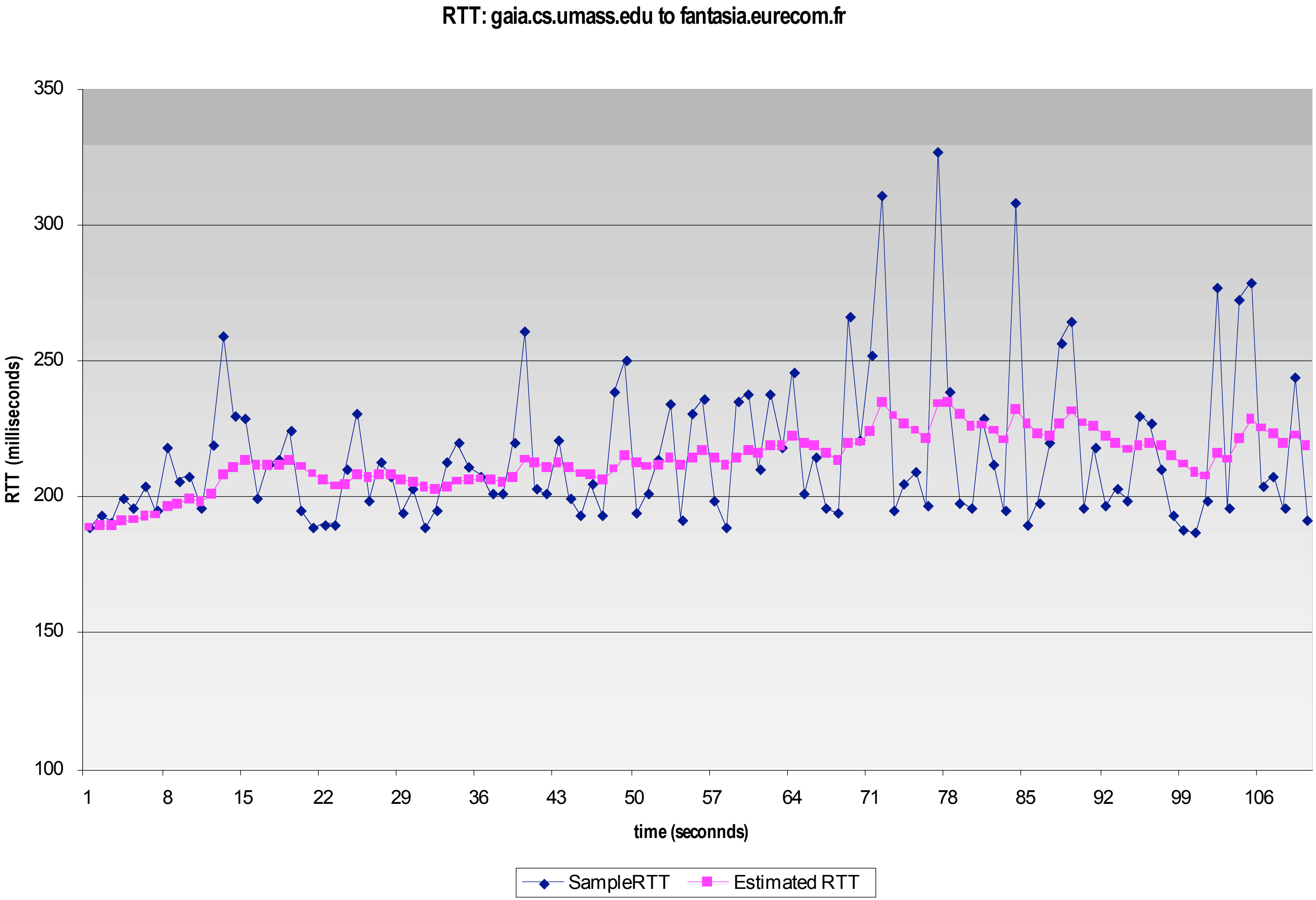
Initialize: SRTT = DevRTT = 3000 msec
RTO = 3000ms

Upon receiving first packet:
SRTT = 400, DevRTT= 200

Upon receiving second packet:
diff = 480 – 400 = 80
SRTT = 400 + 10 = 410
DevRTT= 200 + ¼ (80-200) = 170



Example RTT estimation:



TCP Receiver: when to send ACK?

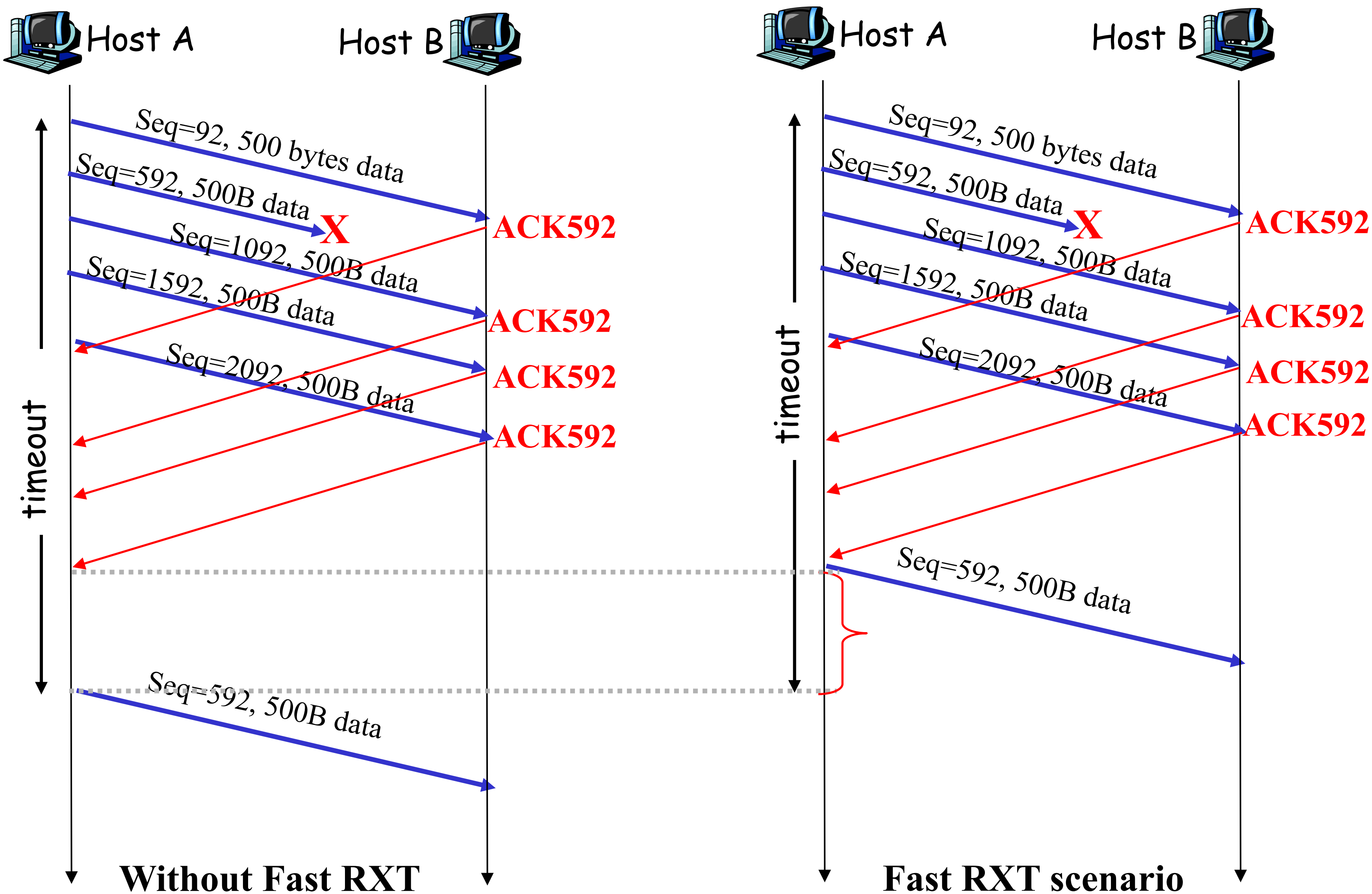
Event at receiver	TCP Receiver action
in-order segment arrival, no gaps, everything earlier already ACKed 	delayed ACK: wait up to 500ms, If nothing arrived, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send one cumulative ACK
out-of-order arrival: higher-than-expect seq. #, gap detected 	Immediately send ACK, indicating <u>seq. # of next expected byte</u>
arrival of segment that partially or completely fills a gap 	immediate send ACK if segment starts at <u>the lower end of the gap</u>

TCP Fast Retransmit

- RTO set to a relatively long value
  - long delay before resending lost packet
- Can detect lost segments via duplicate ACKs.
  - When a segment is lost, next arrival at receiver is out of order
  - When a segment arrived out of order, receiver sends an ack with the seq# of last in-order arrival
- If sender receives 3 duplicate ACKs for seq#(n), it assumes the segment for seq#(n) was lost

→fast retransmit: resend the segment before timer expires

TCP fast retransmit example



Flow control window size vs. seq# field length

- Window size: W
- Seq# field: n bits long
- W ≤ 2^n / 2
  - n = 4, W ≤ 8

TCP Options

Meaning	Reference
End of Option List	[RFC793]
No-Operation	[RFC793]
Maximum Segment Size	[RFC793]
Window Scale	[RFC7323]
SACK Permitted	[RFC2018]
SACK	[RFC2018]
Echo (obsoleted by option 8)	[RFC1072][RFC6247]
Echo Reply (obsoleted by option 8)	[RFC1072][RFC6247]
Timestamps	[RFC7323]
Partial Order Connection Permitted (obsolete)	[RFC1693][RFC6247]
Partial Order Service Profile (obsolete)	[RFC1693][RFC6247]
CC (obsolete)	[RFC1644][RFC6247]
CC.NEW (obsolete)	[RFC1644][RFC6247]
CC.ECHO (obsolete)	[RFC1644][RFC6247]
TCP Alternate Checksum Request (obsolete)	[RFC1146][RFC6247]
TCP Alternate Checksum Data (obsolete)	[RFC1146][RFC6247]
Skeeter	[Stev_Knowles]
Bubba	[Stev_Knowles]
Trailer Checksum Option	[Subbu_Subramaniam][Monroe_Bridges]
MD5 Signature Option (obsoleted by option 29)	[RFC2385]
SCPS Capabilities	[Keith_Scott]
Selective Negative Acknowledgements	[Keith_Scott]
Record Boundaries	[Keith_Scott]
Corruption experienced	[Keith_Scott]
SNAP	[Vladimir_Sukonnik]
TCP Compression Filter	[Steve_Bellovin]
Quick-Start Response	[RFC4782]

TCP Connection Close

- Either end can initiate the close of ***its end*** of the connection at any time

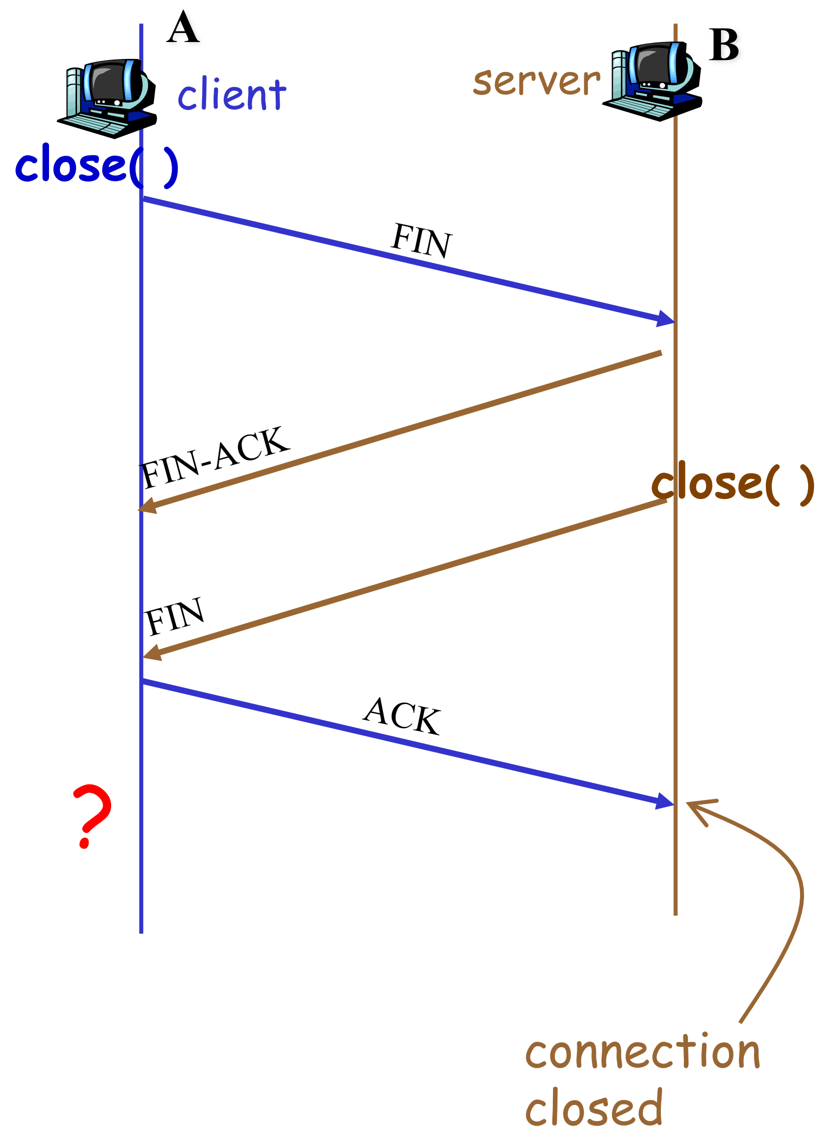
1: one end (A) sends TCP FIN control segment to the other

  - ***No data***

2: the other end (B) receives FIN, replies with FIN_ACK; when it's ready to close too, send FIN

3: A receives FIN, replies with **FIN-ACK**.

