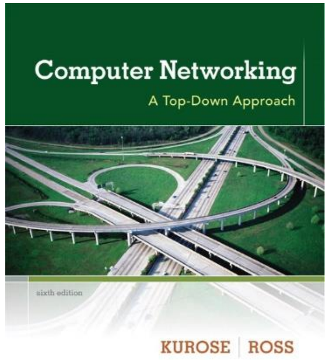


Computer Networks: Transport Layer Protocols

Rajesh Palit, Ph.D.

North South University, Dhaka

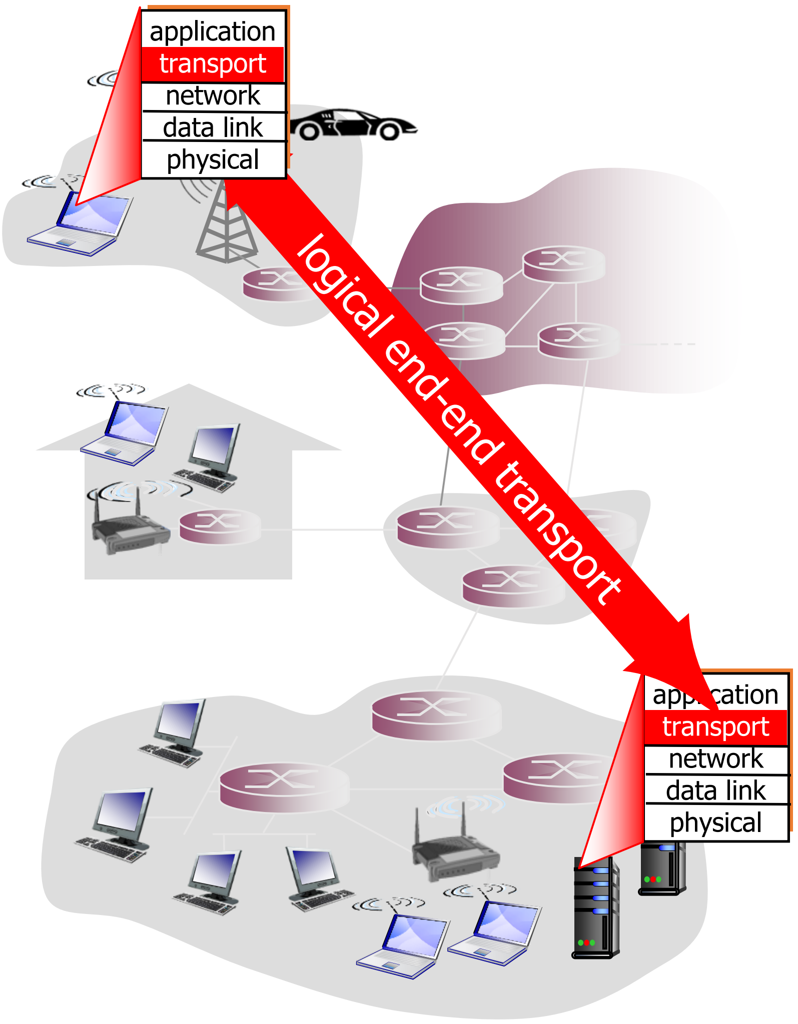
Chapter 3: Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

