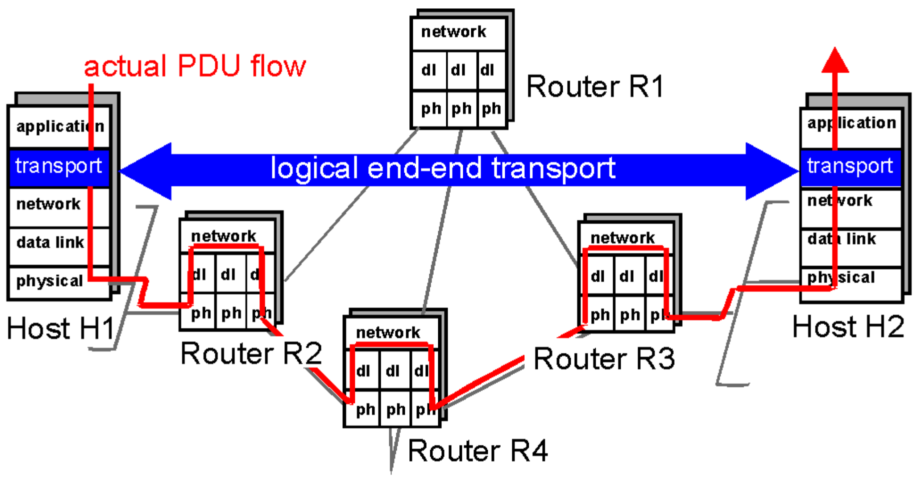
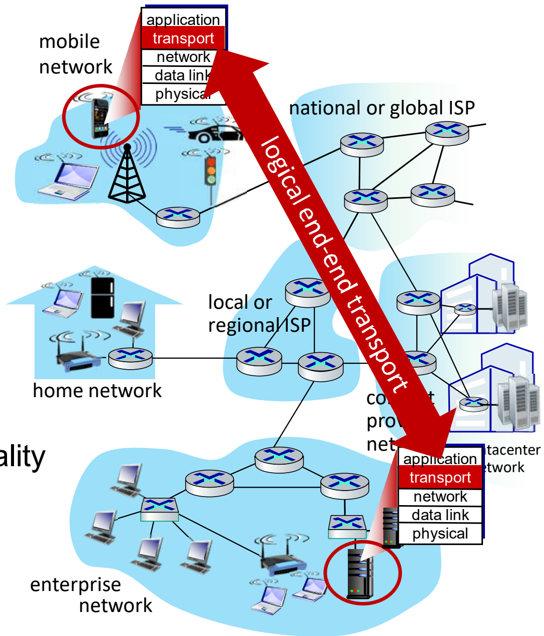
Transport layer



Transport layer: roadmap

- 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

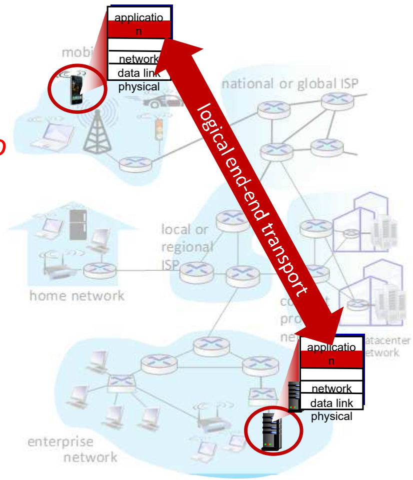


Transport services and protocols

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.

Transport vs network vs DL layers

Data Link layer:

physical communication between (directly connected *) *hosts*

Network layer:

logical communication between *hosts*

Transport layer*:*

logical communication between *processes*

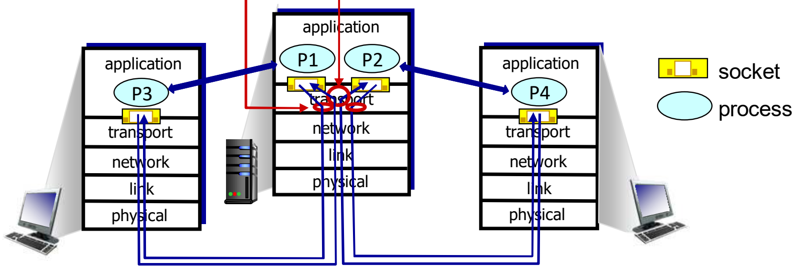
Multiplexing/demultiplexing

multiplexing at sender:

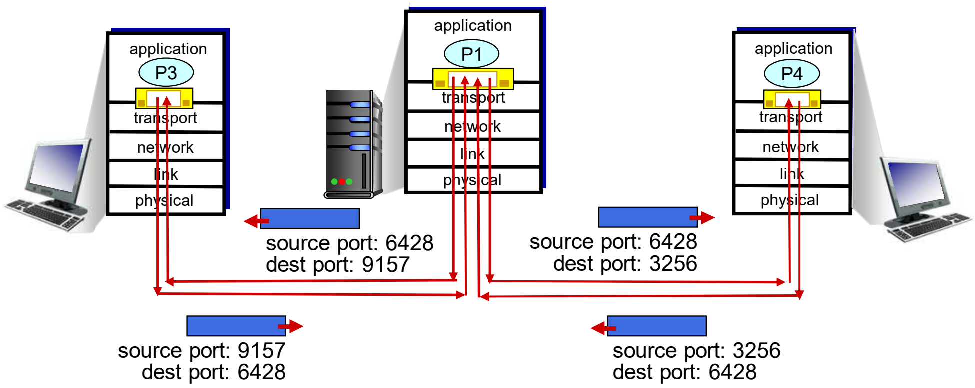
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

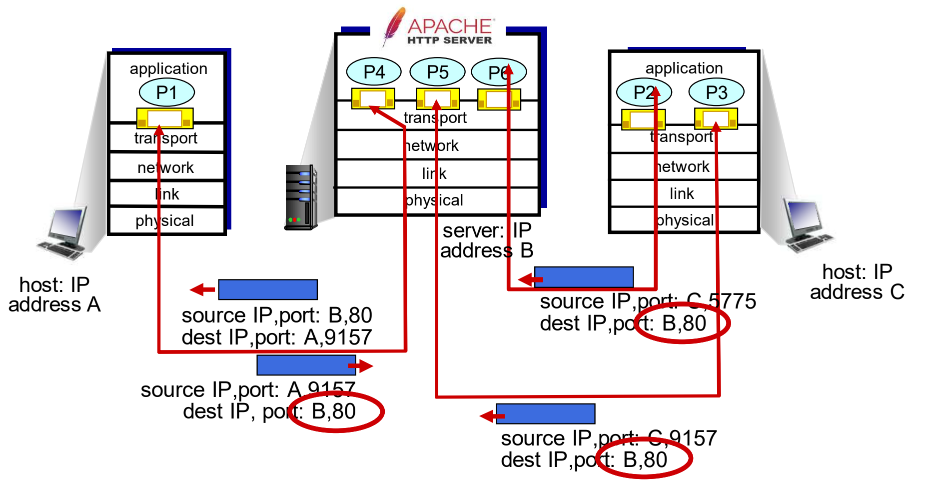
use header info to deliver received segments to correct socket



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

UDP

```
                                                 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
------


                  0      7 8     15 16    23 24    31
                 +--------+--------+--------+--------+
                 |     Source      |   Destination   |
                 |      Port       |      Port       |
                 +--------+--------+--------+--------+
                 |                 |                 |
                 |     Length      |    Checksum     |
                 +--------+--------+--------+--------+
                 |
                 |          data octets ...
                 +---------------- ...
```

UDP: User Datagram Protocol

- "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: User Datagram Protocol

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):

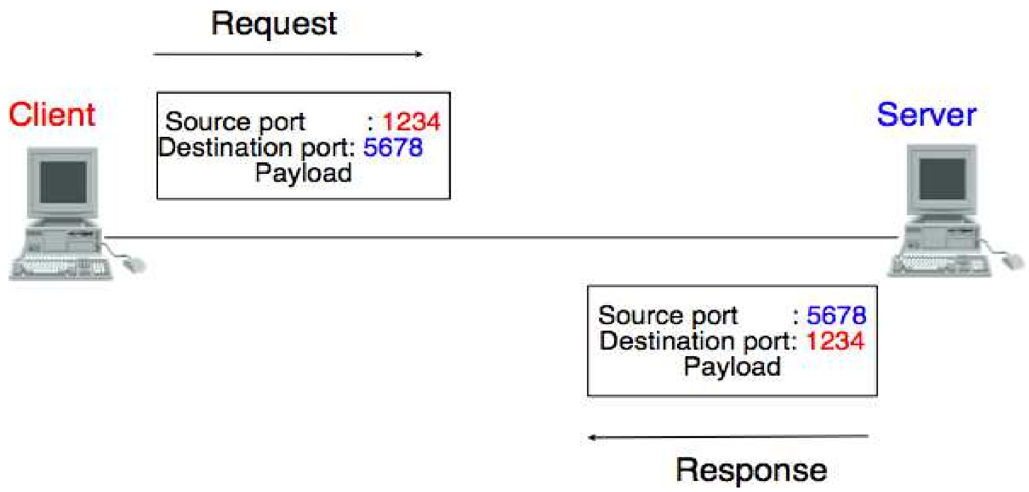
- add needed reliability at application layer
- add congestion control at application layer

Università di Catania

UDP header

0 – 15	16 – 31
Source Port Number(16 bits)	Destination Port Number(16 bits)
Length(UDP Header + Data)16 bits	UDP Checksum(16 bits)
Application Data (Message)	

UDP client-server model



UDP checksum

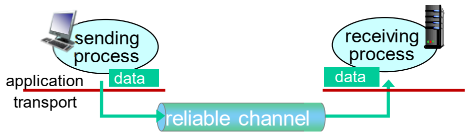
Goal: detect errors (*i.e.,* flipped bits) in received segment

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

receiver-computed checksum ≠ sender-computed checksum (as received)

Principles of reliable data transfer

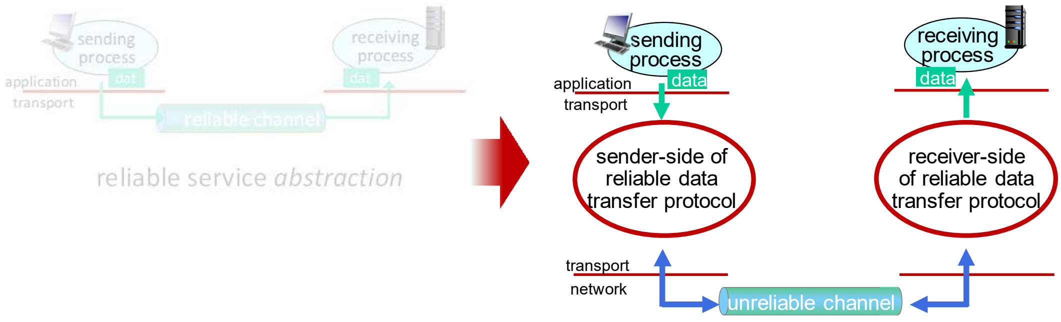
What user wants!



reliable service *abstraction*

Principles of reliable data transfer

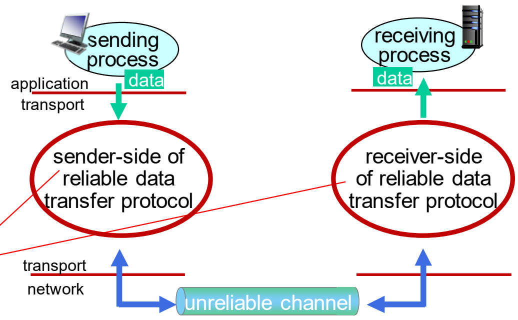
What we have in the real world!



reliable service *implementation*

Principles of reliable data transfer

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

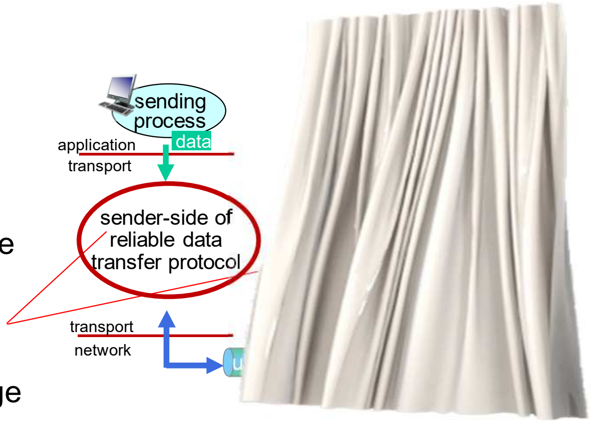


reliable service *implementation*

Principles of reliable data transfer

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

Unless communicated via a message



reliable service *implementation*

Planning

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	**no**	**no**	-	no
RDT 2.0	**yes**	no	ACK / NAK	no
RDT 2.1	yes	**yes**	ACK / NAK	no
RDT 2.2	yes	yes	**ACK**	no
RDT 3.0	yes	yes	ACK	**yes**

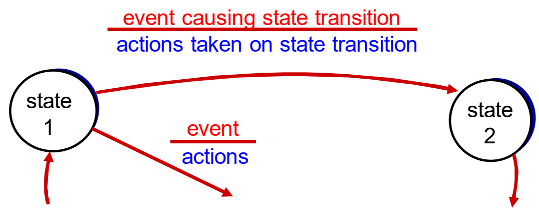
Reliable data transfer: getting started

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

state: when in this "state" next state uniquely determined by next event

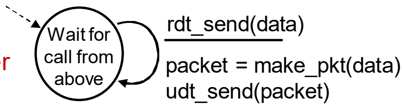


rdt 1.0: reliable transfer over a reliable channel

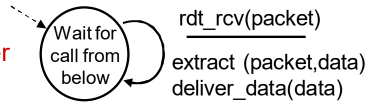
Underlying channel perfectly reliable
- no bit errors
- no loss of packets

Separate FSMs for sender and receiver:
- sender sends data into underlying channel
- receiver reads data from underlying channel

sender



receiver



rdt 2.0: channel with bit errors

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?

rdt 2.0: channel with bit errors

Underlying channel may flip bits in packet

- checksum to detect bit errors

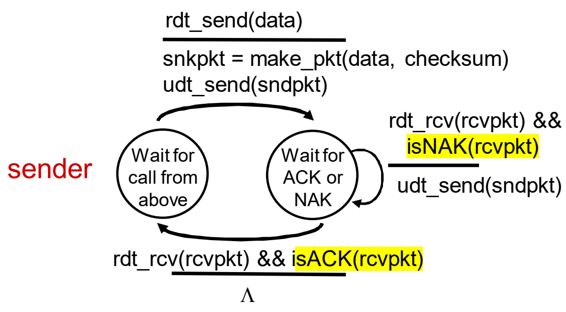
How to recover from errors?

- *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

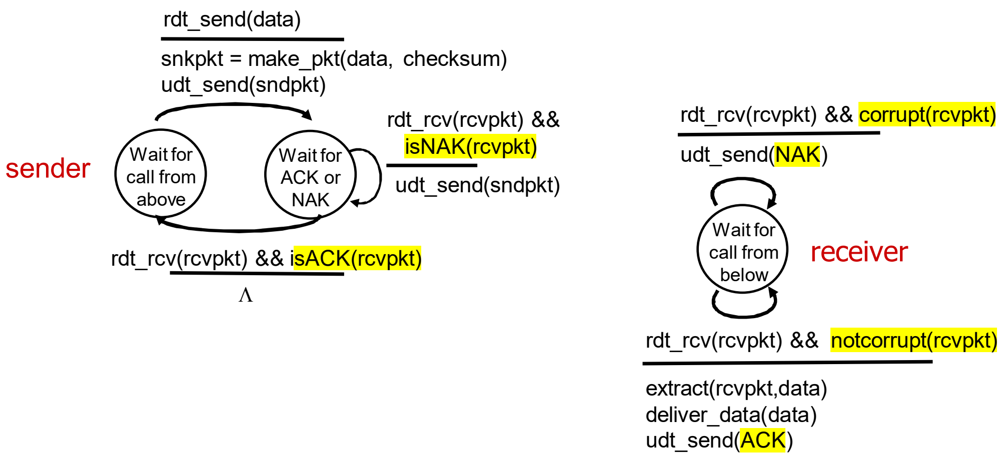
stop and wait

sender sends one packet, then waits for receiver response

rdt 2.0: FSM specifications



rdt 2.0: FSM specifications



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

rdt 2.0 has a fatal flaw!

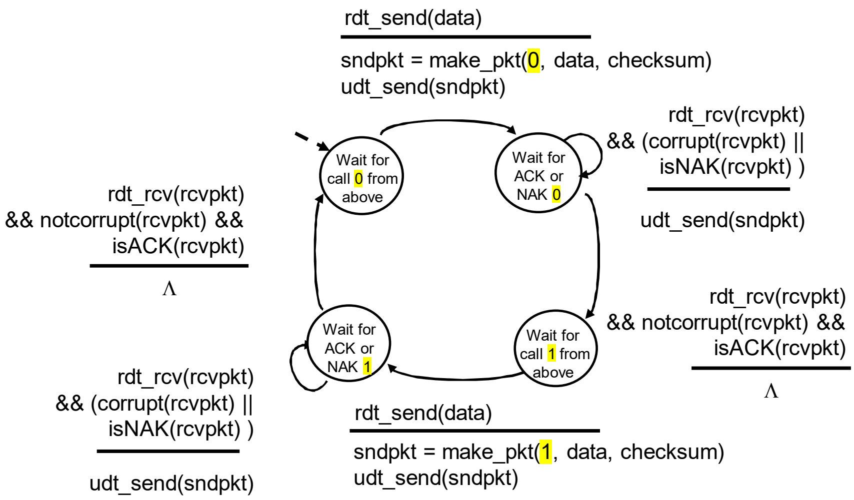
what happens if ACK/NAK arrives corrupted?

- 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

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt)

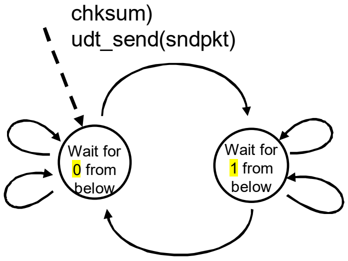
---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt)

---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt 2.1: discussion

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

rdt2.2: a NAK-free protocol

Are NAK necessary?

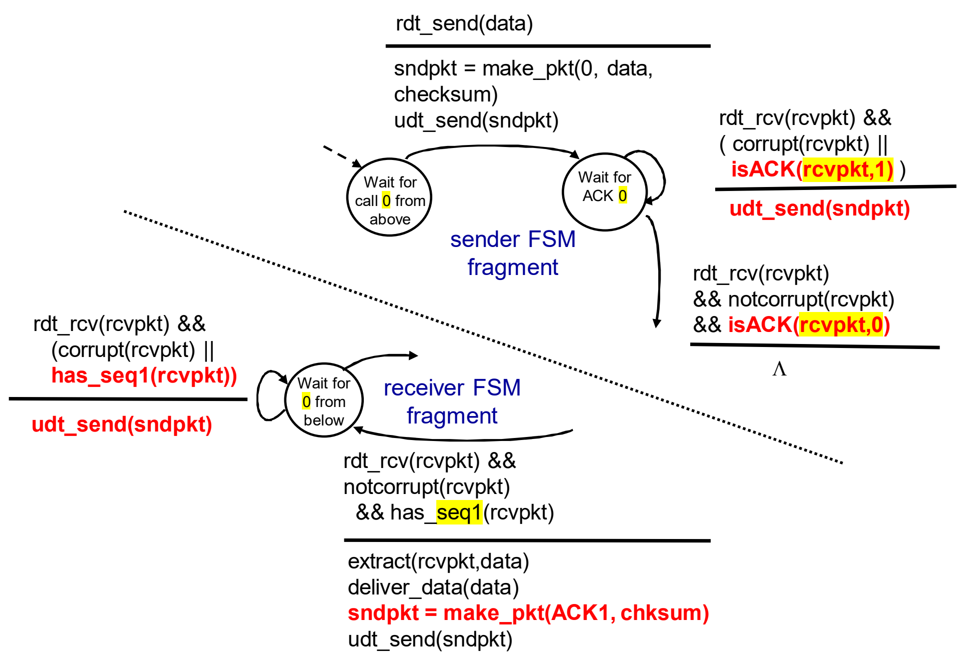
We can design another protocol with the same functionality as rdt 2.1, using ACKs only.