4: B receives ACK, close connection



what should A do after sending ACK?

the well-known “two-army problem”

Q: how can the 2 red armies agree on an attack time?

- Fact: the last one who send a message does not know whether the msg is delivered
- one cannot send an ACK to acknowledge an ACK

TCP Connection Close

1: A sends TCP FIN control segment to the other

2: B receives FIN, replies with FIN_ACK; when it's ready to close too, send FIN

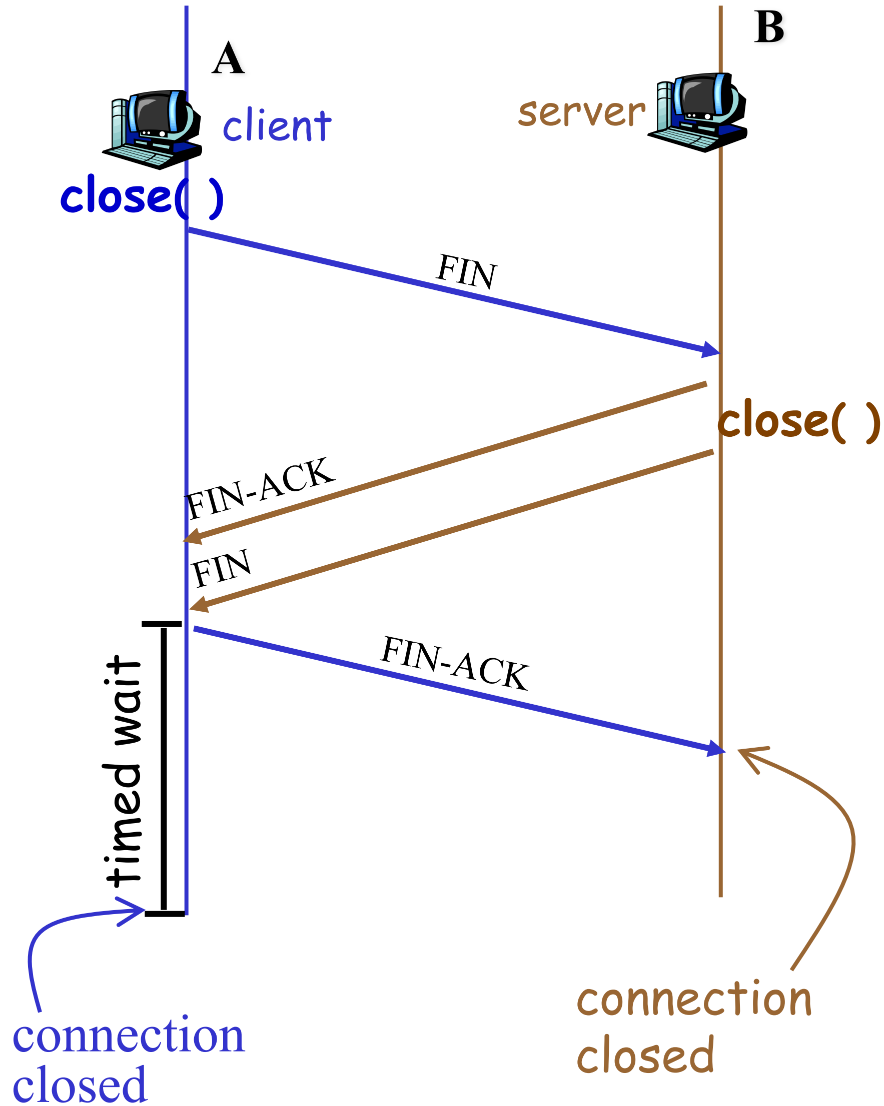
3: A receives FIN, sends FIN_ACK

- A Enters "timed wait", waits for 2 MSL before deleting the connection state

4: B receives FIN_ACK, close connection

5: A closes the connection after waiting for "long enough" time w/0 receiving retransmitted FIN

