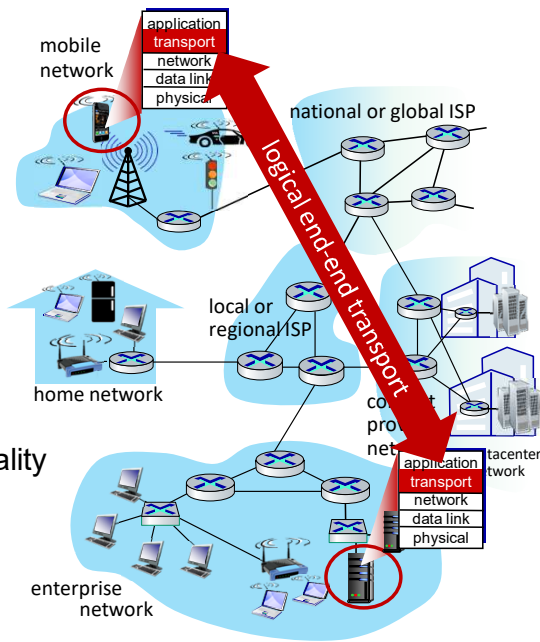


- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



The TL provides a *logical communication* between application processes running on different hosts.

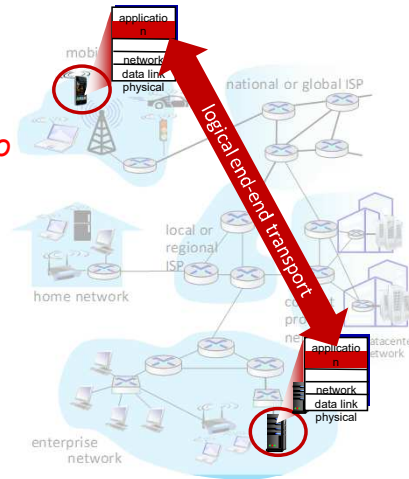
All transport protocols work in an end-to-end way:

- sender: divides application *messages into segments*, passes to network layer
- receiver: reassembles *segments into messages*, passes to application layer

TCP/IP provides two transport protocols for Internet applications

- TCP,
- UDP

Transport layer separates higher layers from lower layers.



Data Link layer:

**physical** communication between (directly connected \* ) *hosts*

Network layer:

**logical** communication between *hosts*

Transport layer:

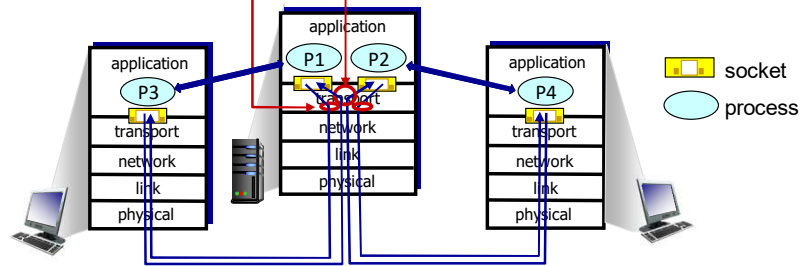
**logical** communication between *processes*

*multiplexing at sender:*

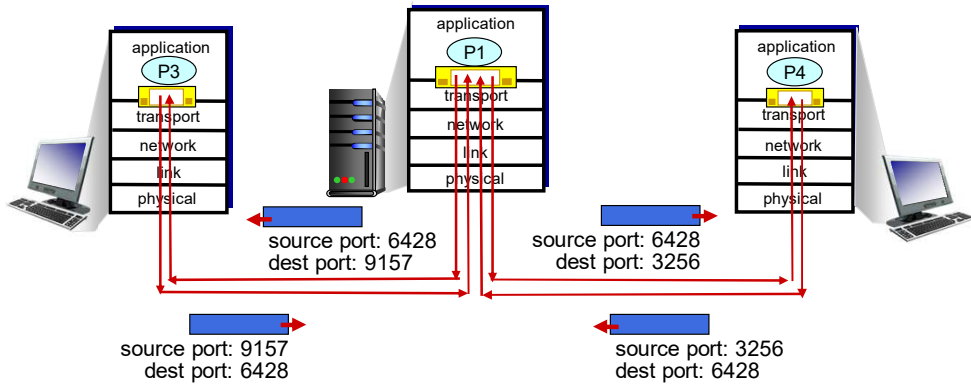
handle data from multiple sockets, add transport header (later used for demultiplexing)

*demultiplexing at receiver:*

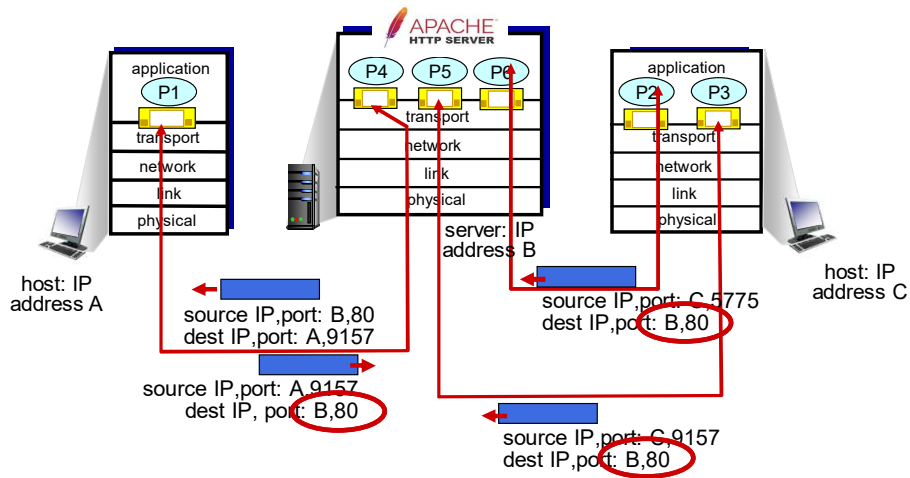
use header info to deliver received segments to correct socket



# Connectionless demultiplexing: an example



## Connection-oriented demultiplexing: example



Three segments, all destined to IP address: B, and dest port: 80 are demultiplexed to *different* sockets

INTERNET STANDARD

RFC 768

J. Postel  
ISI  
28 August 1980

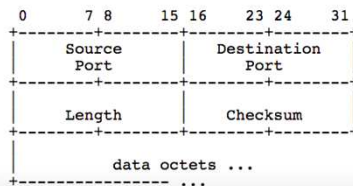
## User Datagram Protocol

### Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

### Format



- “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?**

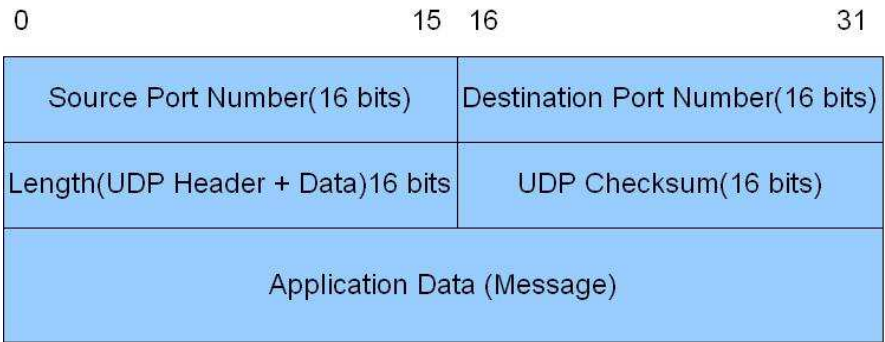
- no connection establishment (which can add RTT delay)
- simple: no connection state at sender or receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

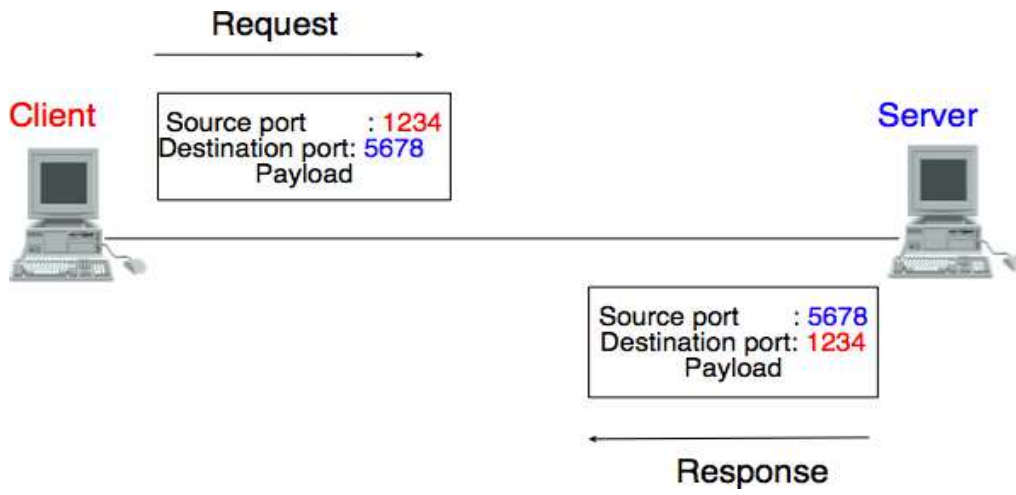
UDP is used by:

- streaming **multimedia** apps (loss tolerant, rate sensitive)
- DNS
- SNMP
- HTTP/3





If **reliable** transfer needed over UDP (e.g., HTTP/3):

- add needed reliability at application layer
- add congestion control at application layer

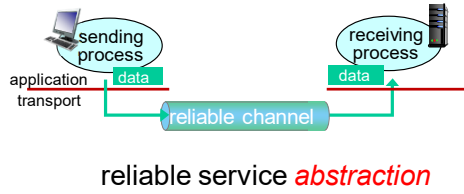




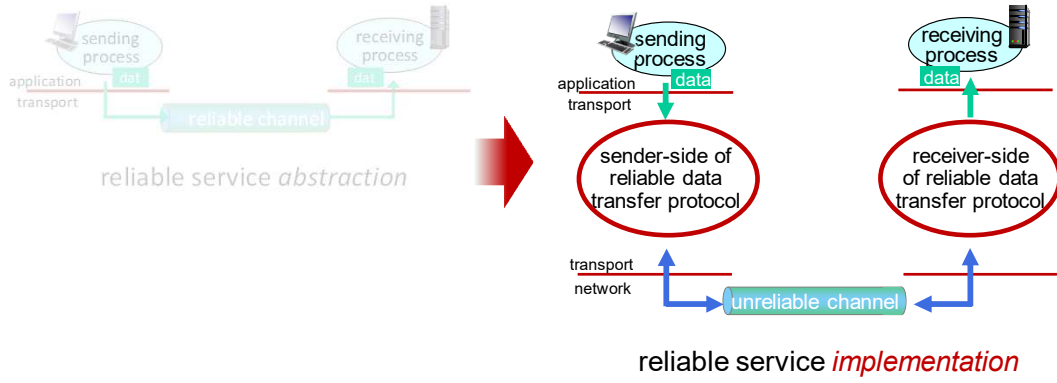
**Goal:** detect errors (*i.e.*, flipped bits) in received segment

	1 <sup>st</sup> number	2 <sup>nd</sup> number	sum
Transmitted:	5	6	11
			
Received:	4	6	11
			
	receiver-computed checksum		sender-computed checksum (as received)
	$\neq$		
			

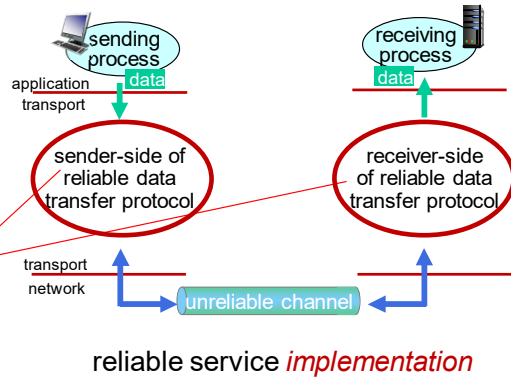
What user wants!



What we have in the real world!

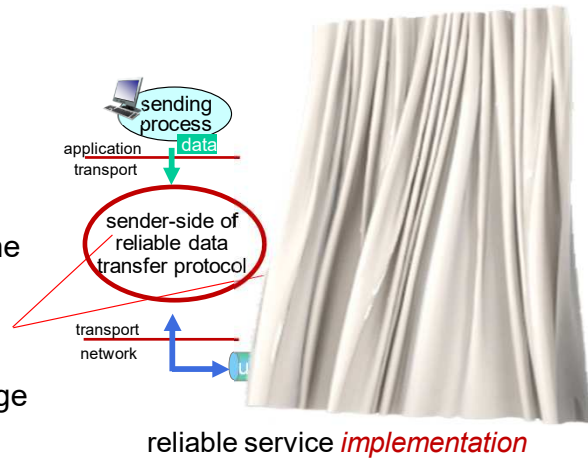


Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



Sender and receiver do not know the "state" of each other, e.g., was a message received?

Unless communicated via a message



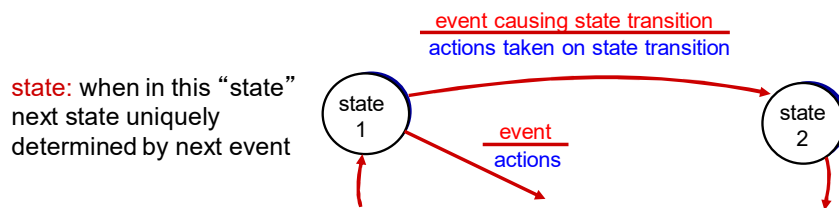
How to obtain a **reliable logical channel** starting from an **unreliable low-level connection**.

	Errors on forward channel	Errors on backward channel	ACK / NAK	Packet loss
RDT 1.0	<b>no</b>	<b>no</b>	-	no
RDT 2.0	<b>yes</b>	no	ACK / NAK	no
RDT 2.1	yes	<b>yes</b>	ACK / NAK	no
RDT 2.2	yes	yes	<b>ACK</b>	no
RDT 3.0	yes	yes	ACK	<b>yes</b>

### We will:

- incrementally develop sender and receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer ( but control info will flow in both directions! )

Use finite state machines (FSM) to specify sender and receiver



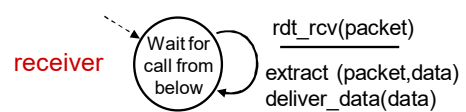
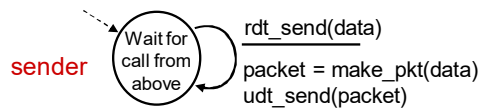
Underlying channel perfectly reliable

- no bit errors
- no loss of packets



**Separate** FSMs for sender and receiver:

- sender sends data into underlying channel
- receiver reads data from underlying channel



Underlying channel may flip bits in packet (no loss!!!)

- use checksum to detect bit errors

How to recover from errors?

*How do humans recover from “errors” during conversation?*

Underlying channel may flip bits in packet

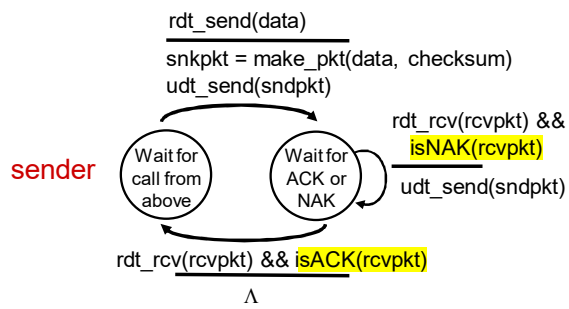
- checksum to detect bit errors

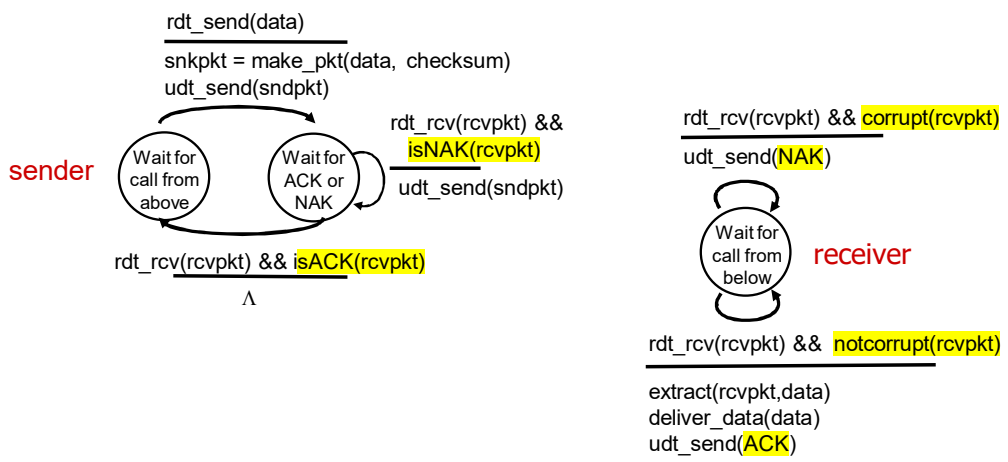
How to recover from errors?

- *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
- *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - sender *retransmits* pkt on receipt of NAK

*stop and wait*

sender sends one packet, then waits for receiver response





Note that sender and receiver must **synchronize** with each other using messages exchange.

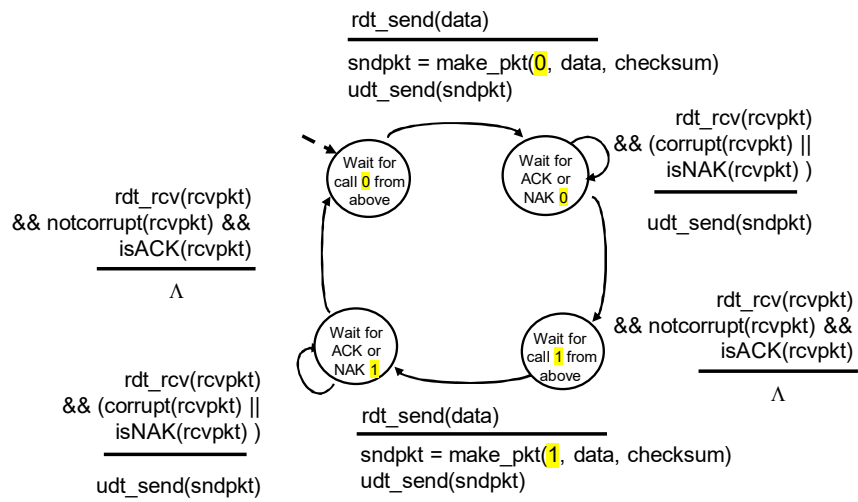
**what happens if ACK/NAK arrives corrupted?**

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate!