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



rdt 3.0: channels with errors *and* loss

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?

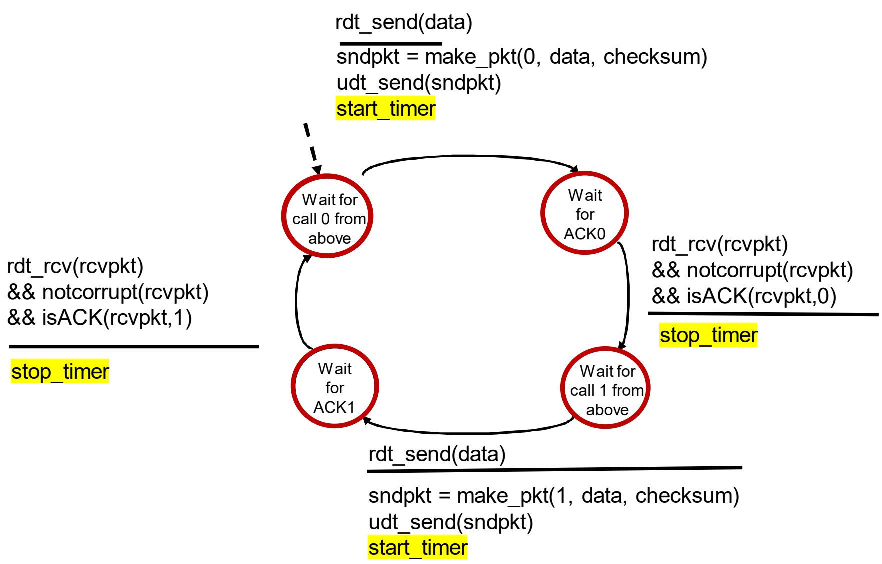
rdt 3.0: channels with errors *and* loss

Approach: sender waits "reasonable" amount of time for ACK

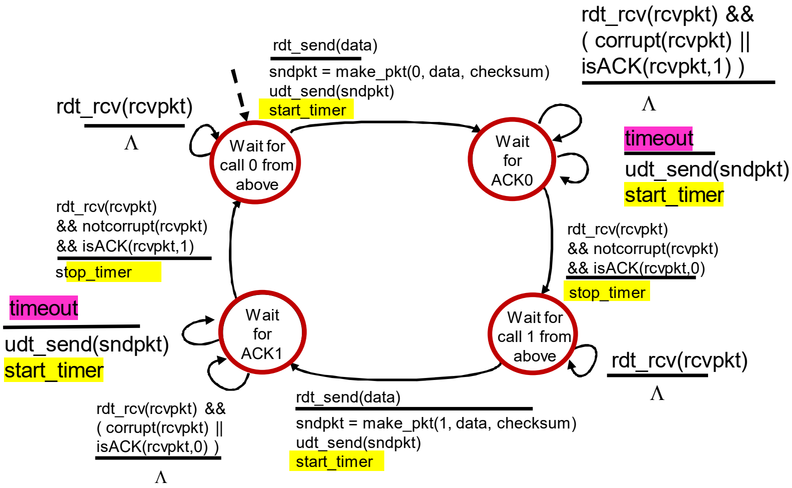
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed

- use countdown timer to interrupt after "reasonable" amount of time

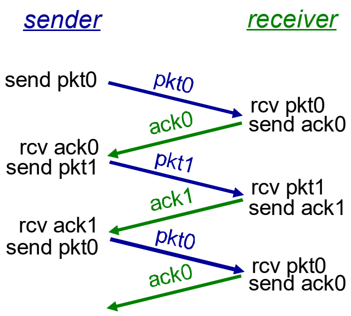
rdt 3.0 sender



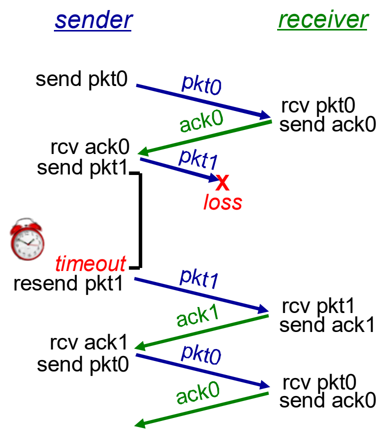
rdt 3.0 sender



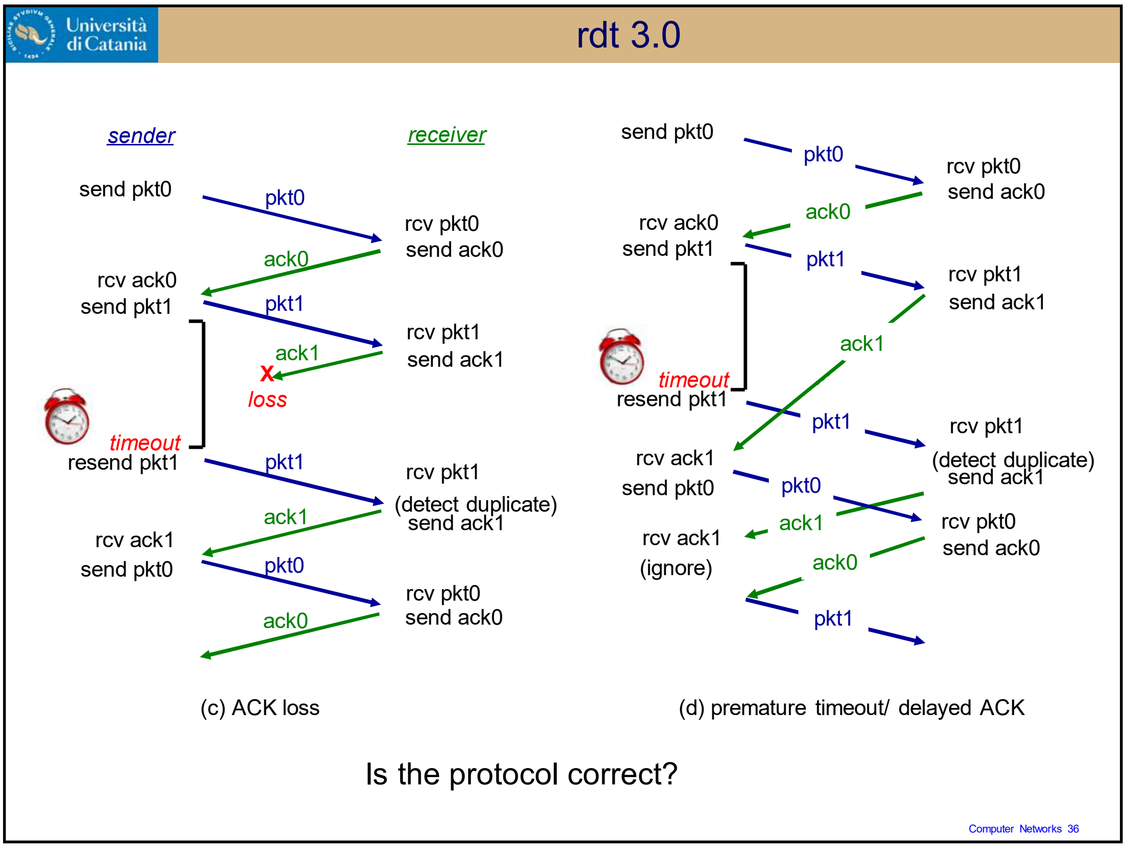
rdt 3.0 in action



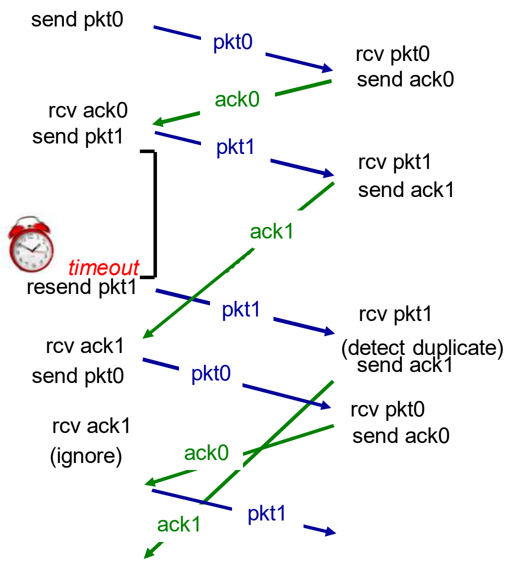
(a) no loss



(b) packet loss

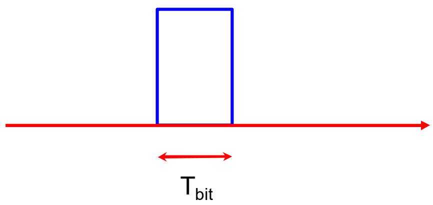

Università di Catania
rdt 3.0
sender
receiver
send pkt0
pkt0
rcv pkt0
send ack0
ack0
rcv ack0
send pkt1
pkt1
rcv pkt1
send ack1
ack1
X
loss
timeout
resend pkt1
pkt1
rcv pkt1
(detect duplicate)
send ack1
ack1
rcv ack1
send pkt0
pkt0
rcv pkt0
send ack0
ack0
(c) ACK loss
send pkt0
pkt0
rcv pkt0
send ack0
ack0
rcv ack0
send pkt1
pkt1
rcv pkt1
send ack1
ack1
timeout
resend pkt1
pkt1
rcv pkt1
(detect duplicate)
send ack1
rcv ack1
send pkt0
pkt0
ack1
rcv pkt0
send ack0
rcv ack1
(ignore)
ack0
pkt1
(d) premature timeout/ delayed ACK
Is the protocol correct?
Computer Networks 36

rdt 3.0



Protocol failure!

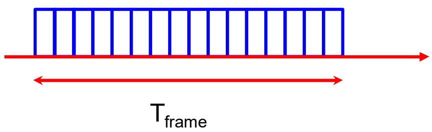
Throughput



BW = 10 Mbps = 10^7 bps

in 1 s => 10.000.000 bits

1 bit => T_{bit} = 0.1 μs

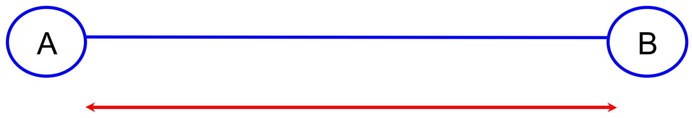


BW = 10 Mbps

Frame of 1500 bytes = 12000 bits

Time to generate a frame: T_{frame} = 1200 μs

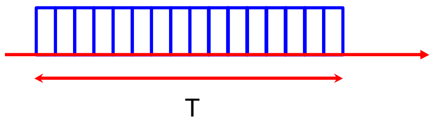
Throughput



1 km channel

v ≃ 200.000 km/s

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

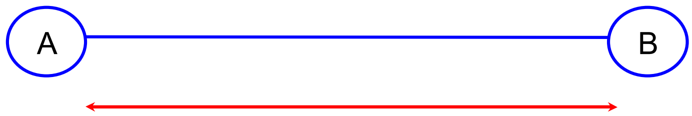


BW = 10 Mbps

in 5 μs => 50 bits

N° of bits simultaneously present in the channel

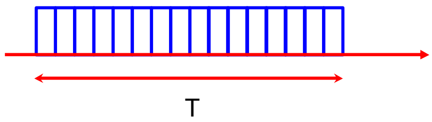
Throughput



10 Mbps - 1 km

$T_{propagation} \backsimeq 5\ \mu s$

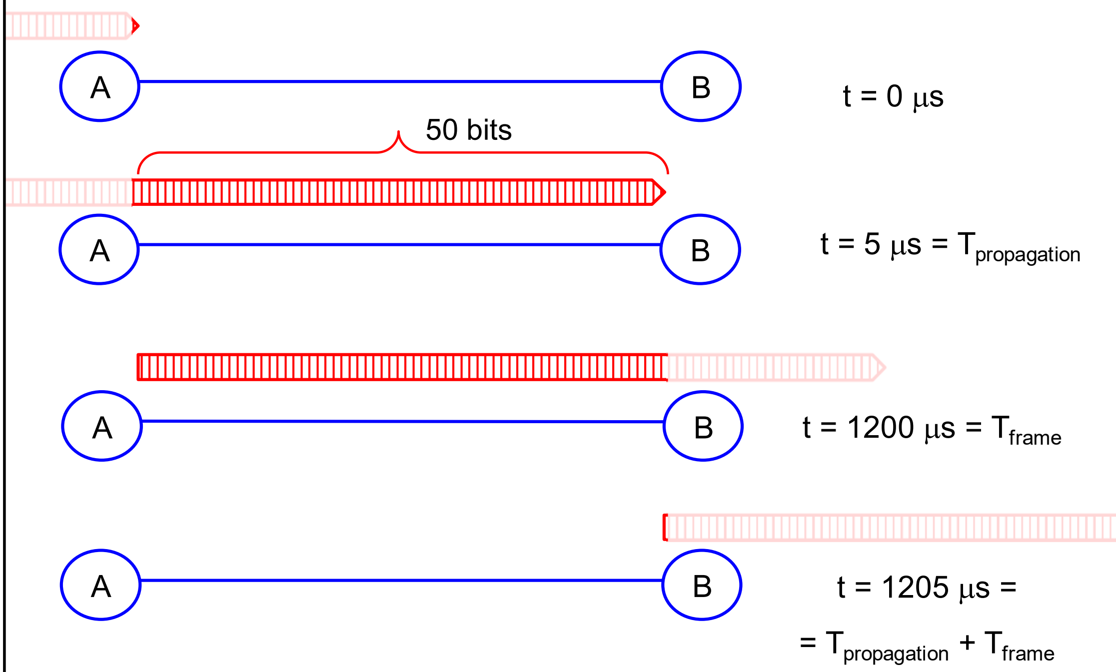
Frame 1500 bytes = 12000 bits



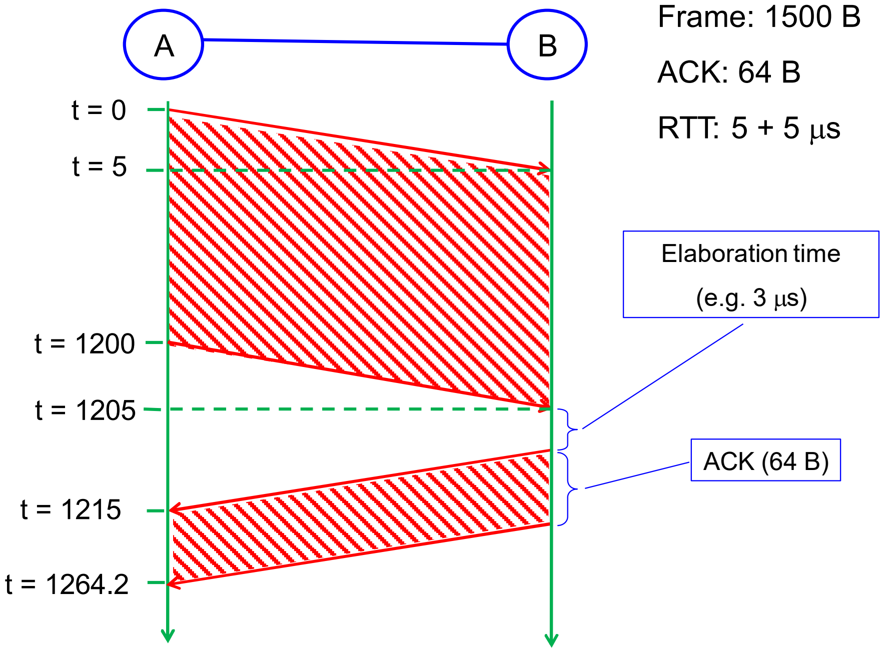
$T_{\ transfer} = 1200\ \mu s + 5\ \mu s$

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

Throughput

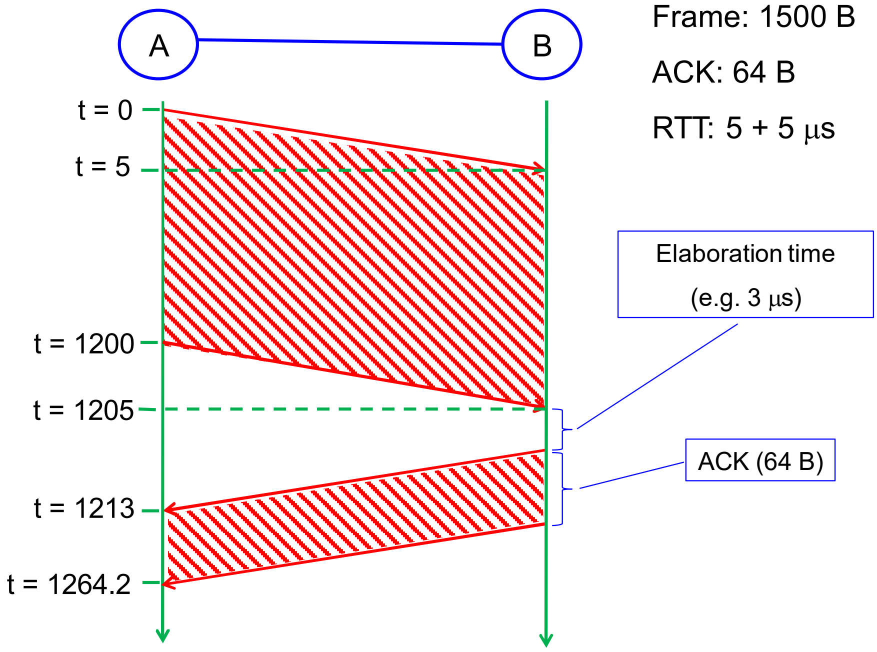

A
B
t = 0 µs
50 bits
A
B
t = 5 µs = T_propagation
A
B
t = 1200 µs = T_frame
A
B
t = 1205 µs =
= T_propagation + T_frame

Actual Throughput



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

Actual Throughput



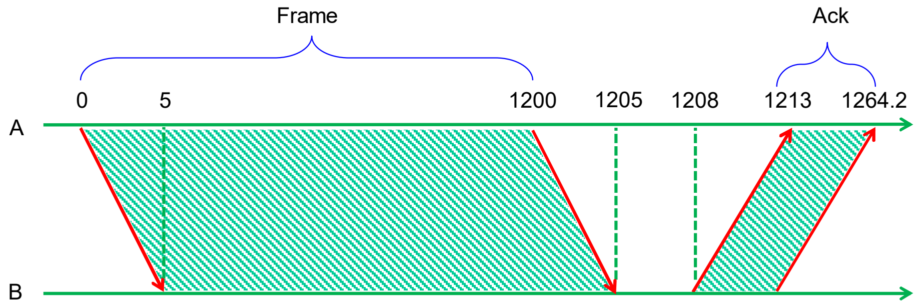
Frame: 1500 B

ACK: 64 B

RTT: 5 + 5 μs

$$T = 1200 + 2 \cdot 5 + 3 + 51.2 = 1264.2 \mu s$$

Actual Throughput



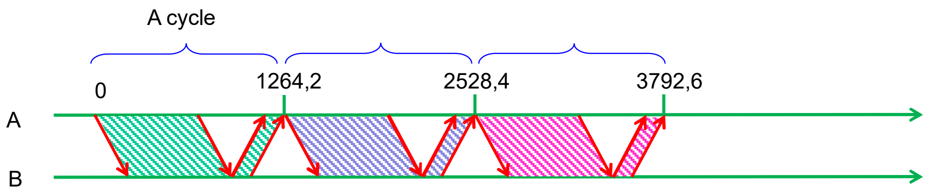
1500 bytes sent in 1264.2 μs

Actual Throughput $= 1500 \cdot 8 / 1264.2 \cdot 10^{-6}$ [b/s]