  - Long enough = 2 x Max. Seg. Lifetime

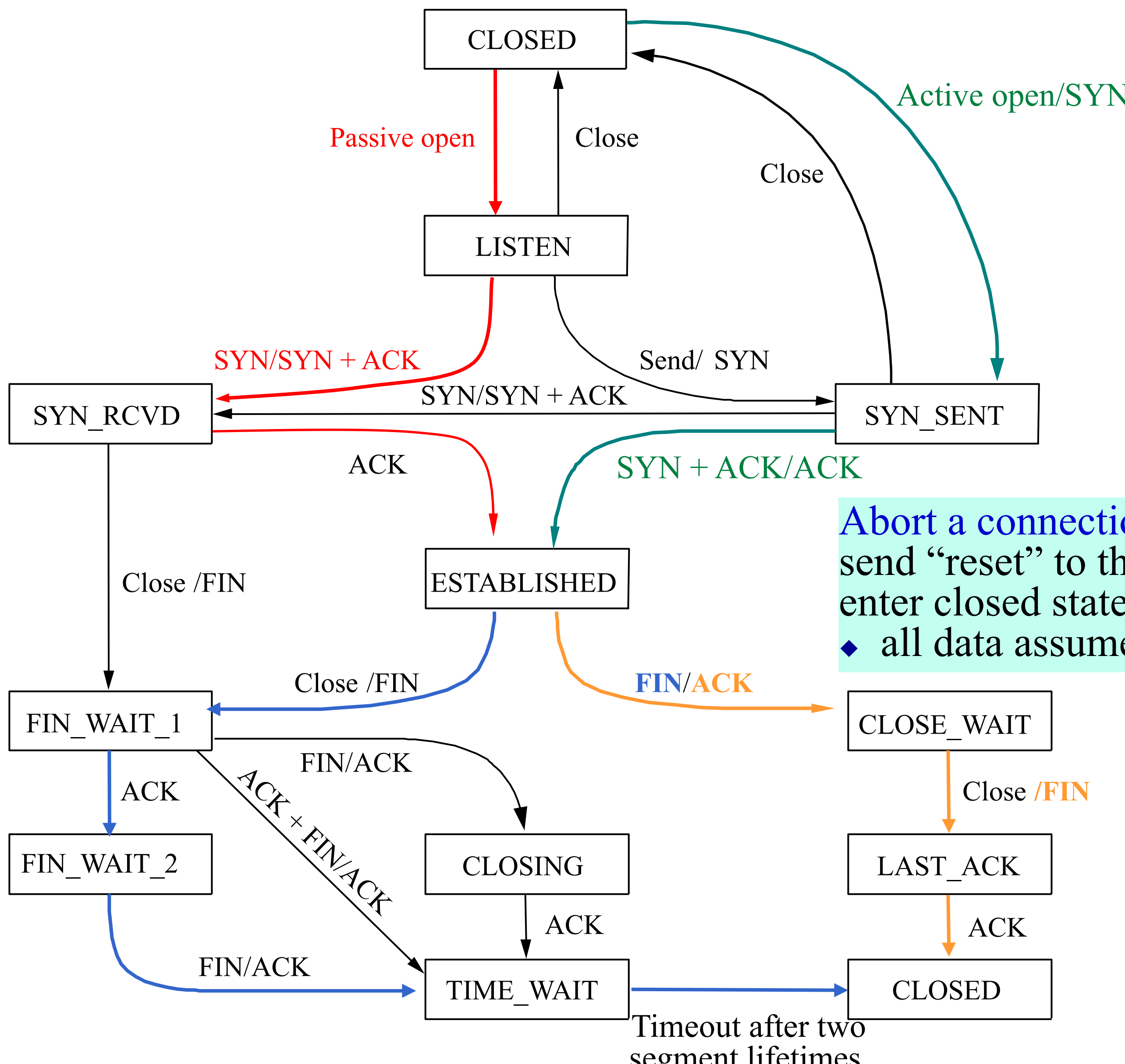


netstat -an -p tcp

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp6       0      0  ::1.587                *.*                    LISTEN
tcp4       0      0  127.0.0.1.587          *.*                    LISTEN
tcp6       0      0  ::1.25                 *.*                    LISTEN
tcp4       0      0  127.0.0.1.25           *.*                    LISTEN
tcp4       0      0  131.179.6.105.52914    188.174.252.22.80      SYN_SENT
tcp4       0      0  131.179.6.105.52911    52.201.115.248.443     CLOSE_WAIT
tcp4       0      0  *.5533                 *.*                    LISTEN
tcp4       0      0  *.*                    *.*                    CLOSED
tcp4       0      0  *.5352                 *.*                    LISTEN
tcp4       0      0  *.53                   *.*                    LISTEN
tcp4      31      0  131.179.6.105.52896    108.160.172.193.443    CLOSE_WAIT
tcp6       0      0  2607:f010:2e9:4:.52894 2607:f8b0:4007:8.443   ESTABLISHED
tcp4       0      4  131.179.6.105.52893    77.38.186.20.14307     ESTABLISHED
tcp4       0      4  131.179.6.105.52892    77.38.172.214.24731    ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52890 2607:f8b0:4007:8.443   ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52883 2001:668:108:9a4.80    ESTABLISHED
tcp4       0      0  131.179.6.105.52865    74.112.184.85.443      CLOSE_WAIT
tcp4       0      0  131.179.6.105.52835    198.189.255.163.80     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52828    198.189.255.163.80     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52827    198.189.255.163.80     CLOSE_WAIT
tcp6       0      0  2607:f010:2e9:4:.52798 2607:f8b0:4007:8.443   ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52797 2607:f8b0:4007:8.80    CLOSE_WAIT
tcp4       0      0  131.179.6.105.52794    74.112.185.182.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52793    74.112.185.182.443     CLOSE_WAIT
tcp6       0      0  2607:f010:2e9:4:.52603 2607:f8b0:4007:8.443   CLOSE_WAIT
tcp4       0      0  131.179.6.105.52565    17.110.241.16.993      ESTABLISHED