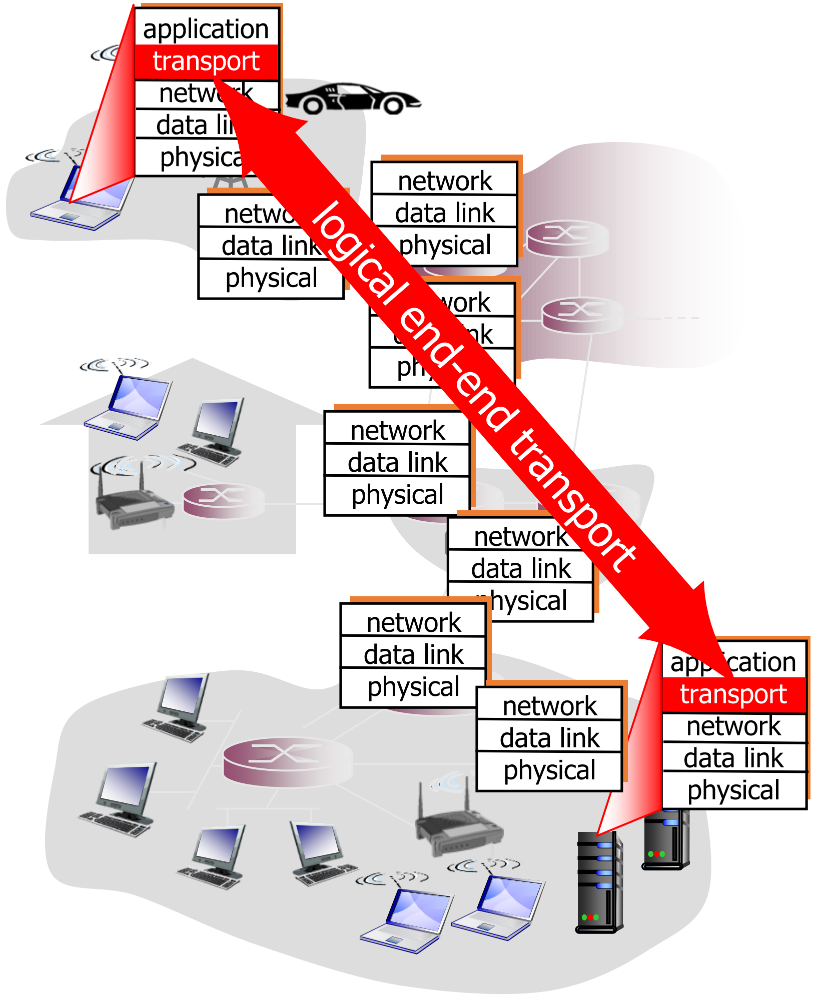
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who mux/demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