$= 9.492\ 10^6$ bps

$= 9.492$ Mbps

Actual Throughput



Frame: 1500 B

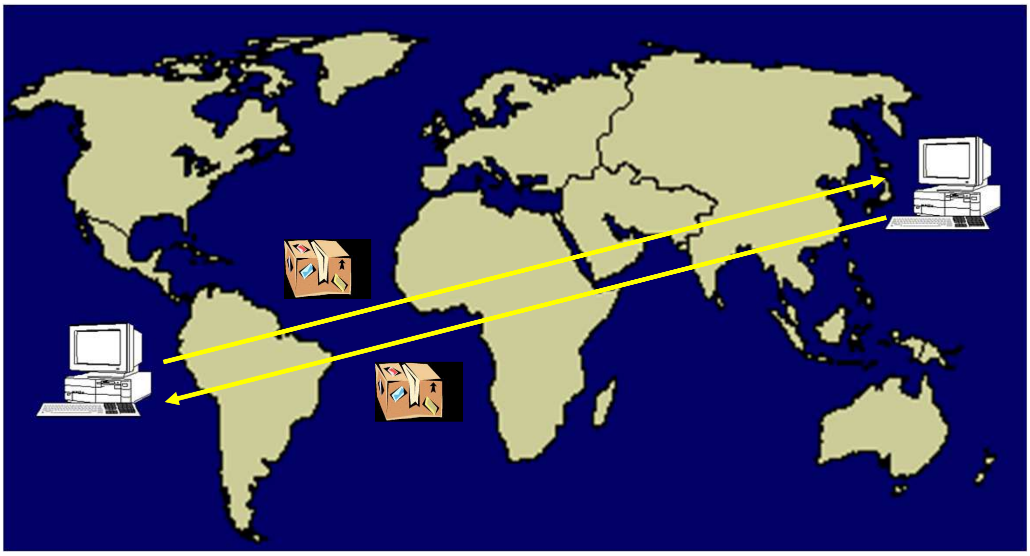
Ack: 64B

BW: 10Mbps

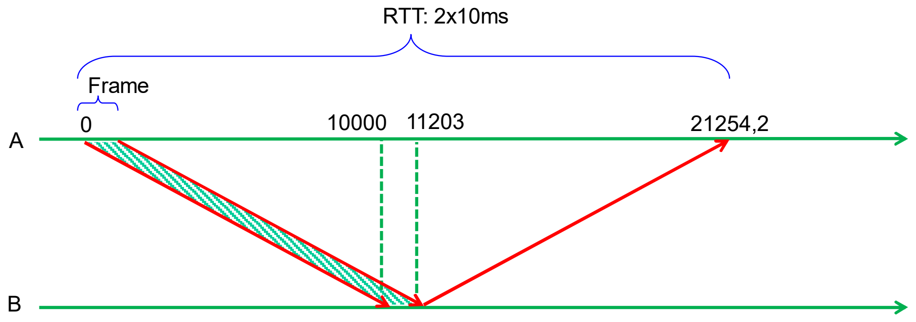
Th_{actual} : 9.492 Mbps

High RTT

Stop and wait protocol



Actual Throughput



Frame: 1500 B

Ack: 64B

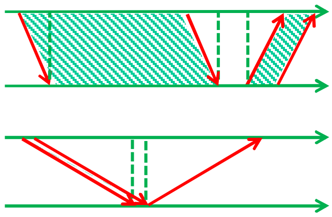
BW: 10Mbps

RTT: 20ms

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

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

Actual Throughput



Frame: 1500 B

Ack: 64B

BW: 10Mbps

RTT: 10 μs   T = 1264.2 μs   9.492 Mbps

RTT: 20 ms   T = 21254.2 μs   0.565 Mbps = 565 Kbps

Test on Internet

```
$ 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
```

Test on Internet

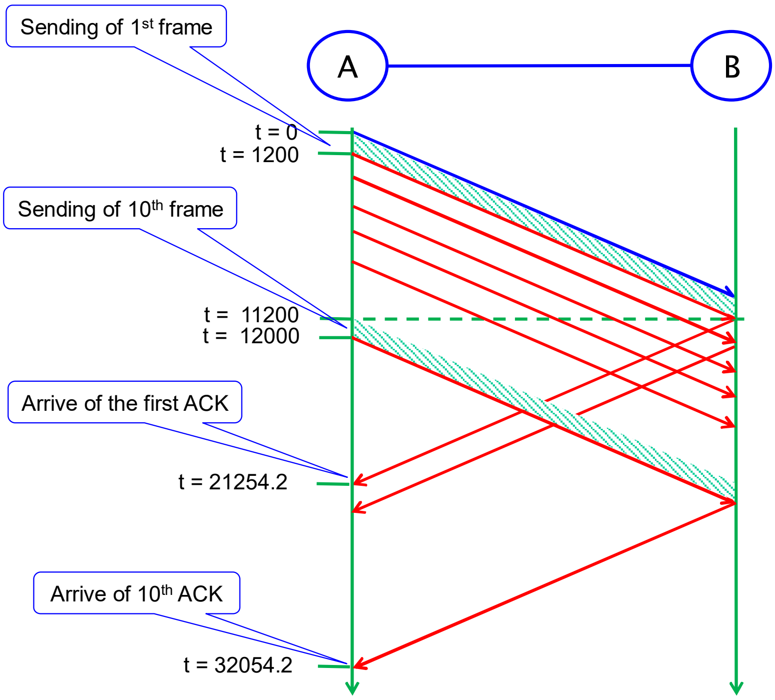
Geolocation data from IP2Location (Product: DB6, 2024-3-1)

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

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

IP ADDRESS: 216.58.205.36	**ISP:** Google LLC
COUNTRY: Italy	**ORGANIZATION:** Google LLC
REGION: Lombardy	**LATITUDE:** 45.4642
CITY: Milan	**LONGITUDE:** 9.18998

Pipeline



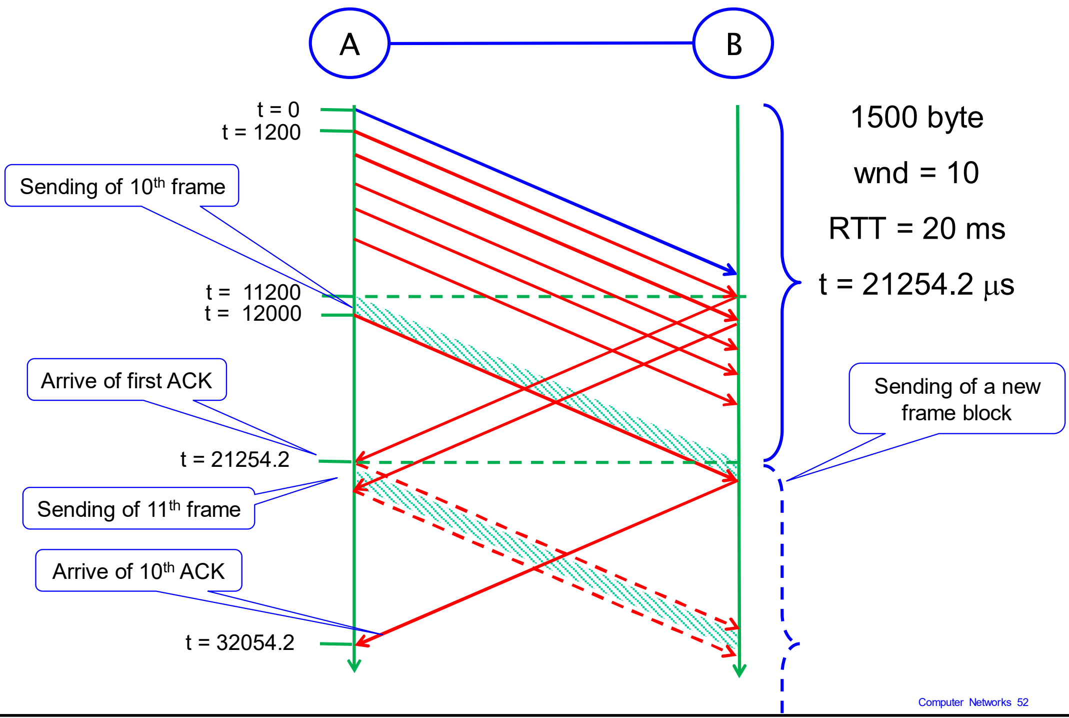
1500 byte

wnd = 10

RTT = 20 ms

t = 32054.2 µs

Pipeline



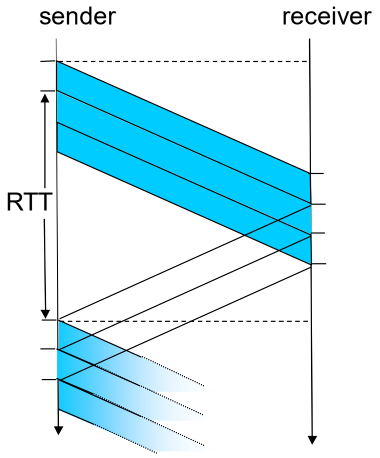
A
B
t = 0
t = 1200
Sending of 10th frame
t = 11200
t = 12000
Arrive of first ACK
t = 21254.2
Sending of 11th frame
Arrive of 10th ACK
t = 32054.2
1500 byte
wnd = 10
RTT = 20 ms
t = 21254.2 µs
Sending of a new frame block

Pipeline

Frame: 1500 B

Ack: 64B

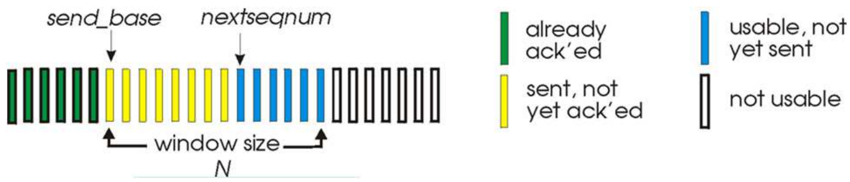
BW: 10Mbps



RTT 10 μs	wnd= 1	t = 1264.2 μs	9.477 Mbps
RTT 20 ms	wnd= 1	t = 21254.2 μs	0.565 Mbps
RTT 20 ms	wnd=10	t = 21254.2 μs	5.646 Mbps

Go-Back-N: sender

- 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

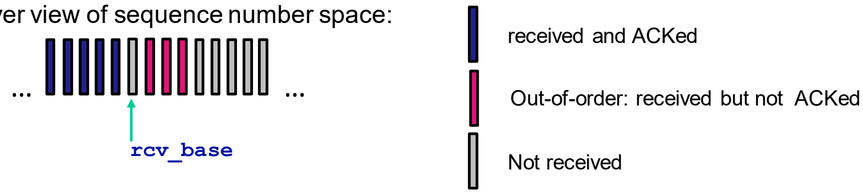
Go-Back-N: receiver

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

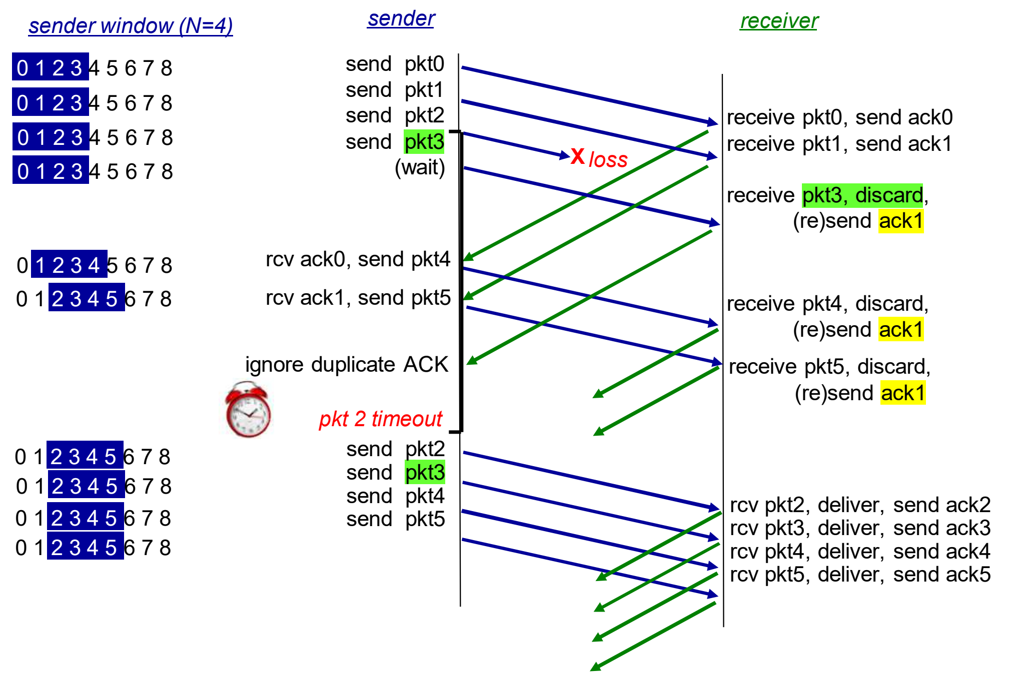
- may generate duplicate ACKs
- need only remember **rcv_base**

- 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:



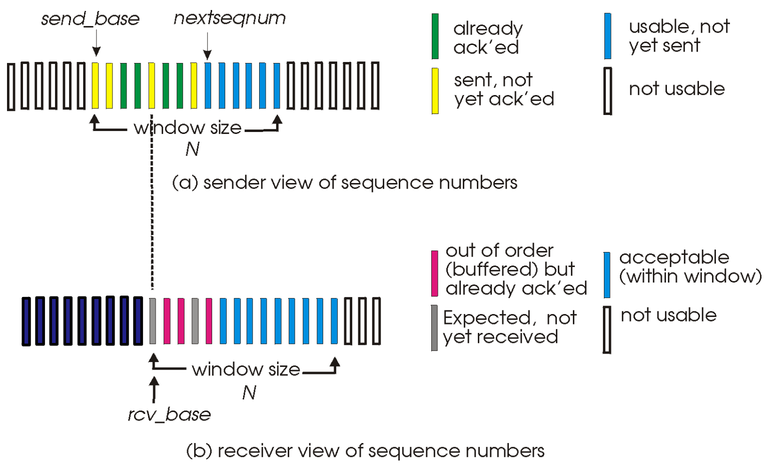
Go back N


sender window (N=4)
sender
receiver
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
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, discard,
(re)send ack1
ignore duplicate ACK
receive pkt5, discard,
(re)send ack1
pkt 2 timeout
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
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

Selective repeat

- 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



Università di Catania

Selective repeat: sender and receiver

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

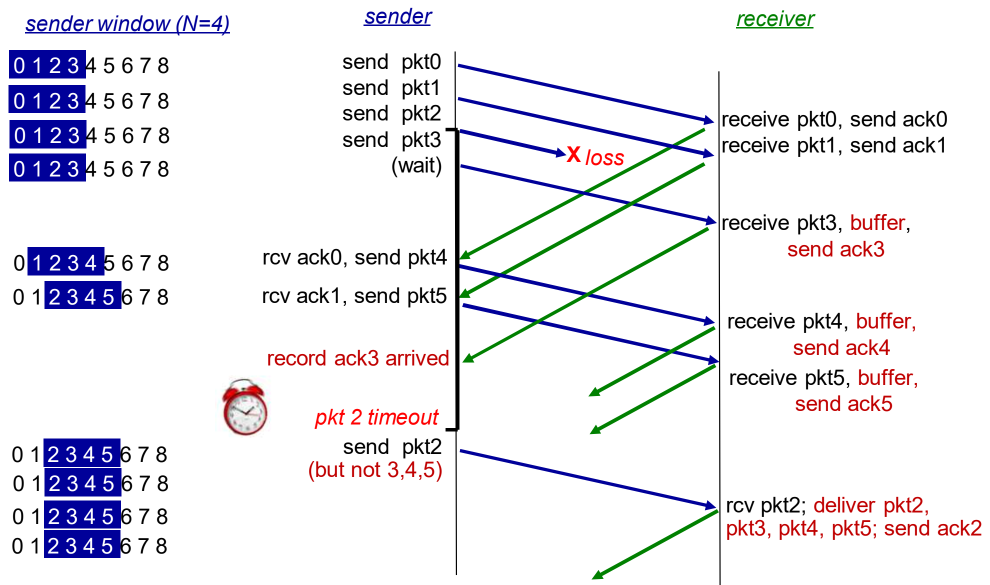
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

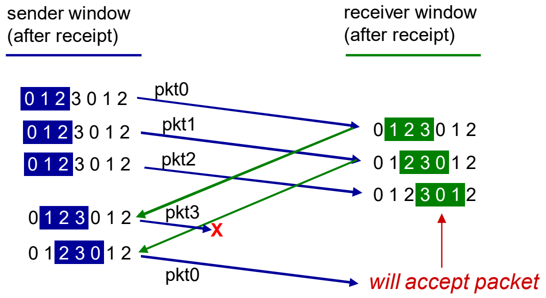
Selective repeat



Q: what happens when ack2 arrives?

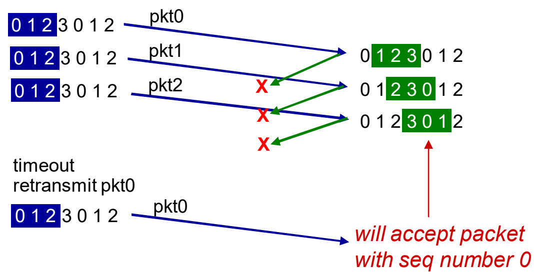
Selective repeat: a dilemma!

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



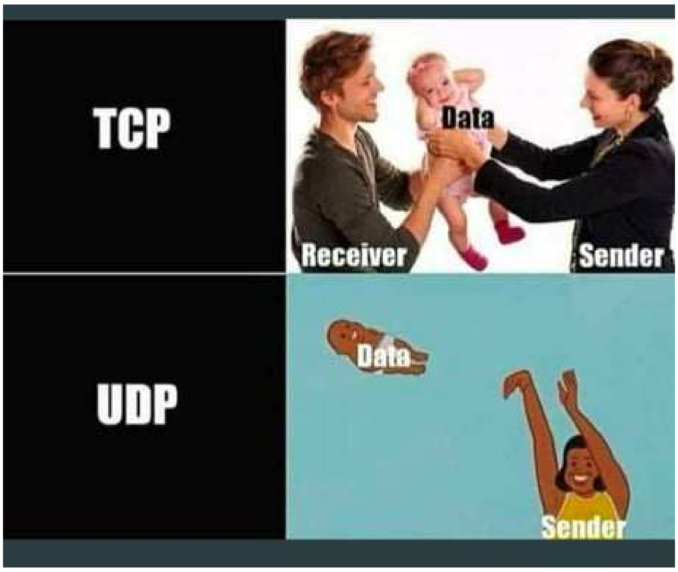
(a) no problem

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



(b) oops!