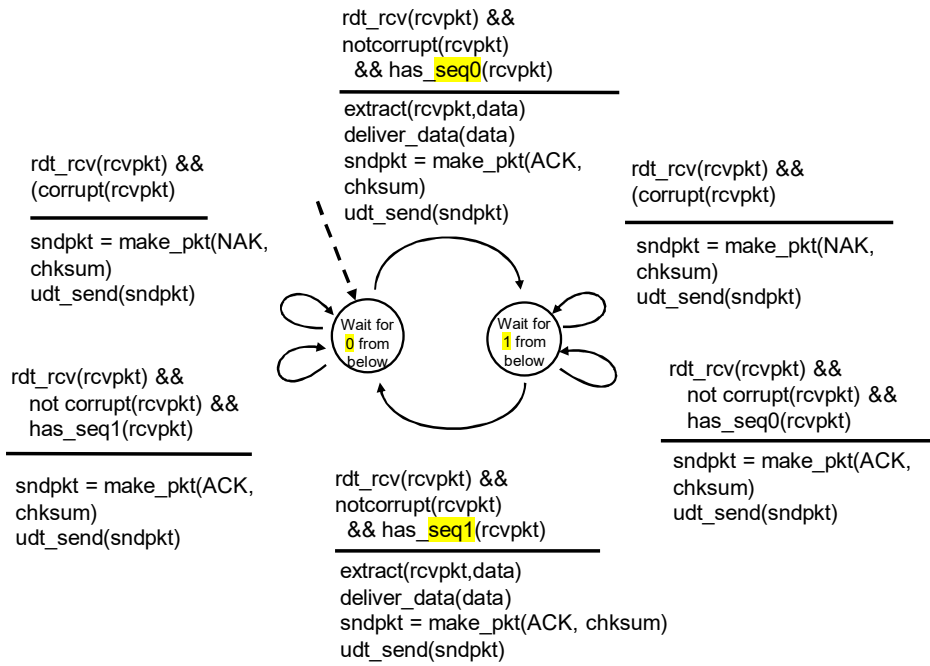
**handling duplicates:**

- sender retransmits current pkt if ACK/NAK is corrupted
- sender adds a *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkts

## rdt 2.1: sender, handling garbled ACK/NAKs



## rdt 2.1: receiver, handling garbled ACK/NAKs



sender:

- seq # added to pkt
- two seq. #s (0,1) will suffice.  
Why?
- must check if received  
ACK/NAK corrupted
- twice as many states
  - state must “remember”  
whether “expected” pkt  
should have seq # of 0 or 1

receiver:

- must check if received  
packet is duplicate
  - state indicates whether 0  
or 1 is expected pkt seq #
- note: receiver can *not* know  
if its last ACK/NAK arrived  
correctly to the sender

### Are NAK necessary?

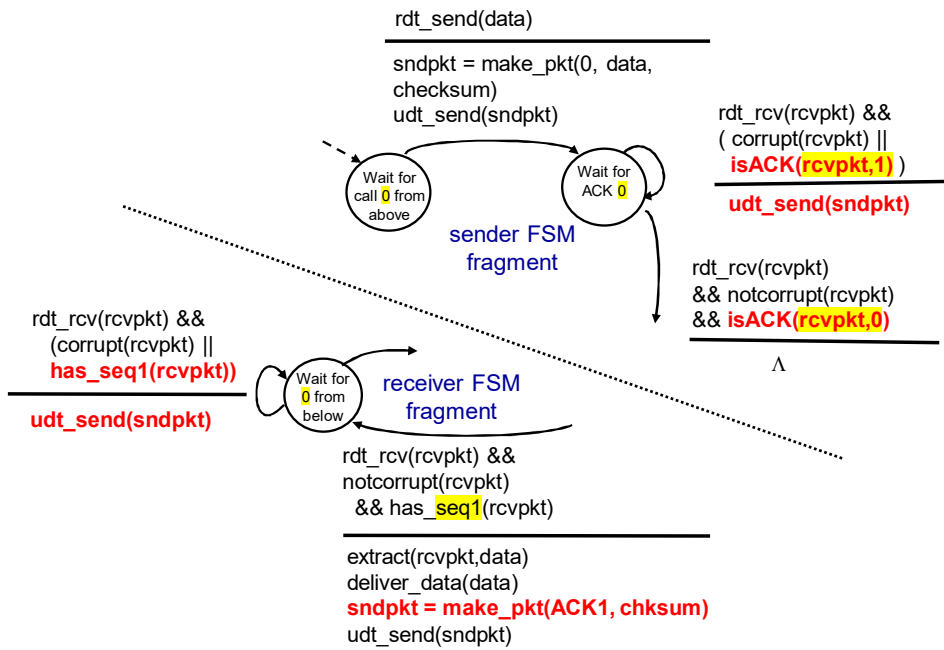
We can design another protocol with the same functionality as rdt 2.1, using ACKs only.

Instead of NAK, receiver **sends ACK for last pkt received correctly**

- receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

As we will see, TCP uses this approach to be NAK-free

## rdt 2.2: sender, receiver fragments



*New channel assumption:* underlying channel can also *lose* packets (data, ACKs)

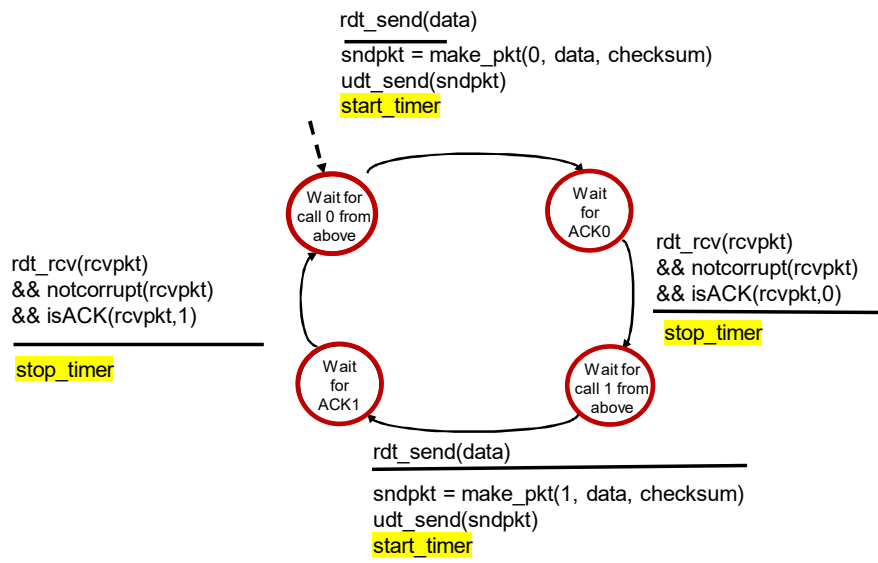
- checksum, sequence #s, ACKs, retransmissions will be of help ... but not quite enough

**Q:** How do *humans* handle lost sender-to-receiver words in conversation?

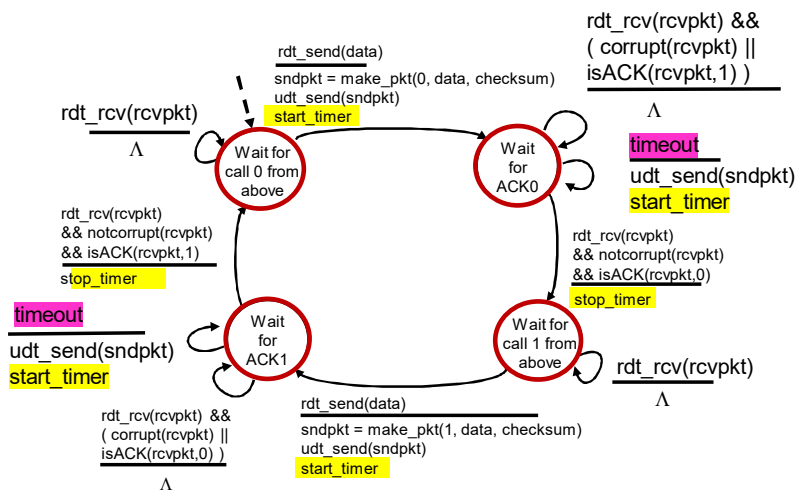
**Approach:** sender waits “reasonable” amount of time for ACK

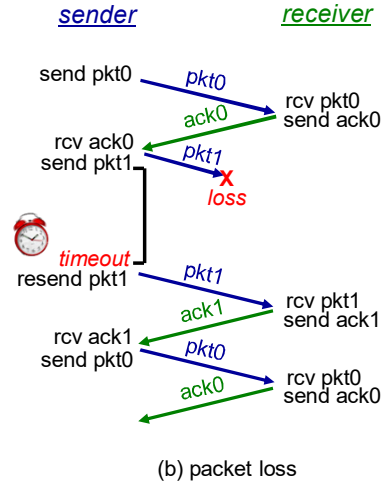
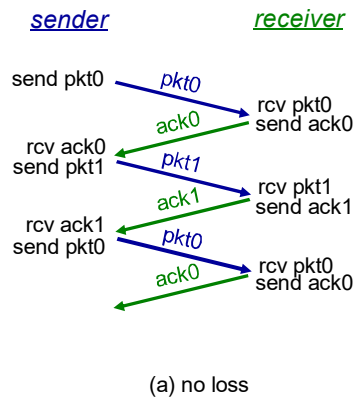
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
  
- use countdown timer to interrupt after “reasonable” amount of time

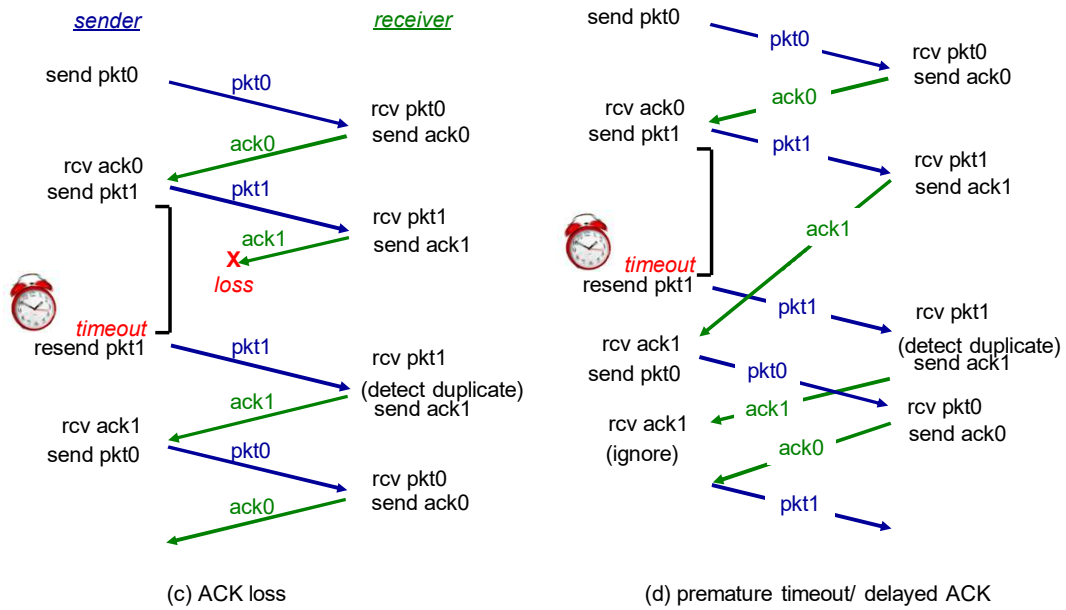




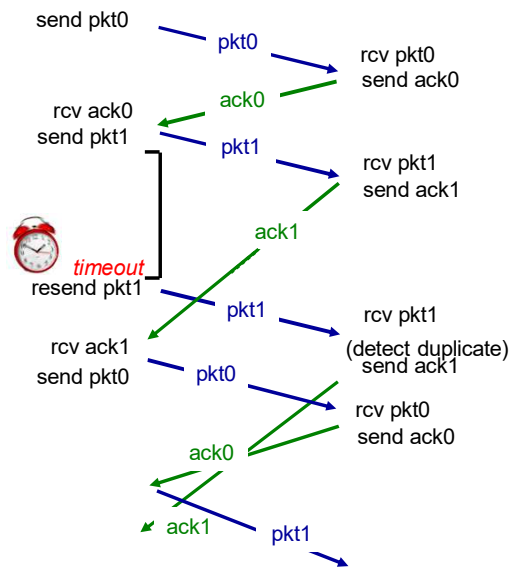
# rdt 3.0 sender



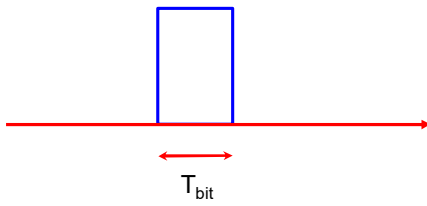




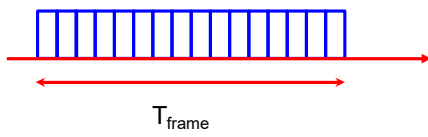
Is the protocol correct?



Protocol failure!



BW = 10 Mbps =  $10^7$  bps  
in 1 s  $\Rightarrow$  10.000.000 bits  
1 bit  $\Rightarrow$   $T_{bit} = 0.1 \mu\text{s}$



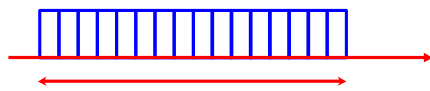
BW = 10 Mbps  
Frame of 1500 bytes = 12000 bits  
Time to generate a frame:  $T_{frame} = 1200 \mu\text{s}$



1 km channel

$v \approx 200.000 \text{ km/s}$

Signal propagation time:  $T_{\text{propagation}} \approx 5 \mu\text{s}$



BW = 10 Mbps

in  $5 \mu\text{s} \Rightarrow 50 \text{ bits}$

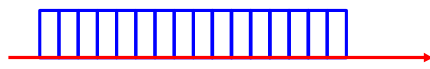
N° of bits simultaneously present in the channel



10 Mbps - 1 km

$T_{\text{propagation}} \approx 5 \mu\text{s}$

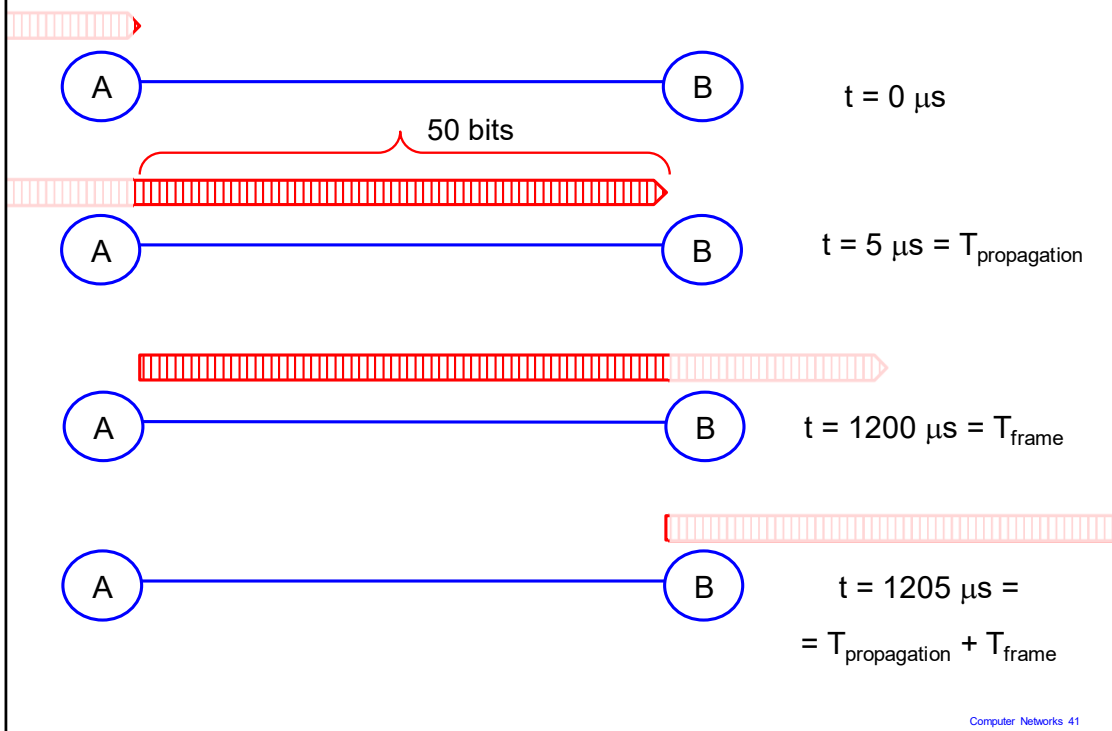
Frame 1500 bytes = 12000 bits



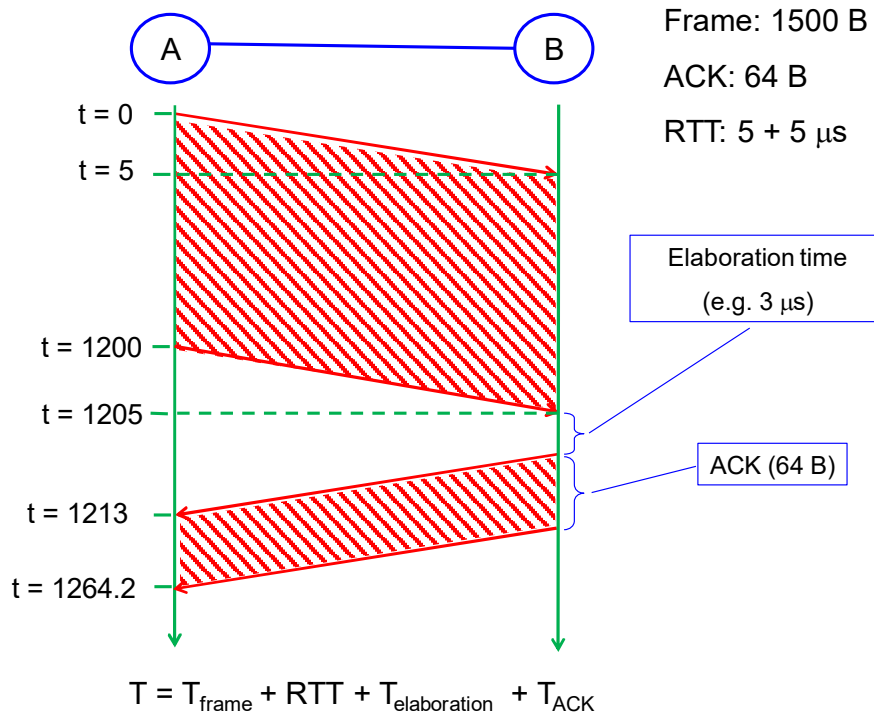
T

$T_{\text{transfer}} = 1200 \mu\text{s} + 5 \mu\text{s}$

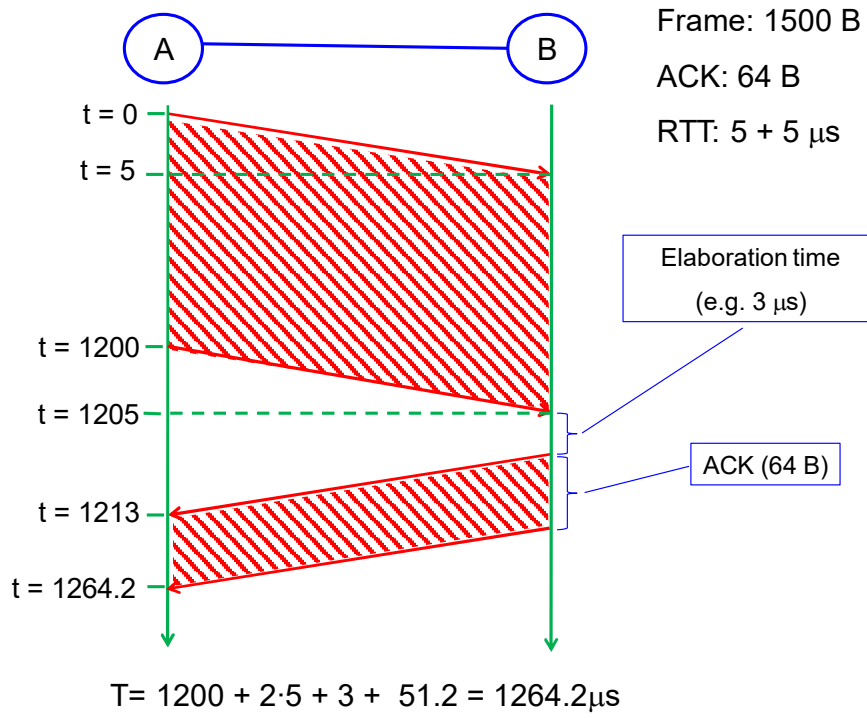
Time needed to completely transfer the frame from node A to node B

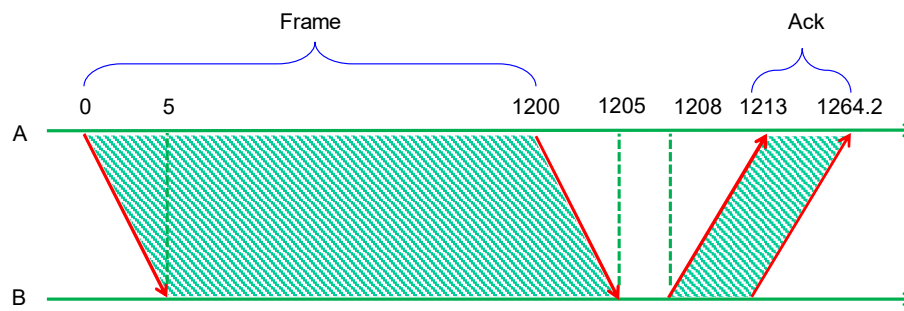


# Actual Throughput



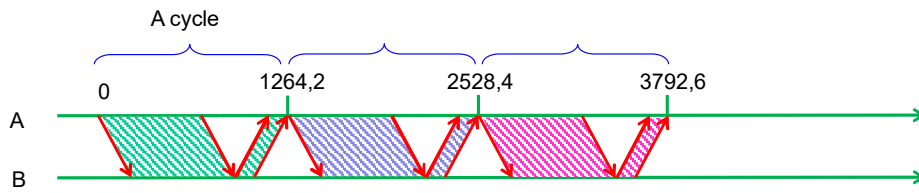
# Actual Throughput





1500 bytes sent in 1264.2  $\mu\text{s}$

$$\begin{aligned}
 \text{Actual Throughput} &= 1500 \cdot 8 / 1264.2 \cdot 10^{-6} \text{ [b/s]} \\
 &= 9.492 \cdot 10^6 \text{ bps} \\
 &= 9.492 \text{ Mbps}
 \end{aligned}$$