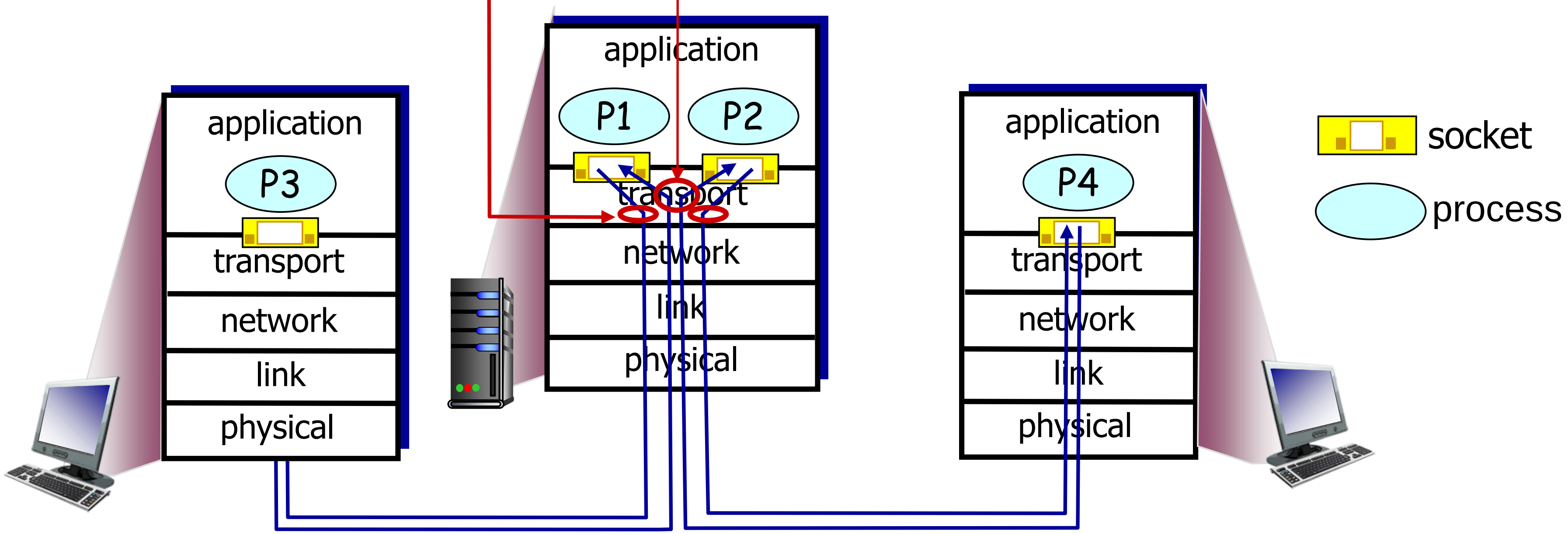
Multiplexing/demultiplexing

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

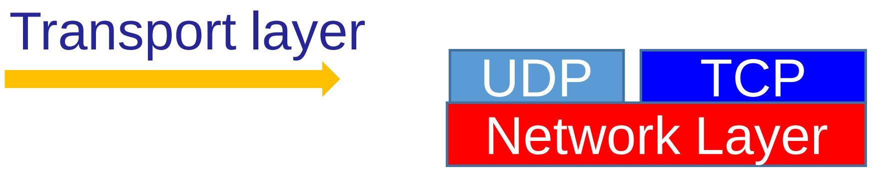


How Mux/DeMux Work?

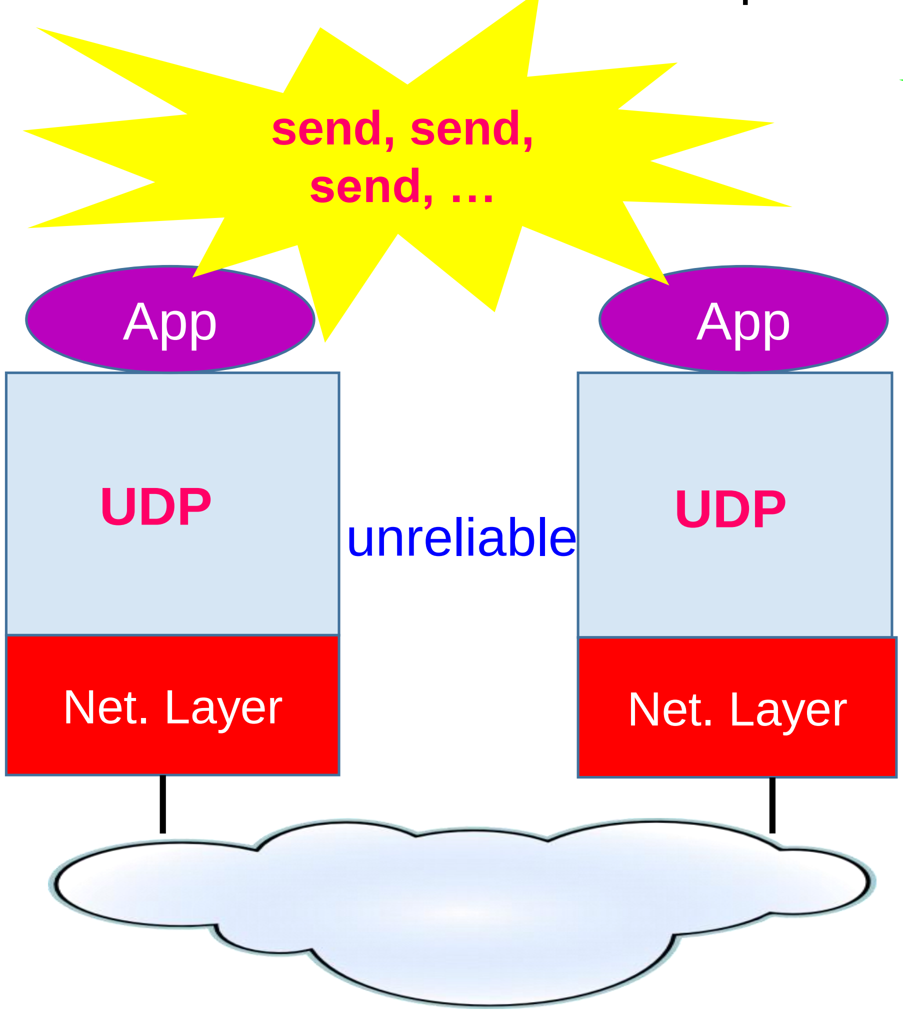
- Suppose you are downloading Web pages while running one FTP session and two Telnet sessions. You therefore have four network application processes running -- two Telnet processes, one FTP process, and one HTTP process. When the transport layer in your computer receives data from the network layer below, it needs to direct the received data to one of these four processes.