TCP vs UDP



TCP

- 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)

TCP features

- 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

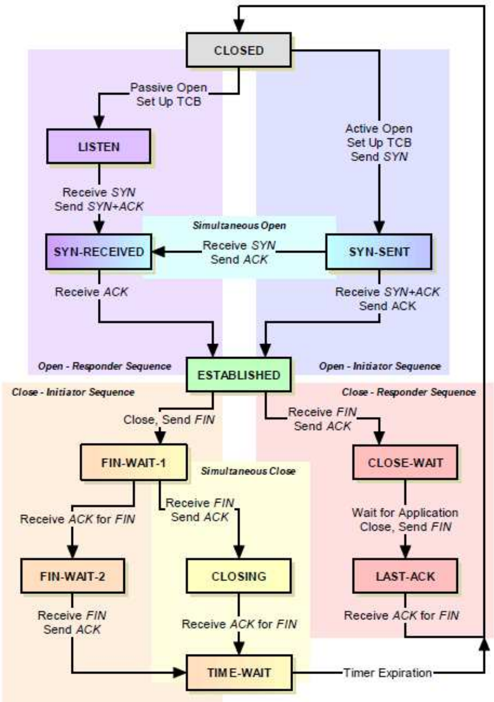
What TCP does not provide

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

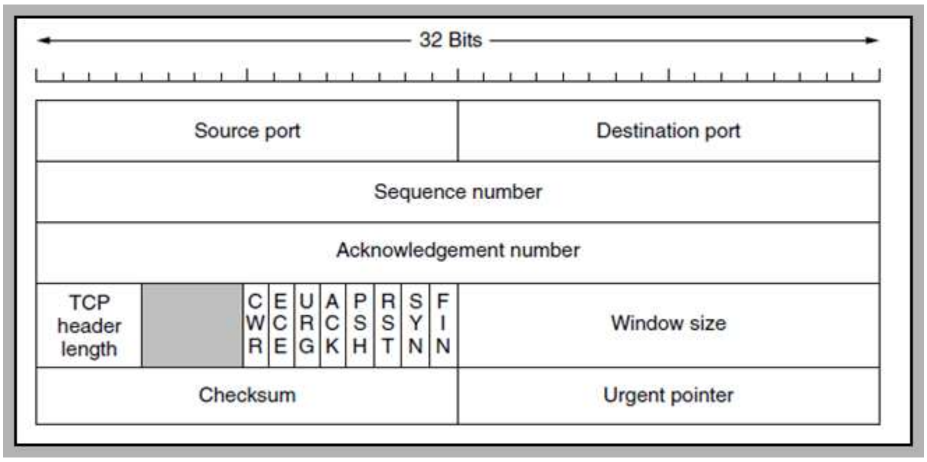
TCP Features

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

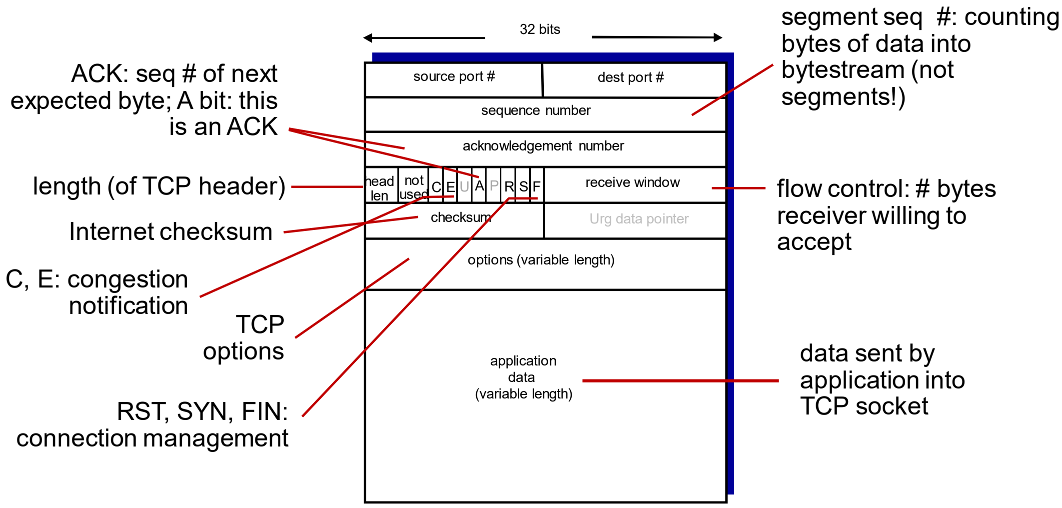
TCP state model



TCP segment structure



TCP segment structure



TCP sequence numbers, ACKs

Sequence numbers:

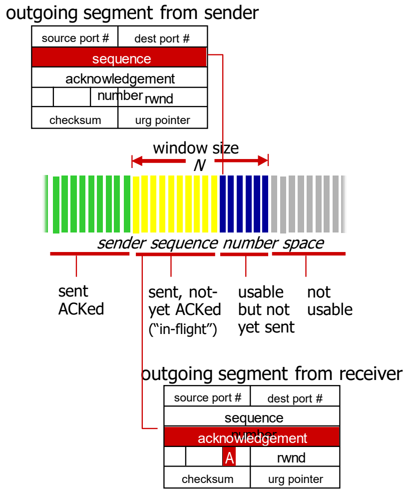
- byte stream "number" of first byte in segment's data

Acknowledgements:

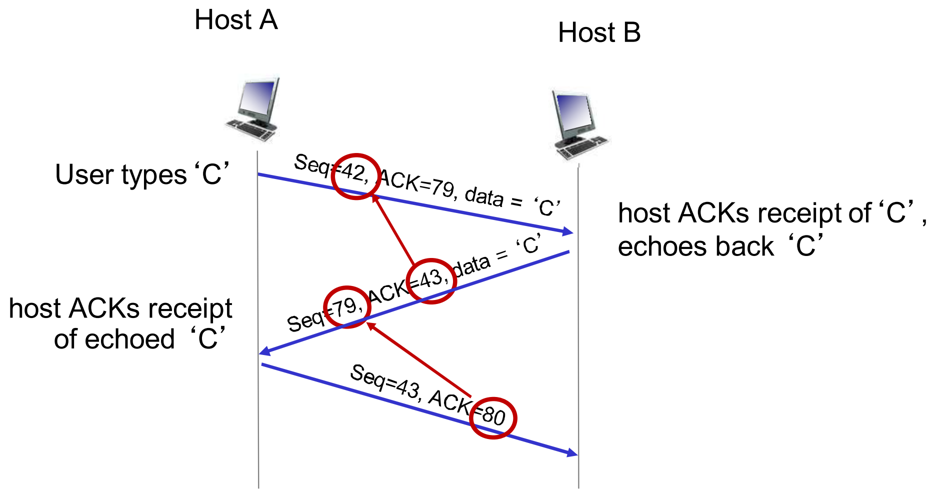
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs



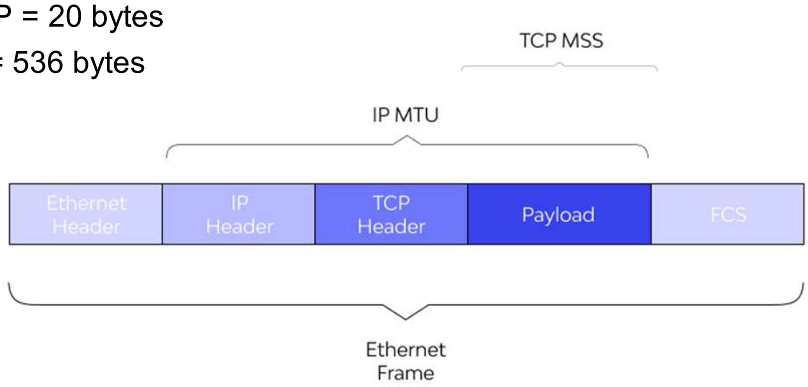
simple telnet scenario

TCP MSS

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



TCP Round Trip Time, timeout

Q: how to set TCP timeout value?

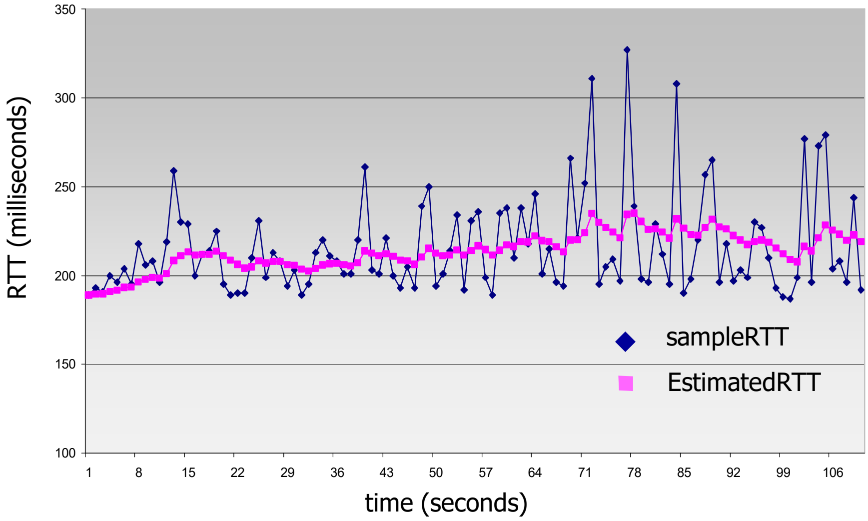
- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions (Karn's algorithm)
- SampleRTT will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current SampleRTT

Round Trip Time

Estimated RTT e Sample RTT



Round Trip Time

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

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

Typical value $\alpha = 0.125$

EWMA

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

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

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

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

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

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

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

$E\_RTT_{n} =$ **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)

Round Trip Time

SampleRTT can be subject to large variation.

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

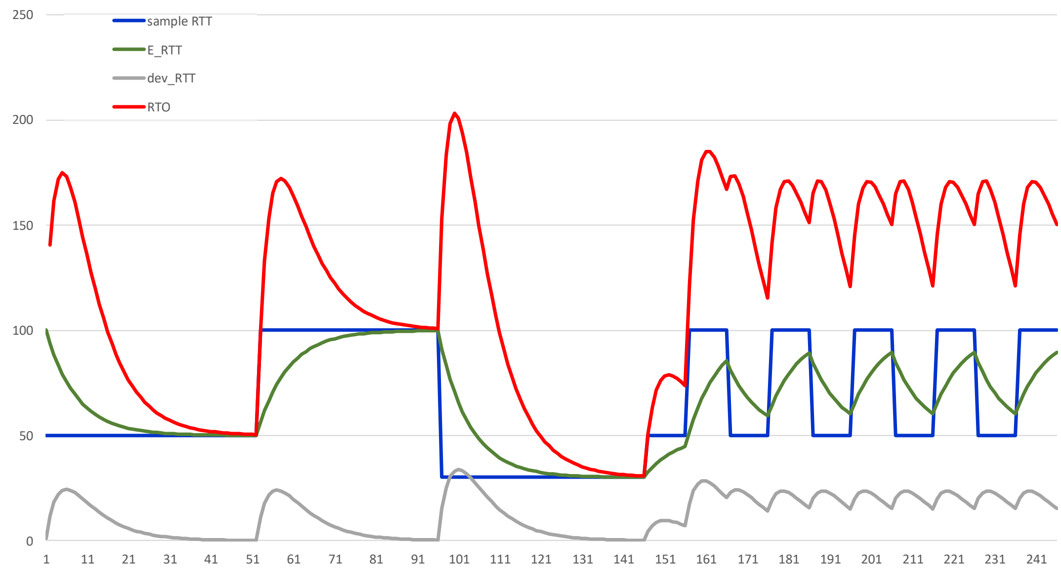
Typical value: $\beta = 0.25$

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

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

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

TCP Timer

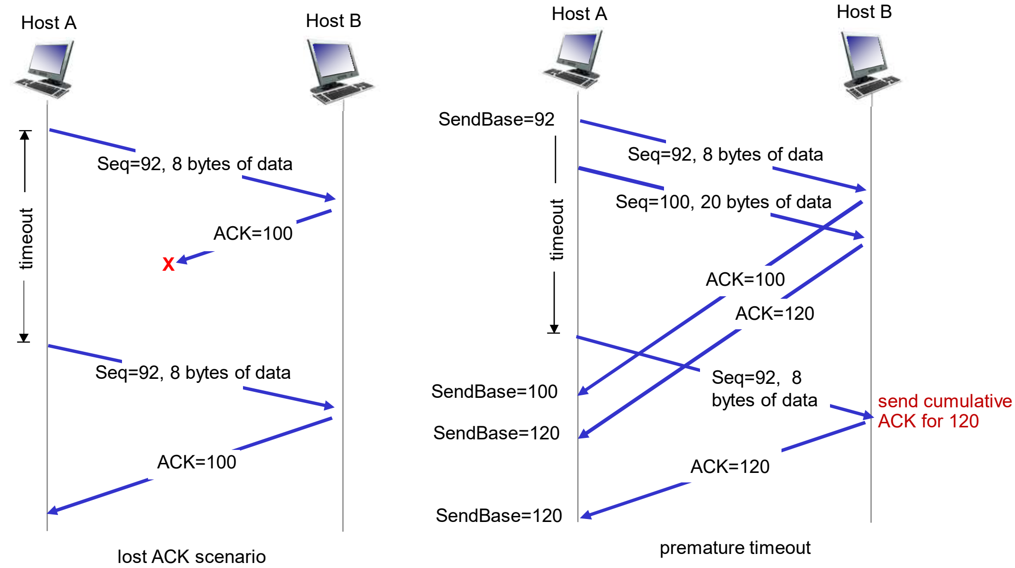


sample RTT
E_RTT
dev_RTT
RTO
250
200
150
100
50
0
1 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161 171 181 191 201 211 221 231 241

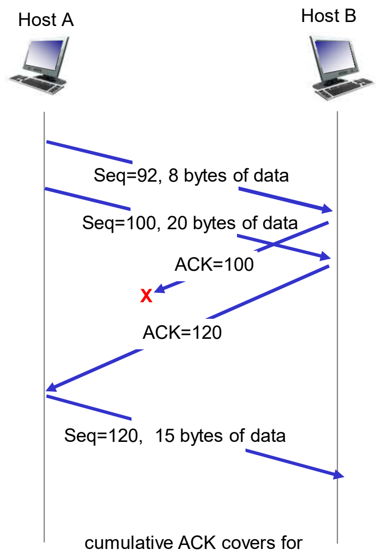
Timer

TCP uses only one timer at a time (RFC 6298), also if there are more than one segment "on the fly".

After the arrival of an ACK a new timer starts, linked with the next not confirmed segment.



Timer



cumulative ACK covers for
earlier lost ACK

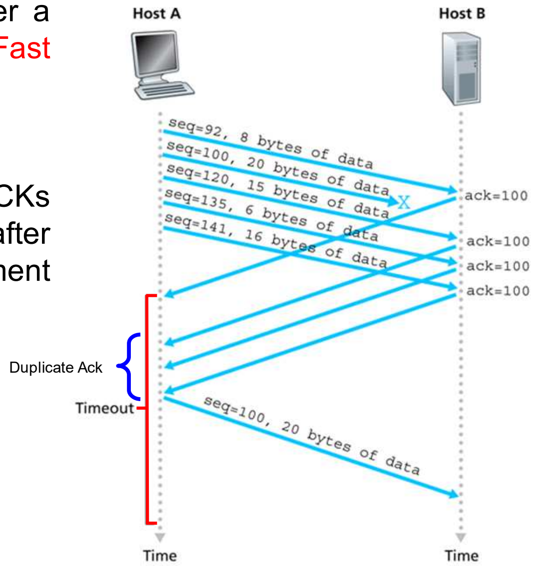
Timer

```
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 */
```

TCP Fast Retransmit

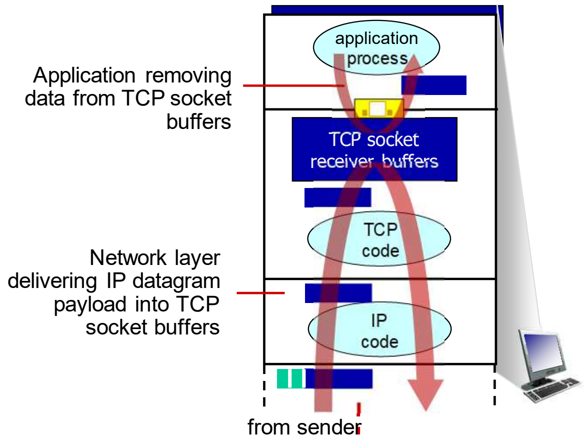
To avoid long waiting time after a packet loss TCP uses the "Fast Retransmit".

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



TCP flow control

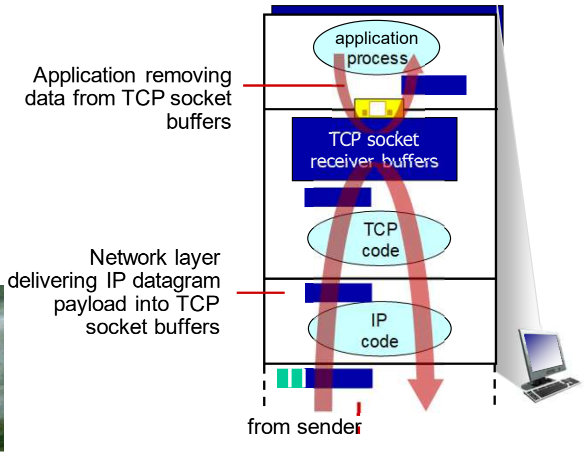
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

TCP flow control

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

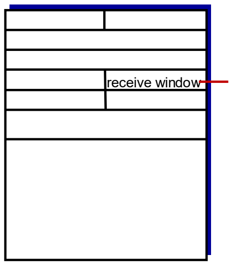


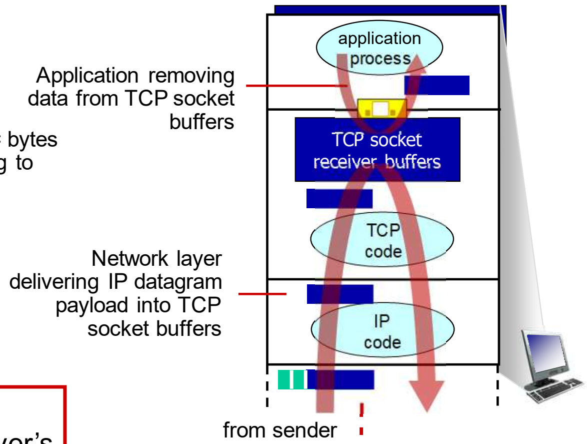
receiver protocol stack

Flow control

TCP flow control

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



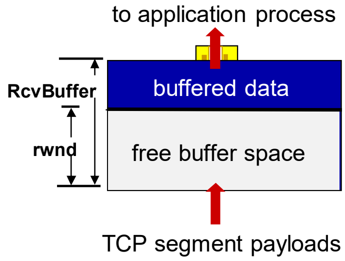


receiver protocol stack

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

TCP flow control

- 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

Nagle's Algorithm (RFC 896)

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
```

Nagle's Algorithm (RFC 896)

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

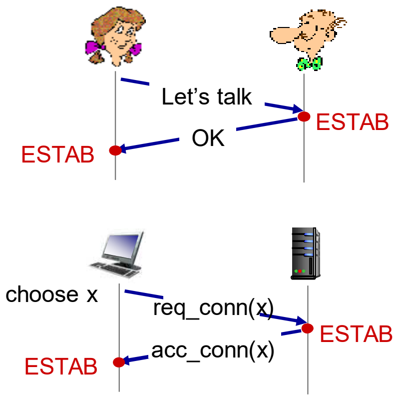
In networks with high RTT, data are bufferized and sent in large packets.

```
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.

Agreeing to establish a connection

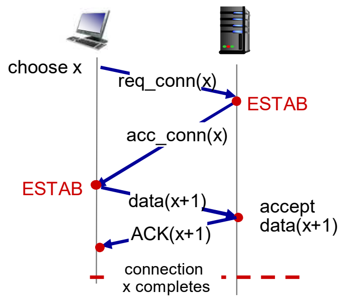
2-way handshake:



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

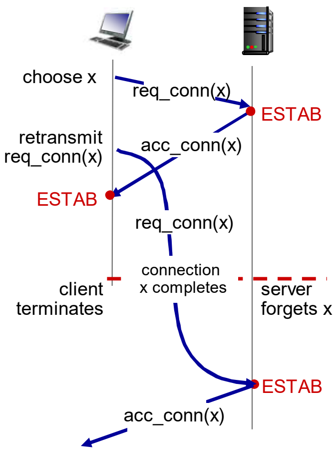
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

2-way handshake scenarios



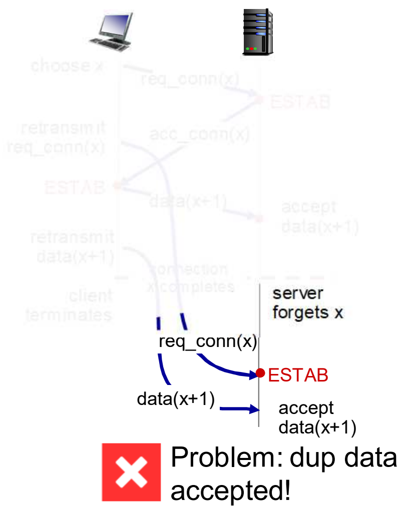
No problem!