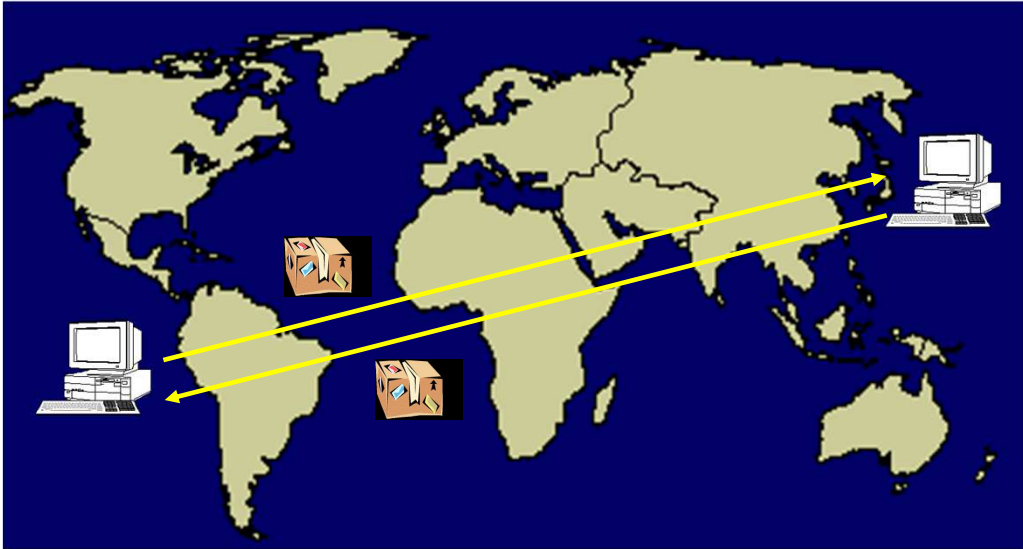
Frame: 1500 B

Ack: 64B

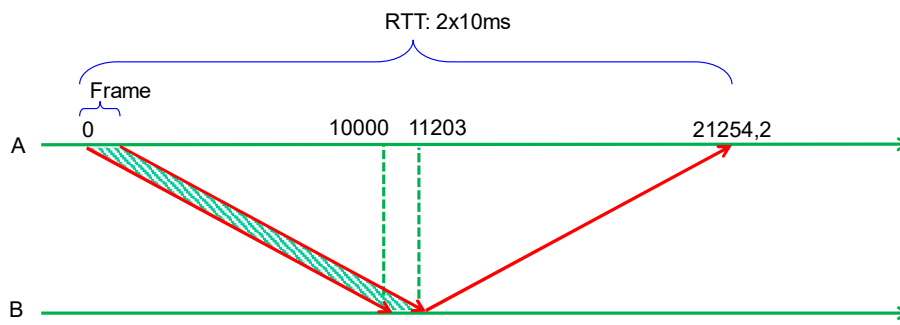
BW: 10Mbps

$T_{\text{actual}}$  : 9.492 Mbps

## Stop and wait protocol



# Actual Throughput



Frame: 1500 B

Ack: 64B

BW: 10Mbps

RTT: 20ms

$$T = T_{\text{frame}} + \text{RTT} + T_{\text{elaboration}} + T_{\text{ACK}}$$

$$T(\text{in } \mu\text{s}) = 1200 + 2 \cdot 10000 + 3 + 51.2 = 21254.2 \mu\text{s}$$

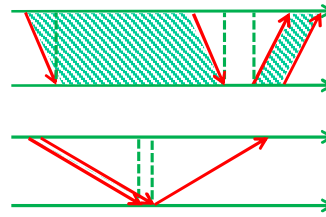
Frame: 1500 B

Ack: 64B

BW: 10Mbps










RTT: 10  $\mu$ s      T = 1264.2  $\mu$ s    9.492 Mbps

RTT: 20 ms      T = 21254.2  $\mu$ s    0.565 Mbps = **565 Kbps**












```
$ ping www.google.com
PING www.google.com (216.58.205.36) 56(84) bytes of data.
64 bytes from mil07s19-in-f4.1e100.net (216.58.205.36):
icmp_seq=1 ttl=117 time=19.2 ms
64 bytes from mil07s19-in-f4.1e100.net (216.58.205.36):
icmp_seq=2 ttl=117 time=19.2 ms
64 bytes from mil07s19-in-f4.1e100.net (216.58.205.36):
icmp_seq=3 ttl=117 time=19.4 ms
64 bytes from mil07s19-in-f4.1e100.net (216.58.205.36):
icmp_seq=4 ttl=117 time=19.3 ms
...
--- www.google.com ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5006ms
rtt min/avg/max/mdev = 19.213/19.270/19.412/0.066 ms
```

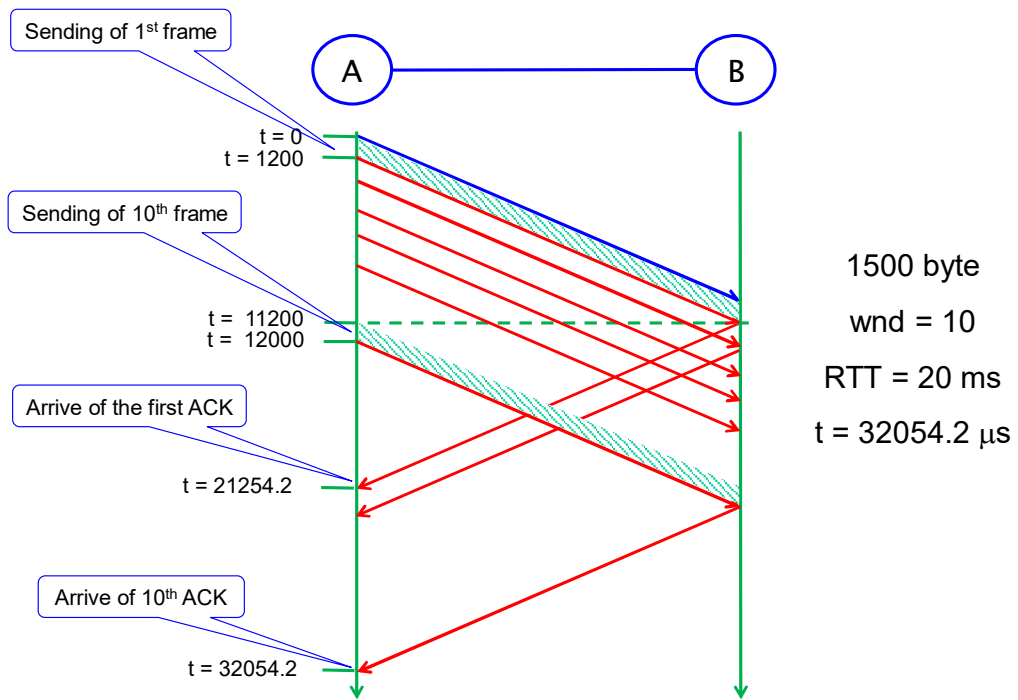
## Geolocation data from IP2Location (Product: DB6, 2024-3-1)

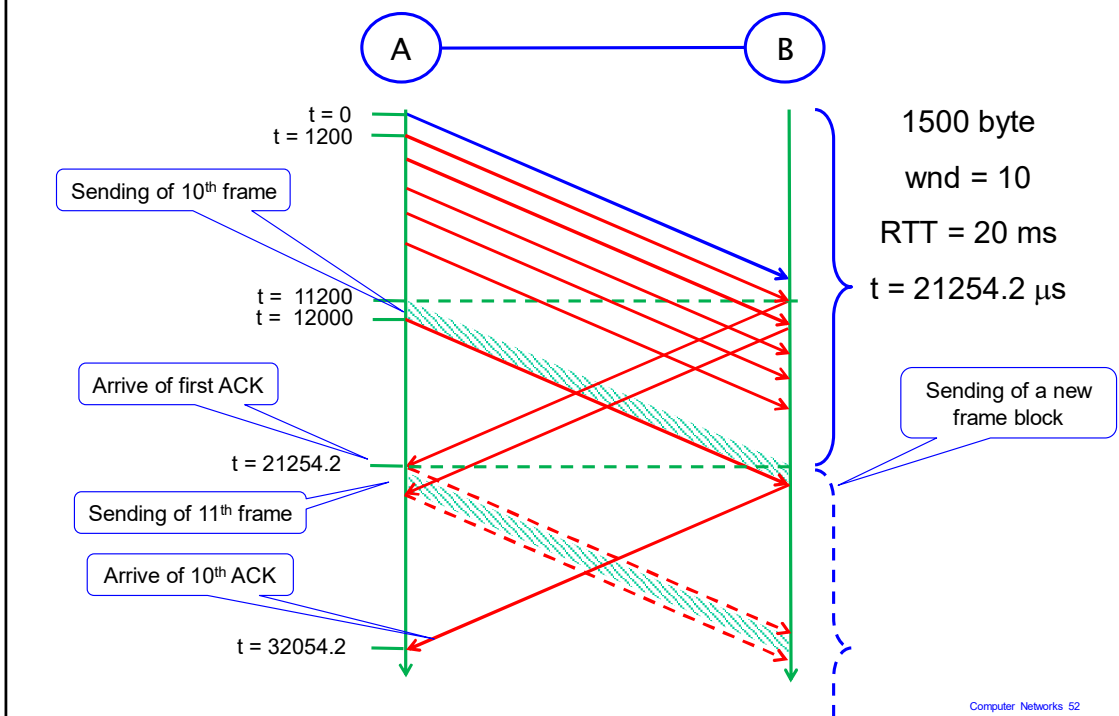
 <b>IP ADDRESS:</b> 216.58.205.36	 <b>ISP:</b> Google LLC
 <b>COUNTRY:</b> United States 	 <b>ORGANIZATION:</b> Not available
 <b>REGION:</b> California	 <b>LATITUDE:</b> 37.4060
 <b>CITY:</b> Mountain View	 <b>LONGITUDE:</b> -122.0785

## Geolocation data from DB-IP (Product: API, real-time)

 <b>IP ADDRESS:</b> 216.58.205.36	 <b>ISP:</b> Google LLC
 <b>COUNTRY:</b> Italy 	 <b>ORGANIZATION:</b> Google LLC
 <b>REGION:</b> Lombardy	 <b>LATITUDE:</b> 45.4642
 <b>CITY:</b> Milan	 <b>LONGITUDE:</b> 9.18998

# Pipeline

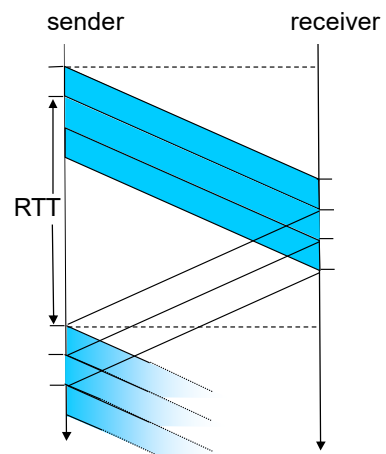




Frame: 1500 B

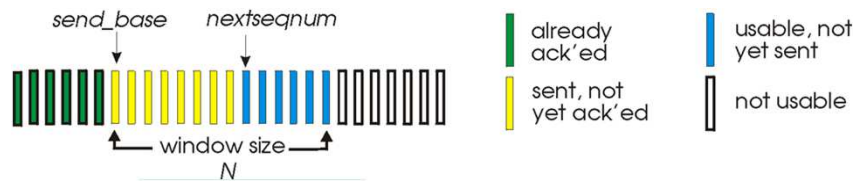
Ack: 64B

BW: 10Mbps



RTT 10 $\mu$ s	wnd= 1	t = 1264.2 $\mu$ s	9.477 Mbps
RTT 20 ms	wnd= 1	t = 21254.2 $\mu$ s	0.565 Mbps
RTT 20 ms	wnd=10	t = 21254.2 $\mu$ s	5.646 Mbps

- sender: “window” of up to  $N$ , consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

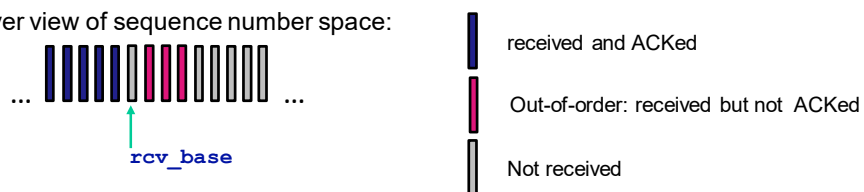


- **cumulative ACK:**  $ACK(n)$ : ACKs all packets up to (*including seq #  $n$* )
  - on receiving  $ACK(n)$ : move window forward to begin at  $n+1$
- timer for oldest in-flight packet
- **timeout( $n$ ):** retransmit packet  $n$  and all higher seq # packets in window

ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



# Go back N

sender window (N=4)

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

sender

send pkt0  
 send pkt1  
 send pkt2  
 send pkt3  
 (wait)

rcv ack0, send pkt4  
 rcv ack1, send pkt5

ignore duplicate ACK



pkt 2 timeout

send pkt2  
 send pkt3  
 send pkt4  
 send pkt5

receiver

receive pkt0, send ack0  
 receive pkt1, send ack1

receive pkt3, discard,  
 (re)send ack1

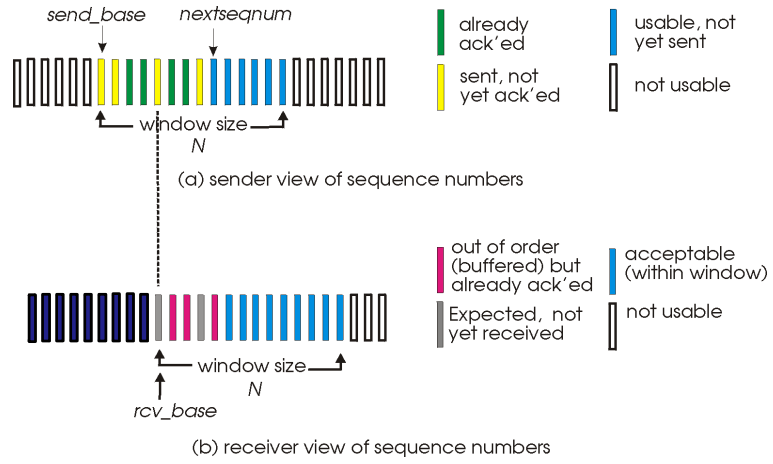
receive pkt4, discard,  
 (re)send ack1

receive pkt5, discard,  
 (re)send ack1

rcv pkt2, deliver, send ack2  
 rcv pkt3, deliver, send ack3  
 rcv pkt4, deliver, send ack4  
 rcv pkt5, deliver, send ack5

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - $N$  consecutive seq #s
  - limits seq #s of sent, unACKed packets

## Selective repeat: sender, receiver windows



## sender

## data from above:

- if next available seq # in window, send packet

timeout( $n$ ):

- resend packet  $n$ , restart timer

ACK( $n$ ) in [sendbase, sendbase+N]:

- mark packet  $n$  as received
- if  $n$  smallest unACKed packet, advance window base to next unACKed seq #

## receiver

packet  $n$  in [rcvbase, rcvbase+N-1]

- send ACK( $n$ )
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

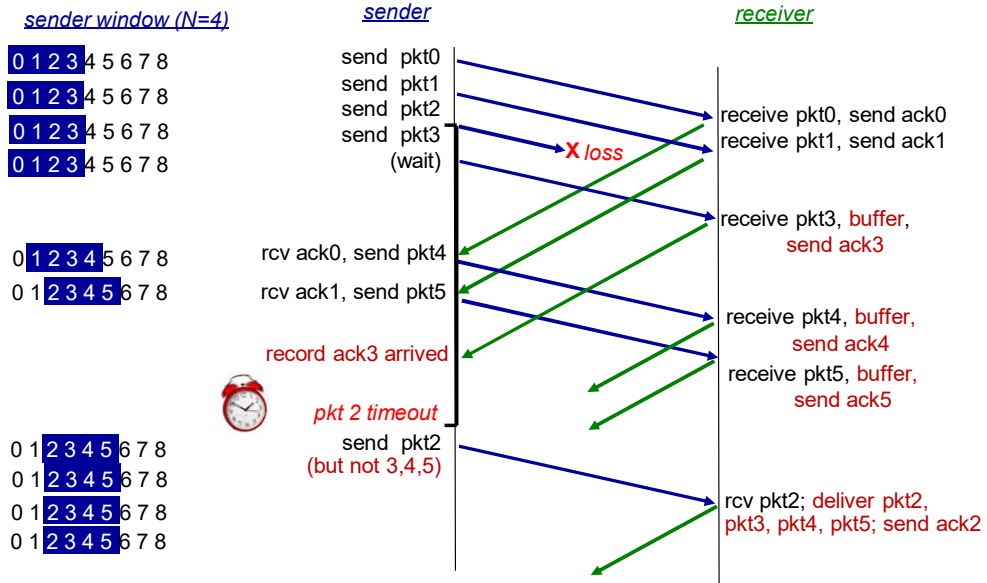
packet  $n$  in [rcvbase-N, rcvbase-1]

- ACK( $n$ )

## otherwise:

- ignore

# Selective repeat

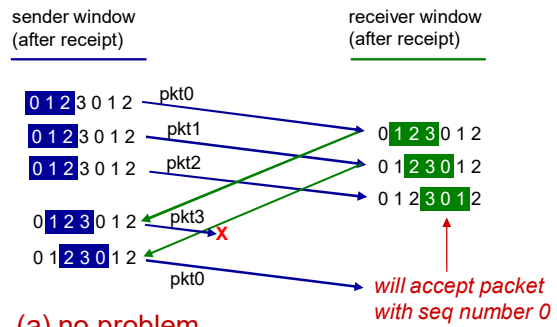


Q: what happens when ack2 arrives?

# Selective repeat: a dilemma!

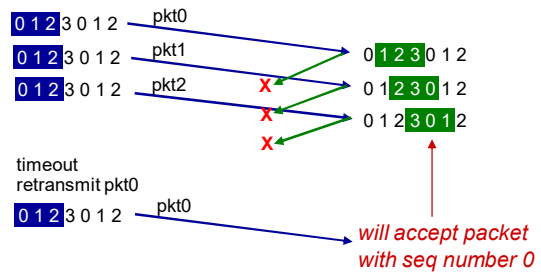
example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

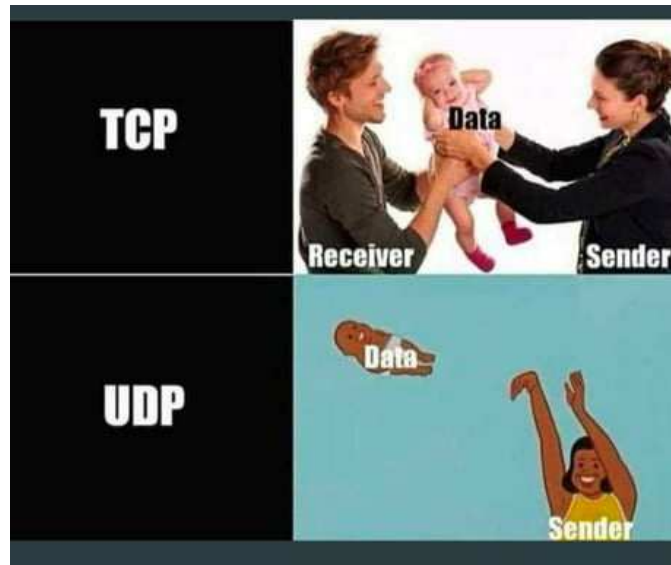


(a) no problem

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?



(b) oops!