- This job of delivering the data in a transport-layer segment to the correct application process is called **de-multiplexing**. The job of gathering data at the source host from different application processes, enveloping the data with header information to create segments, and passing the segments to the network layer is called **multiplexing**.

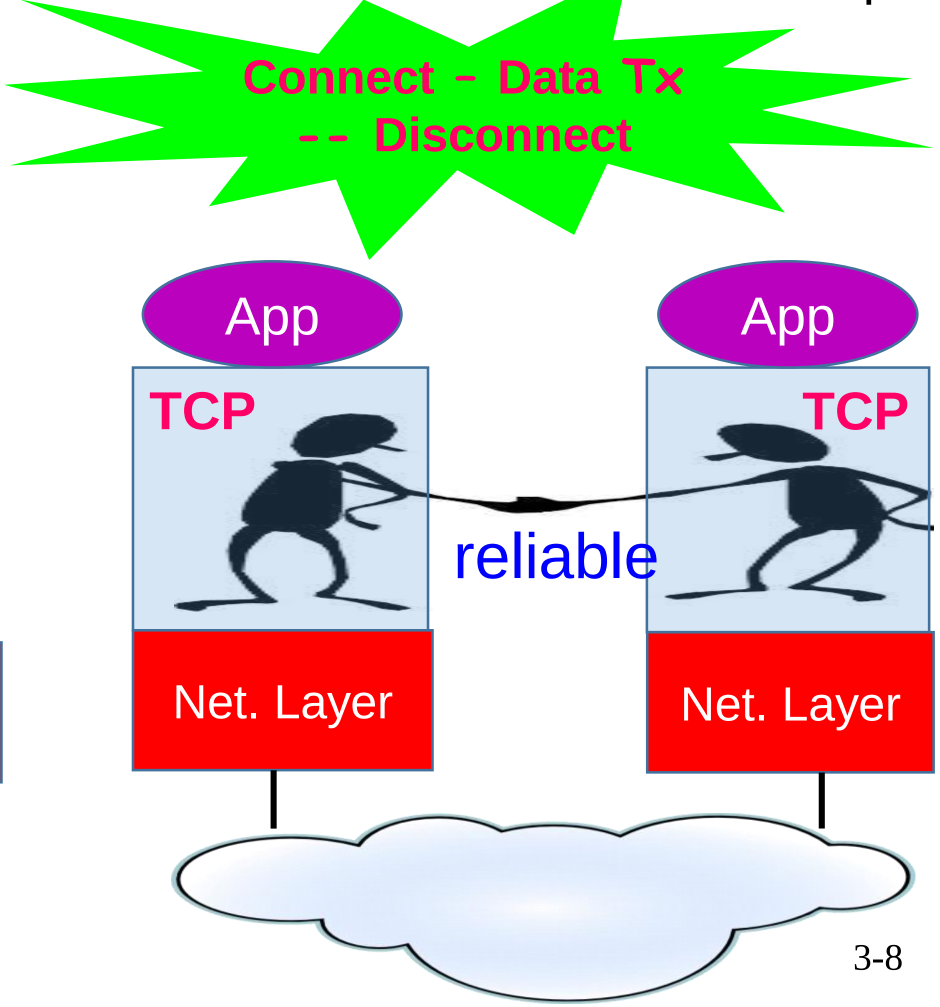
Two Protocols in the Transport Layer



- UDP (User Datagram Protocol)
  - connectionless transport



- TCP (Transmission Control P'col)
  - connection-oriented transport



Unreliable UDP vs. Reliable TCP

UDP	TCP
If segments arrive out-of-sequence from network layer, the receiver does not reorder them.	If segments arrive out-of-sequence from network layer, the receiver reorders them.
If a segment is missing from a sequence, the sender does not retransmit it.	If a segment is missing from a sequence, the sender eventually retransmits it.
Segments are not ACKed.	Segments are ACKed by receiver.
No flow control is performed.	Flow control is performed.
No congestion control is performed.	Congestion control is performed.