tcp4      31      0  131.179.6.105.52547    54.192.139.170.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52479    13.94.234.1.443        ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52439 2607:f8b0:400e:c.5228  ESTABLISHED
tcp4       0      0  131.179.6.105.52437    74.112.184.86.443      ESTABLISHED
tcp4       0      0  131.179.6.105.52401    107.152.24.197.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52387    131.253.34.234.443     ESTABLISHED
tcp4       0      0  131.179.6.105.52382    216.58.217.205.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52378    173.194.202.125.5222   ESTABLISHED
tcp4       0      0  131.179.6.105.52329    74.125.28.109.993      ESTABLISHED
tcp4       0      0  131.179.6.105.52305    17.110.226.165.5223    ESTABLISHED
tcp4       0      0  131.179.6.105.52303    65.52.108.74.443       ESTABLISHED
tcp4       0      0  131.179.6.105.52299    162.125.17.3.443       ESTABLISHED
tcp4       0      0  131.179.6.105.52297    17.110.241.16.993      ESTABLISHED
tcp4       0      0  131.179.6.105.52295    194.68.30.24.4070      ESTABLISHED
tcp4       0      0  131.179.6.105.52293    17.110.228.92.5223     ESTABLISHED
tcp4       0      0  131.179.6.105.52289    91.190.216.55.12350    ESTABLISHED
tcp4       0      0  131.179.6.105.52288    157.55.130.172.40008   ESTABLISHED
tcp4       0      0  131.179.6.105.7313     *.*                    LISTEN
tcp4       0      0  131.179.6.105.53       *.*                    LISTEN
tcp4      31      0  131.179.6.74.52156     199.47.217.97.443      CLOSE_WAIT
tcp4       0      0  131.179.6.74.51067     107.152.24.197.443     CLOSE_WAIT
tcp4       0      0  131.179.6.74.51065     107.152.24.197.443     CLOSE_WAIT
tcp4       0      0  131.179.6.74.51053     173.194.202.125.5222   ESTABLISHED
tcp6       0      0  2607:f010:3f9::1.65165 2607:f8b0:4007:8.443   CLOSE_WAIT
tcp4       0      0  131.179.196.220.64254  74.112.185.87.443      CLOSE_WAIT
tcp6       0      0  2605:e000:1521:5.63475 2607:f8b0:400e:c.5222  ESTABLISHED
tcp6       0      0  2605:e000:1521:5.63473 2607:f8b0:400e:c.5222  ESTABLISHED
tcp6       0      0  2605:e000:1521:5.54836 2607:f8b0:4007:8.443   CLOSE_WAIT