2-way handshake scenarios

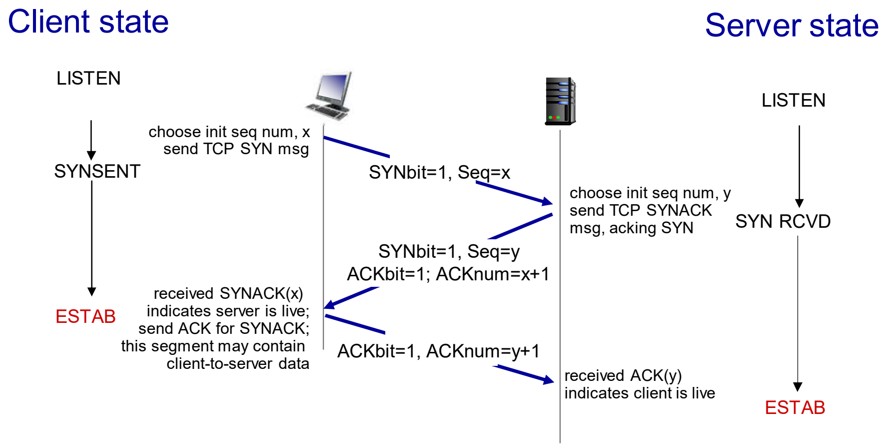


Problem: half open connection! (no client)

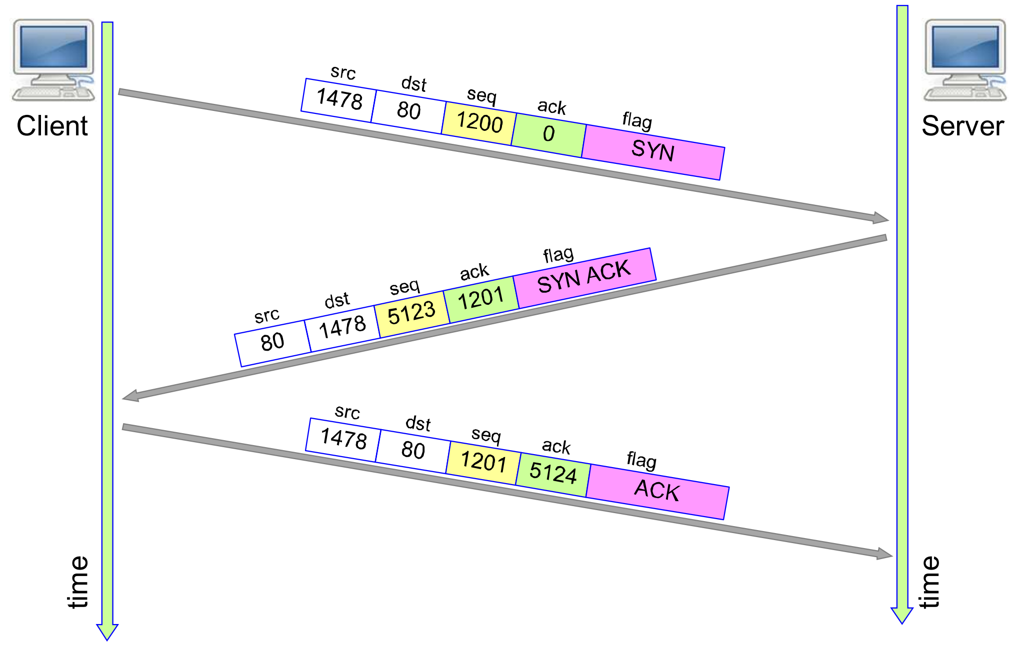
2-way handshake scenarios



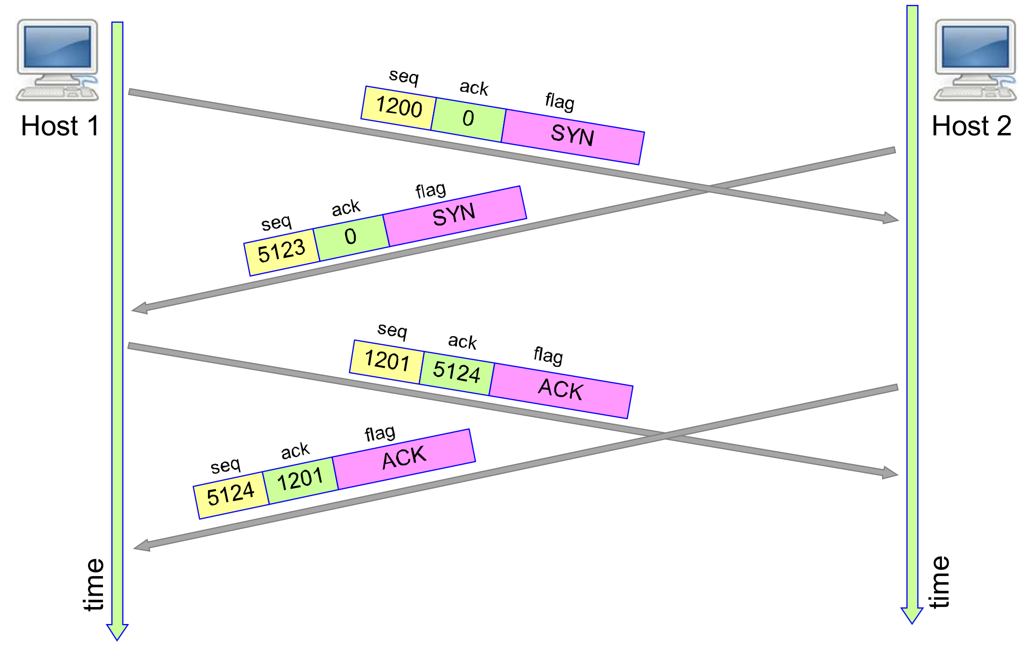
TCP 3-way handshake



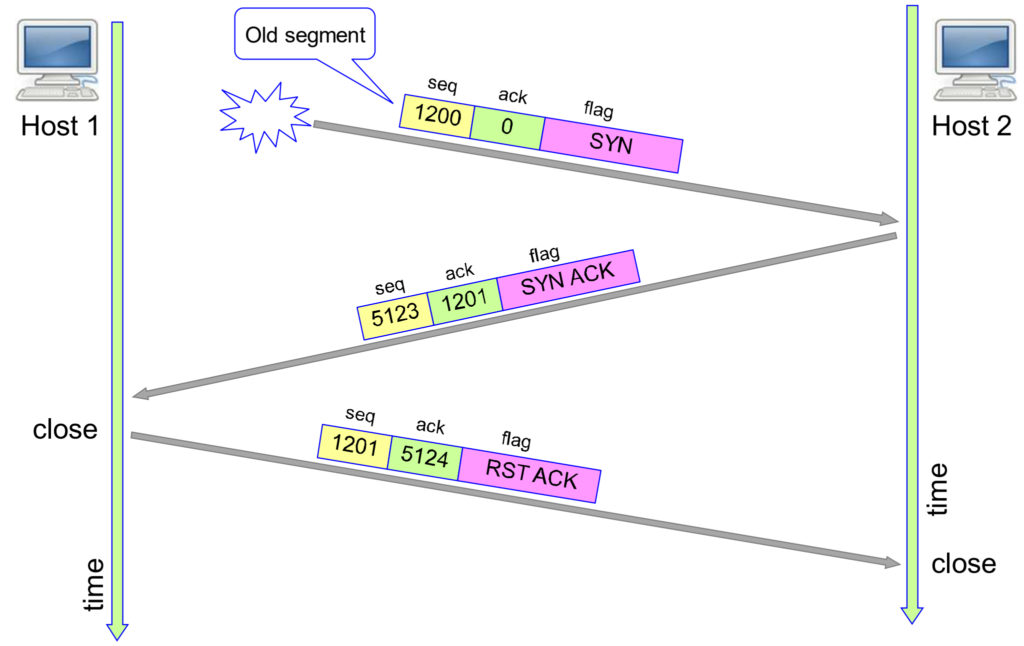
TCP 3-way handshake



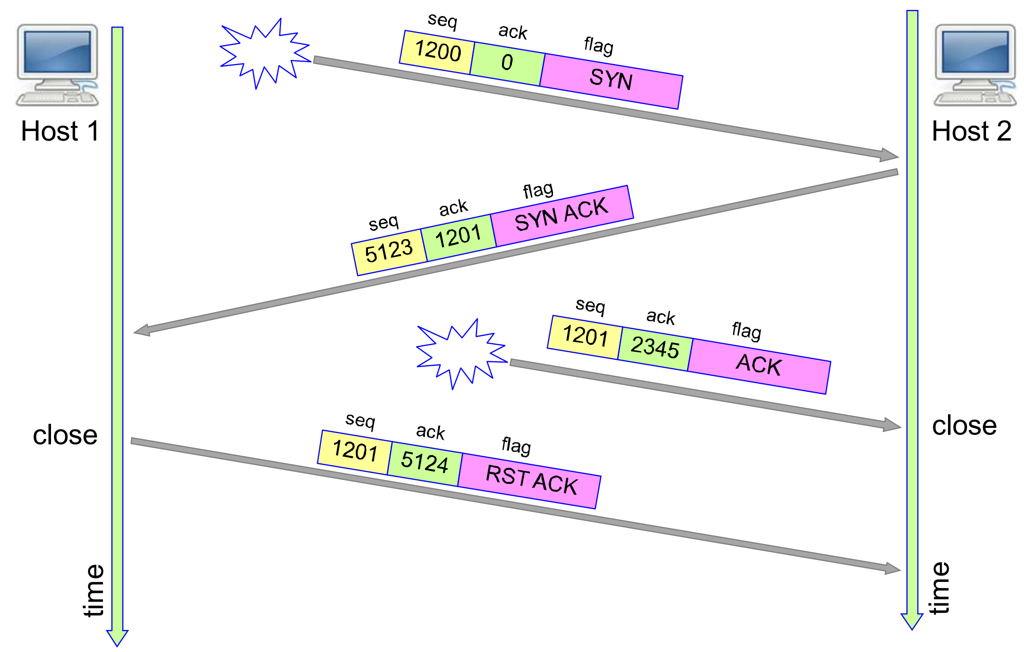
TCP 3-way handshake



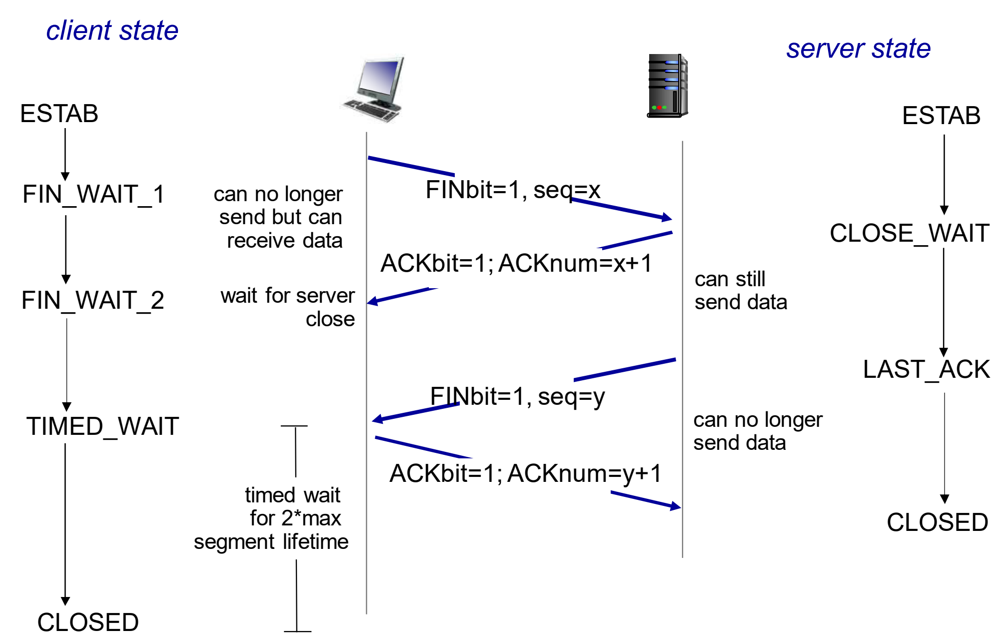
TCP 3-way handshake



TCP 3-way handshake

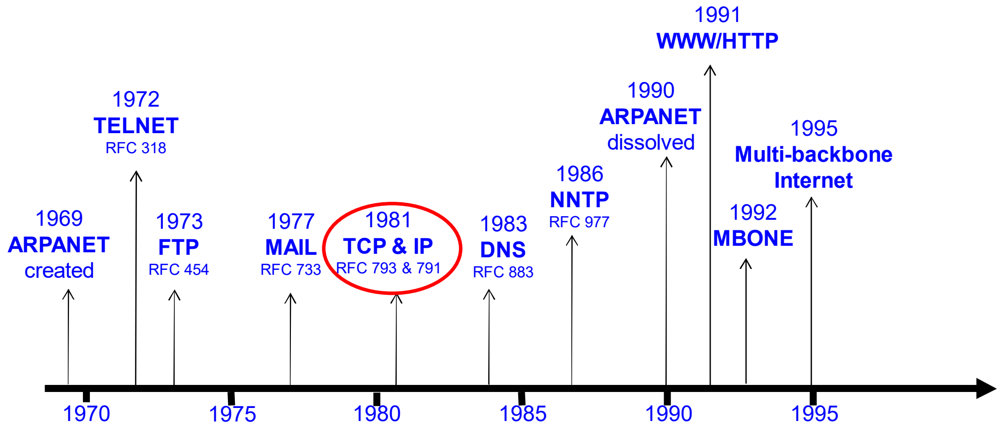


TCP: closing a connection

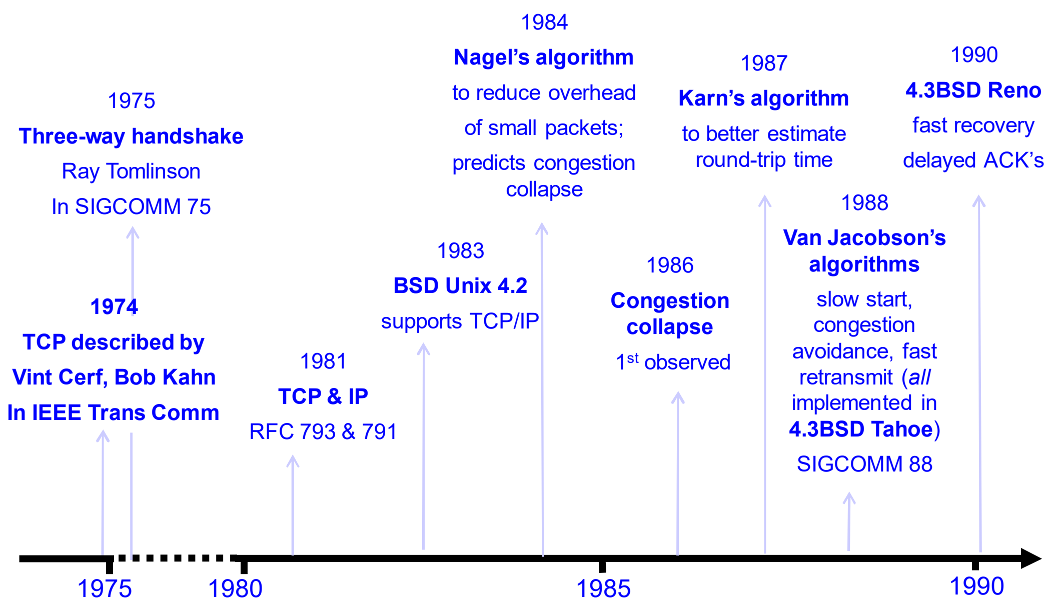


Università di Catania

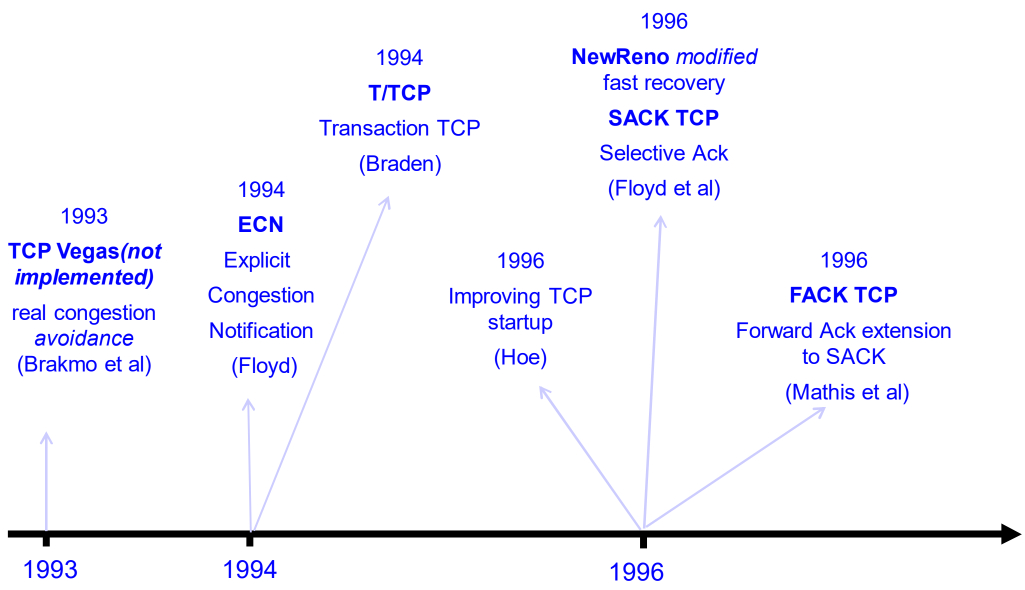
A brief Internet history...