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

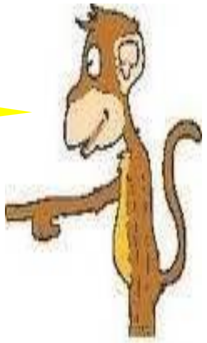
3.6 principles of congestion control

3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

Who wants to have unreliable comm.

Some apps can tolerate unreliable comm.



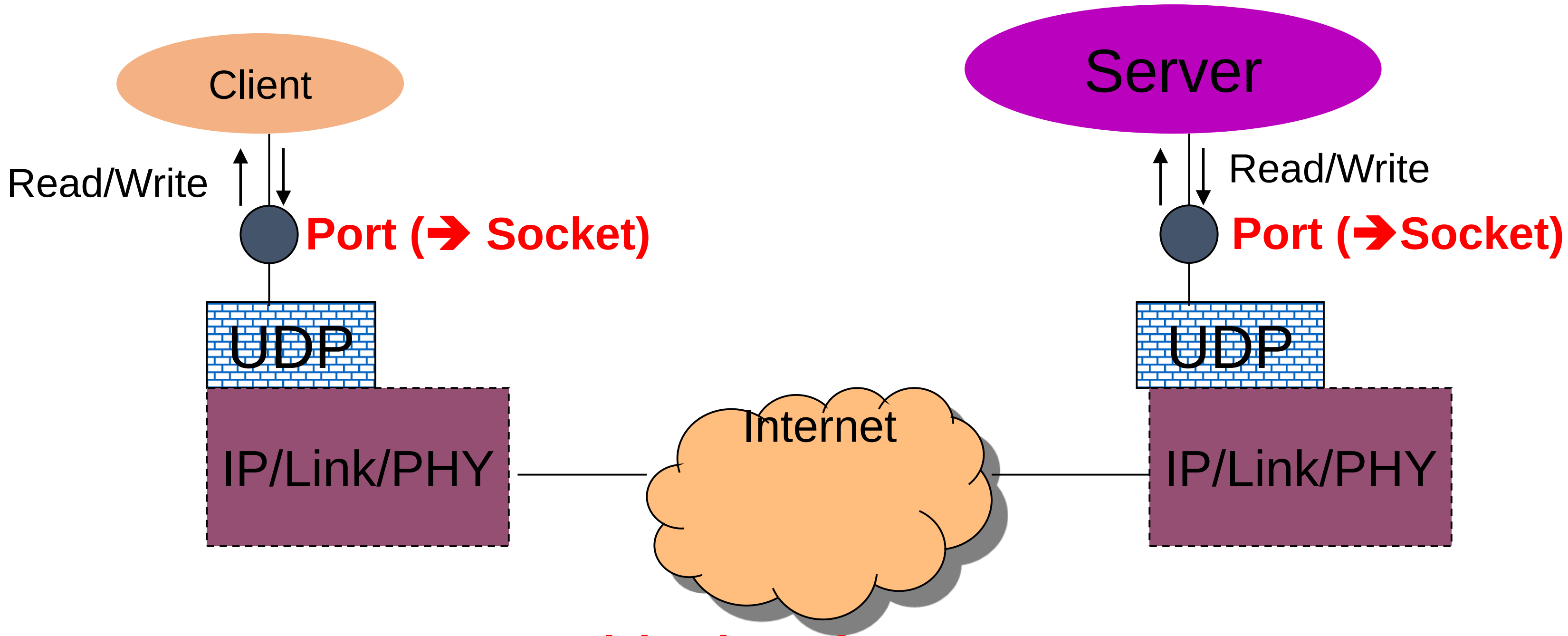
VoIP
(Skype)

RIP

SNMP
(Simple Network Management Protocol)

Note:
It is up to the app to decide what to do:
accept unreliability OR run your own protocol to make it reliable.