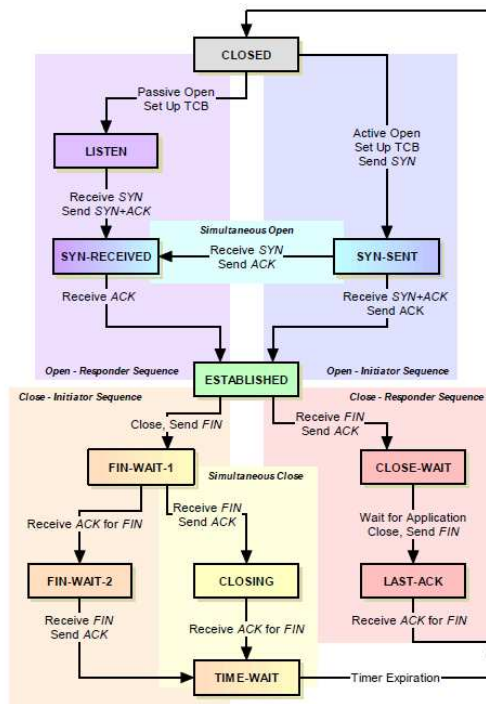
- December 1970: NCP (Network Control Program)
- December 1974: TCP (v1) (Transmission Control *Program*, RFC 675)
- March 1977: TCPv2
- Spring 1978: TCPv3 / IPv3 (Transmission Control *Protocol*)
- September 1981: TCPv4 / IPv4 (RFC 793)

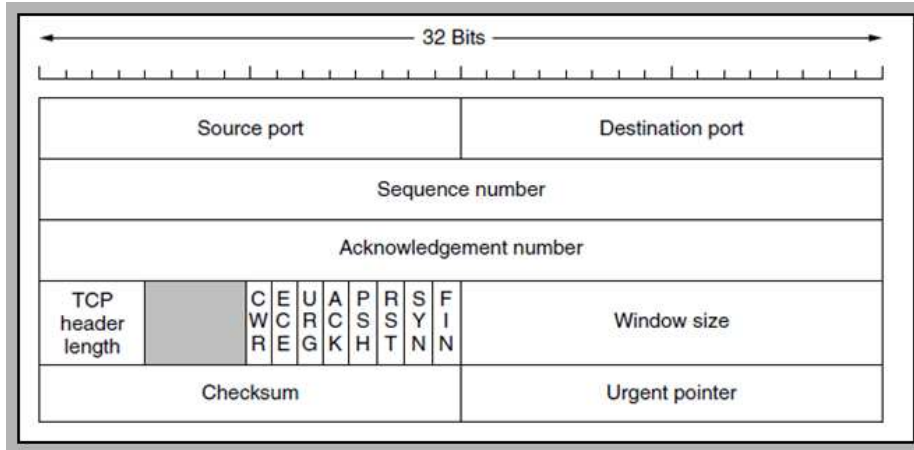
- Addressing/Multiplexing
- Connection Establishment, Management and Termination
- Data Handling and Packaging
- Data Transfer
- Providing Reliability and Transmission Quality Services
- Providing Flow Control and Congestion Avoidance Features

- Specifying Application Use
- Providing Security
- Maintaining Message Boundaries
- Guaranteeing Communication

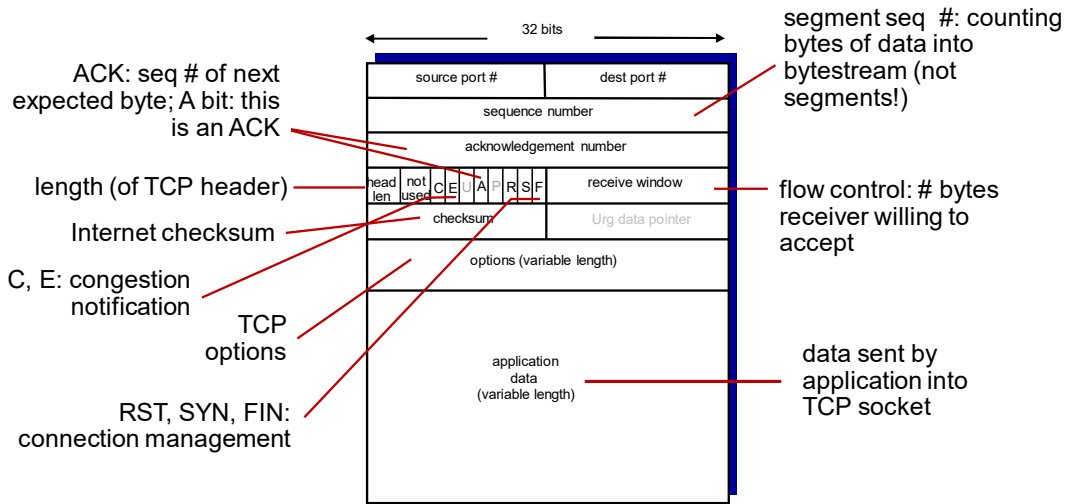
- Connection-Oriented
- Bidirectional
- Multiply-Connected and Endpoint-Identified
- Reliable
- Acknowledged
- Stream-Oriented
- Data-Unstructured
- Data-Flow-Managed

# TCP state model





# TCP segment structure



**Sequence numbers:**

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

**Acknowledgements:**

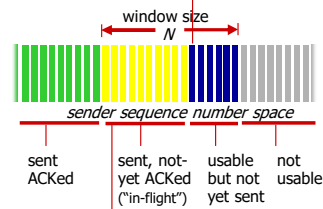
- seq # of next byte expected from other side
- cumulative ACK

**Q:** how receiver handles out-of-order segments

**A:** TCP spec doesn’t say, - up to implementor

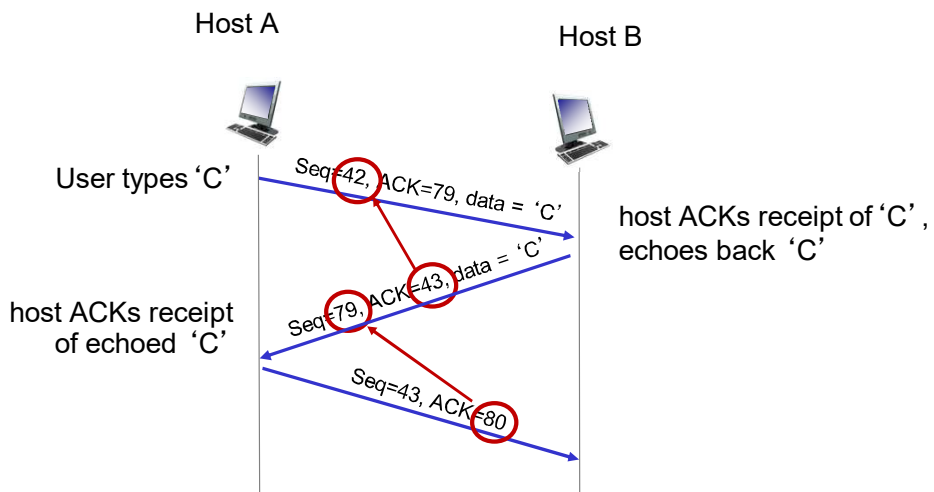
outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
rwnd	
checksum	urg pointer



outgoing segment from receiver

source port #	dest port #
sequence number	
acknowledgement number	
A	rwnd
checksum	urg pointer



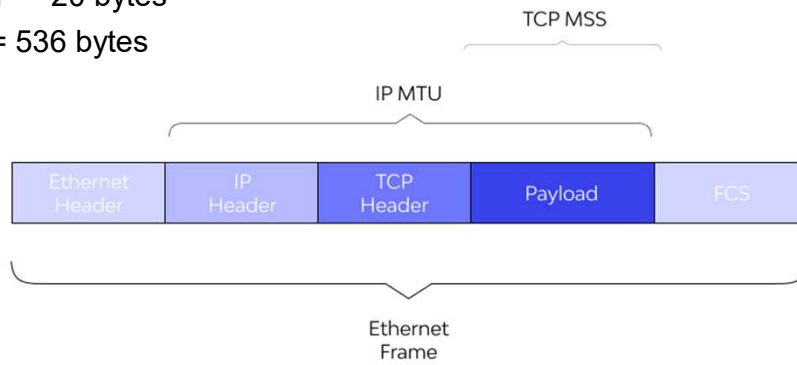
simple telnet scenario

**MSS:** Maximum Segment Size (of TCP payload).

**MTU:** Maximum Transmission Unit

TCP requires all networks to transfer IP packets with at least 576 bytes, that means MSS of 536 bytes.

- Header IP = 20 bytes
- Header TCP = 20 bytes
- TCP Data = 536 bytes



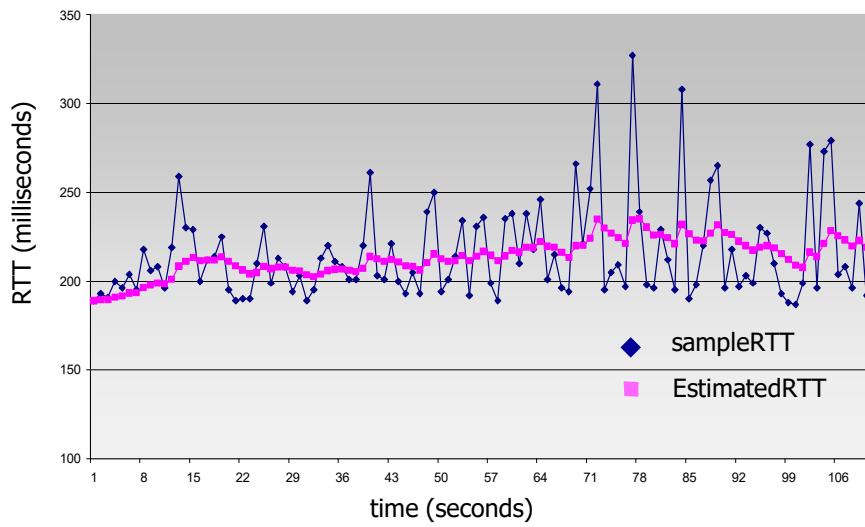
Q: how to set TCP timeout value?

- 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 (Karn's algorithm)
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **SampleRTT**

## Estimated RTT e Sample RTT



A correct timer value can improve performance, avoiding congestion.

TCP estimates the RTT using:

- **SampleRTTs**:
- **EWMA** (Exponential weighted moving average): the influence of past sample decreases exponentially fast

$$\text{Estimated\_RTT}_n = (1-\alpha) \cdot \text{Estimated\_RTT}_{n-1} + \alpha \cdot \text{Sample\_RTT}_n$$

Typical value  $\alpha = 0.125$

$$E\_RTT_{n-2} = (1-\alpha) E\_RTT_{n-3} + \alpha S\_RTT_{n-2}$$

$$E\_RTT_{n-1} = (1-\alpha) E\_RTT_{n-2} + \alpha S\_RTT_{n-1}$$

$$E\_RTT_n = (1-\alpha) E\_RTT_{n-1} + \alpha S\_RTT_n \quad \alpha = 0.125 \quad 1-\alpha = 0.875$$

$$E\_RTT_n = (1-\alpha) ((1-\alpha) E\_RTT_{n-2} + \alpha S\_RTT_{n-1}) + \alpha S\_RTT_n$$

$$E\_RTT_n = (1-\alpha)^2 E\_RTT_{n-2} + \alpha (1-\alpha) S\_RTT_{n-1} + \alpha S\_RTT_n$$

$$E\_RTT_n = (1-\alpha)^2 ((1-\alpha) E\_RTT_{n-3} + \alpha S\_RTT_{n-2}) + \alpha (1-\alpha) \cdot S\_RTT_{n-1} + \alpha \cdot S\_RTT_n$$

$$E\_RTT_n = (1-\alpha)^3 E\_RTT_{n-3} + \alpha (1-\alpha)^2 \cdot S\_RTT_{n-2} + \alpha (1-\alpha) \cdot S\_RTT_{n-1} + \alpha \cdot S\_RTT_n$$

$$E\_RTT_n = 0,6699 E\_RTT_{n-3} + 0,0957 S\_RTT_{n-2} + 0,1094 S\_RTT_{n-1} + 0,125 S\_RTT_n$$

(third to last)                      (second to last)                      (last)

SampleRTT can be subject to large variation.

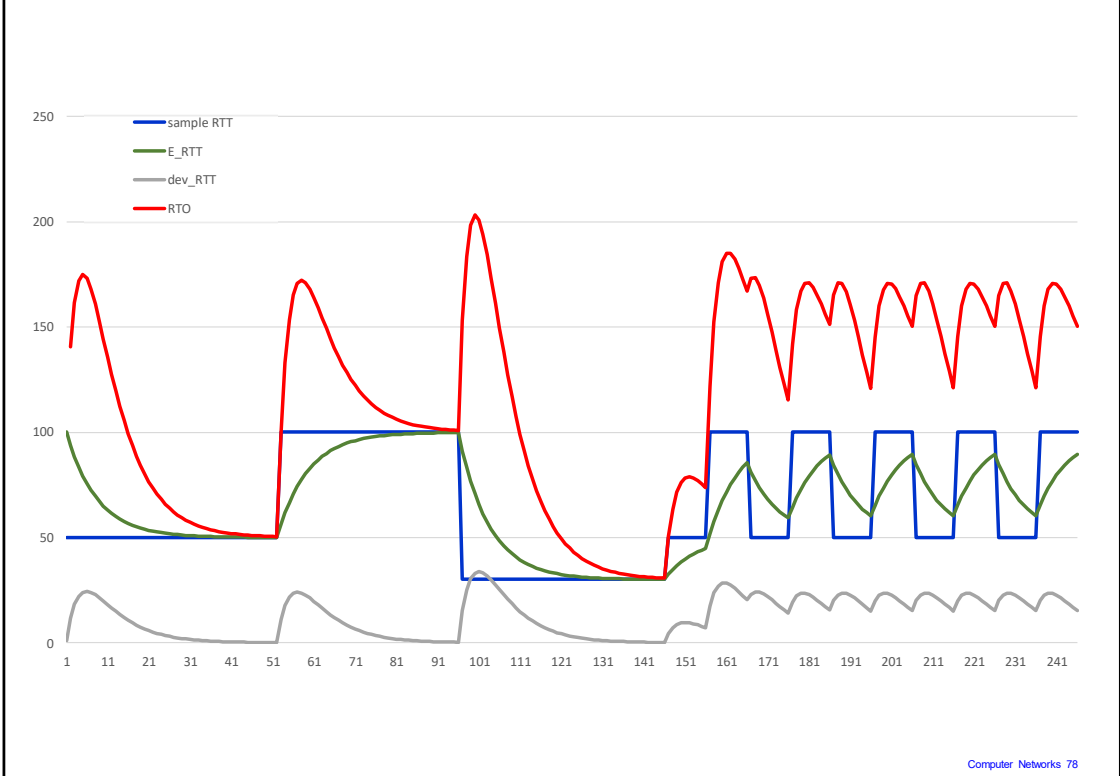
$$\text{DevRTT} = (1-\beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

Typical value:  $\beta = 0.25$

$$\text{RTO} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

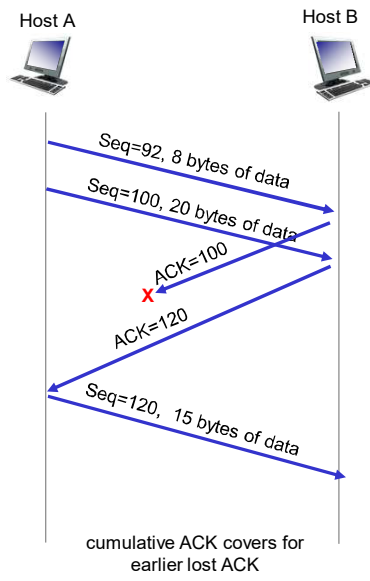
The initial recommended value for RTO is 1s (RFC 6298)

After a timer expiration, the value of RTO will be doubled.





# Timer



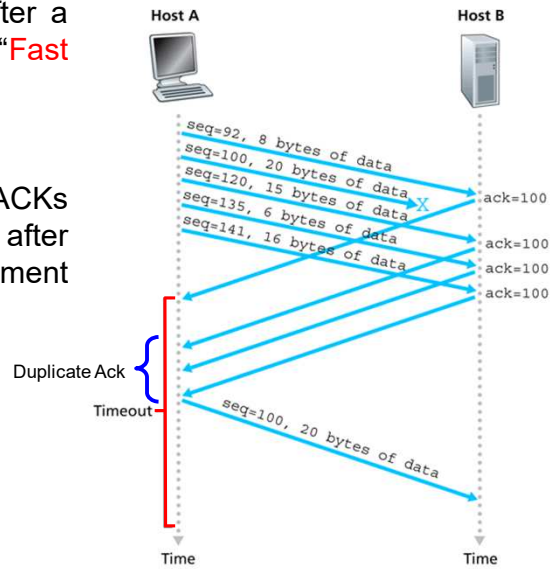
```
loop (forever) {
  switch(event)
  event: data received from application above
    create TCP segment with sequence number NextSeqNum
    if (timer currently not running)
      start timer
    pass segment to IP
    NextSeqNum=NextSeqNum+length(data)
    break;

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

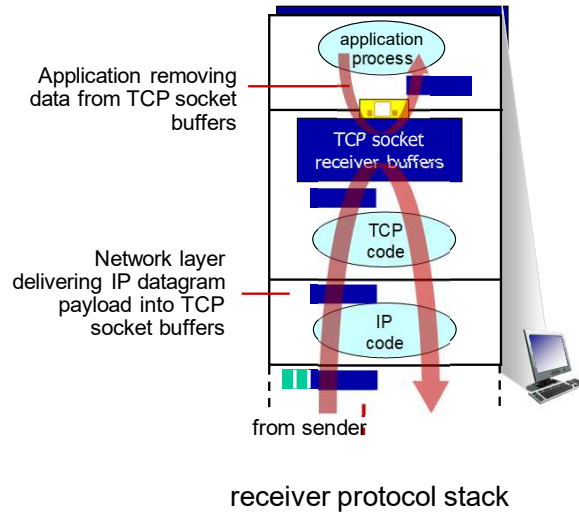
  event: ACK received, with ACK field value of y
    if (y > SendBase) {
      SendBase = y
      if (there are currently any not-yet-acknowledged segments)
        start timer
    }
    break;
} /* end of loop forever */
```

To avoid long waiting time after a packet loss TCP uses the “**Fast Retransmit**”.

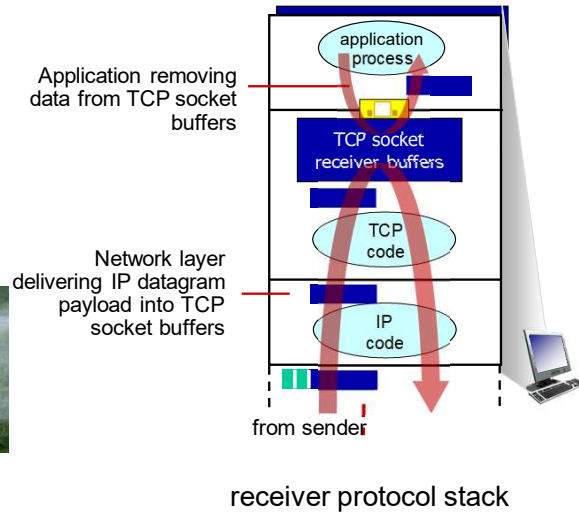
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



**Q:** What happens if network layer delivers data faster than application layer removes data from socket buffers?



**Q:** What happens if network layer delivers data faster than application layer removes data from socket buffers?



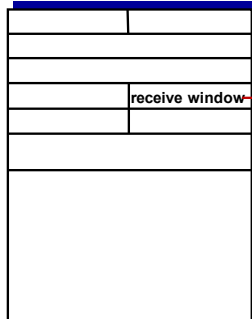
TCP



UDP

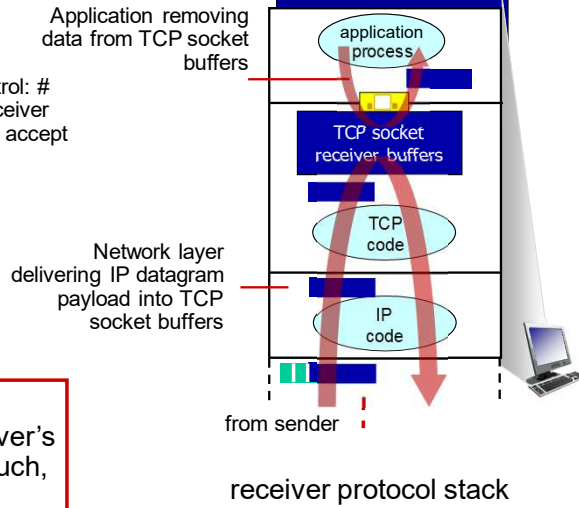


**Q:** What happens if network layer delivers data faster than application layer removes data from socket buffers?

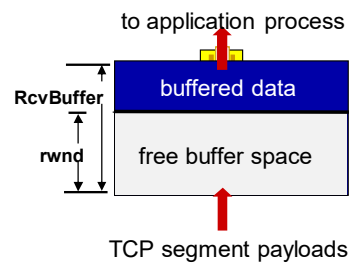


flow control: # bytes receiver willing to accept

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



- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

Telnet sends packets with minimum size (1 byte), wasting bandwidth.

The algorithm of **Nagle** tries to reduce this overhead, buffering the data to send.

```
if available_data > 0 then
  if window_size ≥ MSS & available_data ≥ MSS then
    send_a_MSS_segment
  else
    if waiting_for_an_ack == true then
      enqueue_data /* until an acknowledge is received */
    else
      send_data
    end if
  end if
end if
```

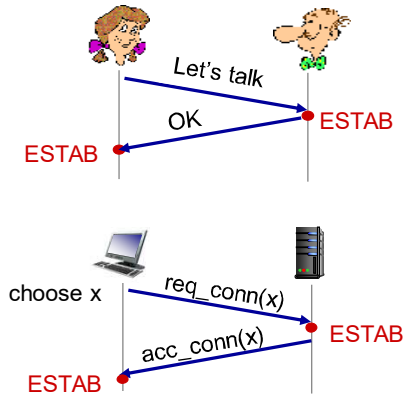
In networks with low RTT, Nagle's algorithm sends small packets with high frequency.