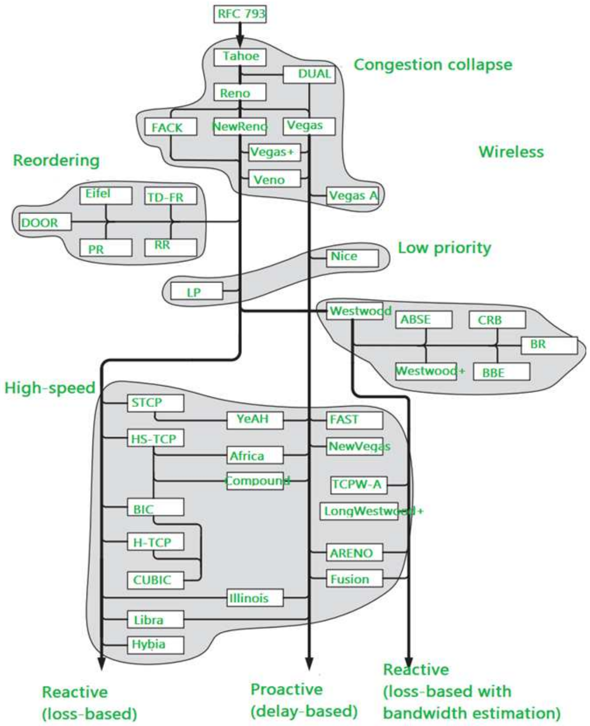
Evolution of TCP



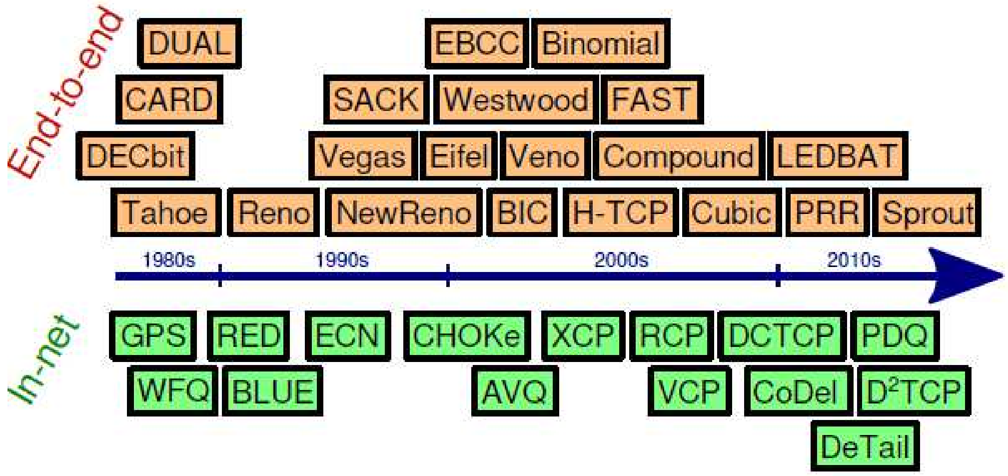
Evolution of TCP



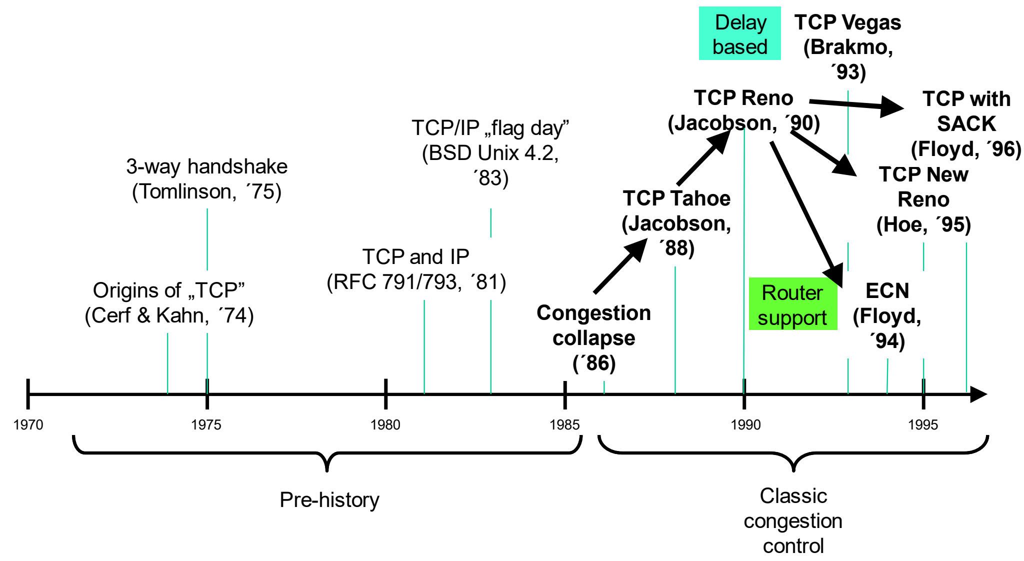
More than 100 TCP variants …



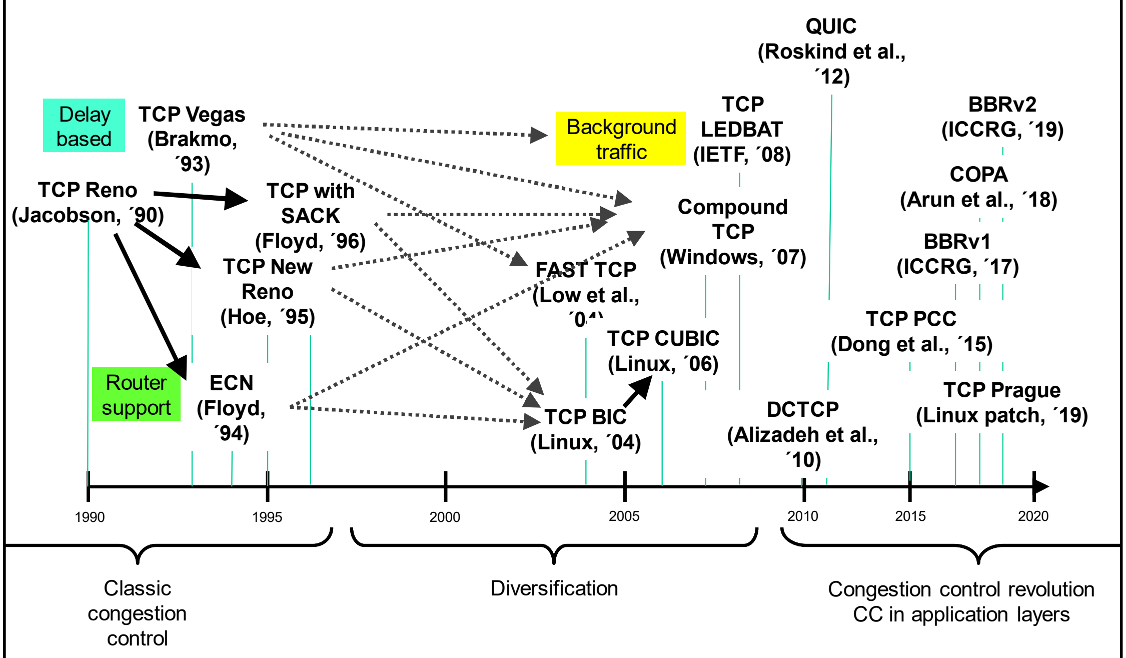
TCP variants



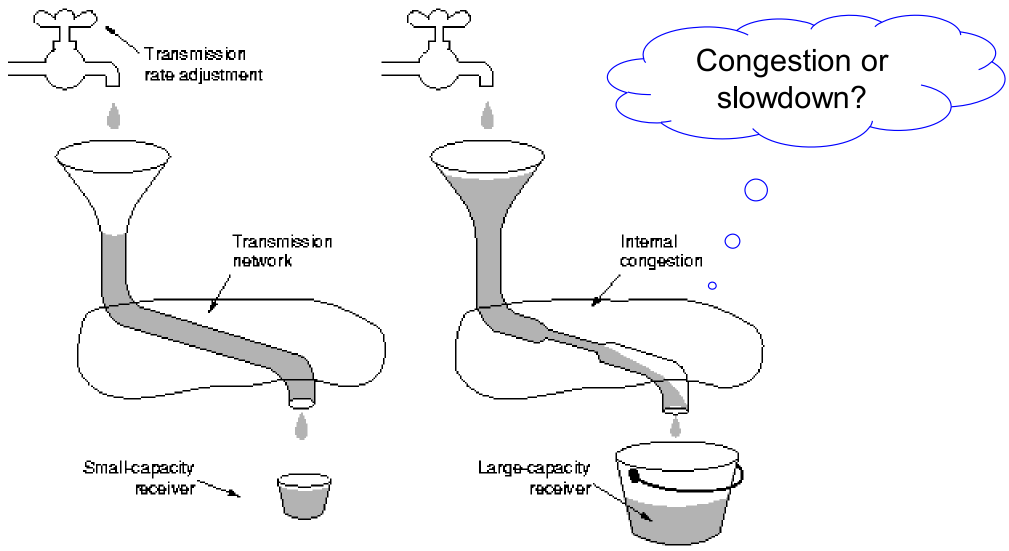
TCP variants



TCP variants



Flow control vs Congestion control



Università di Catania

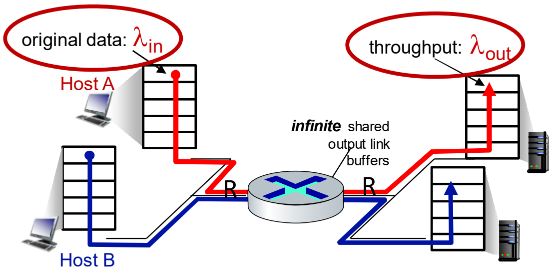
Congestion control

Congestion or delay?

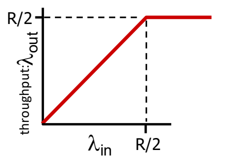
Causes/costs of congestion: scenario 1

Simplest scenario:

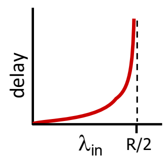
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed



Q: What happens as arrival rate λ_{in} approaches R/2?



maximum per-connection throughput: R/2



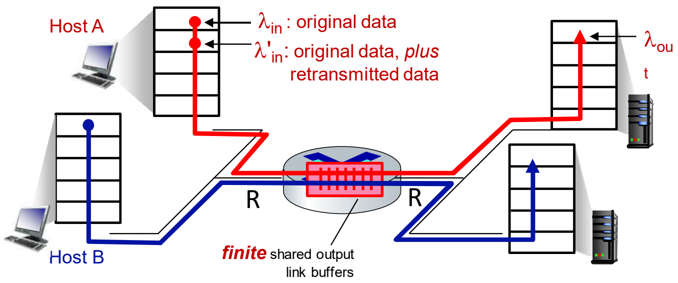
large delays as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

One router, *finite* buffers

sender retransmits lost, timed-out packet

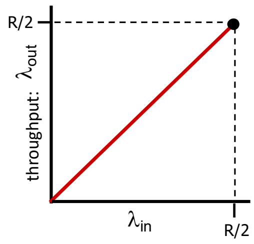
- application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
- transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

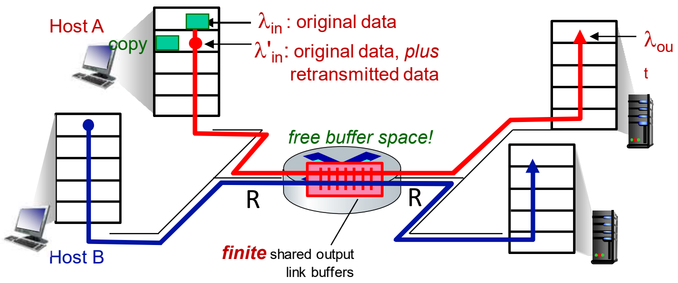


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

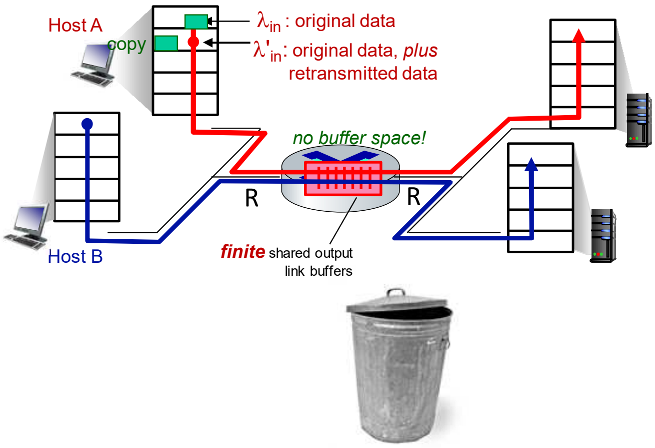




Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

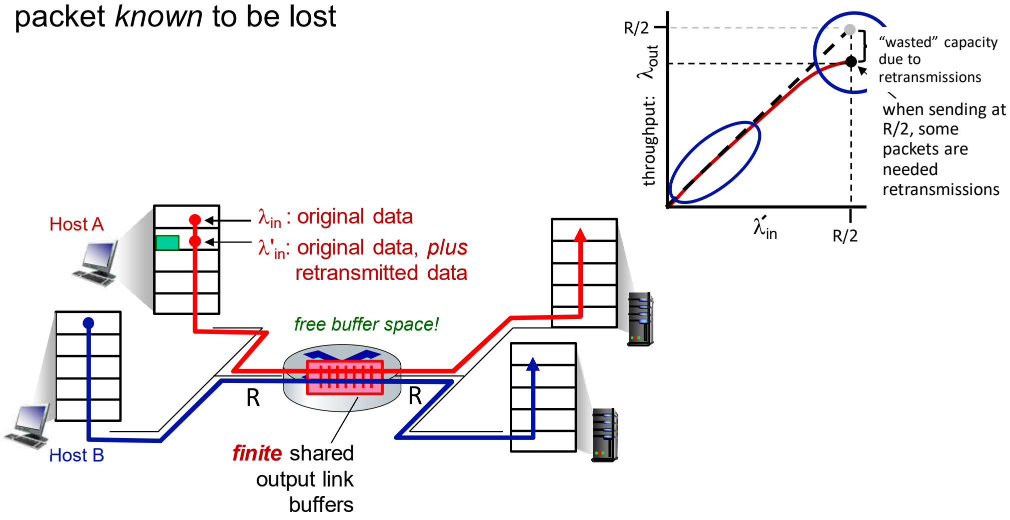
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

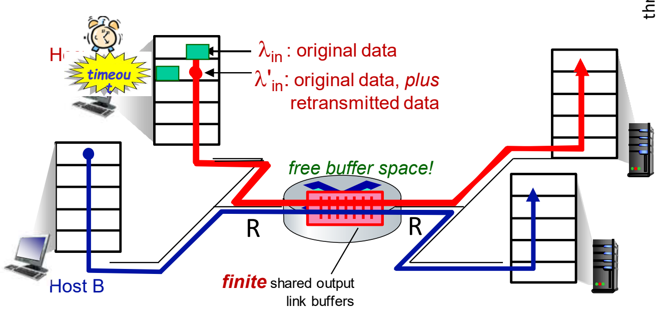
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

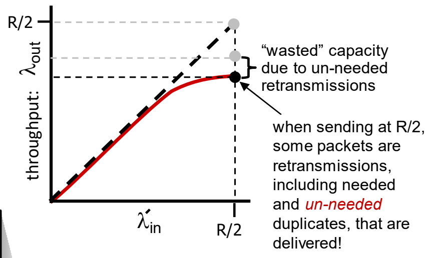


Causes/costs of congestion: scenario 2

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

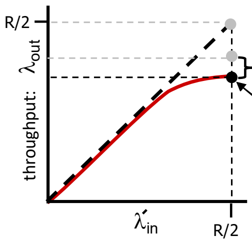




Causes/costs of congestion: scenario 2

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:

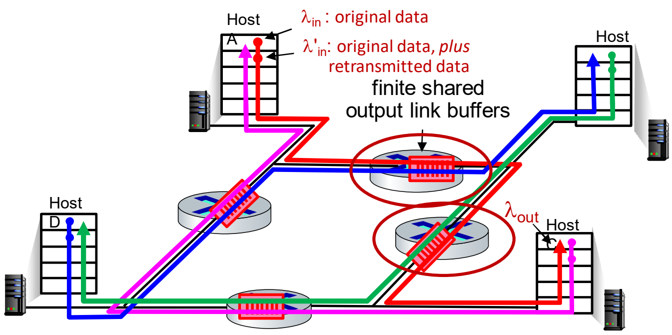
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3

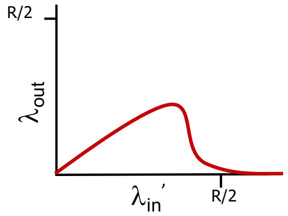
- *four* senders
- *multi-hop* paths
- timeout/retransmit

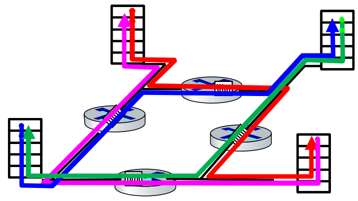
Q: what happens as λ_{in} and λ_{in}' increase ?

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



Causes/costs of congestion: scenario 3



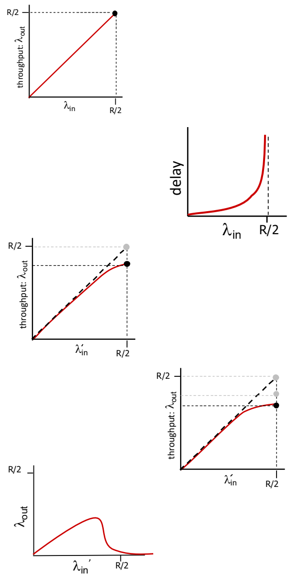


another "cost" of congestion:

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

Causes/costs of congestion: insights

- 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

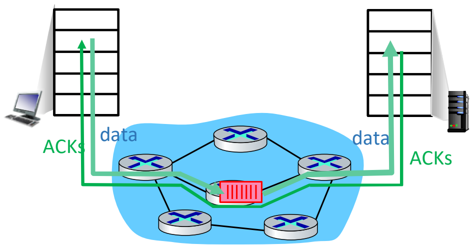


Approaches towards congestion control

End-end congestion control:

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

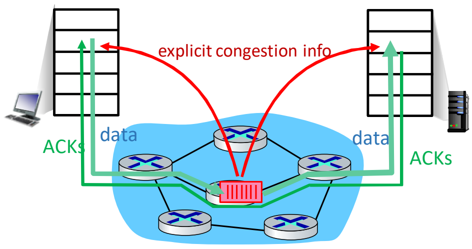
approach taken by TCP



Approaches towards congestion control

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

TCP congestion control: AIMD

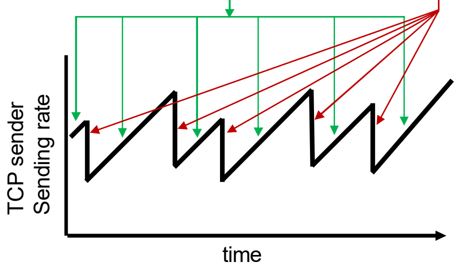
Approach: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

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

TCP AIMD: more

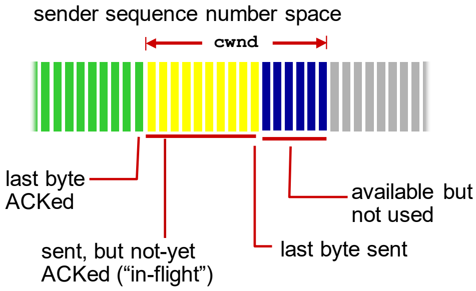
Multiplicative decrease detail: sending rate is

- 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 congestion control: details



TCP sending behavior:

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

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

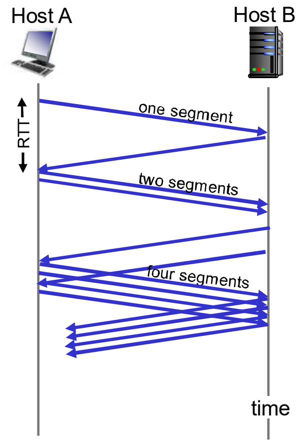
- TCP sender limits transmission:

  LastByteSent - LastByteAcked ≤ cwnd

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

TCP slow start

- 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

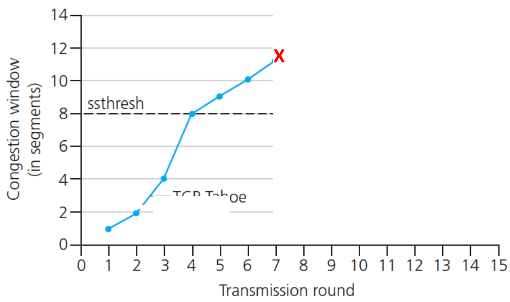
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

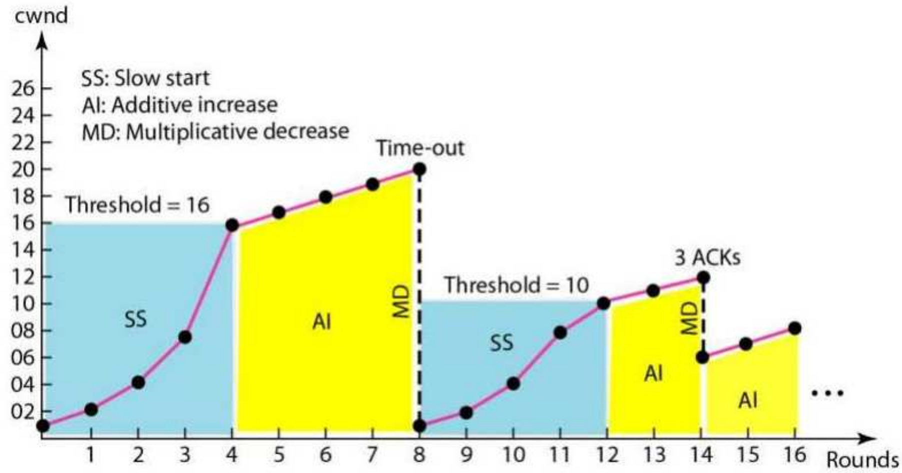
A: when **cwnd** gets to 1/2 of its value before timeout.