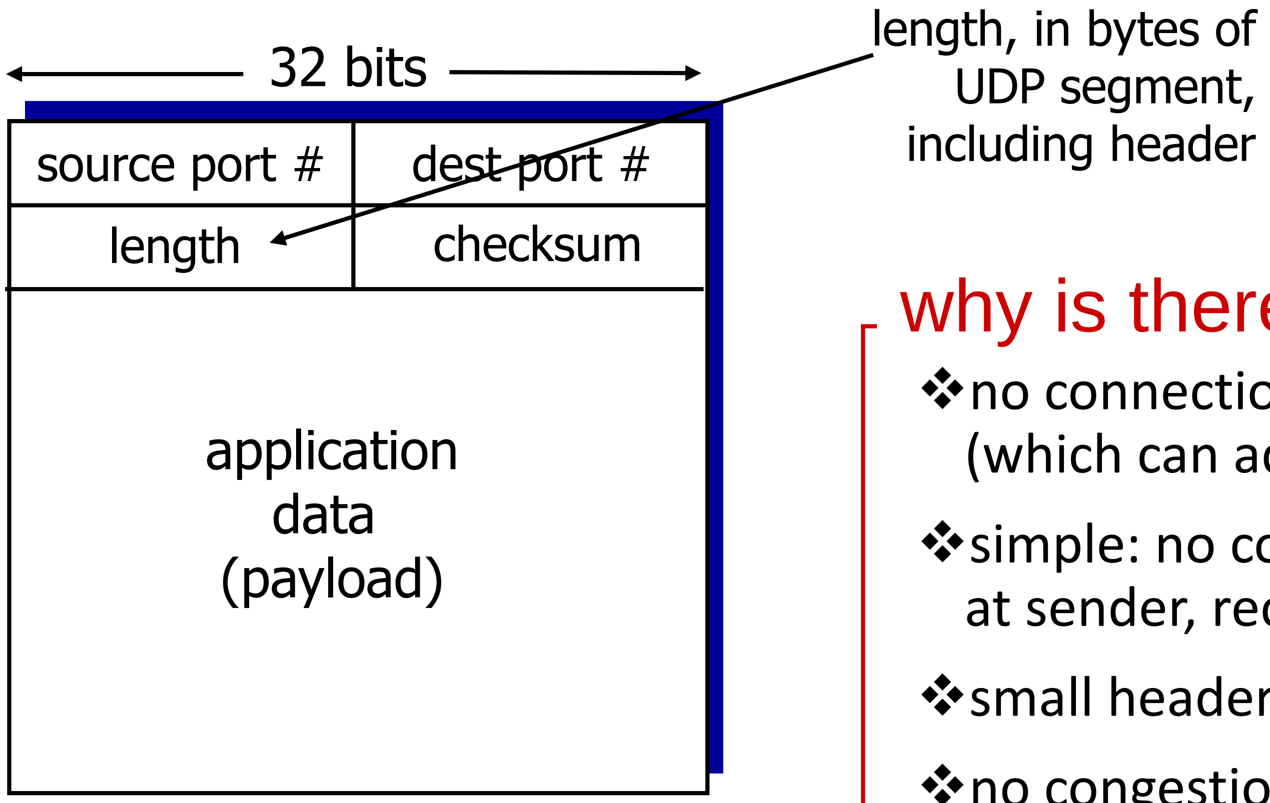
UDP: Application Context



Two kinds of Ports:

- **Reserved** for well-known services
  - **RIP** is attached at UDP port #520
- **Free ports**

UDP: segment header



UDP segment format

why is there a UDP?

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

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: example

example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

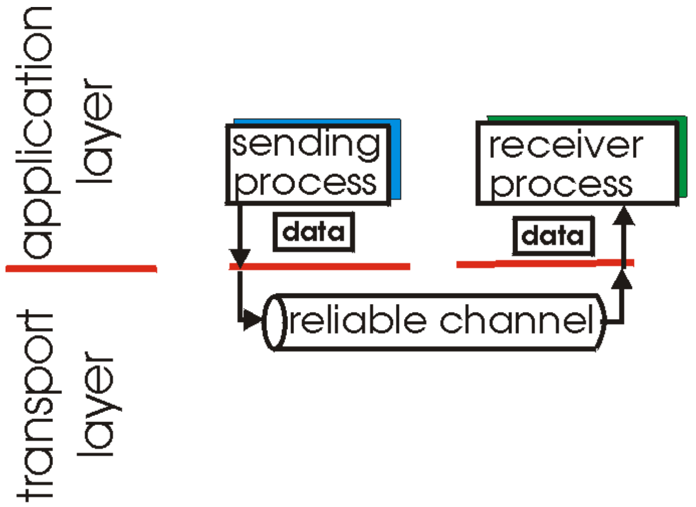
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