In networks with high RTT, data are bufferized and sent in large packets.

```
if available_data > 0 then
  if window_size ≥ MSS & available_data ≥ MSS then
    send_a_MSS_segment
  else
    if waiting_for_an_ack == true then
      enqueue_data /* until an acknowledge is received */
    else
      send_data
    end if
  end if
end if
```

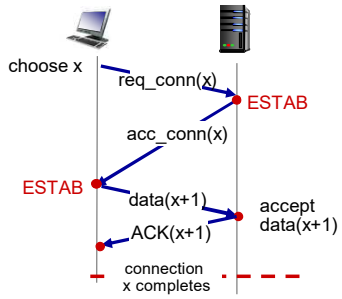
Sometime, to obtain strong reactivity, the O.S. disables the algorithm.

2-way handshake:



Q: will 2-way handshake always work in network?

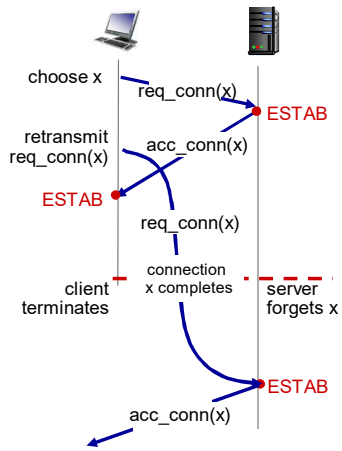
- variable delays
- retransmitted messages (e.g. req\_conn(x)) due to message loss
- message reordering
- can't "see" other side



No problem!

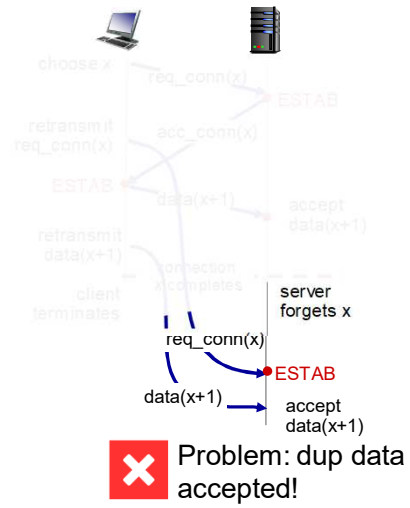


## 2-way handshake scenarios

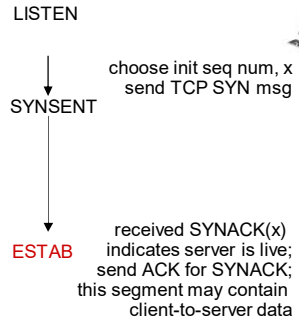


Problem: half open connection! (no client)

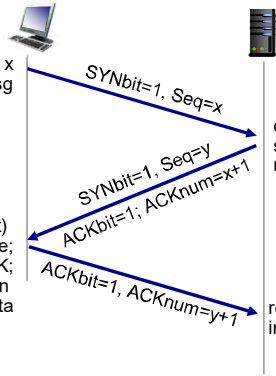
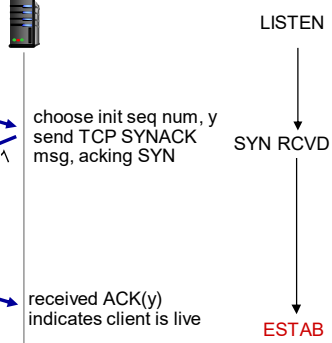
## 2-way handshake scenarios



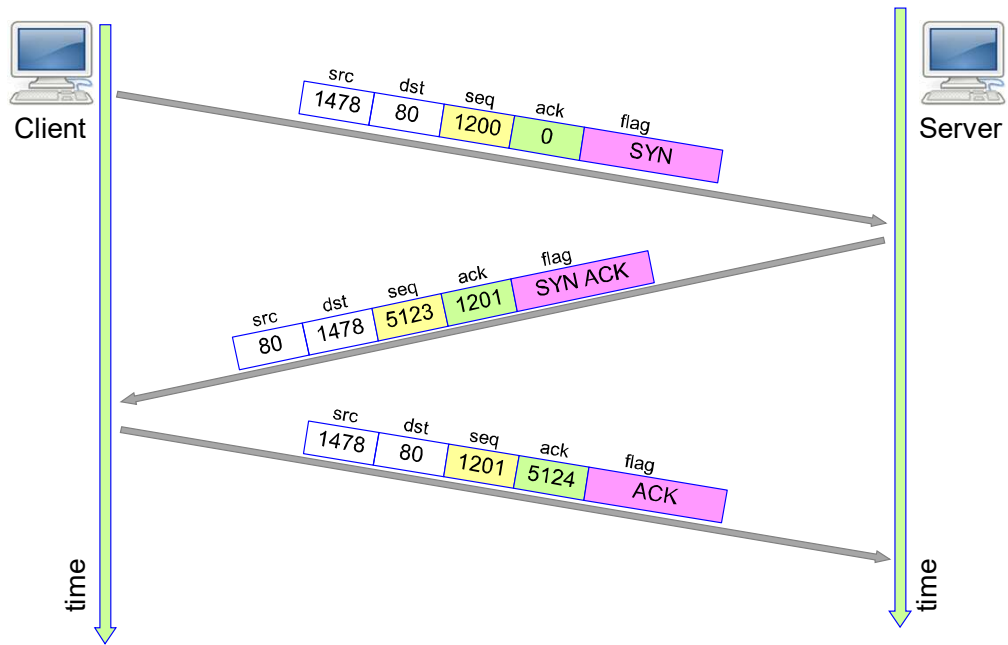
## Client state



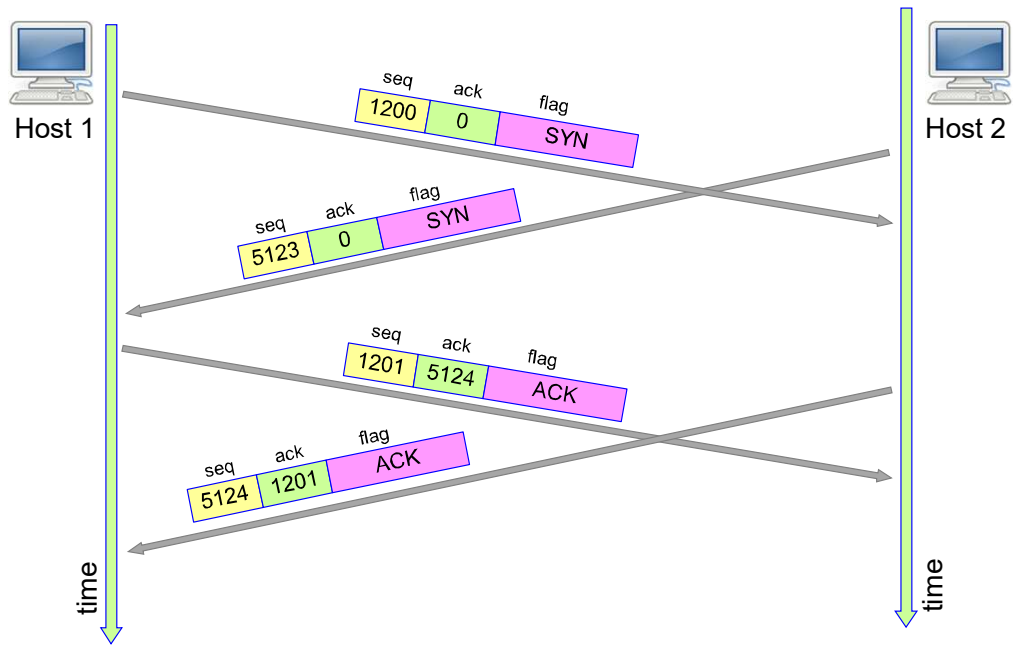
## Server state



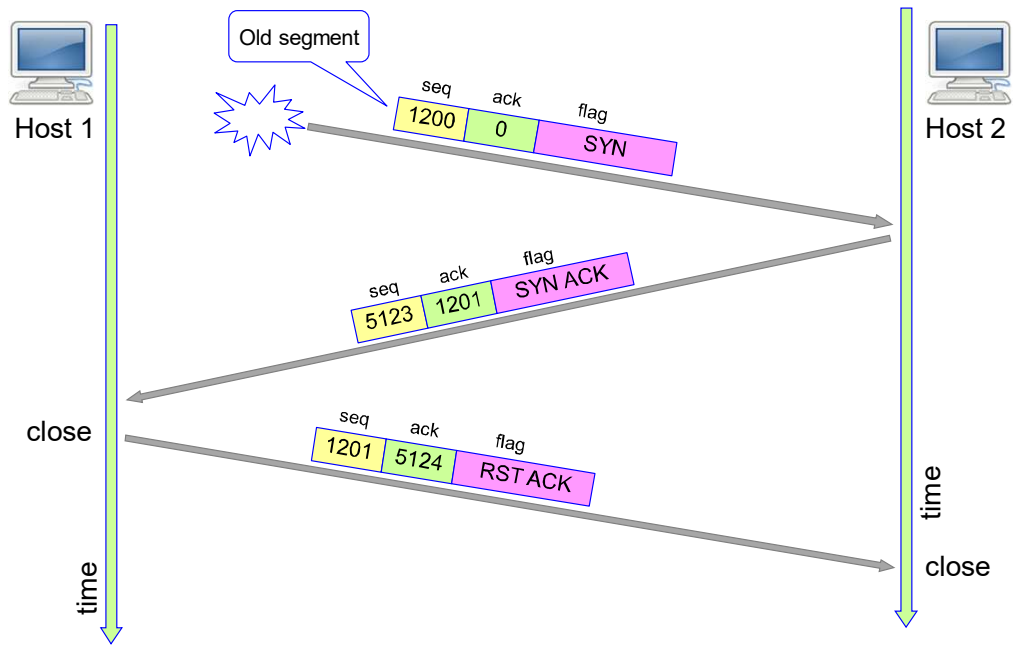
# TCP 3-way handshake



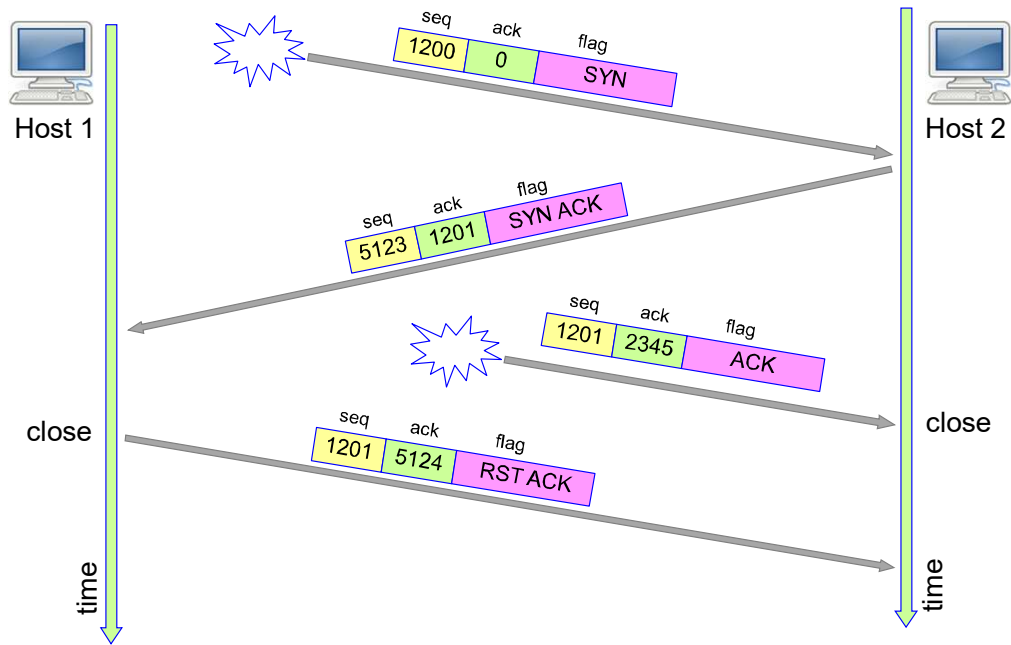
# TCP 3-way handshake



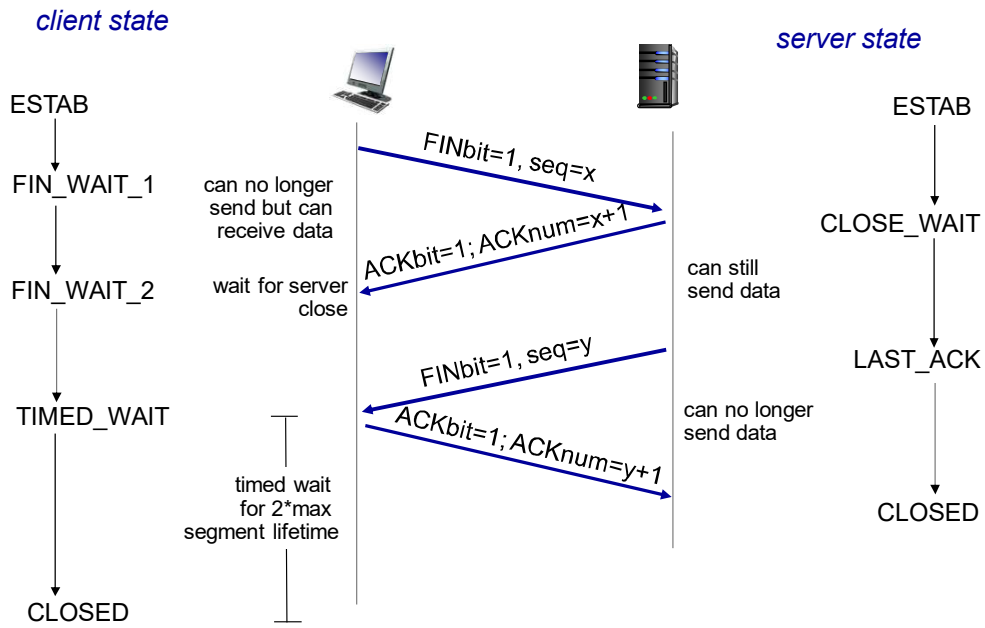
# TCP 3-way handshake



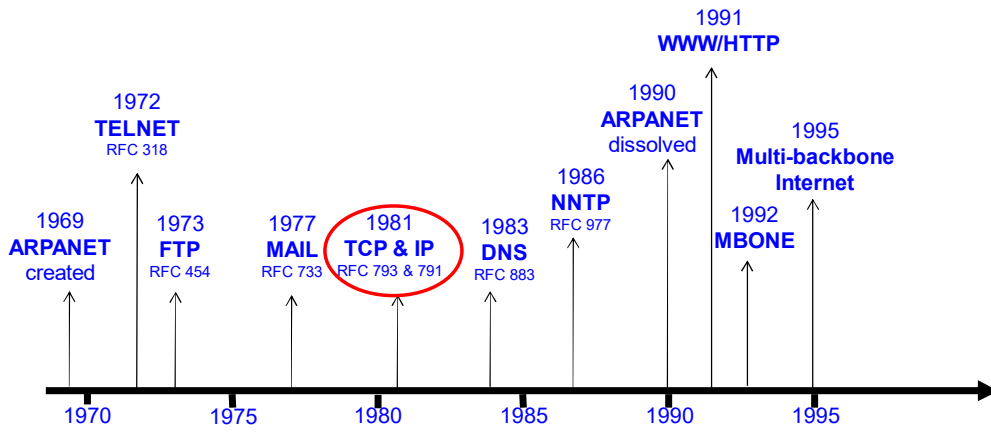
# TCP 3-way handshake



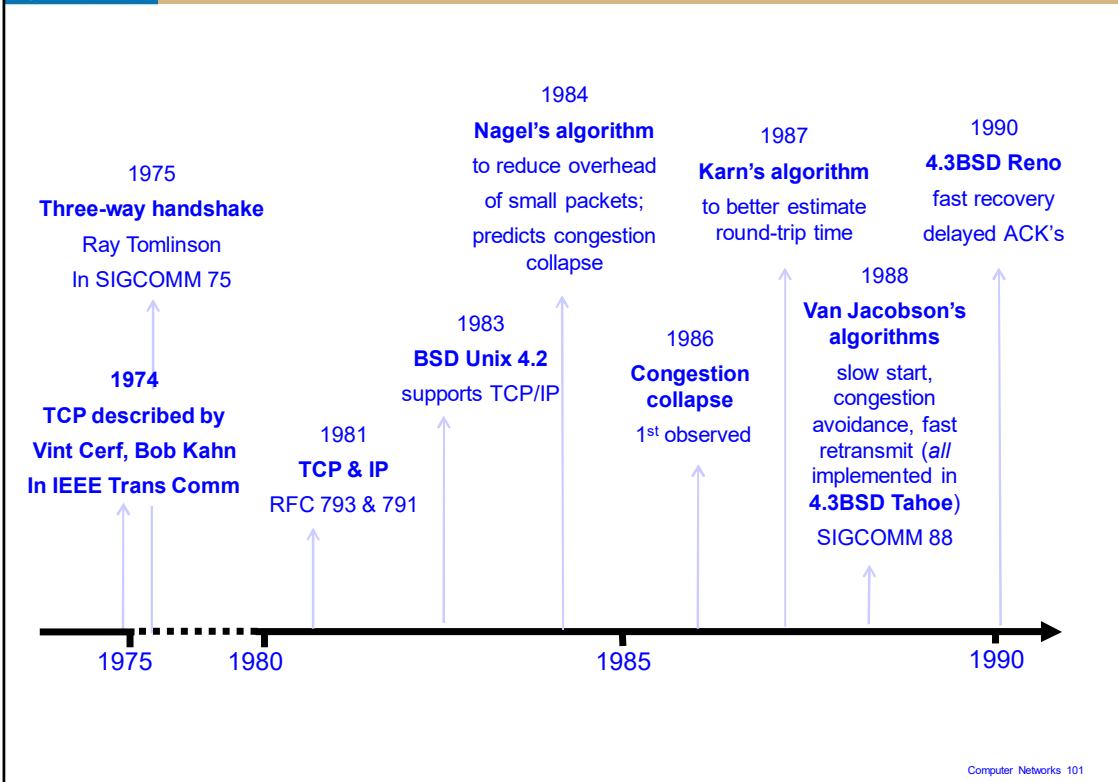
# TCP: closing a connection



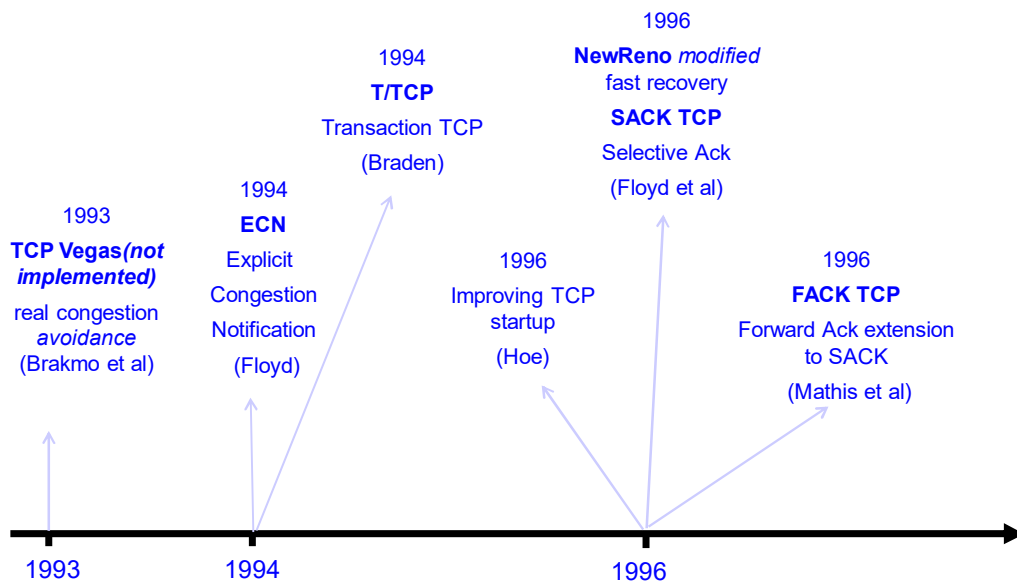
# A brief Internet history...