tcp4       0      0  131.179.196.220.62360  131.179.196.228.17500  ESTABLISHED
....
```

TCP state-transition diagram

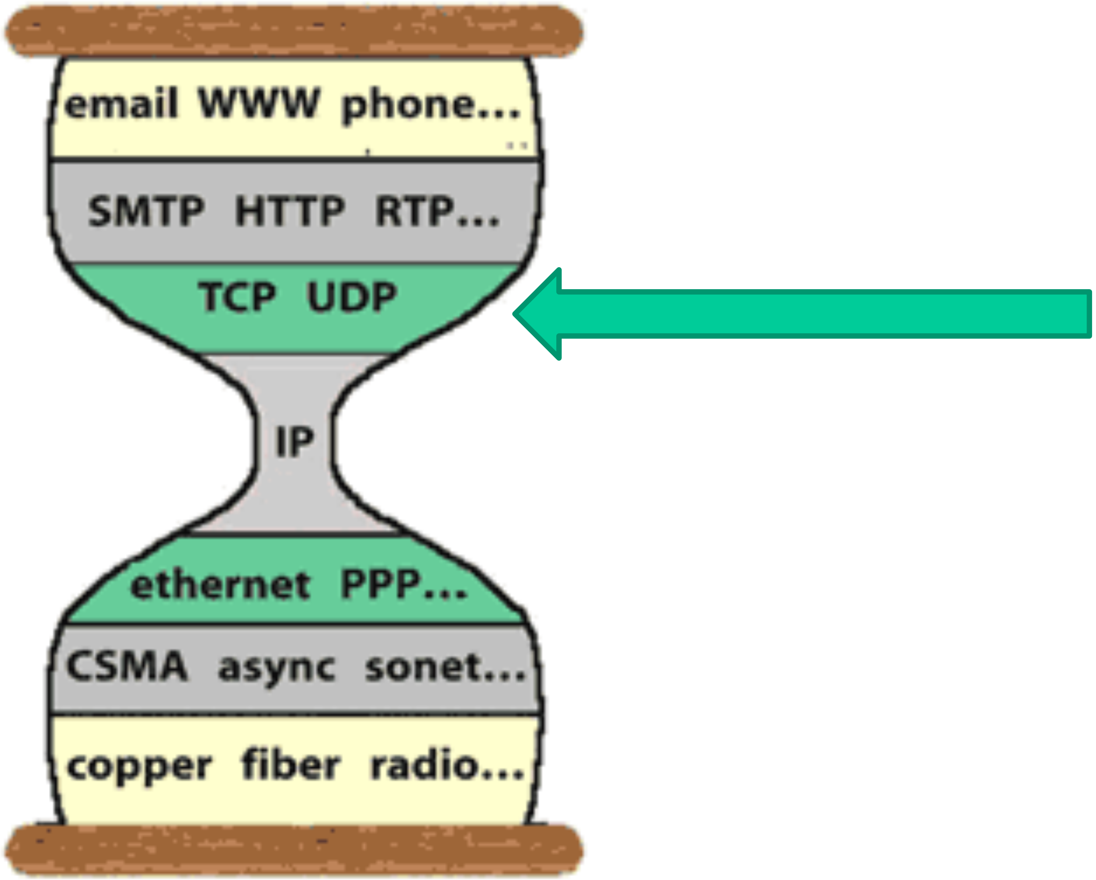
rfc793_state_machine.pptx



Abort a connection: either end may send "reset" to the other end, then enter closed state immediately

- all data assumed lost

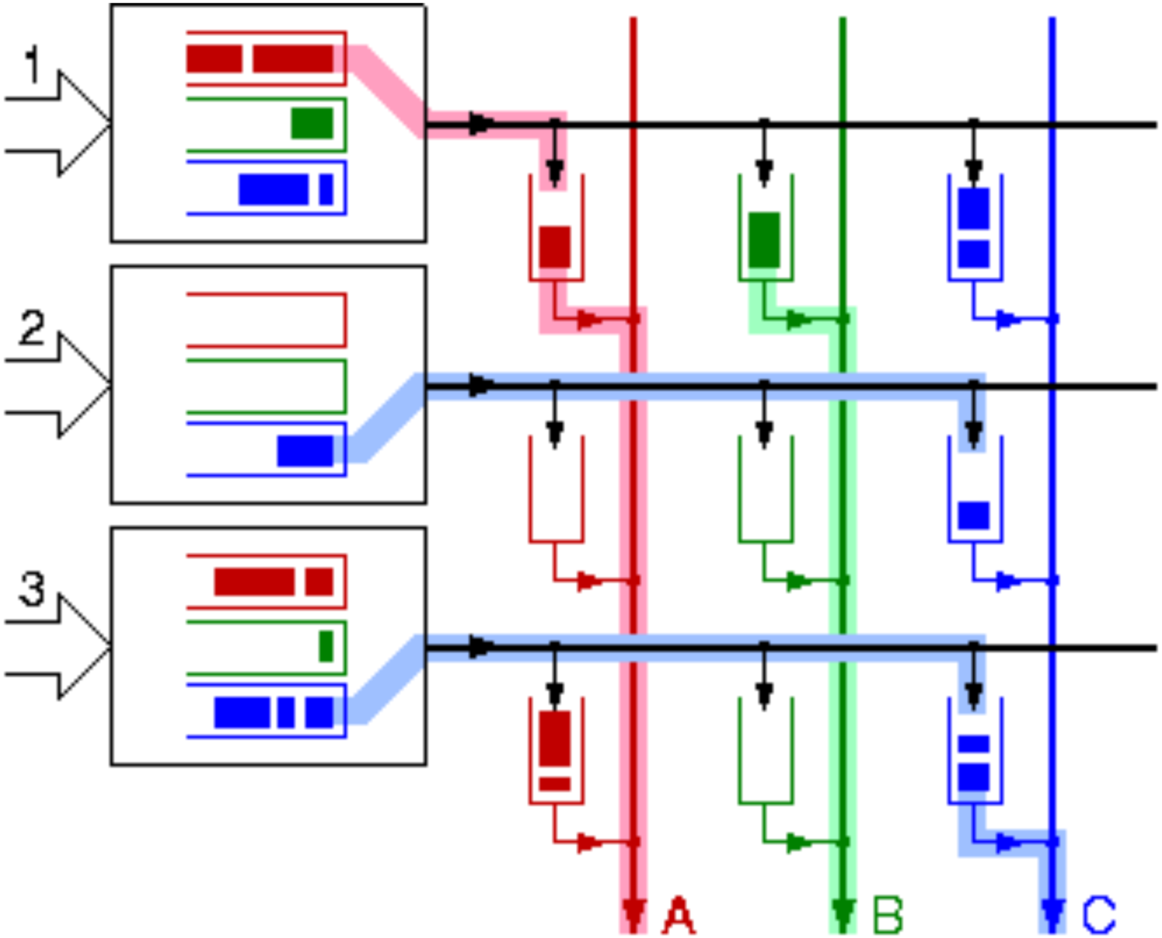
Wrapping Up Transport Layer



Chapter 3

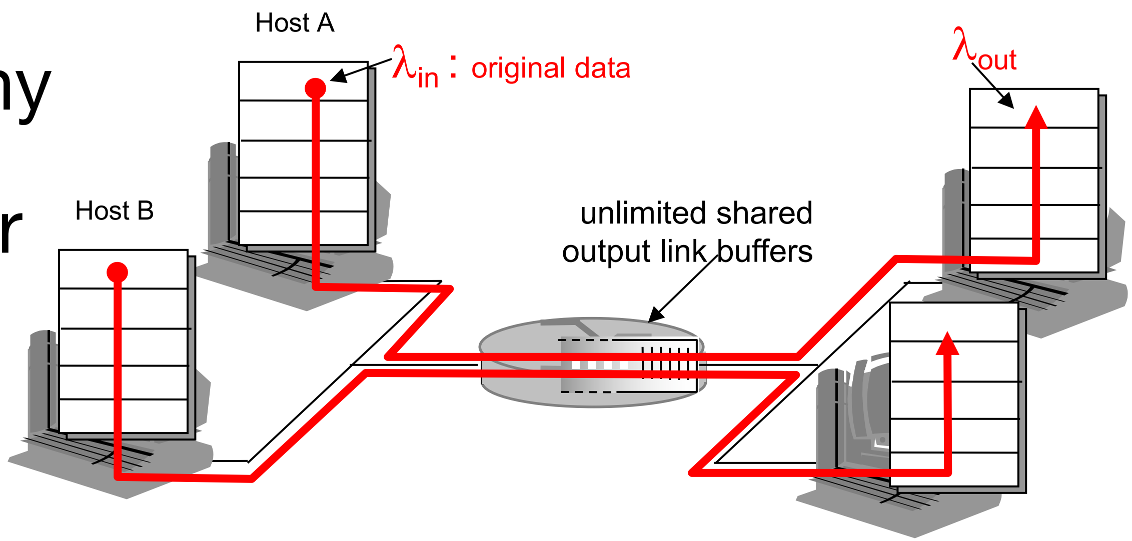
3.5 TCP Congestion Control

Role of Buffers in Packet Switching



Network Congestion

Congestion: "too many sources sending too much data too fast for *network* to handle"

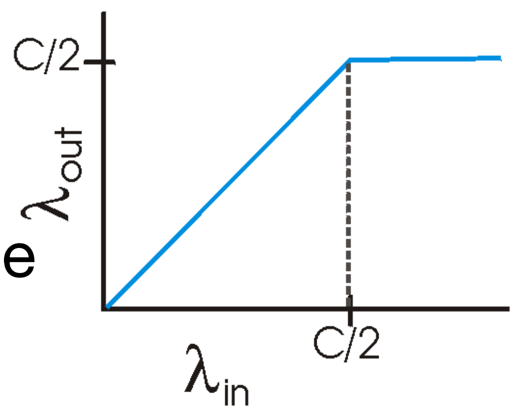


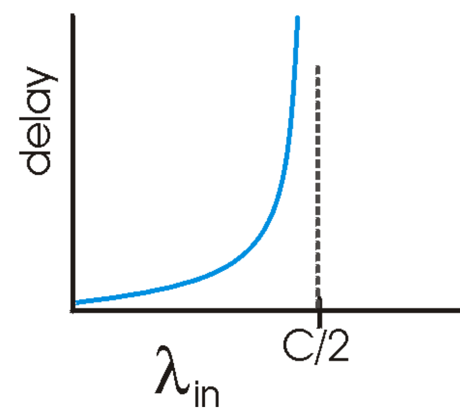
Scenario 1

- 2 senders, 2 receivers
- one router w/ *infinite* buffer
- no retransmission

When congested:

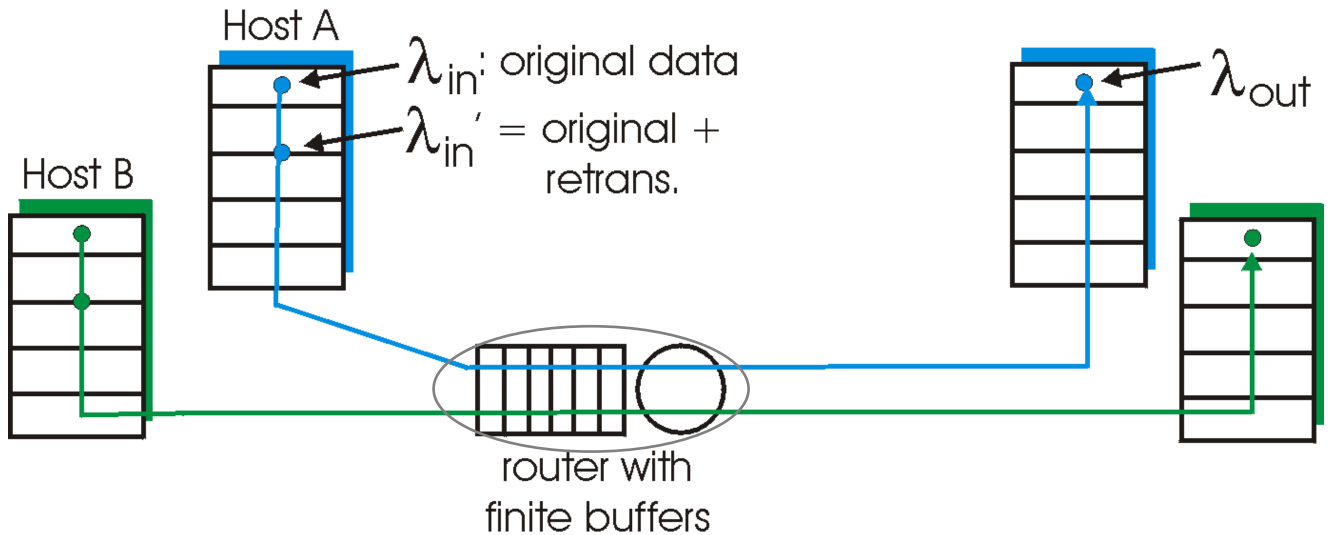
- long delays
- Achieve maximum possible throughput





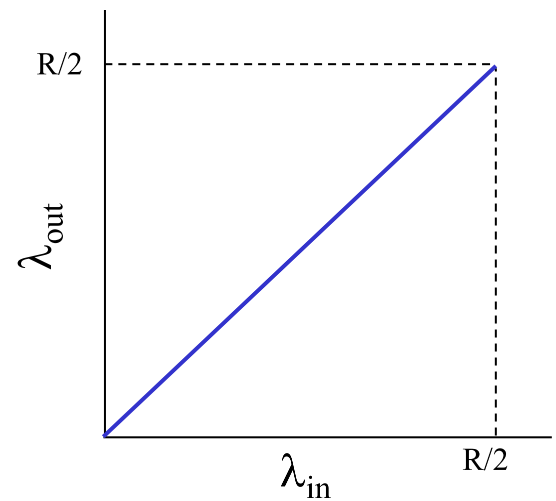
Congestion: scenario 2

- **one router, *finite* buffer**
- **senders *retransmit* when timeout**



Ideal case:

- Each sender takes turns and sends only when router buffer available

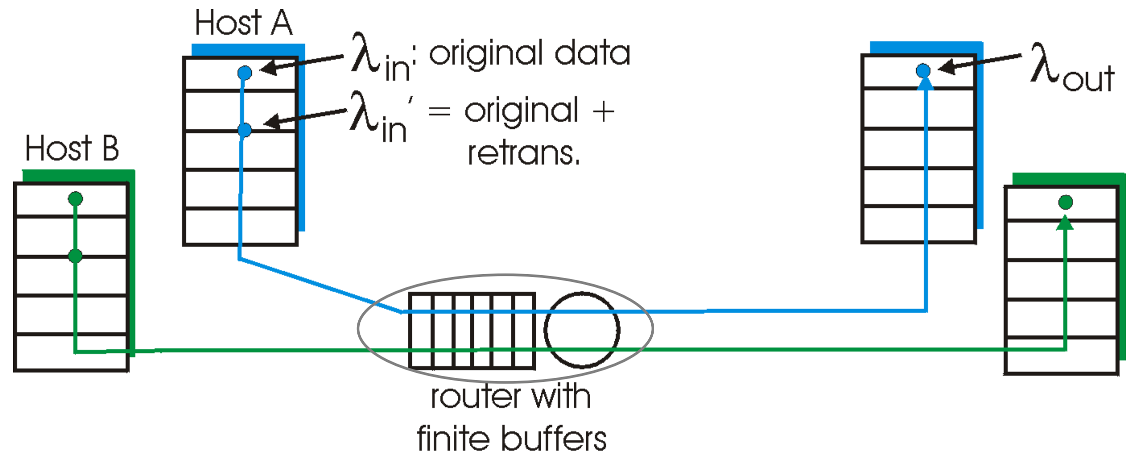


$$\lambda_{in} = \lambda_{out}$$

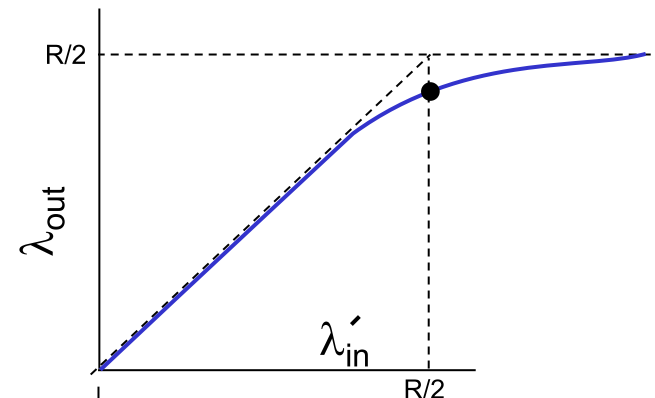
Congestion: scenario 2

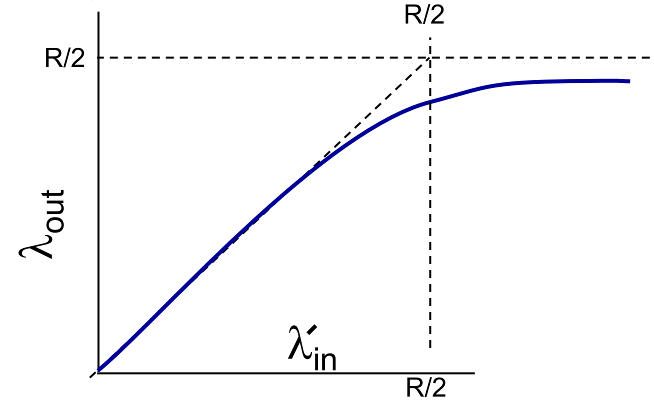
one router, *finite* buffer

senders *retransmit* when timeout

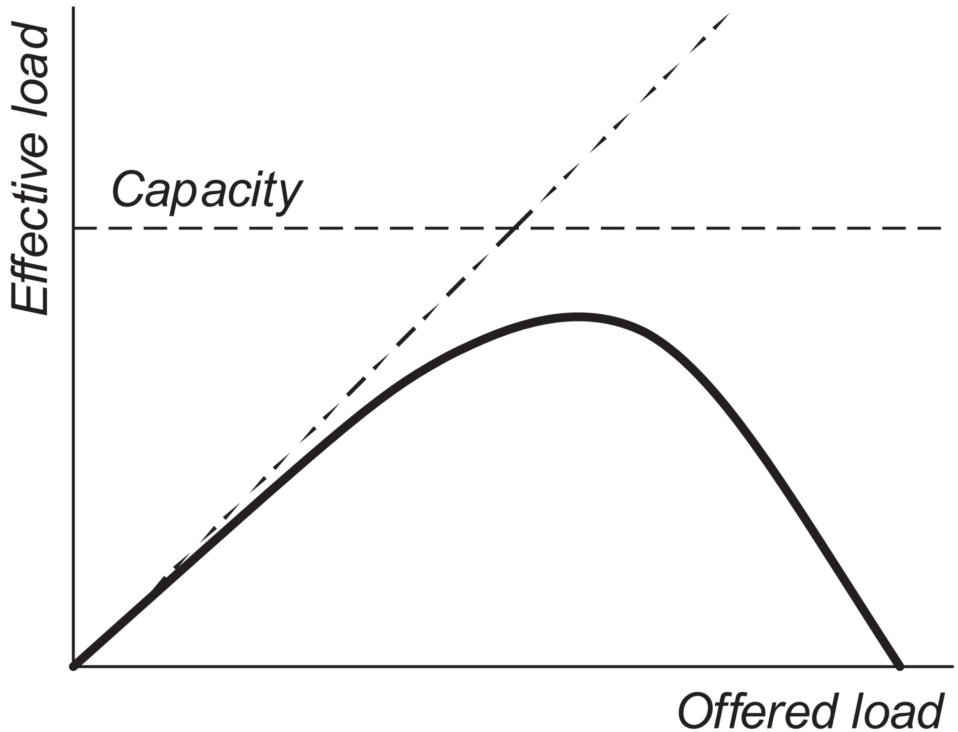


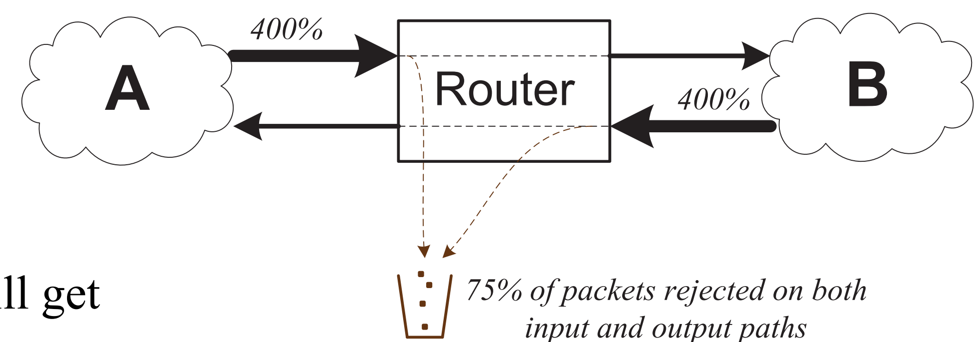
- Packets may get dropped at router due to buffer full
- **Known loss case**: sender only retransmits if packet known to be lost
- **Duplicates**: sender times out prematurely and retransmits, *some duplicates* are delivered





Congestion Collapse

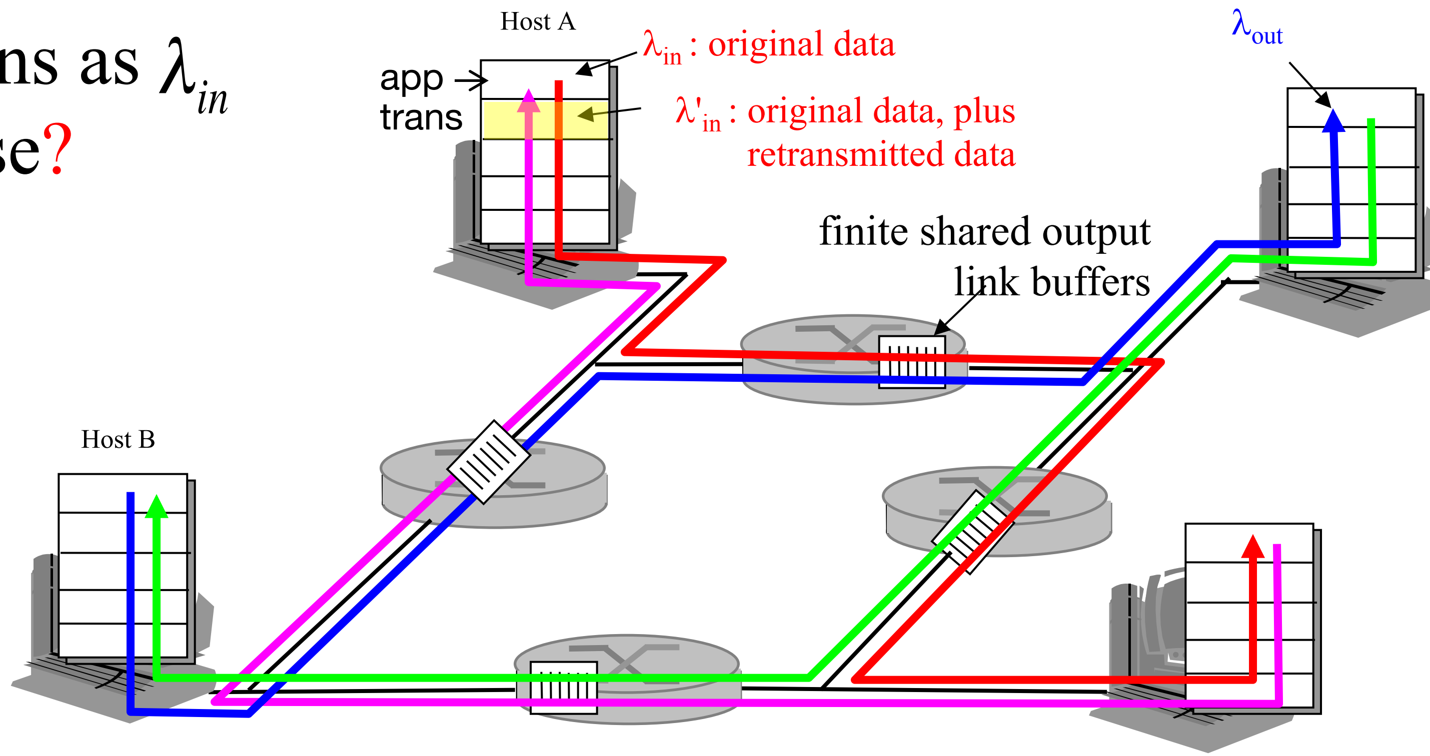




Think of traffic on freeways:
- if there are too many cars, the freeways will get gridlocked and nobody will be going anywhere

Congestion scenario 3

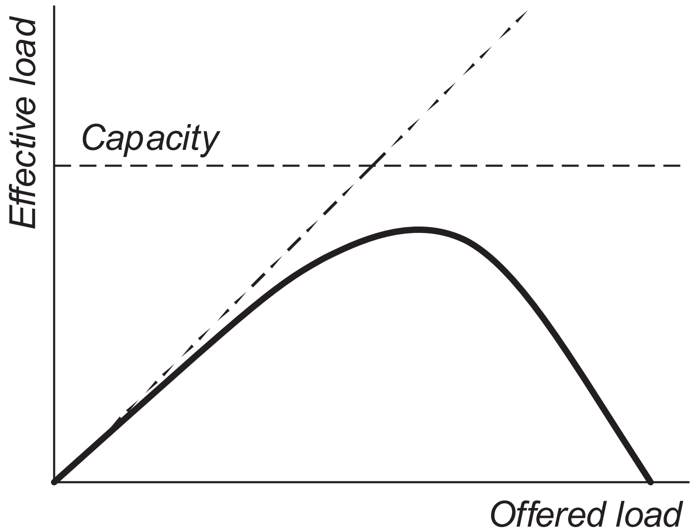
Q: what happens as λ_{in} and λ'_{in} increase?

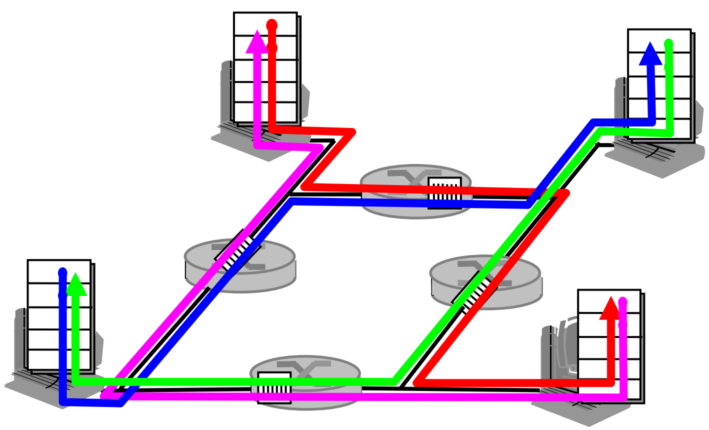


- Long delays
- superfluous retransmissions
- when a packet is dropped, any "upstream transmission capacity" used for that packet was wasted!