Implementation:

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



TCP: from slow start to congestion avoidance



TCP Tahoe - TCP Reno

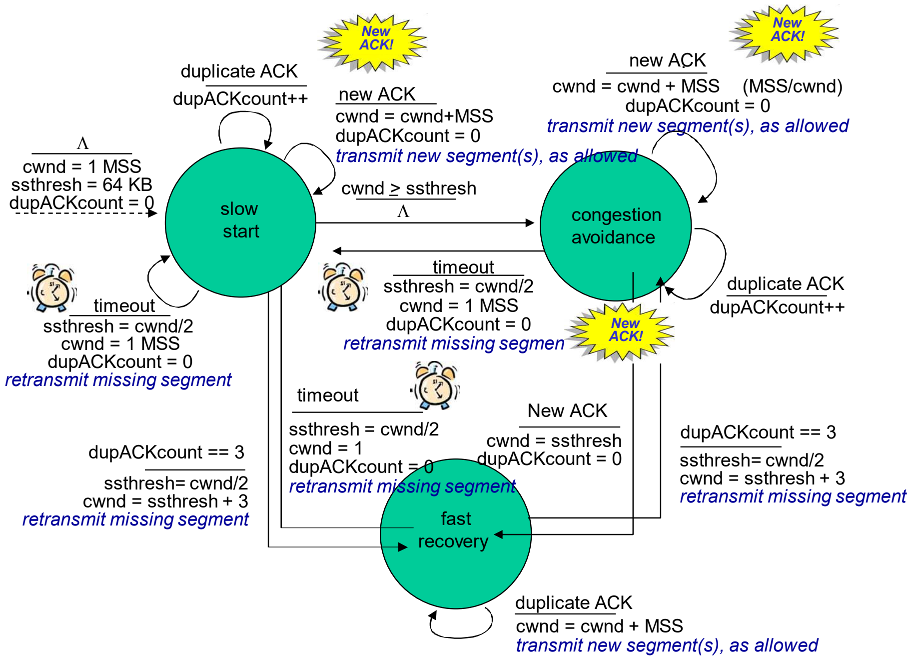
TCP-Tahoe: implements:

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.

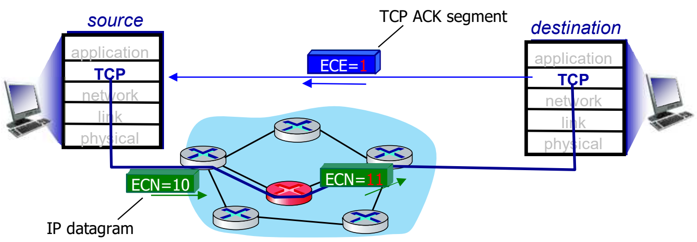
Summary: TCP congestion control



Explicit congestion notification (ECN)

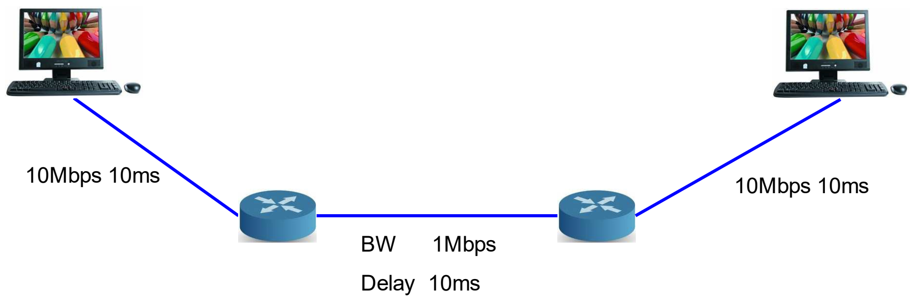
TCP deployments often implement *network-assisted* congestion control:

- 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)



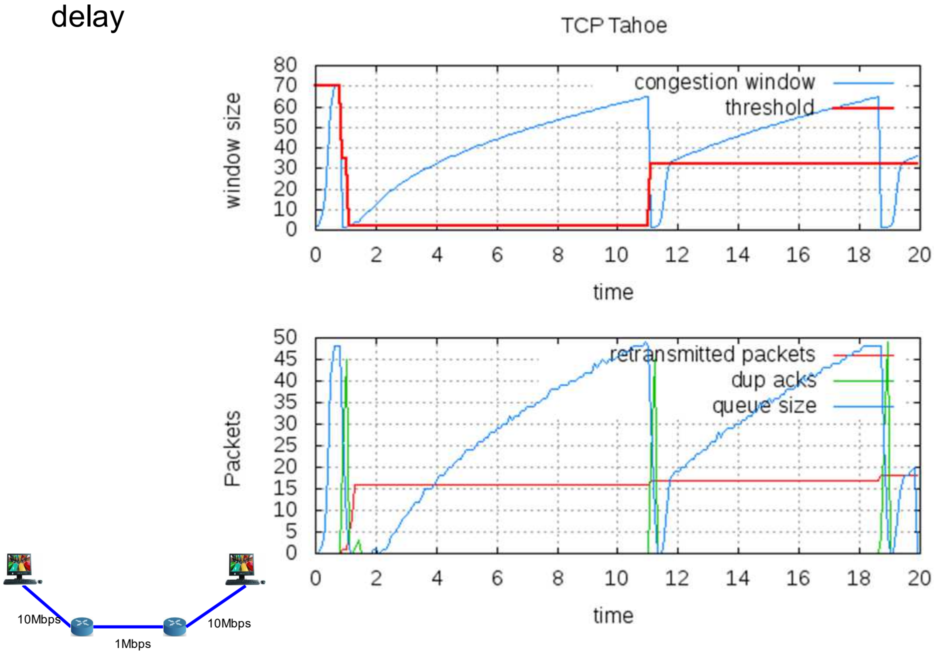
TCP congestion control: simulations

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



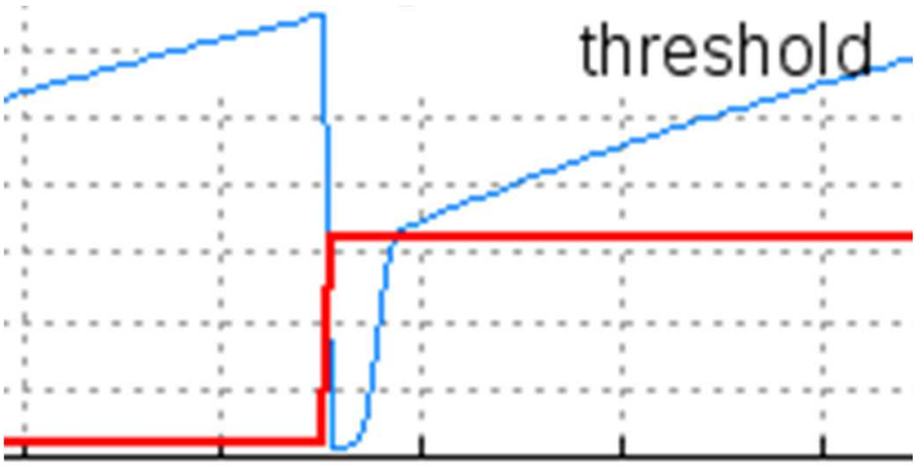
TCP congestion control: simulations

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



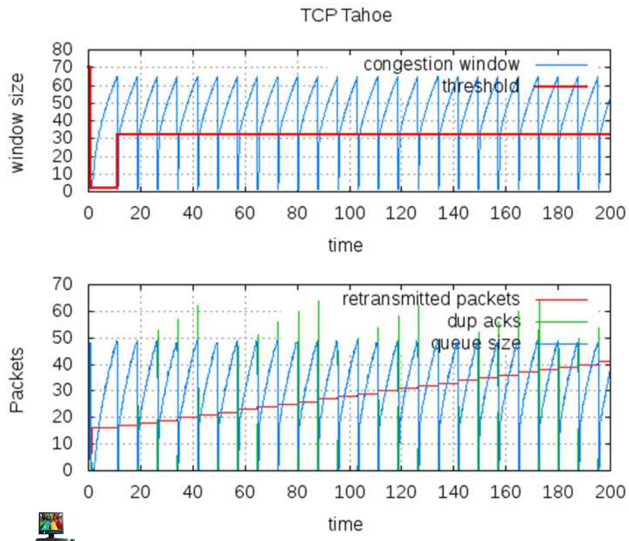
TCP congestion control: simulations

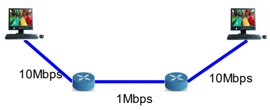
Case study 1: a TCP connection in a lossless channel, with constant delay



TCP congestion control: simulations

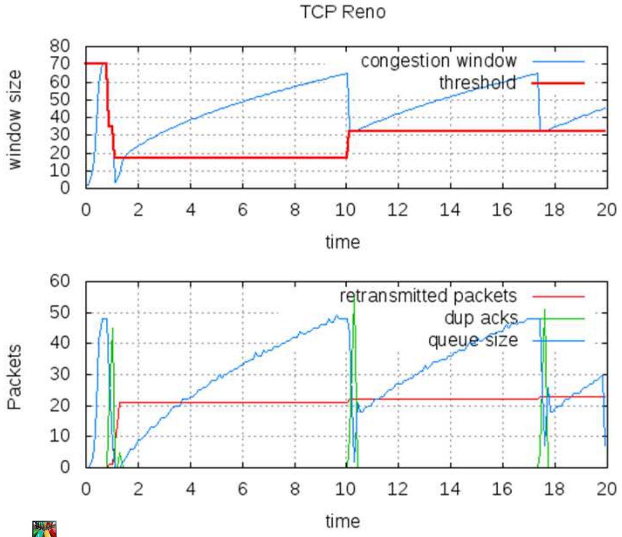
Case study 1: a TCP connection in a lossless channel, with constant delay

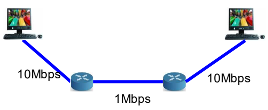




TCP congestion control: simulations

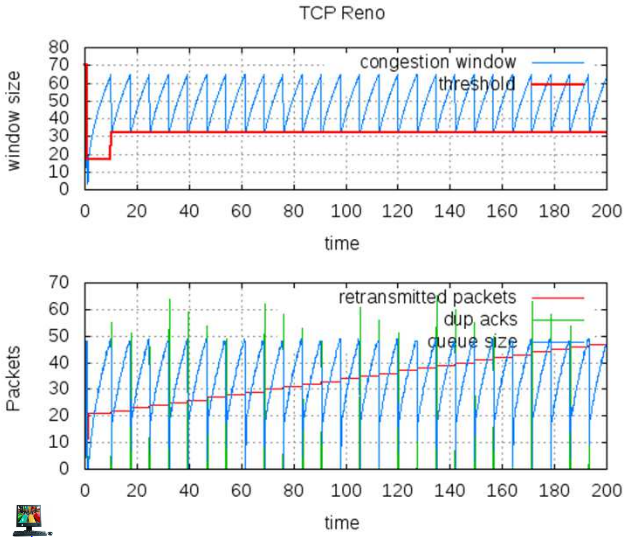
Case study 1: a TCP connection in a lossless channel, with constant delay

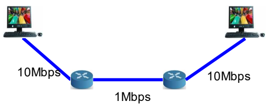




TCP congestion control: simulations

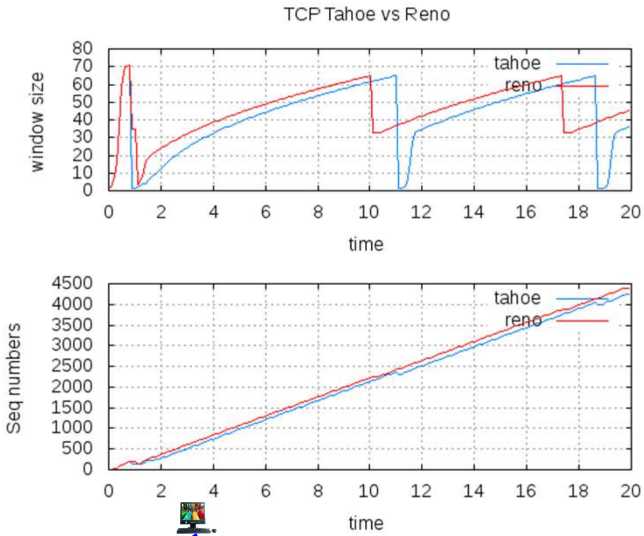
Case study 1: a TCP connection in a lossless channel, with constant delay

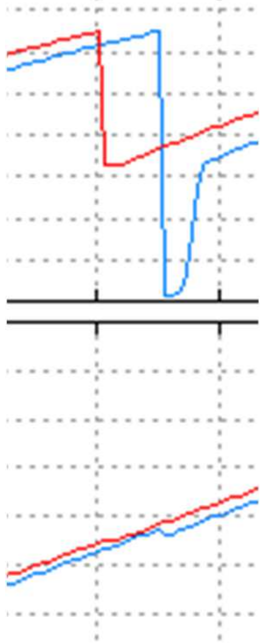


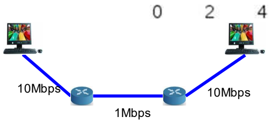


TCP congestion control: simulations

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

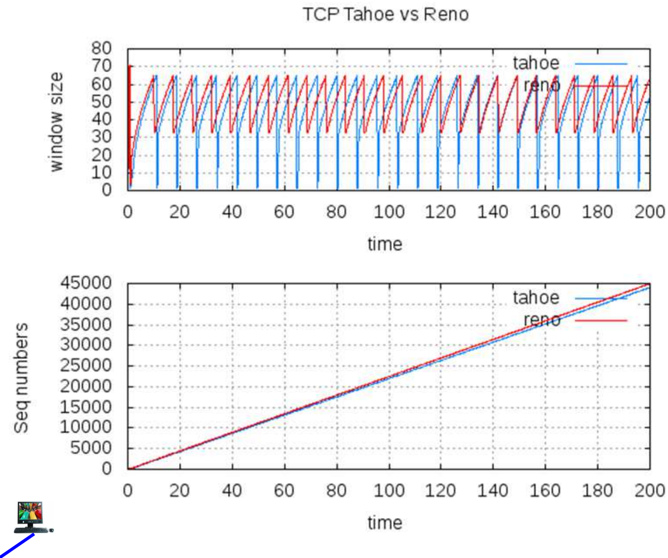


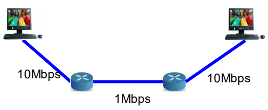




TCP congestion control: simulations

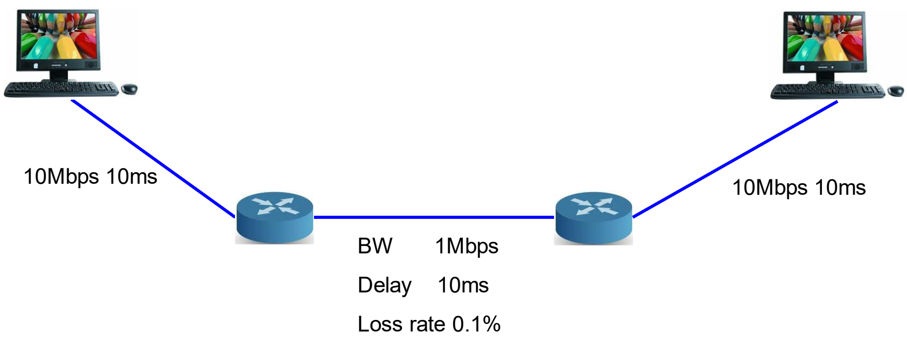
Case study 1: a TCP connection in a lossless channel, with constant delay





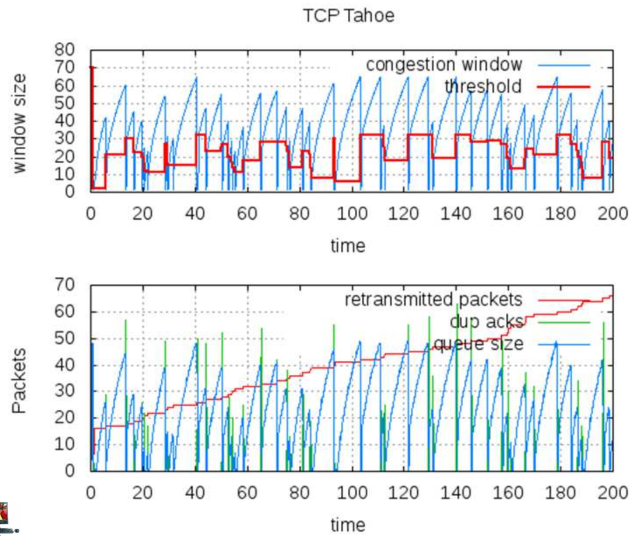
TCP congestion control: simulations

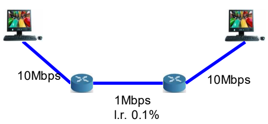
Case study 2: a TCP connection in a lossy channel, with constant delay



TCP congestion control: simulations

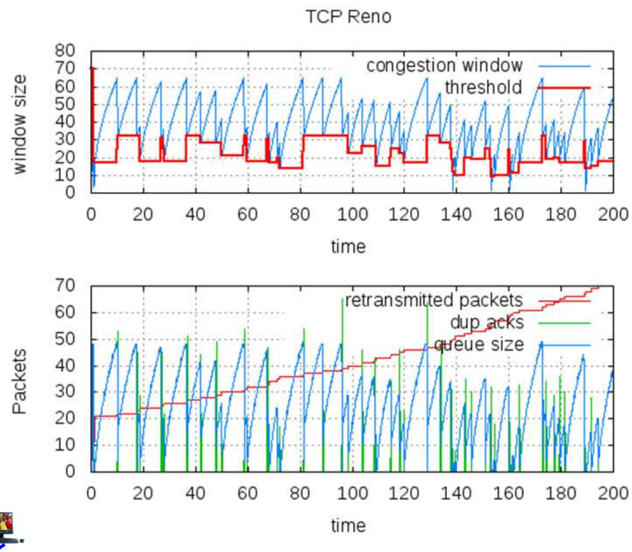
Case study 2: a TCP connection in a lossy channel, with constant delay

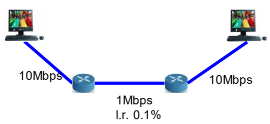




TCP congestion control: simulations

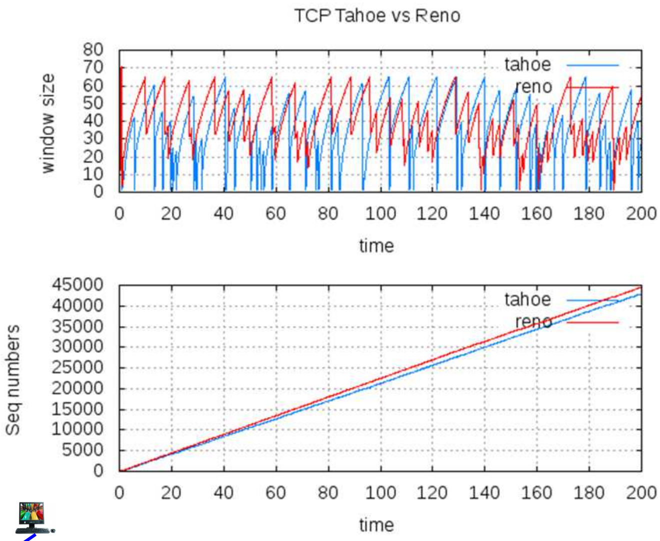
Case study 2: a TCP connection in a lossy channel, with constant delay

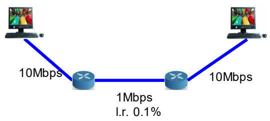




TCP congestion control: simulations

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

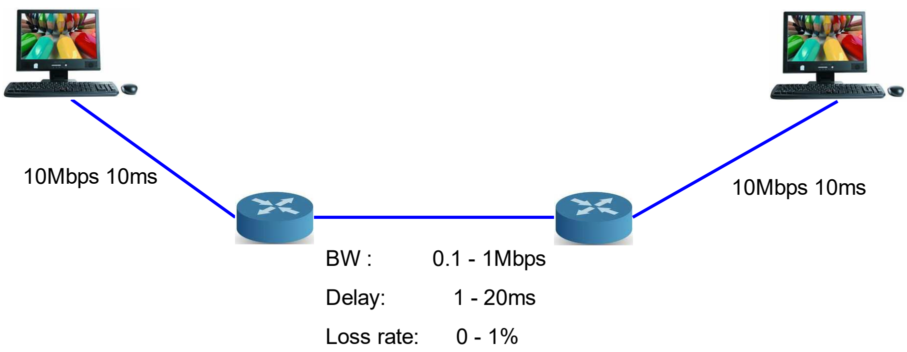




TCP congestion control: simulations

Case study 3:

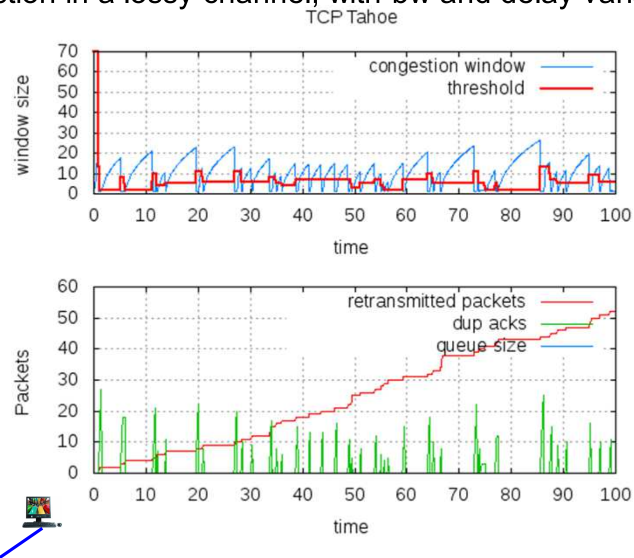
a TCP connection in a lossy channel, with bw and delay variable

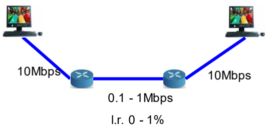


TCP congestion control: simulations

Case study 3:

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

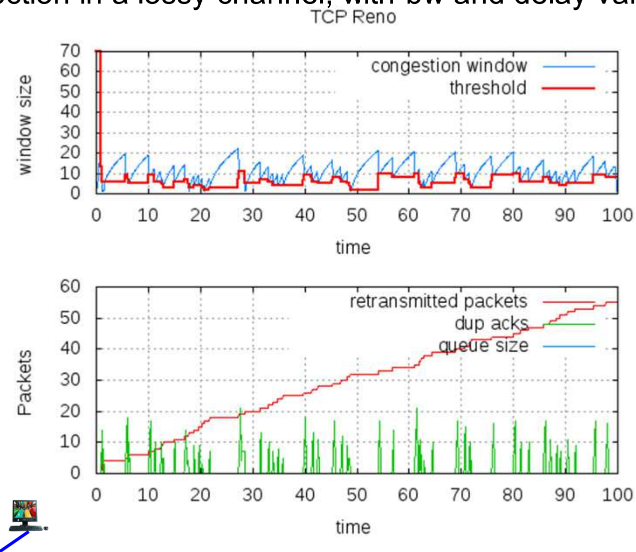


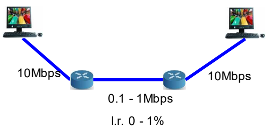


TCP congestion control: simulations

Case study 3:

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

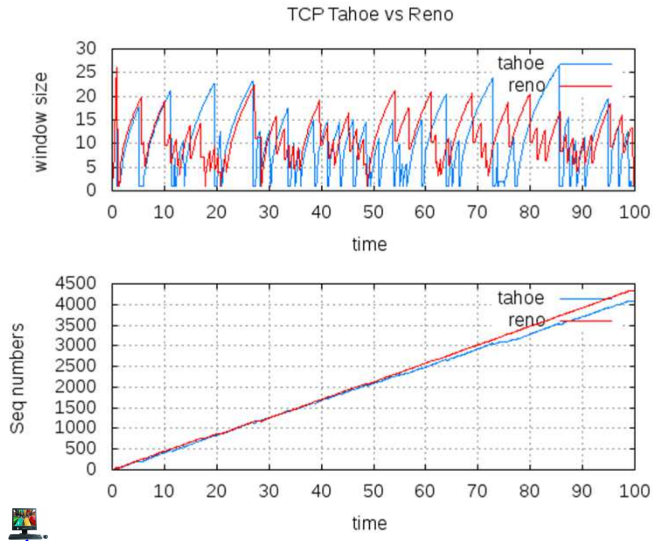


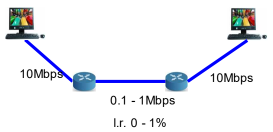


TCP congestion control: simulations

Case study 3:

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

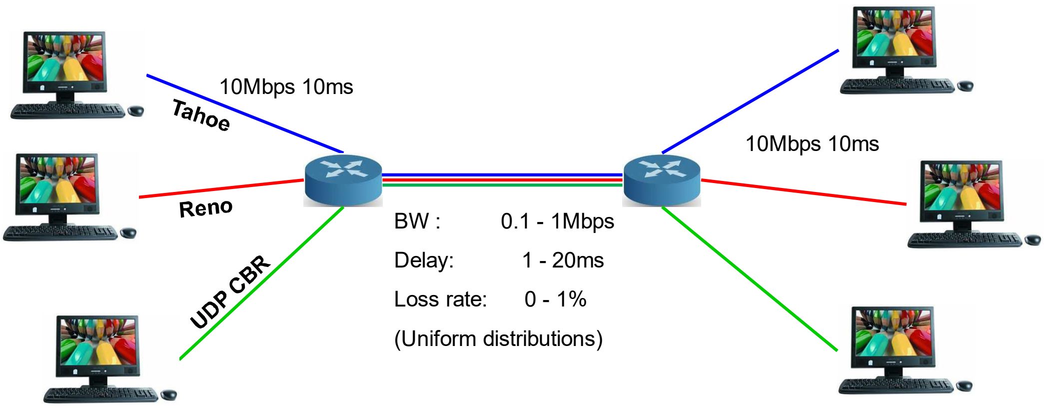




TCP congestion control: simulations

Case study 4:

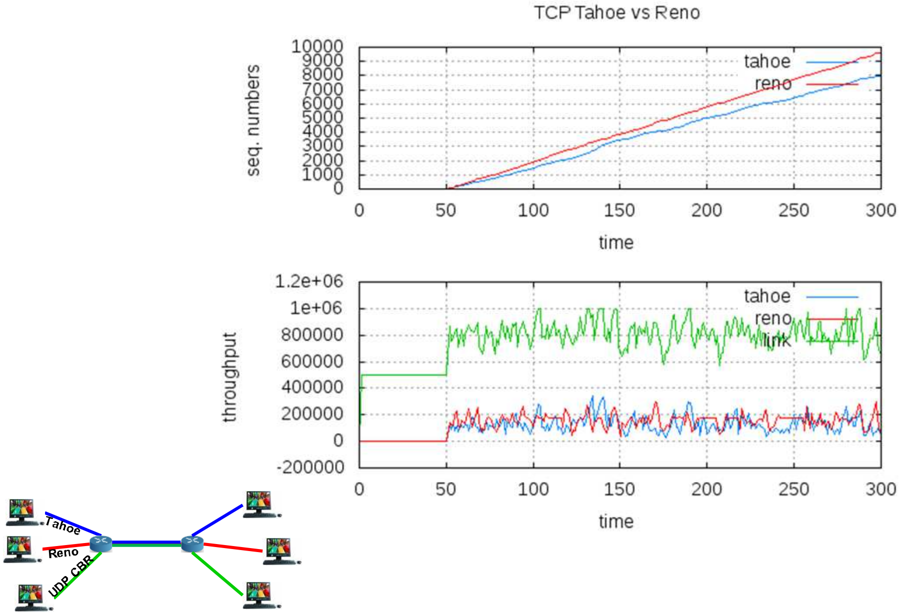
- a TCP Tahoe connection
- A TCP Reno connection
- A UDP CBR flow
- lossy channel, with variable bandwidth and delay



TCP congestion control: simulations

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

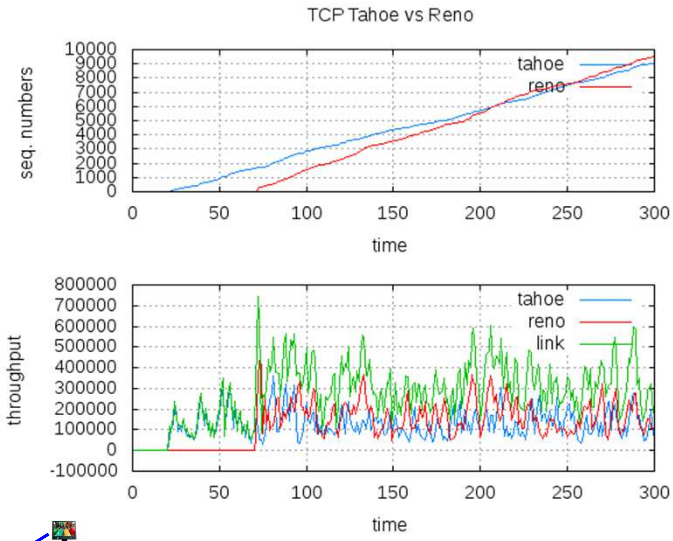
UDP starts at t=0s, TCPs start at t=50s

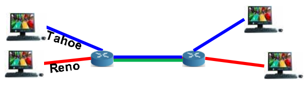


TCP congestion control: simulations

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

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



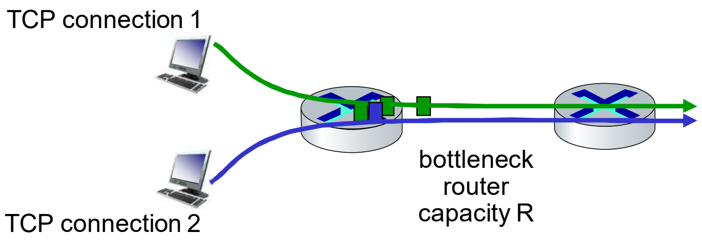


TCP fairness

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

(Collins)

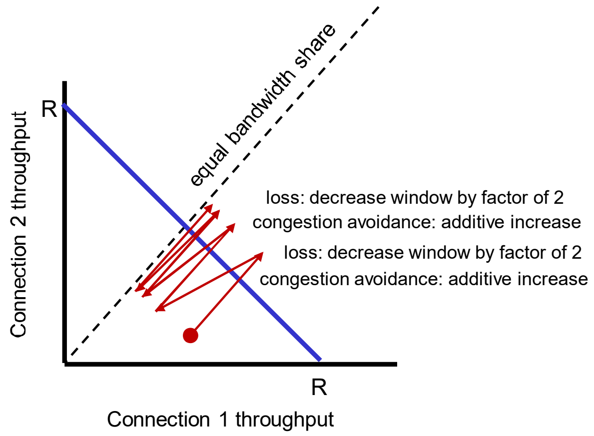
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

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

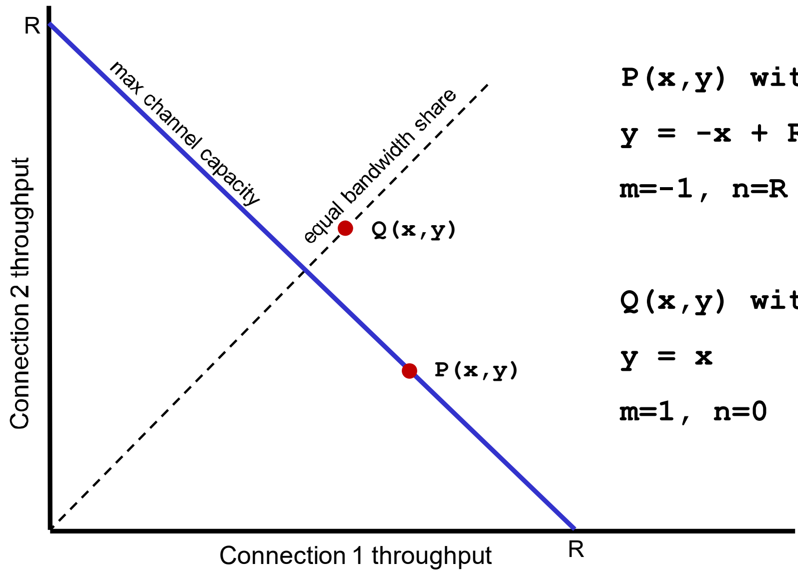


Is TCP fair?

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

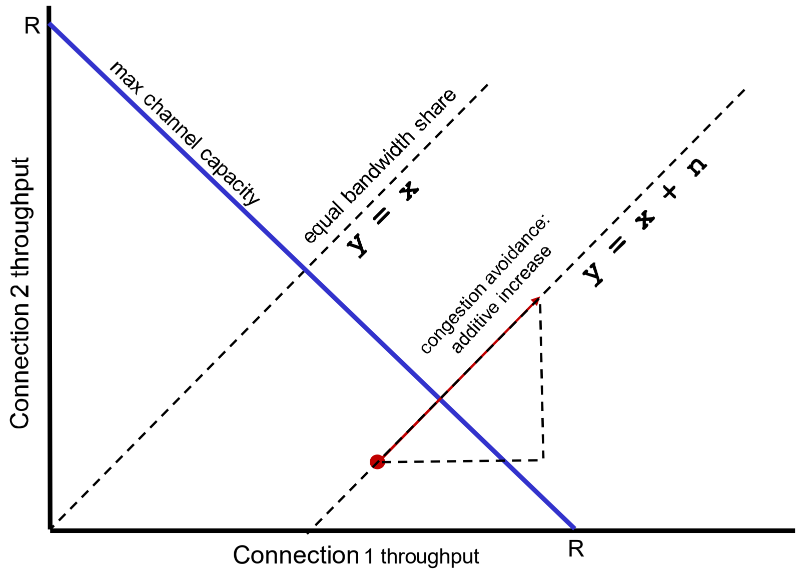
m=-1, n=R

Q(x,y) with x=y

y = x

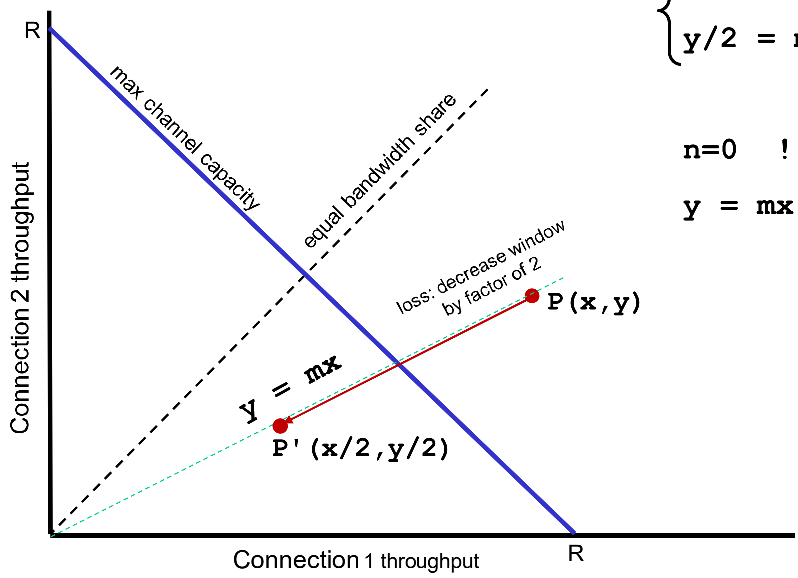
m=1, n=0

Q: is TCP Fair?



R
max channel capacity
equal bandwidth share
y = x
congestion avoidance: additive increase
y = x + n
Connection 2 throughput
Connection 1 throughput
R

Q: is TCP Fair?

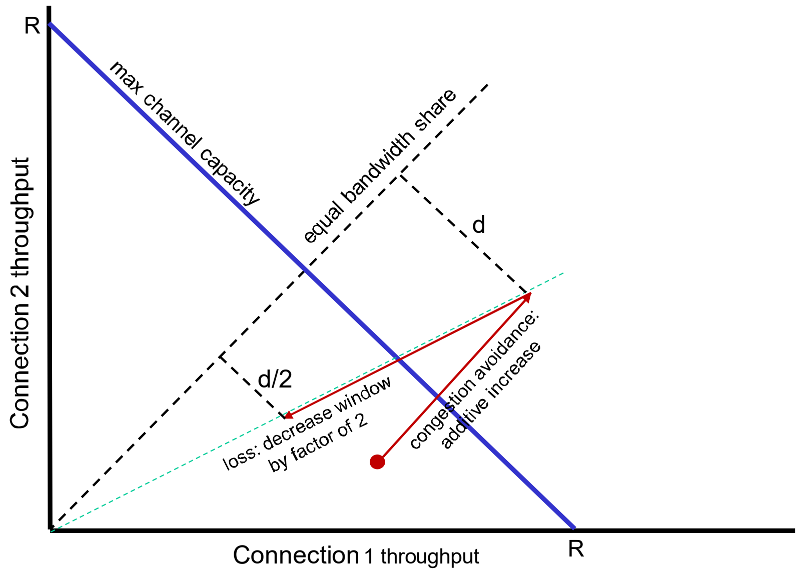


$$\begin{cases} y = mx + n \\ y/2 = mx/2 + n \end{cases}$$

n=0 !

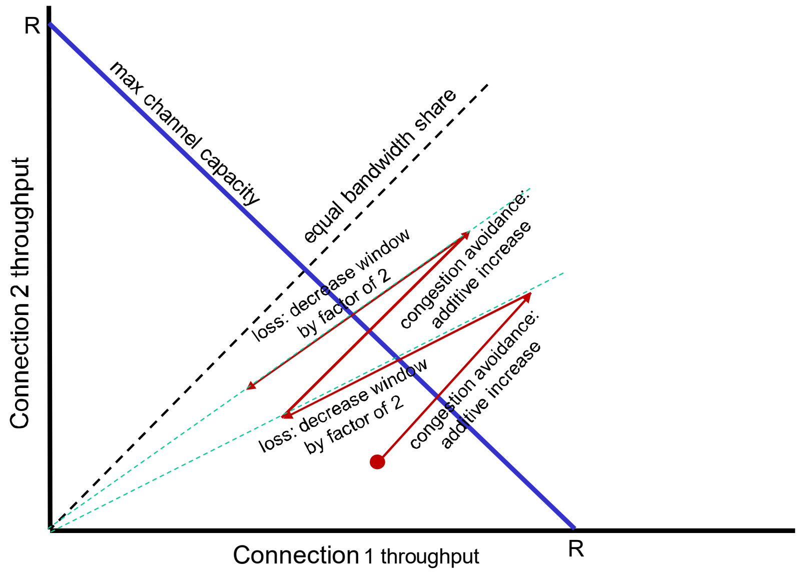
y = mx

Q: is TCP Fair?



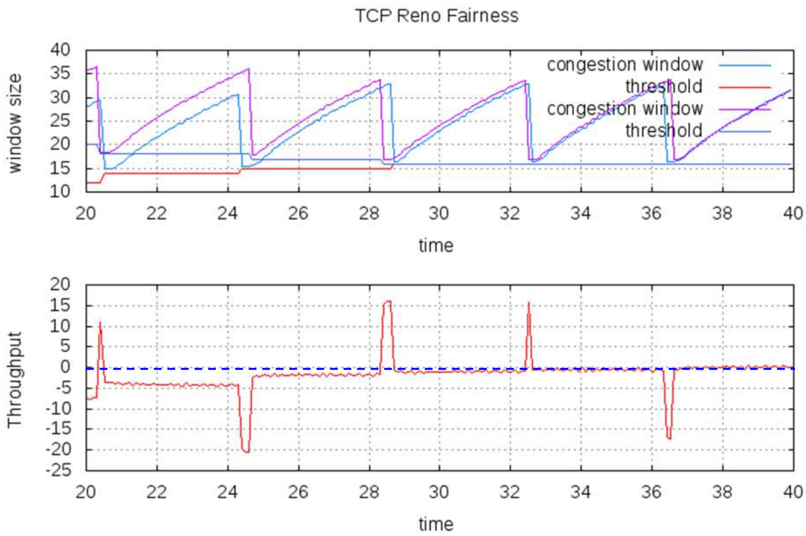
R
max channel capacity
equal bandwidth share
d
d/2
Connection 2 throughput
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 1 throughput
R

Q: is TCP Fair?



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

Fairness


TCP Reno Fairness
window size
time
congestion window
threshold
congestion window
threshold
Throughput
time

Evolving transport-layer functionality

- 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