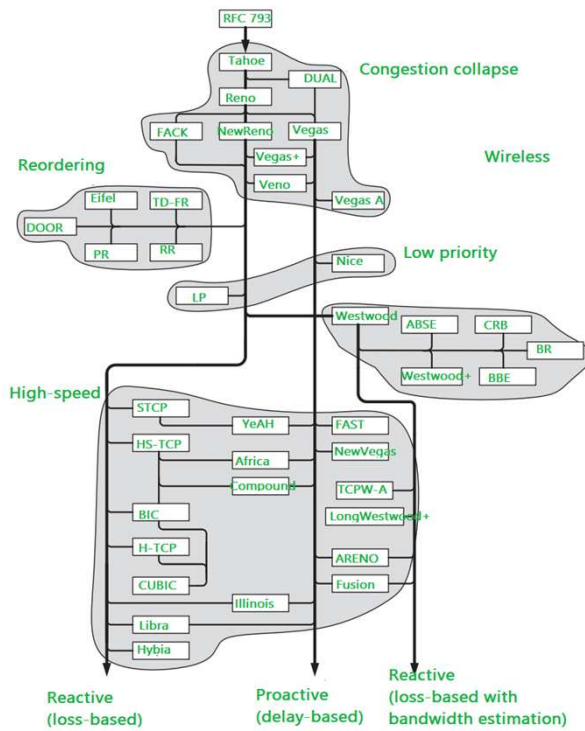
# Evolution of TCP

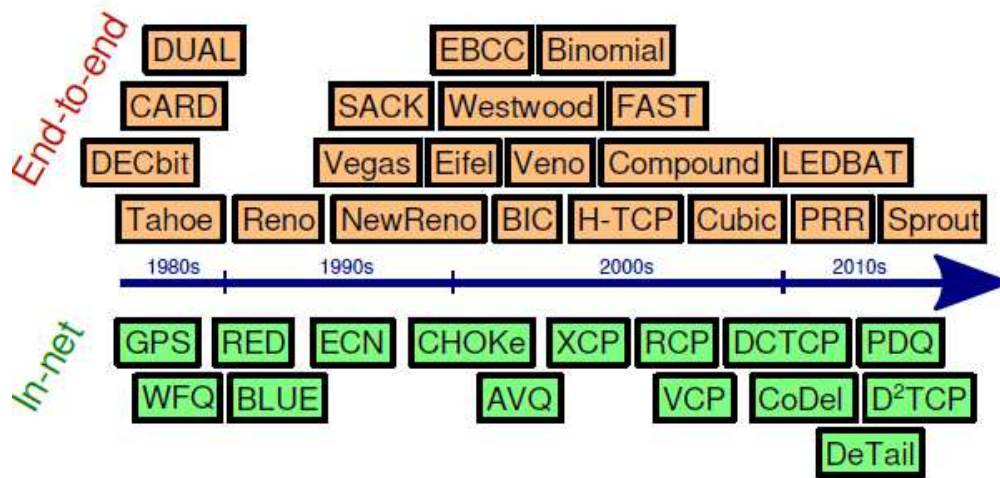


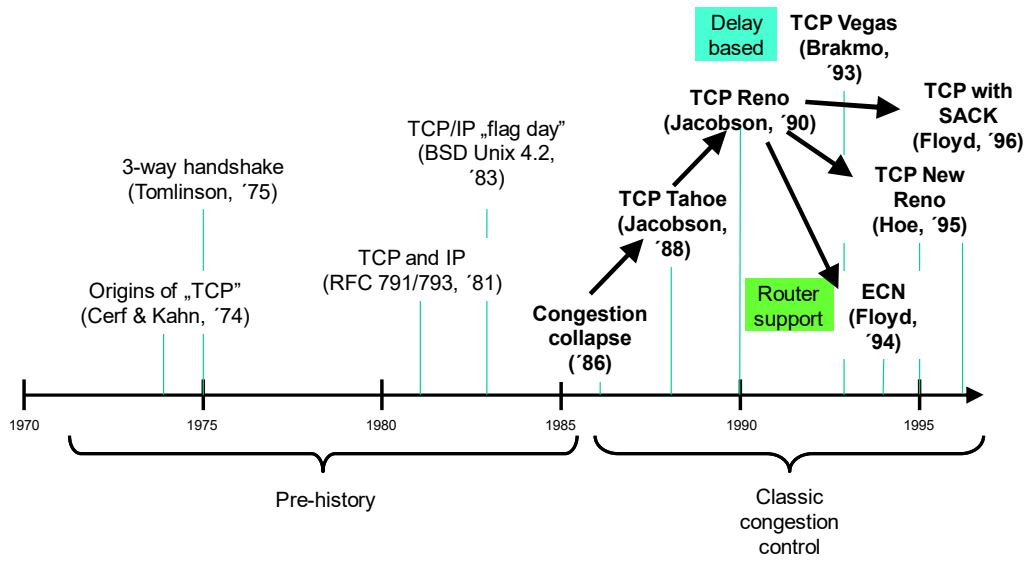
# Evolution of TCP



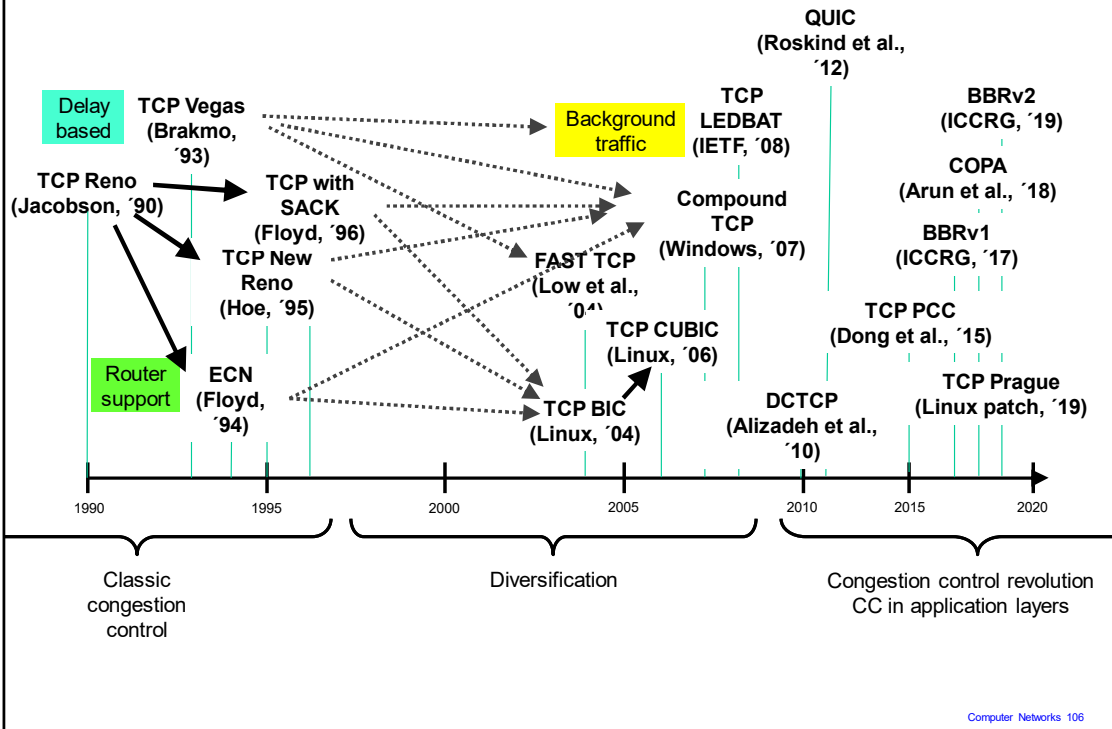
# More than 100 TCP variants ...



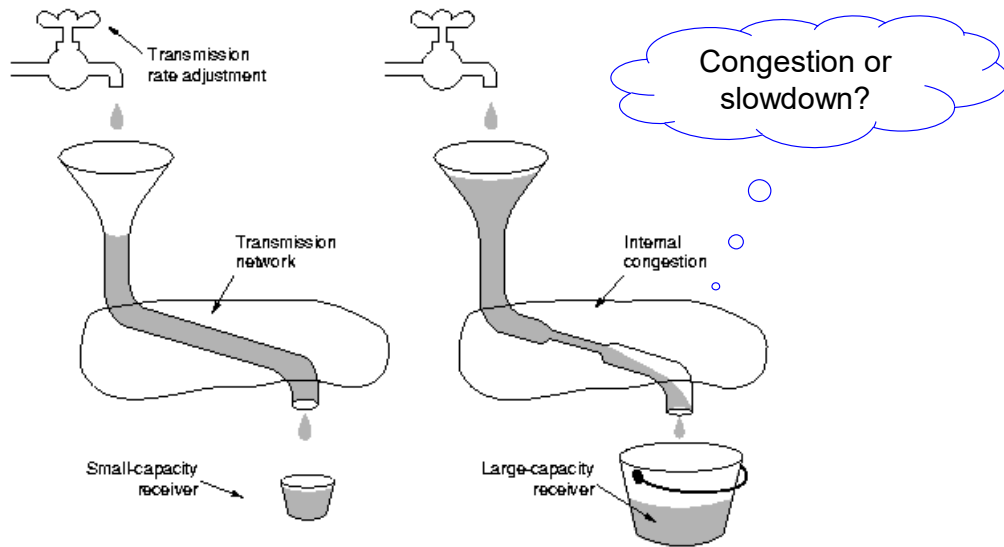




# TCP variants



# Flow control vs Congestion control

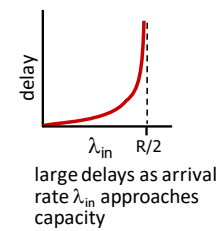
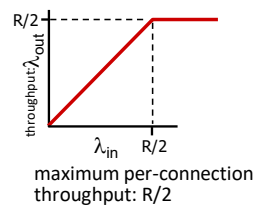
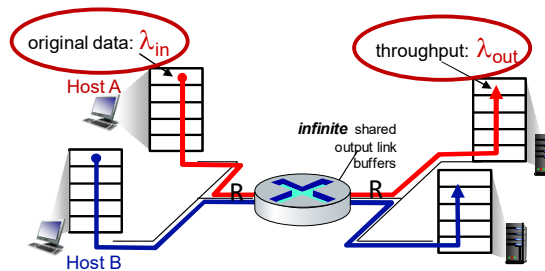




Congestion or delay?

Simplest scenario:

- one router, infinite buffers
- input, output link capacity:  $R$
- two flows
- no retransmissions needed

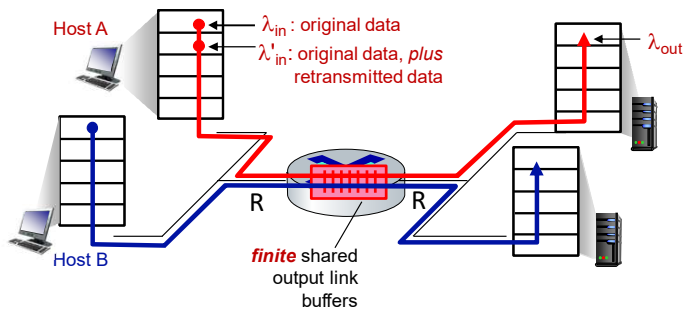


**Q:** What happens as arrival rate  $\lambda_{in}$  approaches  $R/2$ ?

One router, *finite* buffers

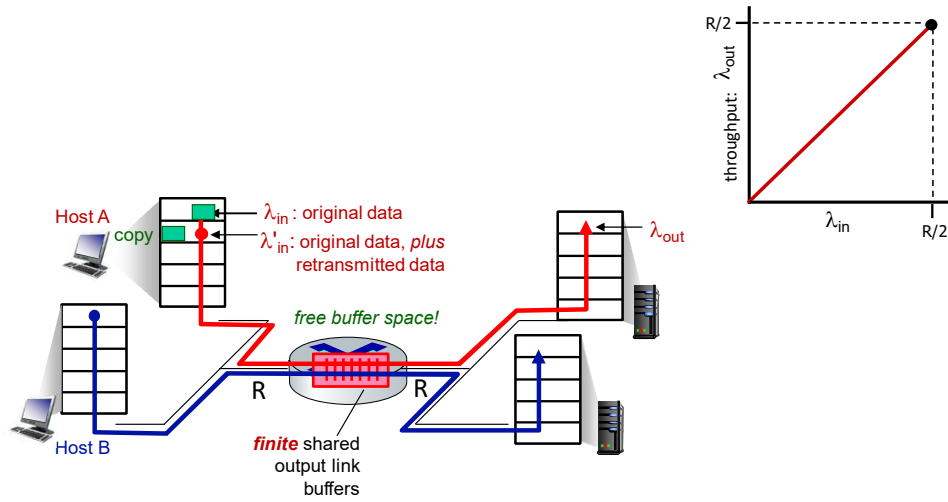
sender retransmits lost, timed-out packet

- application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
- transport-layer input includes *retransmissions* :  $\lambda'_{in} \geq \lambda_{in}$



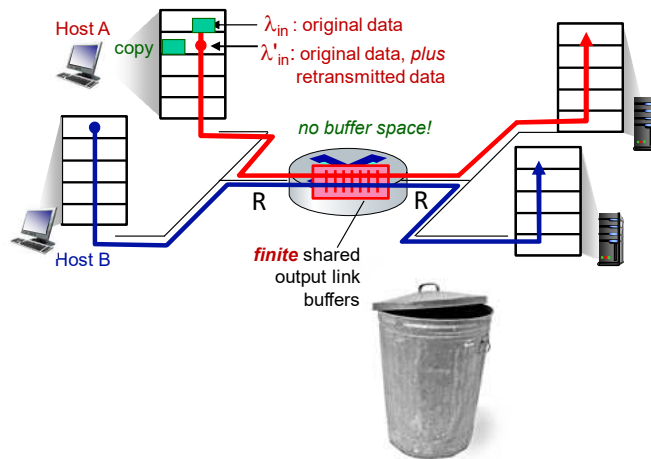
Idealization: perfect knowledge

- sender sends only when router buffers available



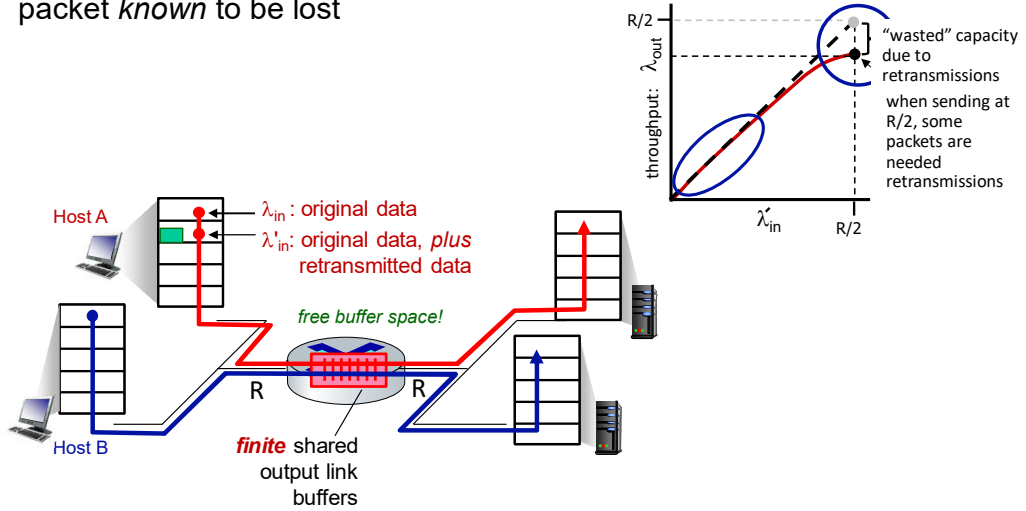
Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



Idealization: *some* perfect knowledge

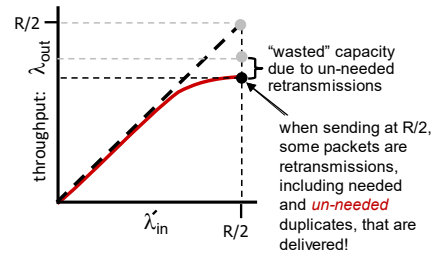
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost





### Realistic scenario: *un-needed duplicates*

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



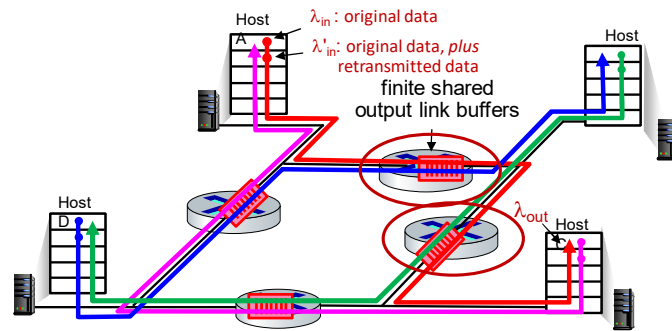
### "costs" of congestion:

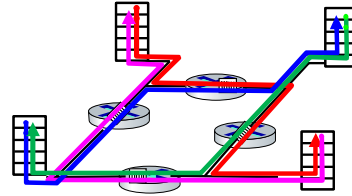
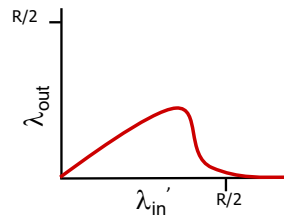
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

- four senders
- multi-hop paths
- timeout/retransmit

**Q:** what happens as  $\lambda_{in}$  and  $\lambda_{in}'$  increase ?

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

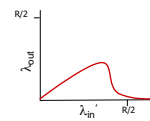
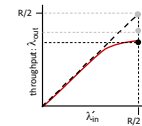
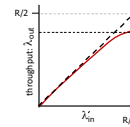
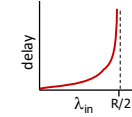
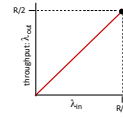




another “cost” of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

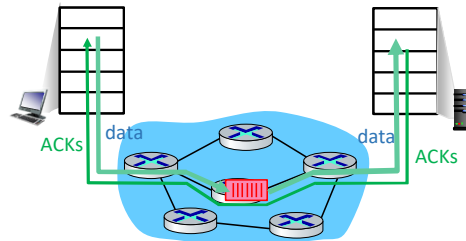
- Connection throughput can never exceed link capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



## End-end congestion control:

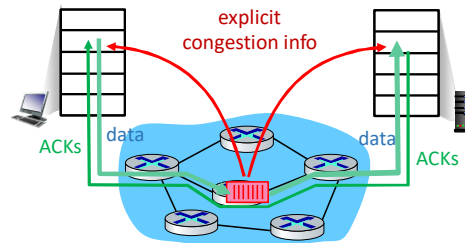
- no explicit feedback from network
- congestion *inferred* from observed loss and/or delay

approach taken by TCP



### Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate



Examples:

TCP ECN, ATM, DECbit protocols

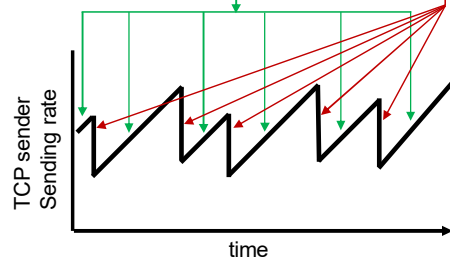
**Approach:** senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

**Additive Increase**

increase sending rate by 1 maximum segment size every RTT until loss detected

**Multiplicative Decrease**

cut sending rate in half at each loss event



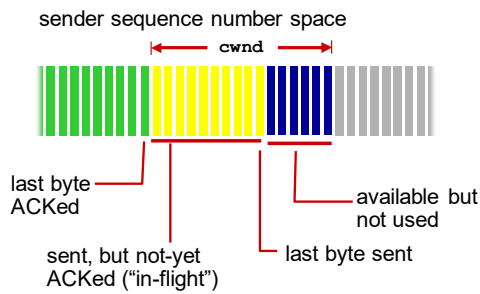
**AIMD** sawtooth behavior: *probing* for bandwidth

*Multiplicative decrease* detail: sending rate is

- cut in half on loss detected by triple duplicate ACK (TCP Reno)
- cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable *stability* properties



TCP sending behavior:

- *roughly*: send **cwnd** bytes, wait RTT for ACKS, then send more bytes

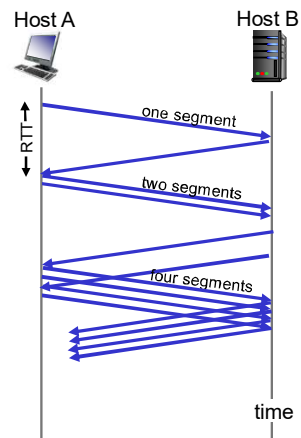
$$\text{TCP rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bps}$$

- TCP sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

- **cwnd** is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

- when connection begins, increase rate **exponentially** until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received



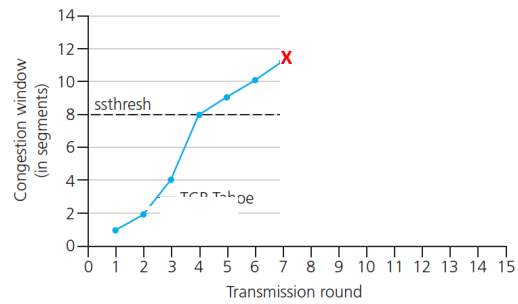
**summary:** initial rate is slow, but ramps up exponentially fast

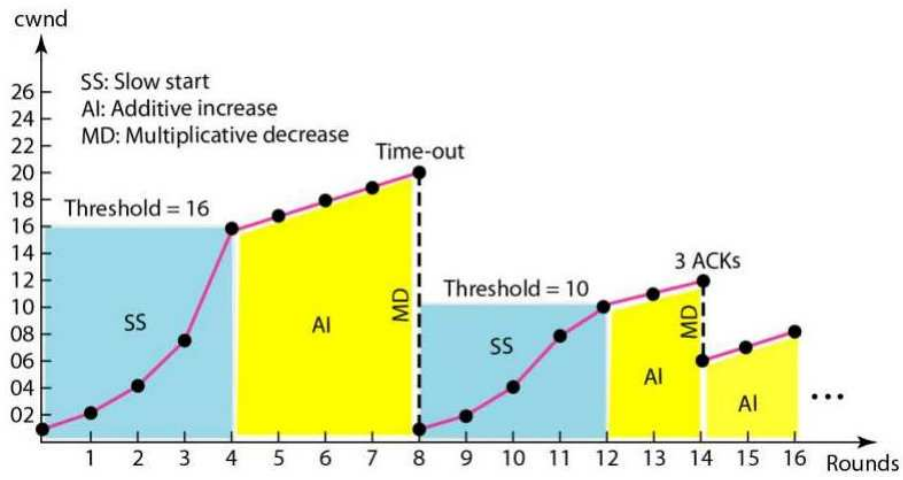
**Q:** when should the exponential increase switch to linear?