Cost of congestion

- unneeded retransmissions: bottleneck link transmitted multiple copies of the same packet, reduce effective throughput
- when a packet dropped further down the road, any "upstream" transmission capacity used for that packet was wasted!





Congestion control: options

Two broad approaches for congestion control:

Network-assisted congestion control: routers provide feedback to end hosts, such as

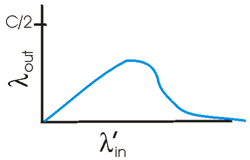
- Single bit congestion indication, or
- Explicit rate that sender should send at

End-end congestion control: no explicit feedback from network

- Hosts infer congestion from observed loss or delay
  - approach taken by TCP

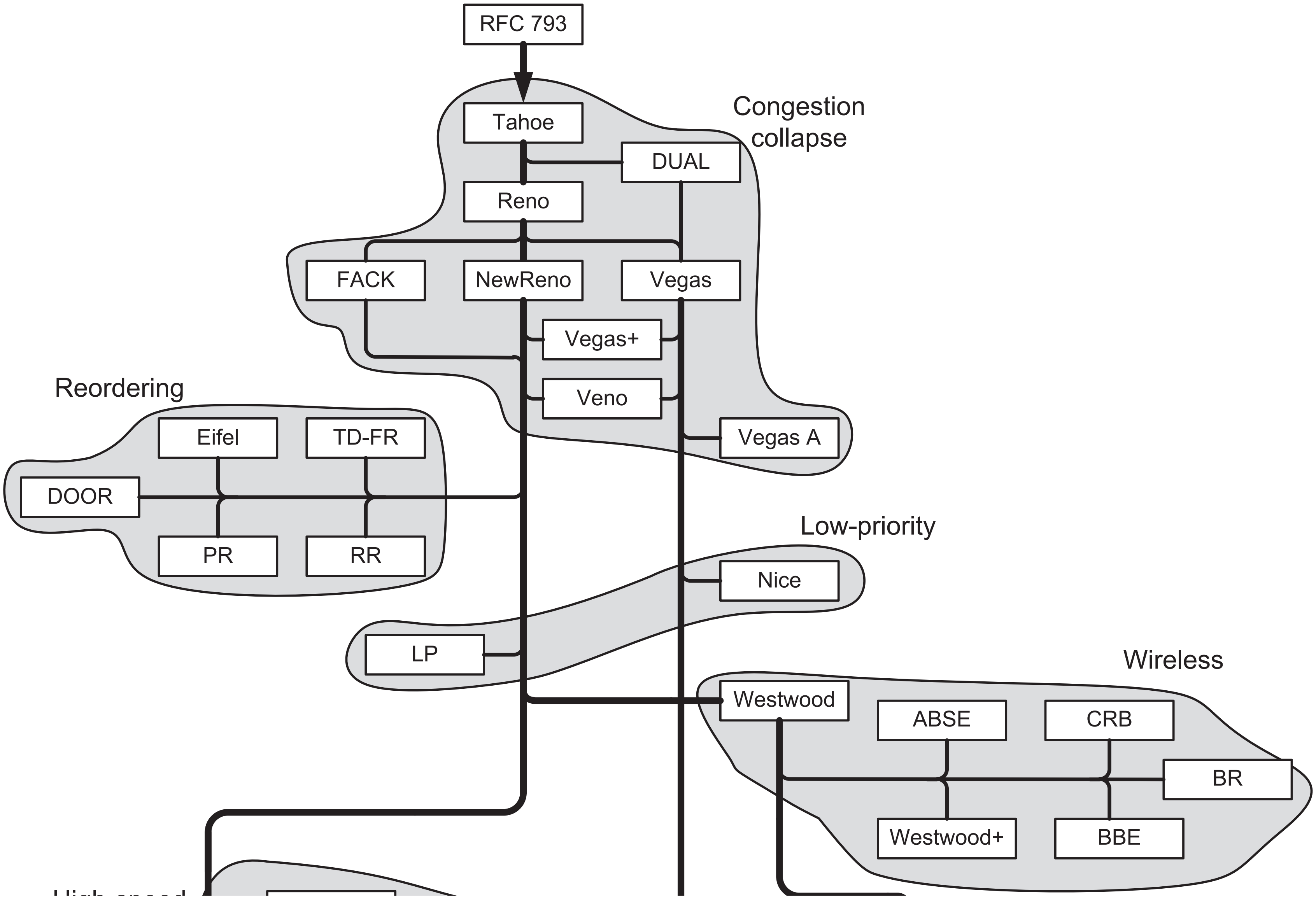
A Bit of The History

- 1974: 3-way handshake
- 1978: TCP and IP split into TCP/IP
- 1983: January 1, ARPAnet switches to TCP/IP
- **1986: Internet begins to suffer congestion collapses**

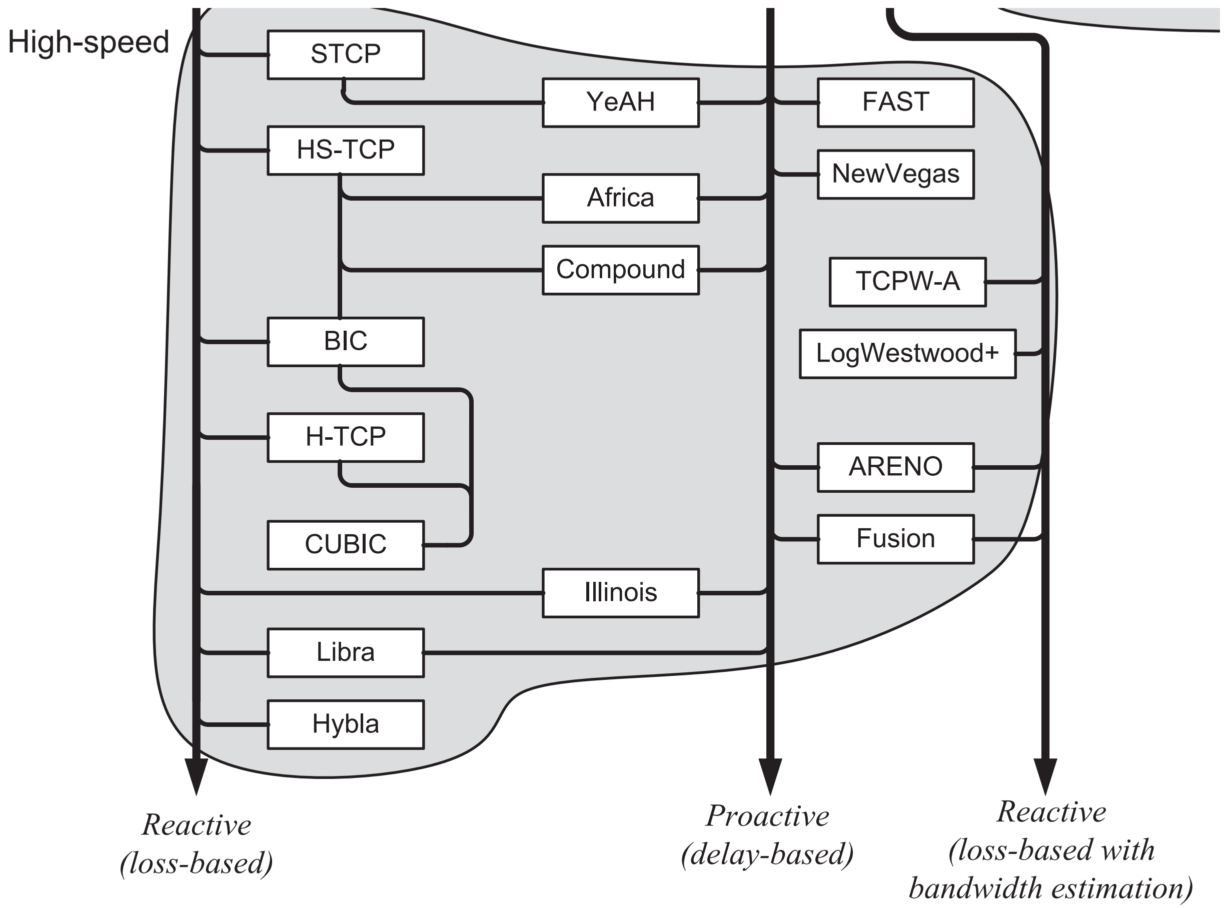


- 1987-8: Van Jacobson fixes TCP, publishes seminal paper (Tahoe)
- 1990: Fast recovery and fast retransmit added (Reno)

A “Little” Bit More



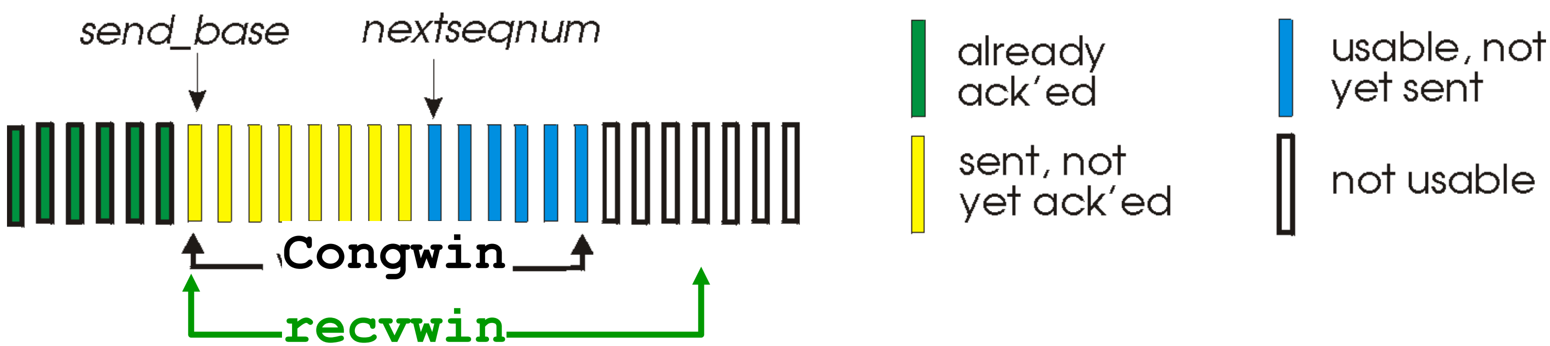
And More



High-speed
STCP
YeAH
HS-TCP
Africa
Compound
BIC
H-TCP
CUBIC
Illinois
Libra
Hybla
FAST
NewVegas
TCPW-A
LogWestwood+
ARENO
Fusion
Reactive
(loss-based)
Proactive
(delay-based)
Reactive
(loss-based with
bandwidth estimation)

TCP Congestion Control

- Add a "congestion control window" **`cwnd`** on top of flow-control window
- Sender limits: **`LastByteSent-LastByteAcked`** ≤ **`cwnd`**



- **`cwnd`** initialized to 1 packet, increases until congestion
  - How sender infers congestion: packet loss (timeout, or 3 dup. ACKs)
- Upon loss: *decrease* **`cwnd`**, then begin increasing again
  - Two phases: (1) slow start, (2) congestion avoidance
  - a threshold **`(sshthresh)`** defines the boundary between the two

MSS: maximum segment size

"Slow" Start

- **Objective: quickly gauge the pipeline size**

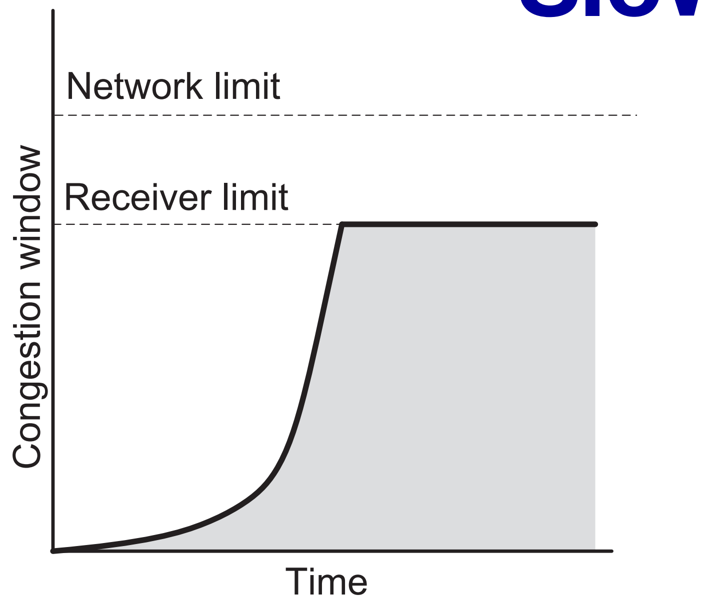
1. Start with cwnd = 1 MSS
2. Send cwnd packets
3. If all packets got acked
   - cwnd = 2 * cwnd
   - goto 2
4. Else
   have gone too far
   - ss-thresh = cwnd / 2
   - cwnd = 1 MSS
   - goto 2

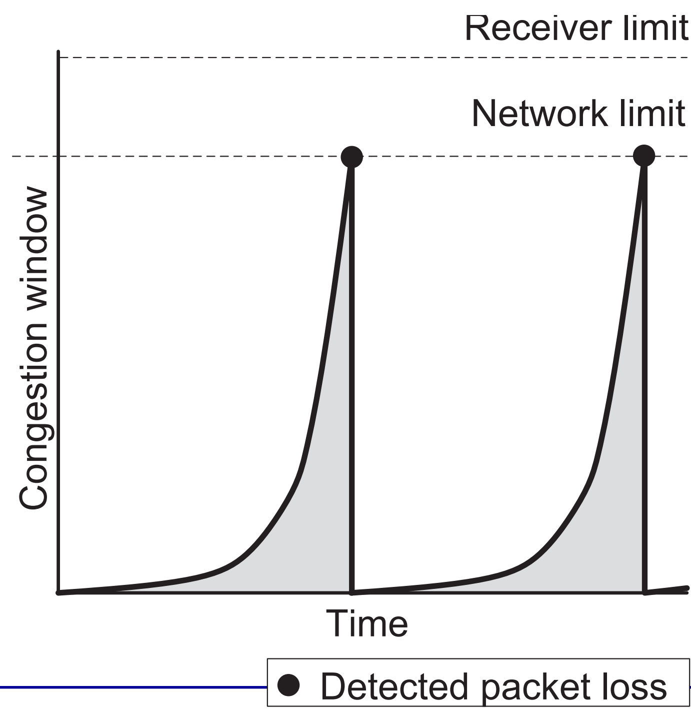
Same, but using "self-clocking" of TCP

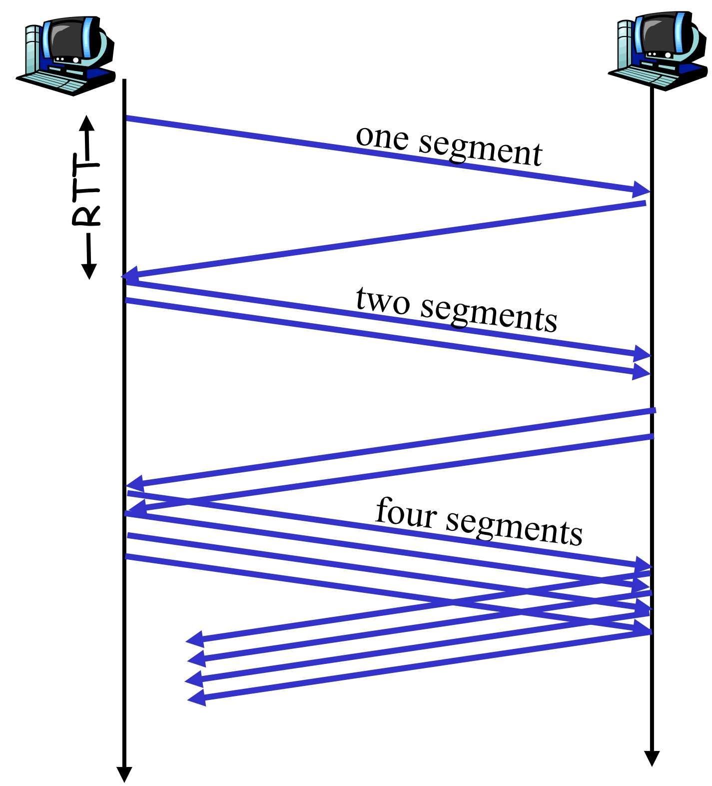
1. Start with cwnd = 1 MSS
2. Send cwnd packets
3. If ack
   - cwnd = cwnd + 1 MSS
   - send packets
4. If timeout
   - sshthresh = cwnd / 2
   - cwnd = 1 MSS
   - goto 2

Multiplicative Increase, Reset to 1 Decrease

Slow Start


Network limit
Receiver limit
Congestion window
Time


Receiver limit
Network limit
Congestion window
Time
Detected packet loss


RTT
one segment
two segments
four segments

Congestion Avoidance

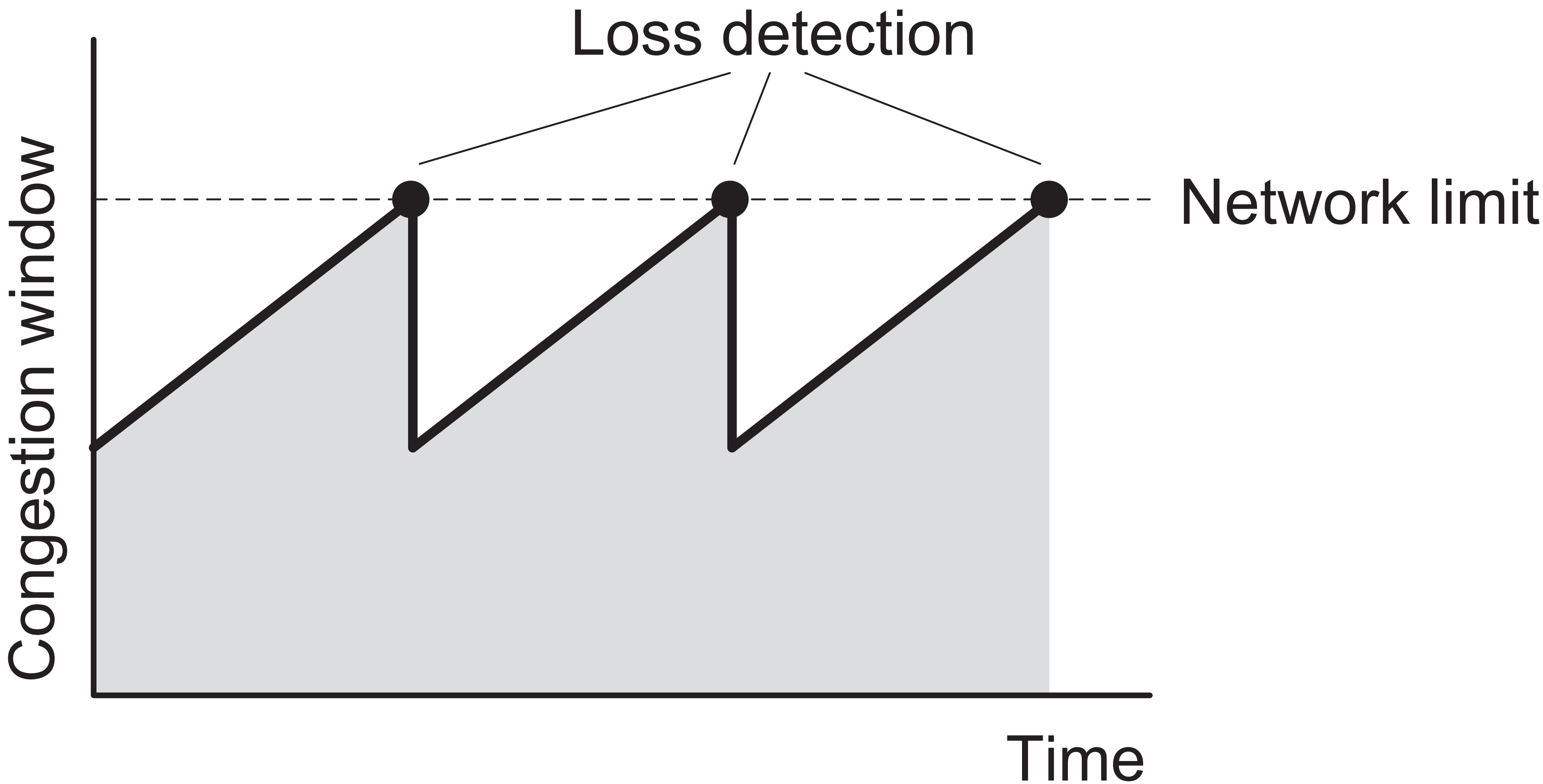
- Objective: maintain steady state, probe for unused resources, avoid conflicts

- Send cwnd packets
- If all sent packets got ack
  - cwnd = cwnd + 1 MSS
- Else
  - cwnd = cwnd / 2

> Send cwnd packets
> If ack
> - cwnd = cwnd + $(1\ \text{MSS})^2$/cwnd
>
> If timeout
> - cwnd = cwnd / 2

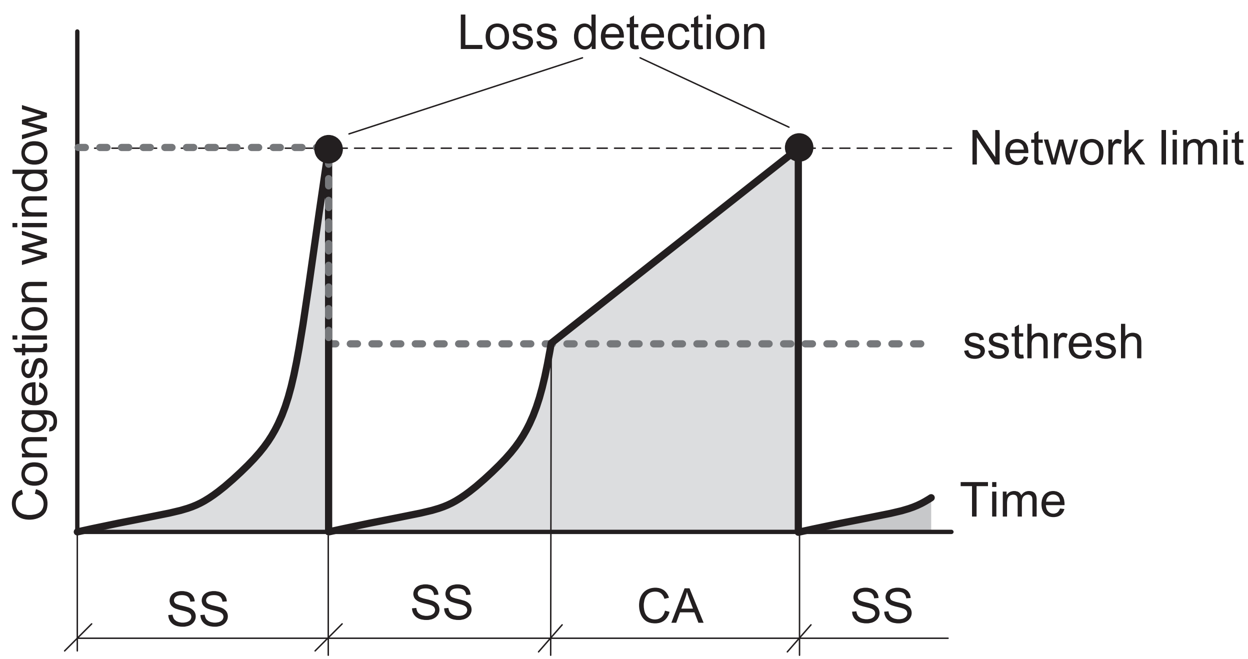
- Additive Increase, Multiplicative Decrease (AIMD)

Congestion Avoidance



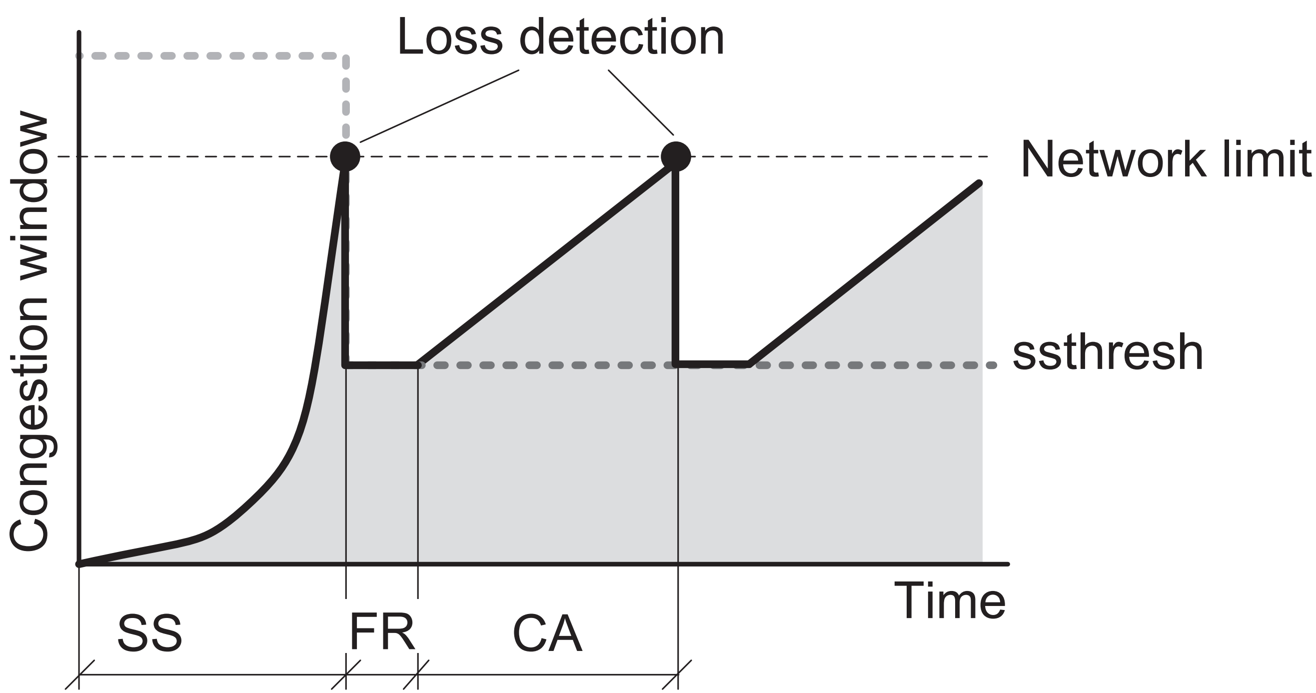
Combining Slow Start with Congestion Avoidance (Tahoe)

- Set initial `sshthresh`
- When `cwnd < sshthresh`, use Slow Start
  - sshthresh will get updated
- when `cwnd` ≥ `sshthresh`, use Congestion Avoidance

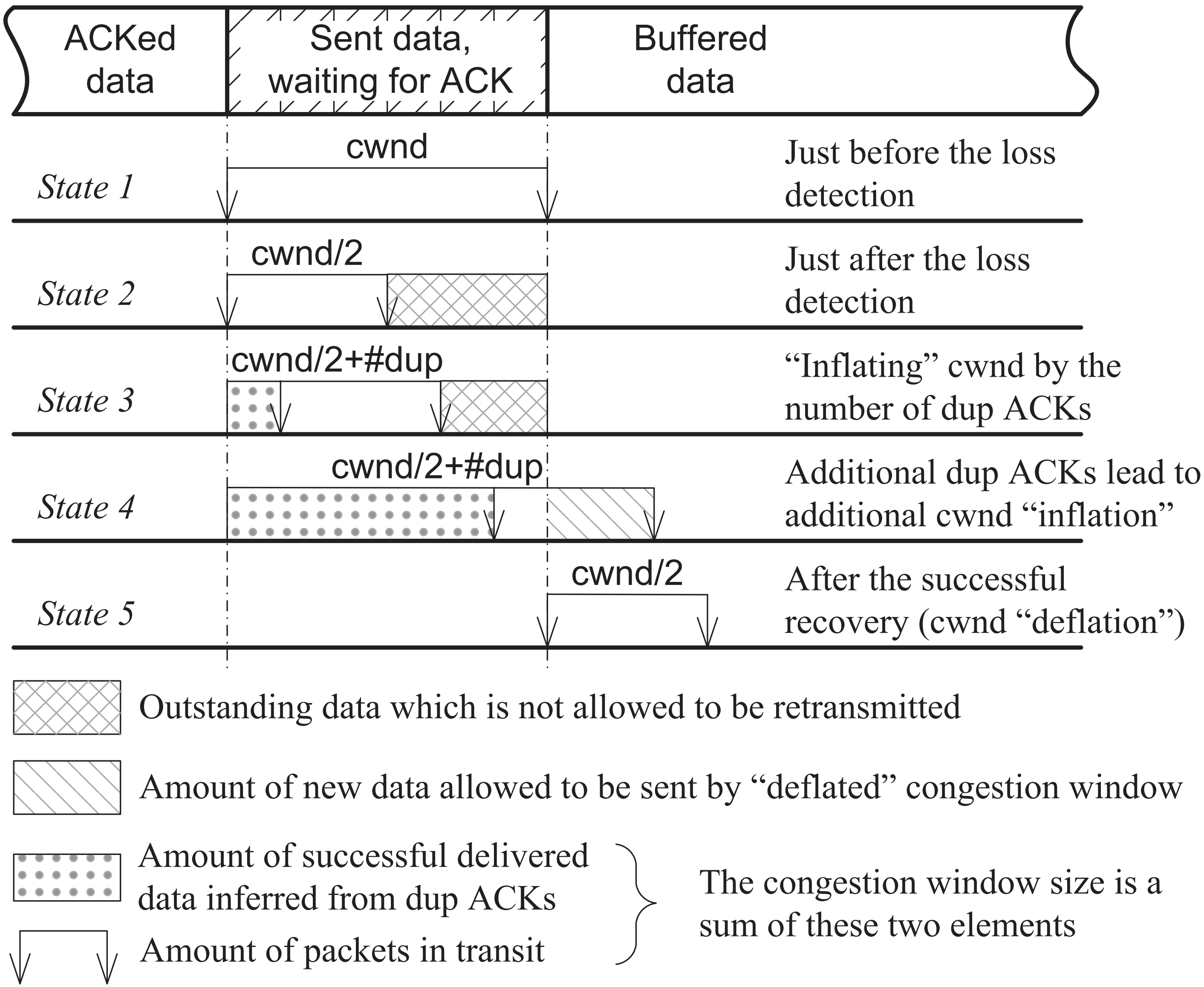


Slow Start and Congestion Avoidance (Reno)

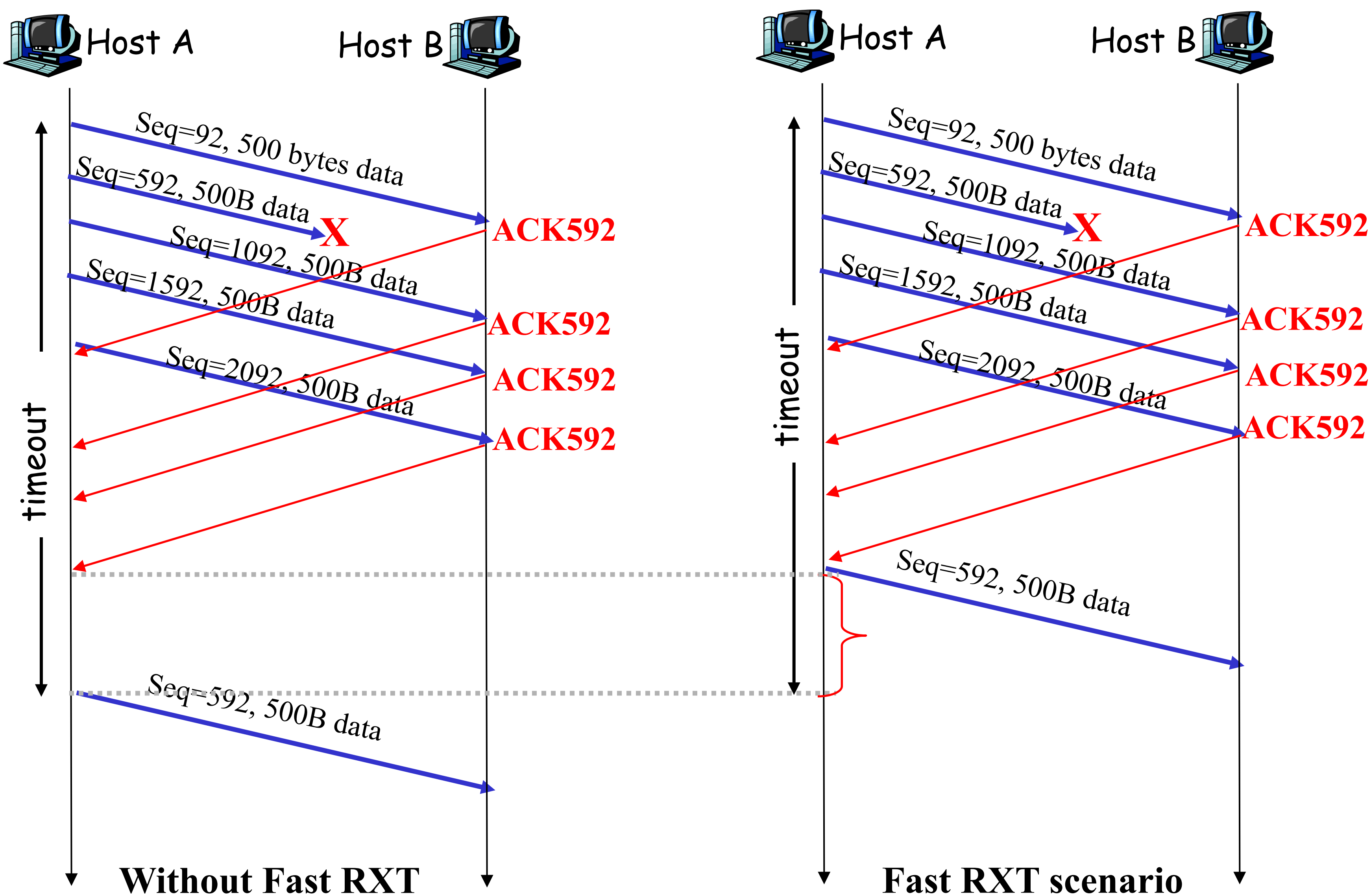
- If just one packet is lost, dropping cwnd to 1 is an overreaction
- What if loss detected through 3 dup acks, we just reduce cwnd by half, and quickly recover from the loss (Fast Recovery)



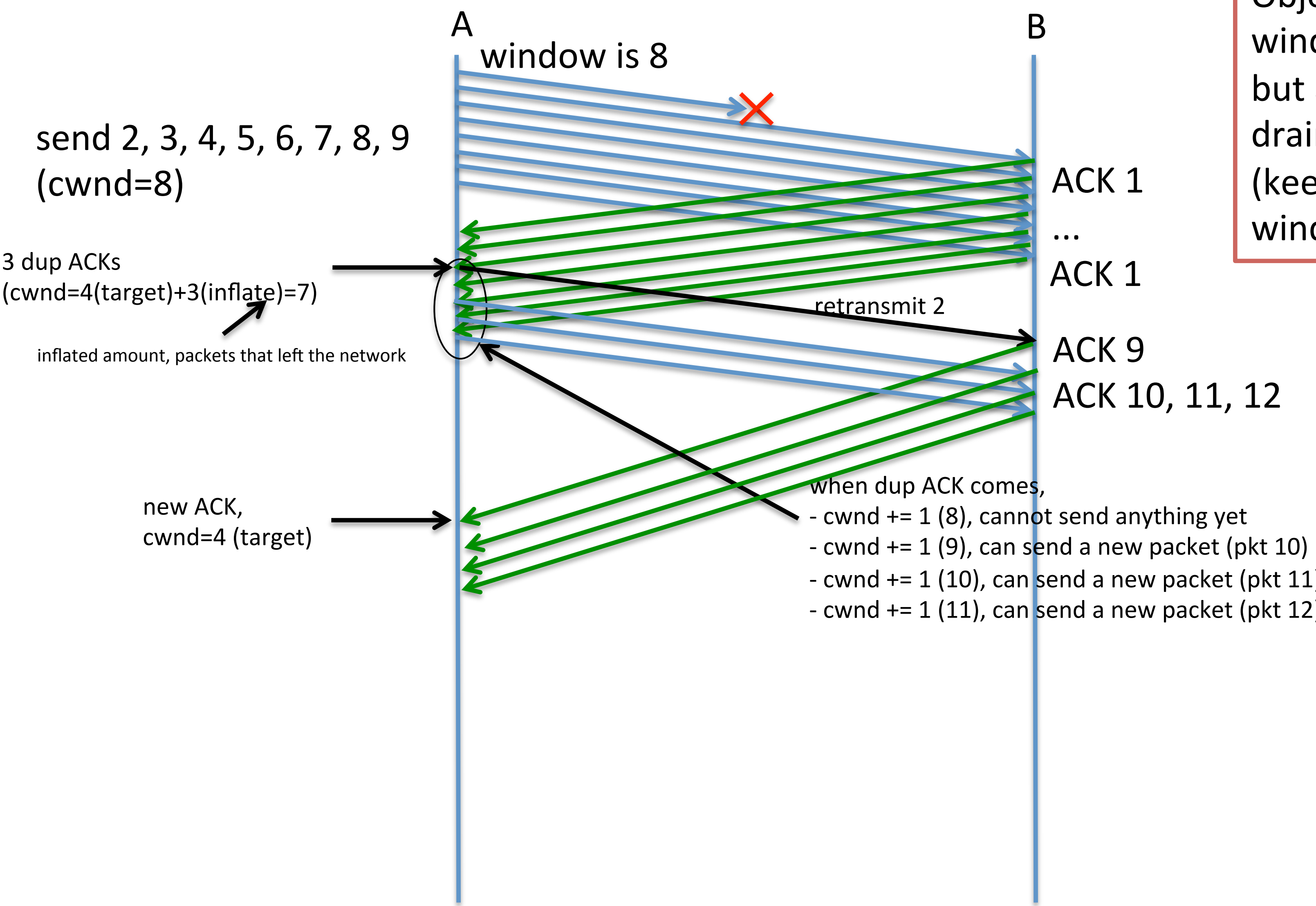
Fast Recovery/Retransmit (Reno)



TCP fast retransmit example



Fast Retransmit / Fast Recovery



Objective: cut window by half, but avoiding draining the pipe (keep ~half of window in flight)

Is TCP fair?

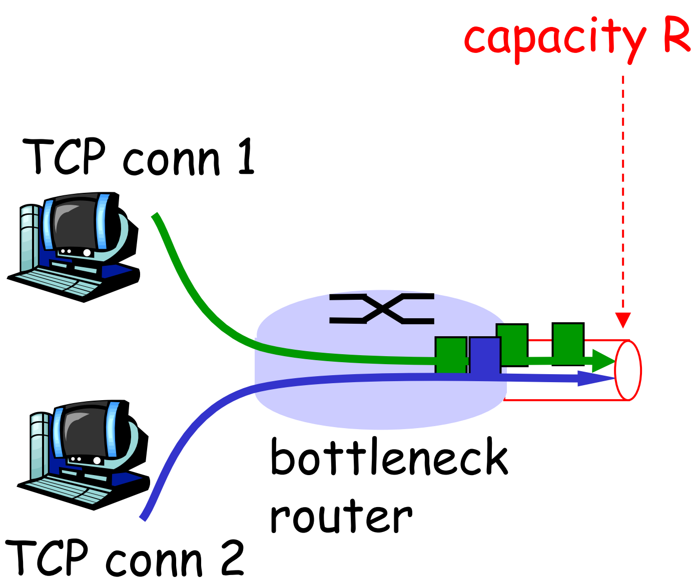
Fairness: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity

Jain's fairness index

n is the number of users sharing the path

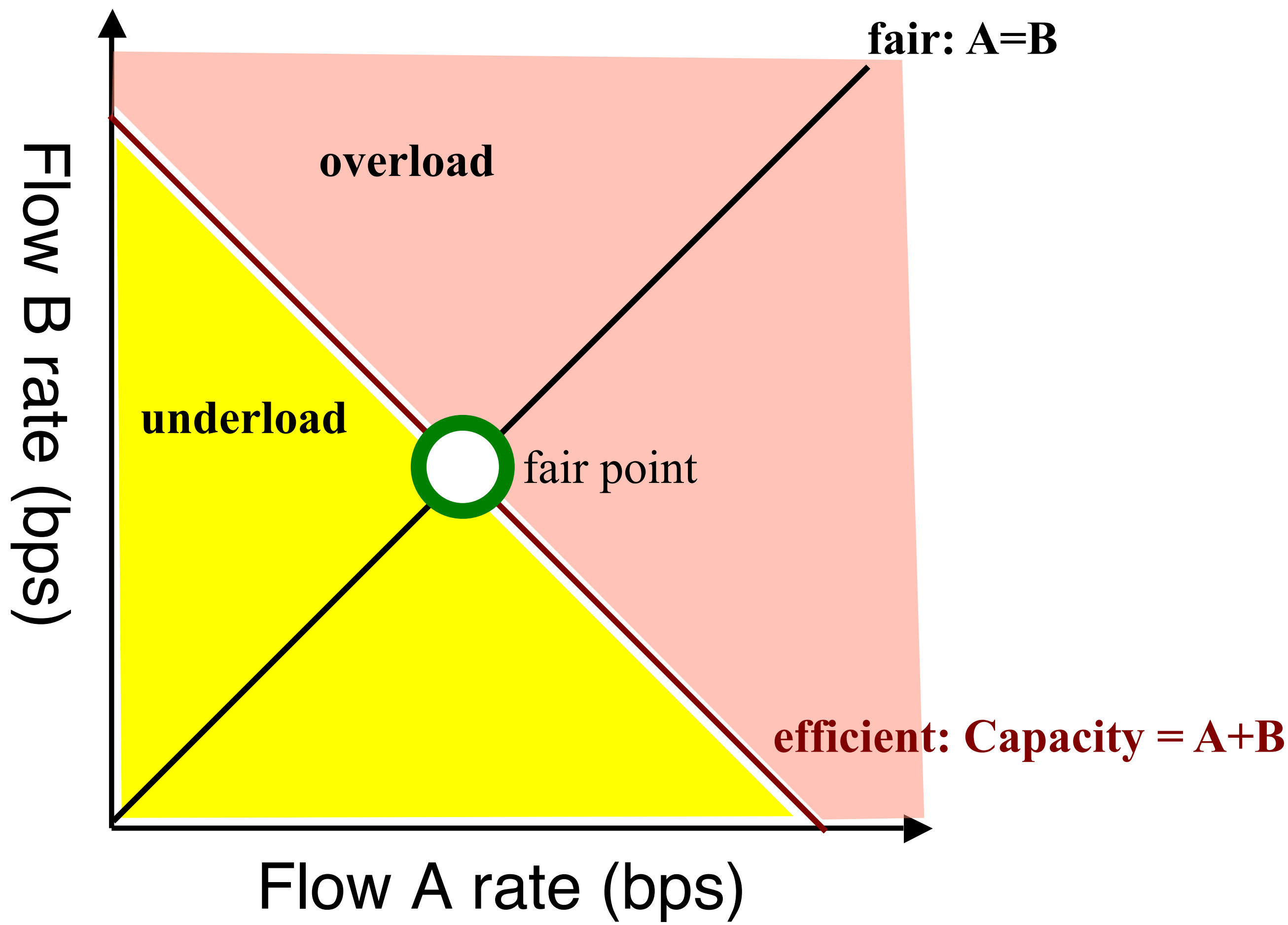
f_i is the network share of i^{th} user

$$F = \frac{(\sum_{i}^{n} f_i)^2}{n \cdot \sum_{i}^{n} f_i^2}$$