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

**Implementation:**

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





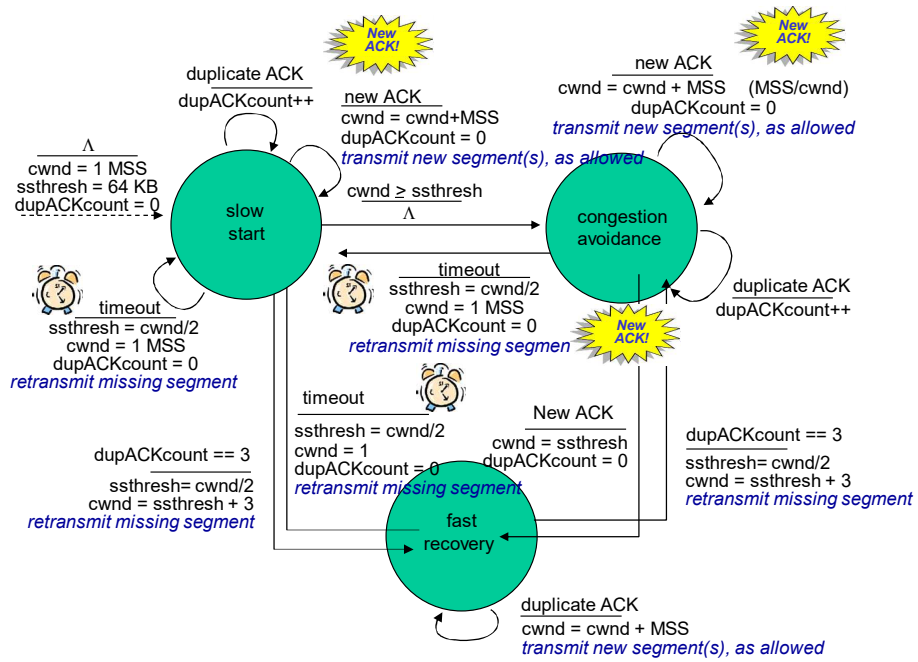
**TCP-Tahoe:** implements:

1. the slow start,
2. congestion avoidance,
3. fast retransmit algorithms.

**TCP-Reno:** implements:

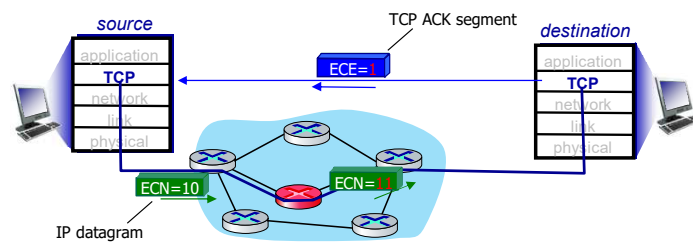
1. the slow start,
2. congestion avoidance,
3. fast retransmit,
4. fast recovery.

# Summary: TCP congestion control

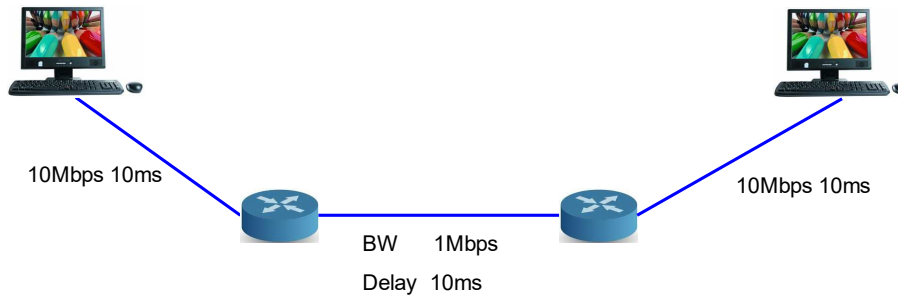


TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)

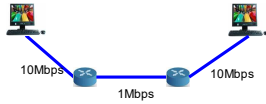
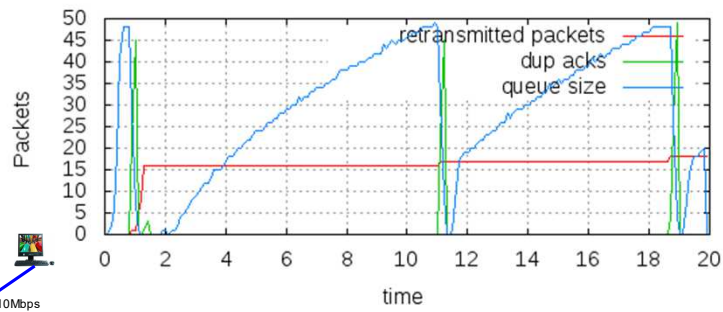
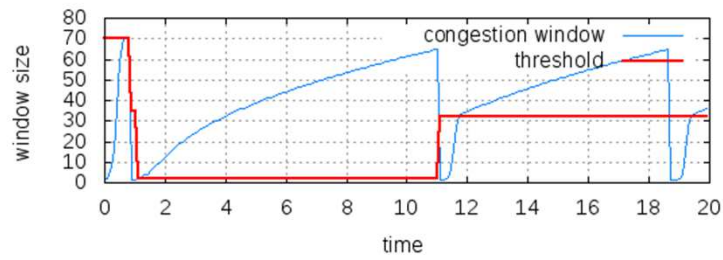


Case study 1: a TCP connection in a lossless channel, with constant delay

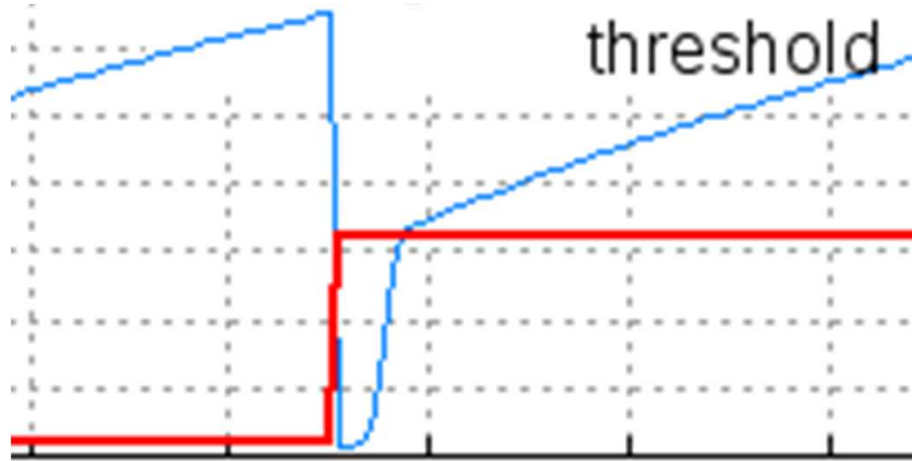


Case study 1: a TCP connection in a lossless channel, with constant delay

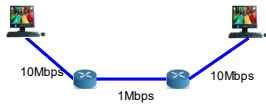
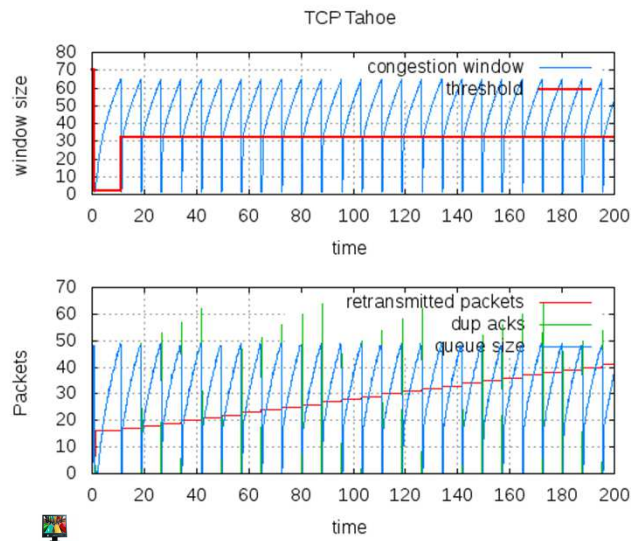
TCP Tahoe



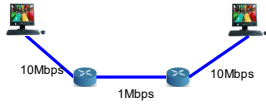
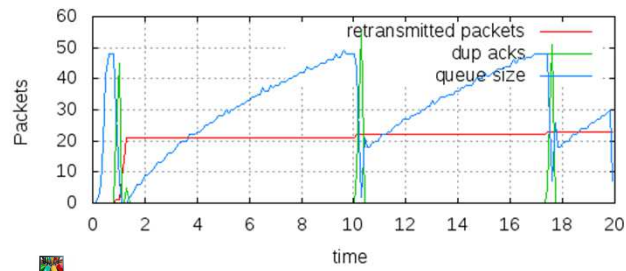
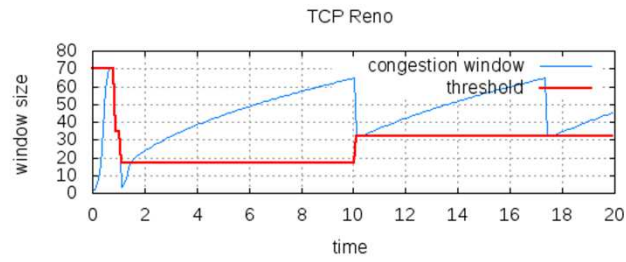
Case study 1: a TCP connection in a lossless channel, with constant delay



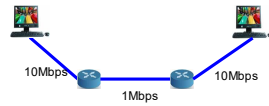
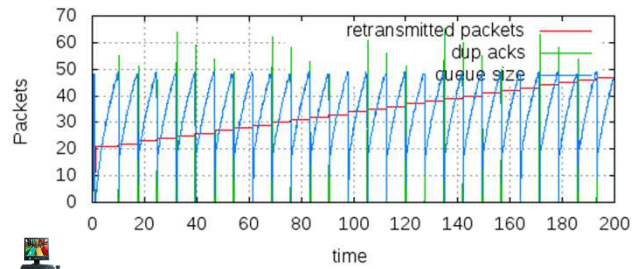
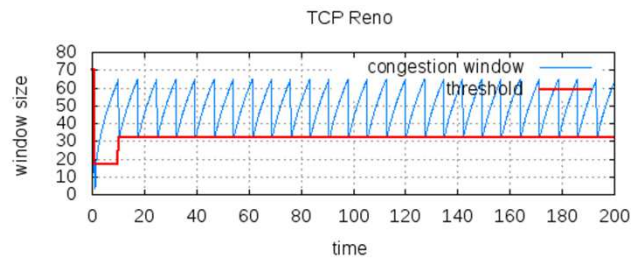
Case study 1: a TCP connection in a lossless channel, with constant delay



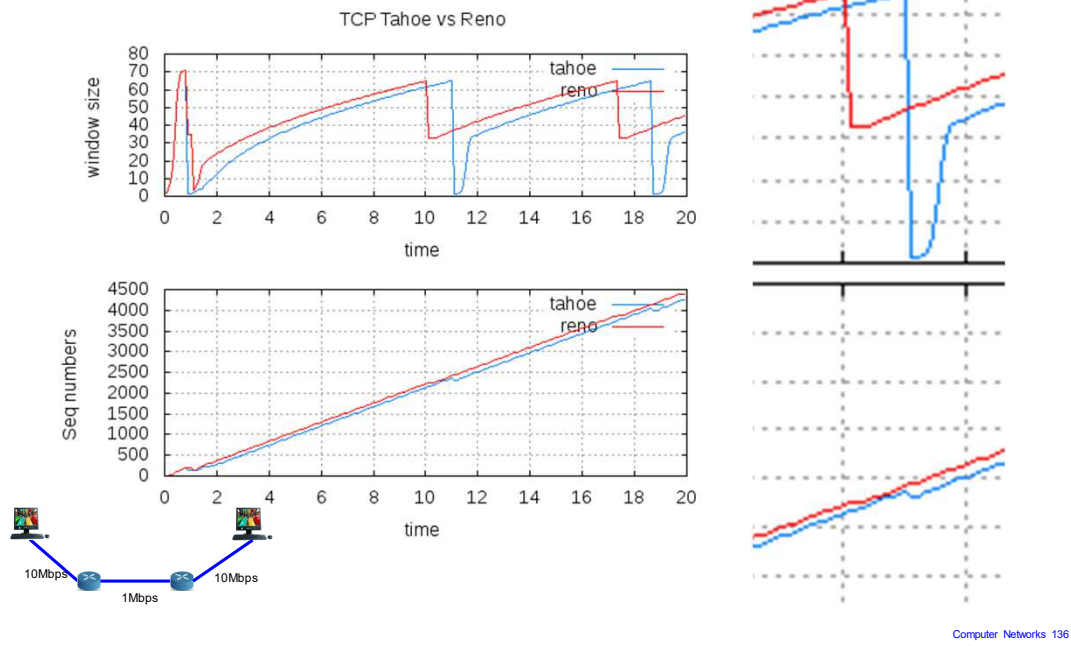
Case study 1: a TCP connection in a lossless channel, with constant delay



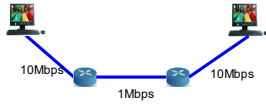
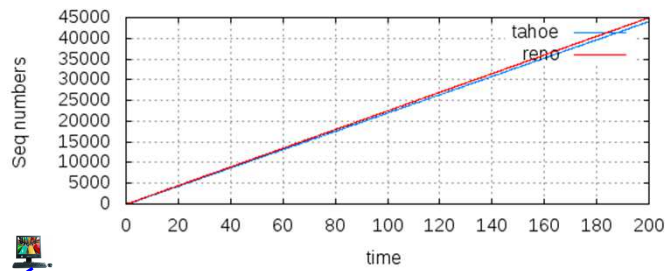
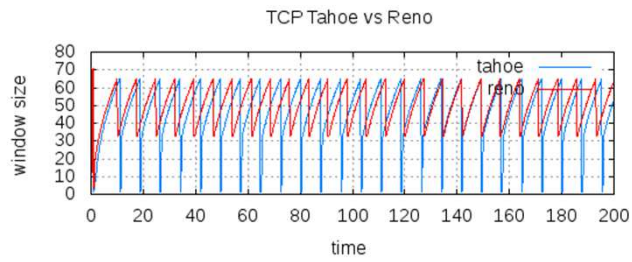
Case study 1: a TCP connection in a lossless channel, with constant delay



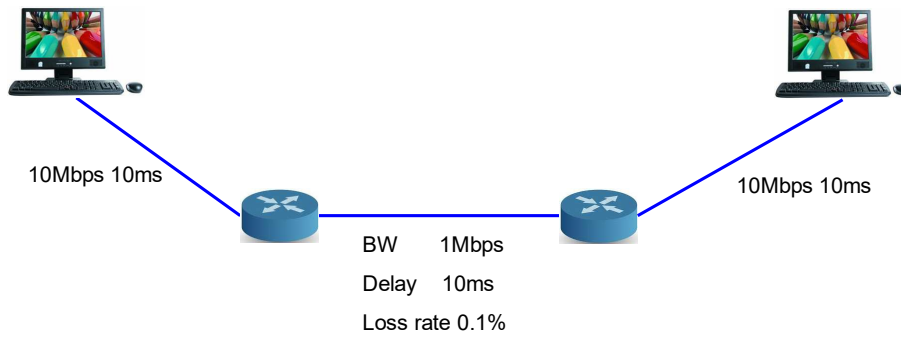
Case study 1: a TCP connection in a lossless channel, with constant delay



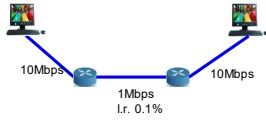
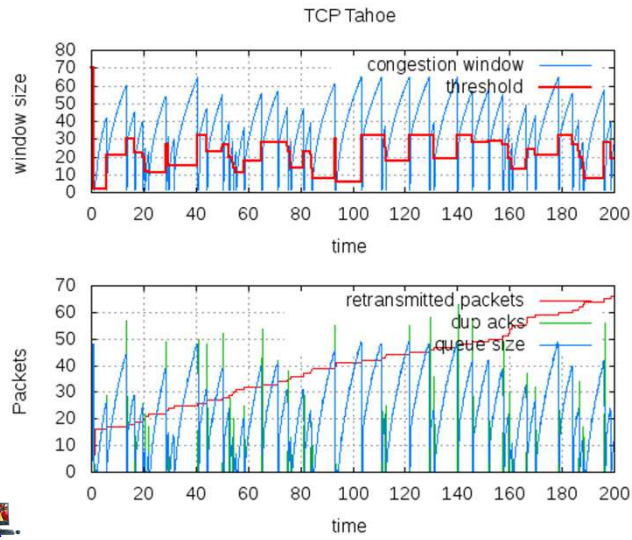
Case study 1: a TCP connection in a lossless channel, with constant delay



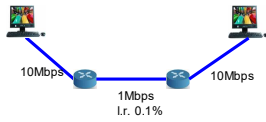
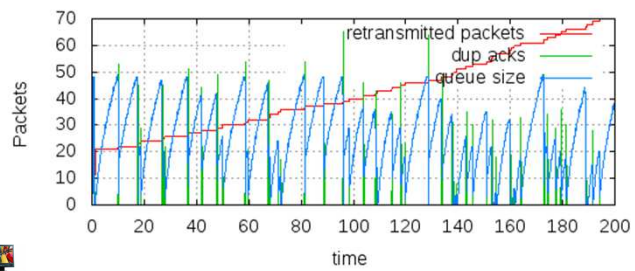
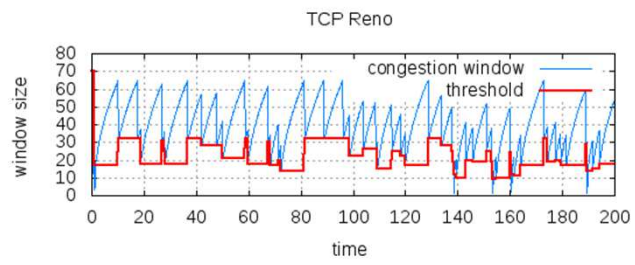
Case study 2: a TCP connection in a lossy channel, with constant delay



## Case study 2: a TCP connection in a lossy channel, with constant delay

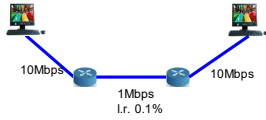
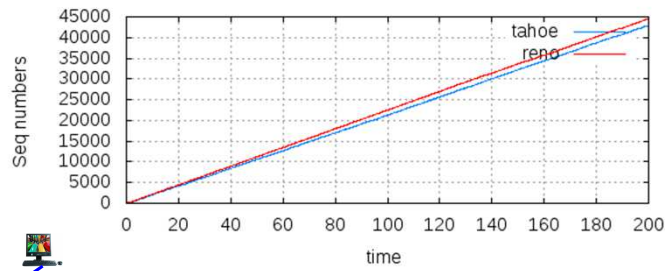
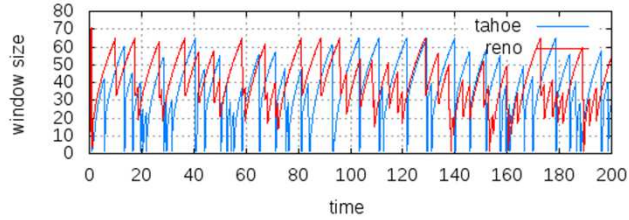