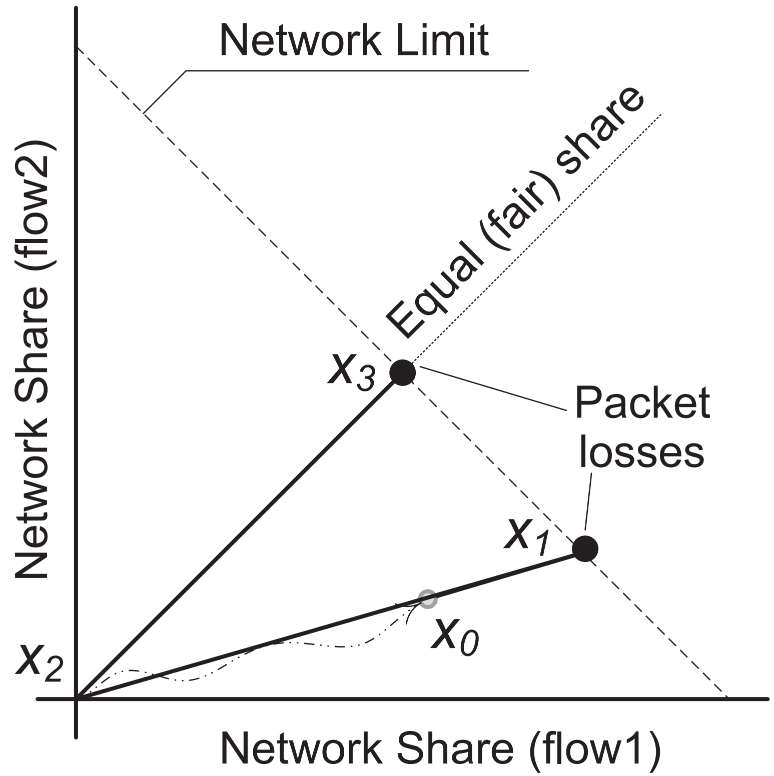


bigger cwnd, larger share TCP flow can "take"

Chiu Jain Phase Plots

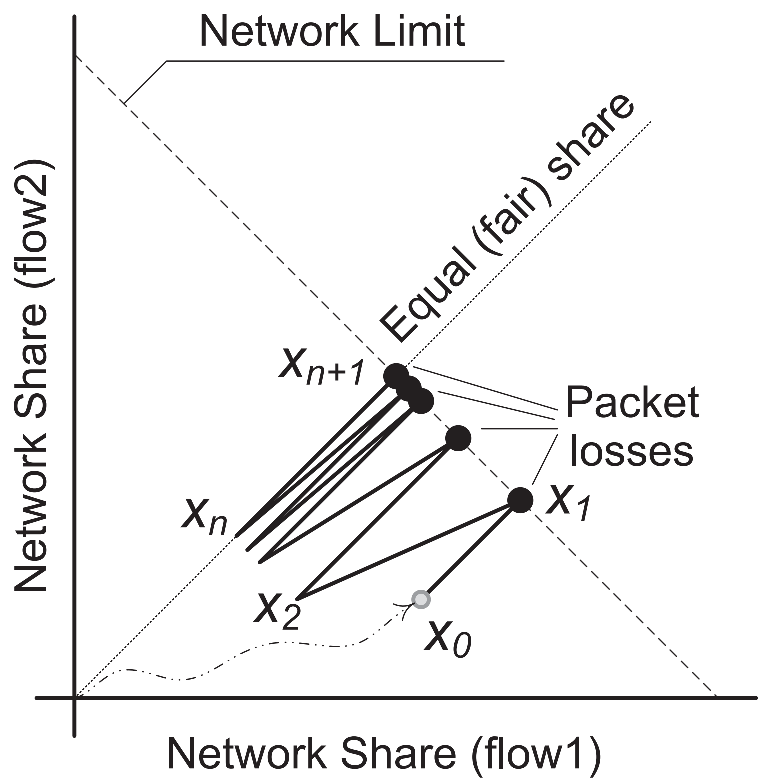


Is Slow Start of TCP Fair?



$x_0 - x_1, \ldots, x_n - x_{n+1}$ multiplicative increase (both flows have the same increase rate of their congestion windows)
$x_1 - x_2$ equalization of the congestion window sizes

Is Congestion Avoidance Is Fair?



$x_0 - x_1, \ldots, x_n - x_{n+1}$ additive increase (both flows have the same increase rate of their congestion windows)
$x_1 - x_2, \ldots, x_{n-1} - x_n$ multiplicative decrease (a flow with the larger congestion window decreases more than a flow with the smaller)

Summary: TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	Received ACK 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)	Received ACK 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 3 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 Throughput

- What's TCP throughout as a function of window size and RTT?
- Ignore slow start: let W = window-size when loss occurs
  - When window is W: throughput = W / RTT
  - Just after loss

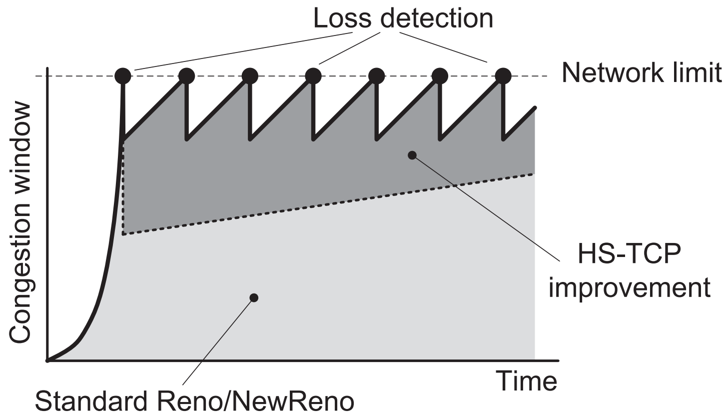
    window → W/2, throughput → W/2RTT
  - Average throughout: 0.75 W/RTT

Why Do We Need Other TCP Variants?

TCP Throughput (Mbps)	RTTs Between Losses	W	P
1	5.5	8.3	0.02
10	55.5	83.3	0.0002
100	555.5	833.3	0.000002
1000	5555.5	8333.3	0.00000002
10000	55555.5	83333.3	0.0000000002

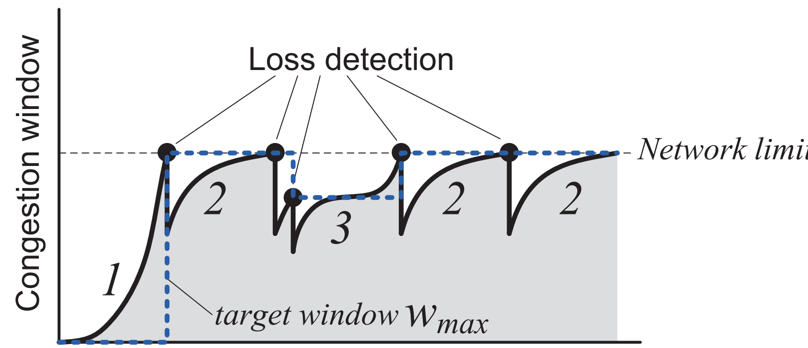
- HS TCP (High-Speed TCP)
  - OS X, available in Linux
- C-TCP (Compound TCP)
  - default on Windows, available in Linux
- CUBIC TCP
  - default in Linux

HS-TCP and CUBIC TCP



HS-TCP

AIMD, but with increased additive increase and decreased multiplicative decrease

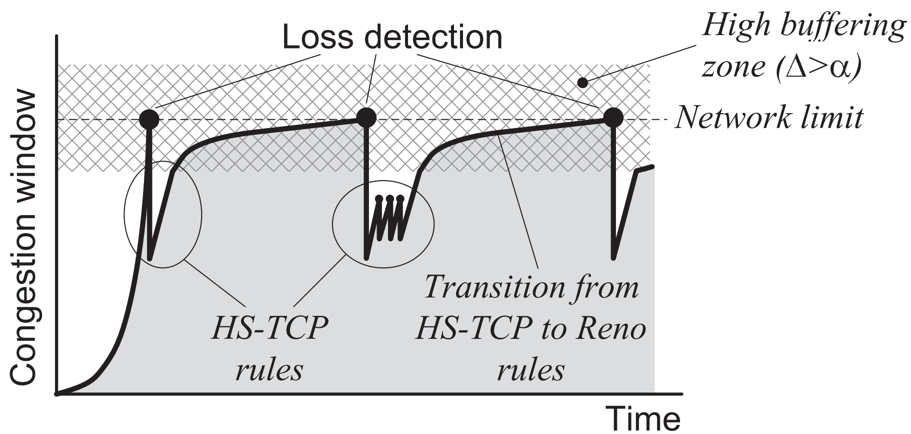


1 – right branch of cubic function
2 – left branch of cubic function
3 – left and right branches of cubic function

CUBIC-TCP

AIMD, but with increased additive increase as a cubic function (fast at first, slow later) and decreased multiplicative decrease

Compound TCP



C-TCP

AIMD with multiple zones of different coefficients, depending on estimated buffering in the network

TCP Variants Timeline