## Case study 2: a TCP connection in a lossy channel, with constant delay



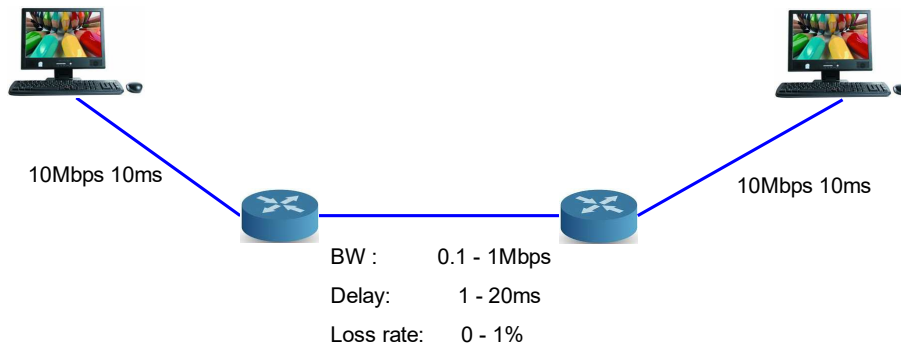
Case study 2: a TCP connection in a lossy channel, with constant delay

TCP Tahoe vs Reno



## Case study 3:

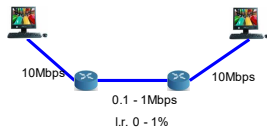
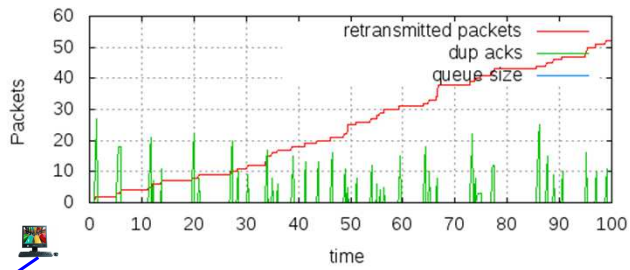
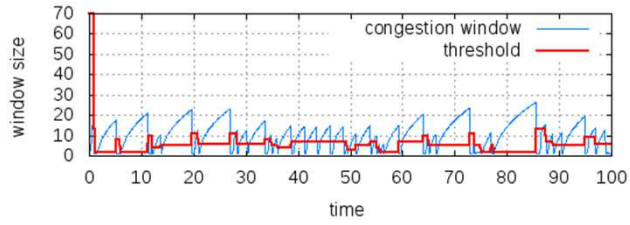
a TCP connection in a lossy channel, with bw and delay variable



Case study 3:

a TCP connection in a lossy channel, with bw and delay variable

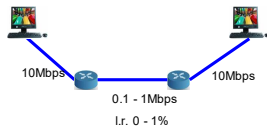
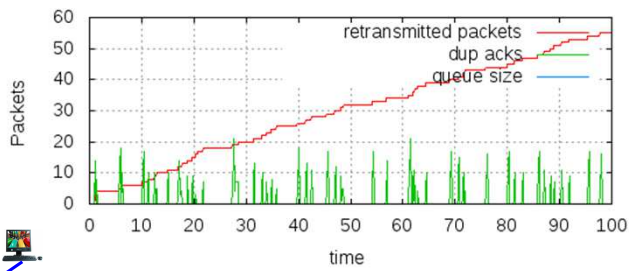
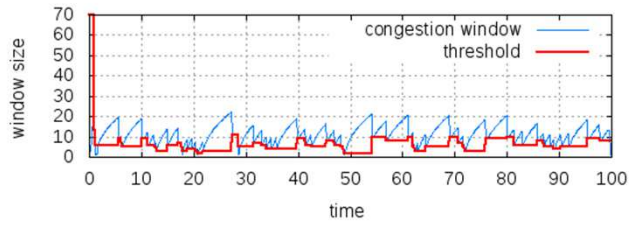
TCP Tahoe



## Case study 3:

a TCP connection in a lossy channel, with bw and delay variable

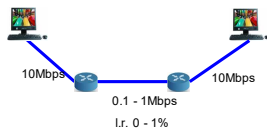
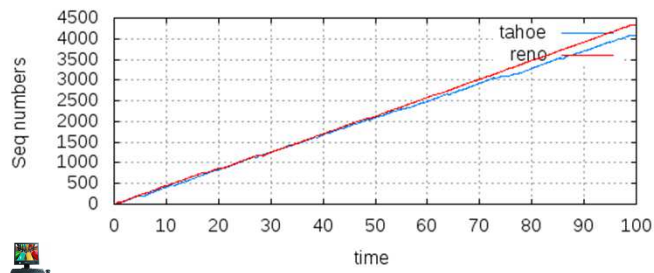
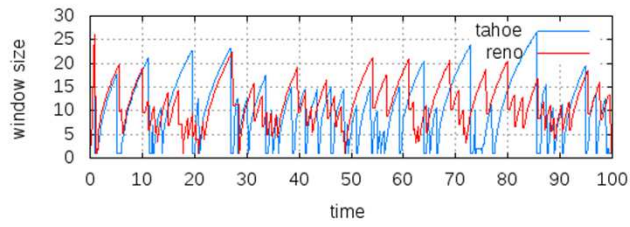
TCP Reno



Case study 3:

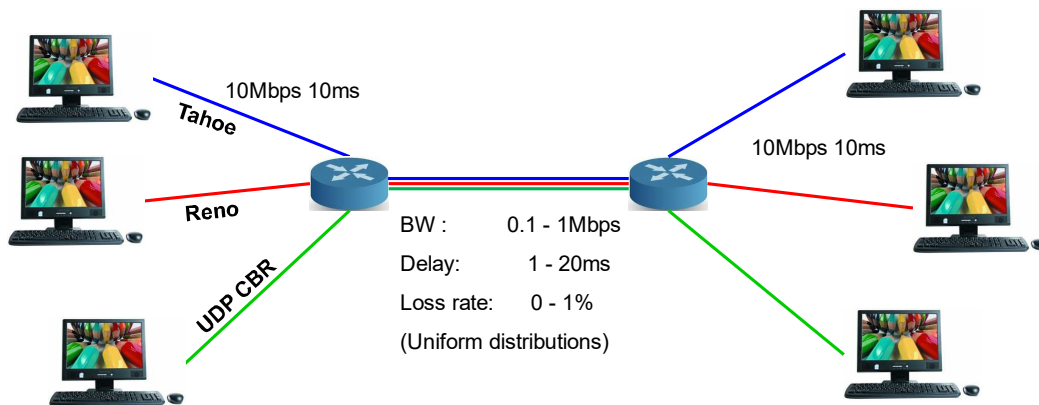
a TCP connection in a lossy channel, with bw and delay variable

TCP Tahoe vs Reno



## Case study 4:

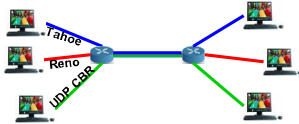
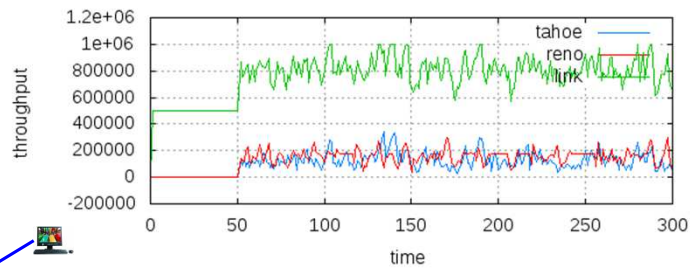
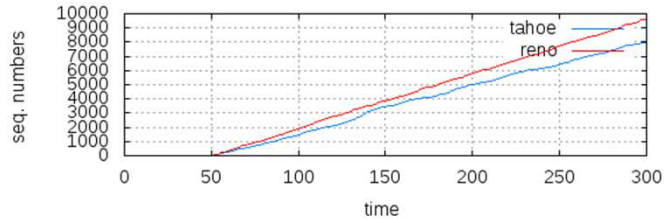
- a TCP Tahoe connection
- A TCP Reno connection
- A UDP CBR flow
- lossy channel, with variable bandwidth and delay



Case study 4a: three concurrent flows in the same channel

UDP starts at t=0s, TCPs start at t=50s

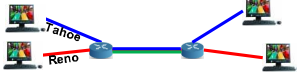
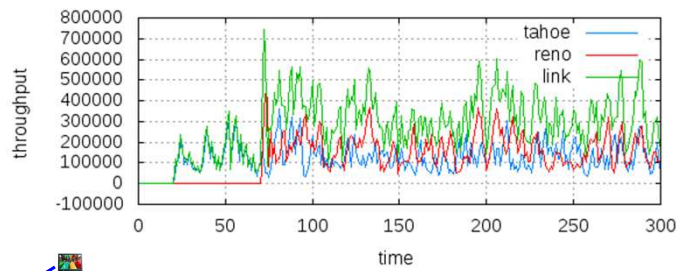
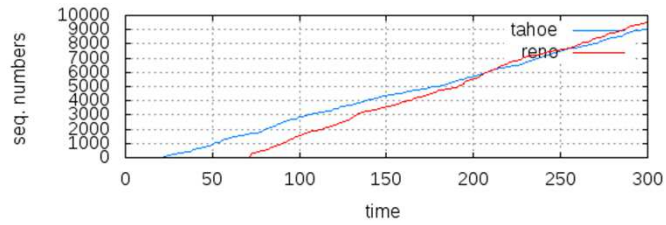
TCP Tahoe vs Reno



Case study 4b: two concurrent flows in the same channel

TCP Tahoe starts at t=25s, TCP Reno starts at t=70s

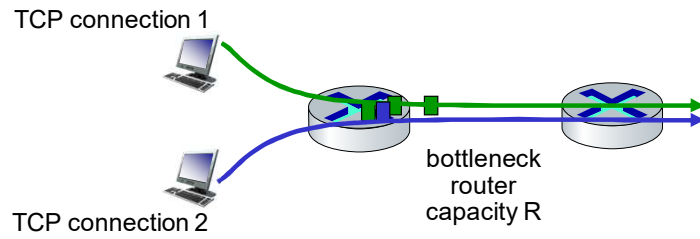
TCP Tahoe vs Reno



*"Fairness is the quality of being reasonable, right, and just."*

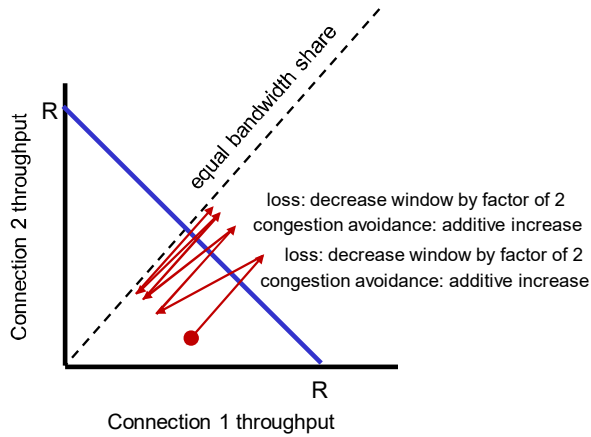
(Collins)

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



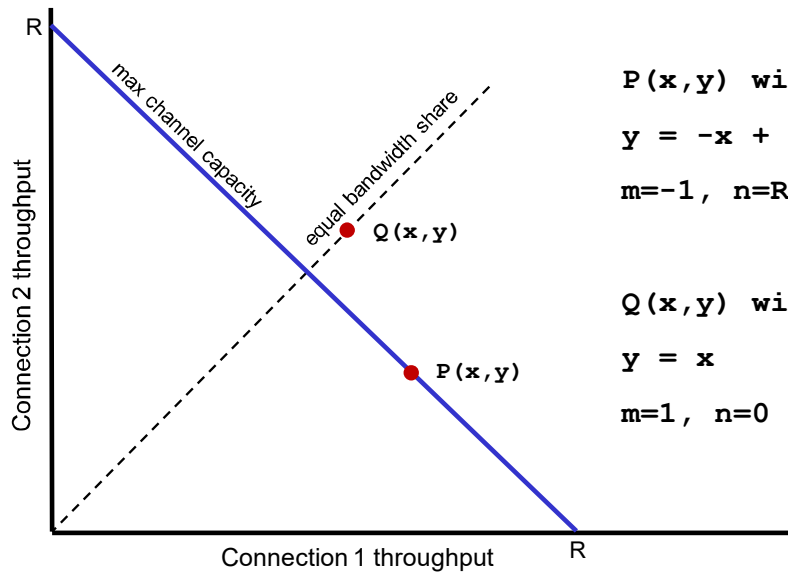
Example: two competing TCP sessions:

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



**Is TCP fair?**  
**A:** Yes, under idealized assumptions:  
 ▪ same RTT  
 ▪ fixed number of sessions only in congestion avoidance

### Q: is TCP Fair?



$$y = mx + n$$

$P(x, y)$  with  $x+y=R$

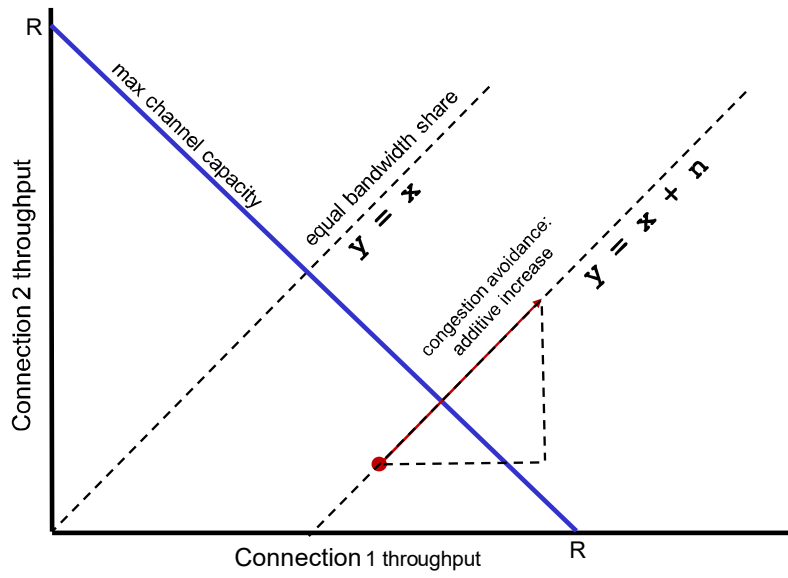
$$y = -x + R$$

$$m=-1, n=R$$

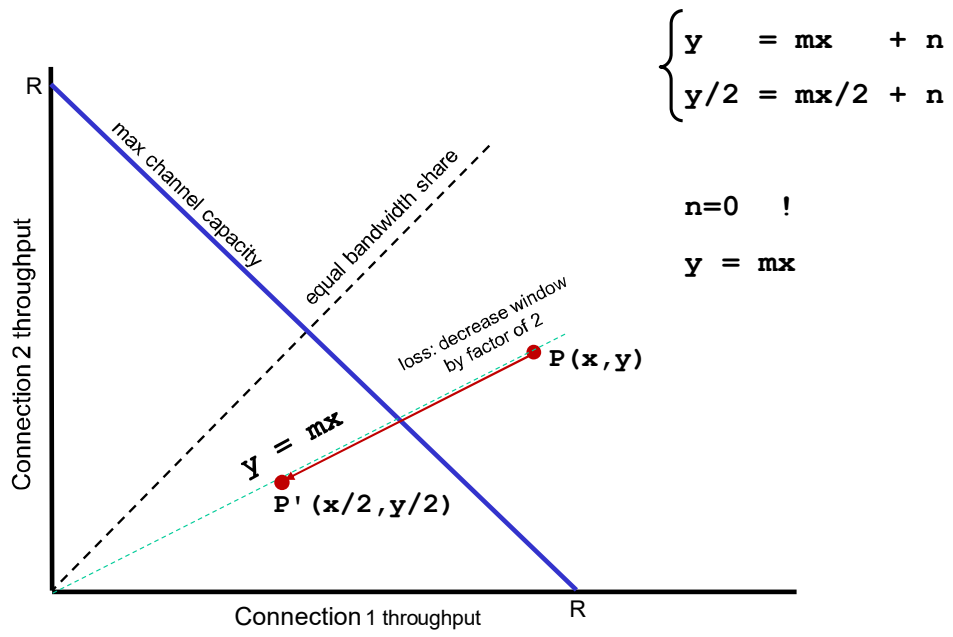
$Q(x, y)$  with  $x=y$

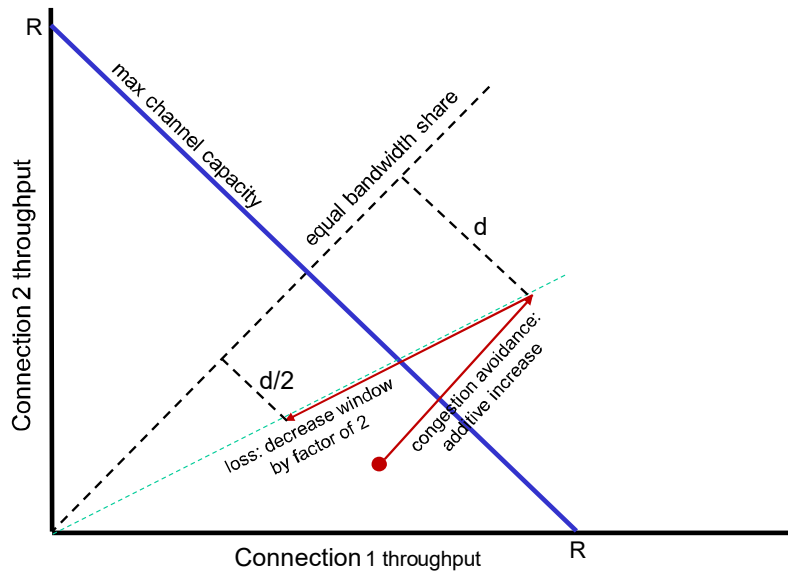
$$y = x$$

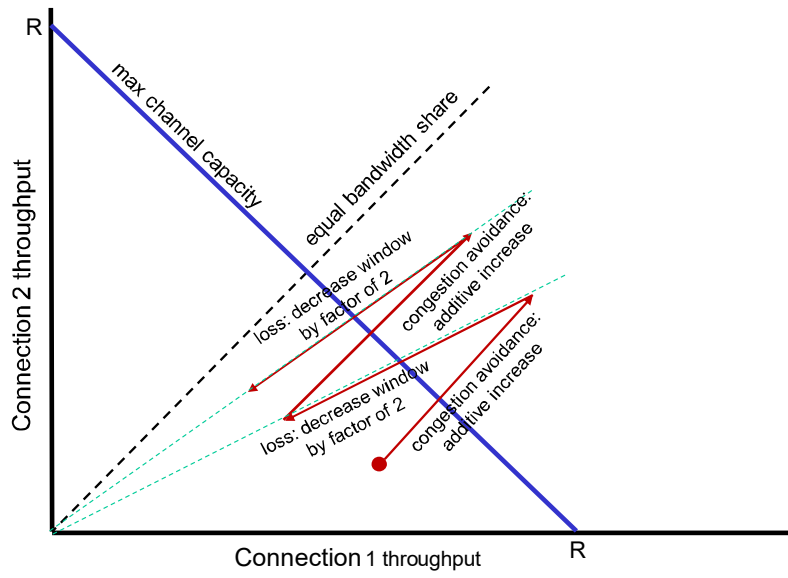
$$m=1, n=0$$

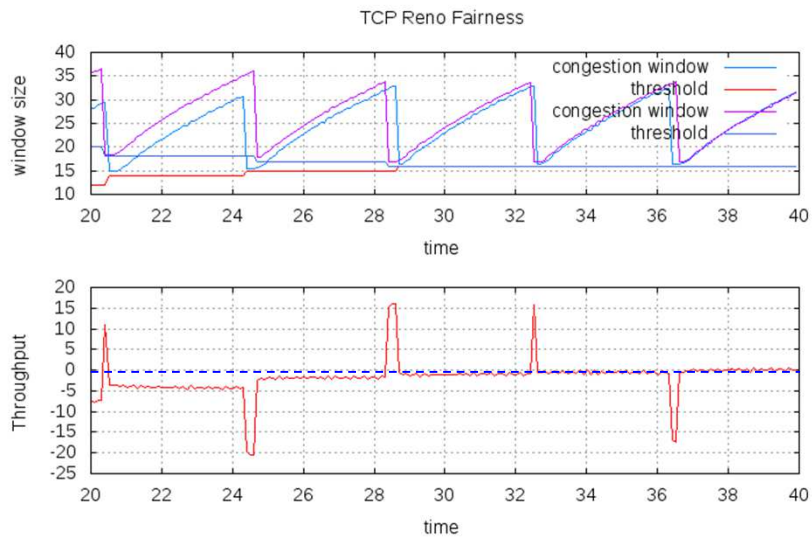


### Q: is TCP Fair?









- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport-layer functions to application layer, on top of UDP
  - HTTP/3: QUIC