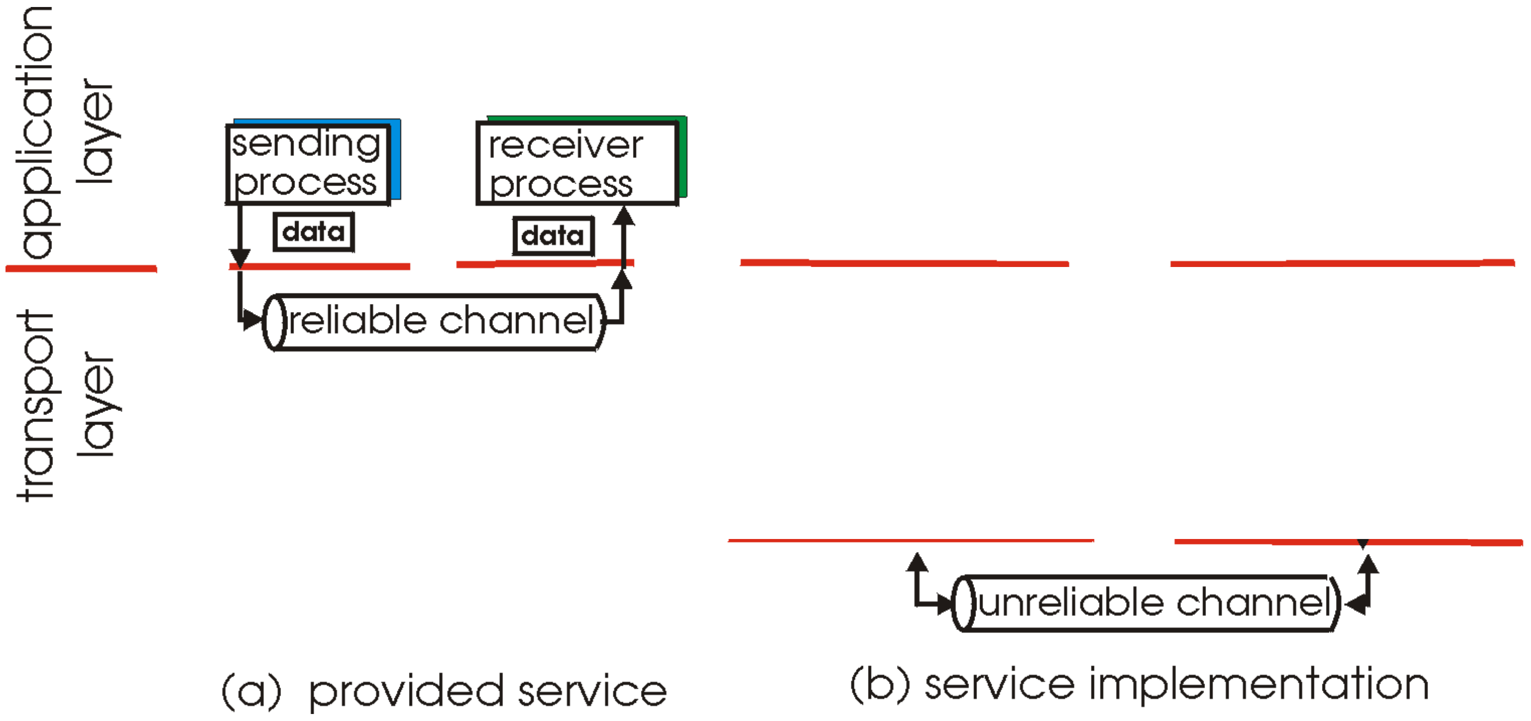


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

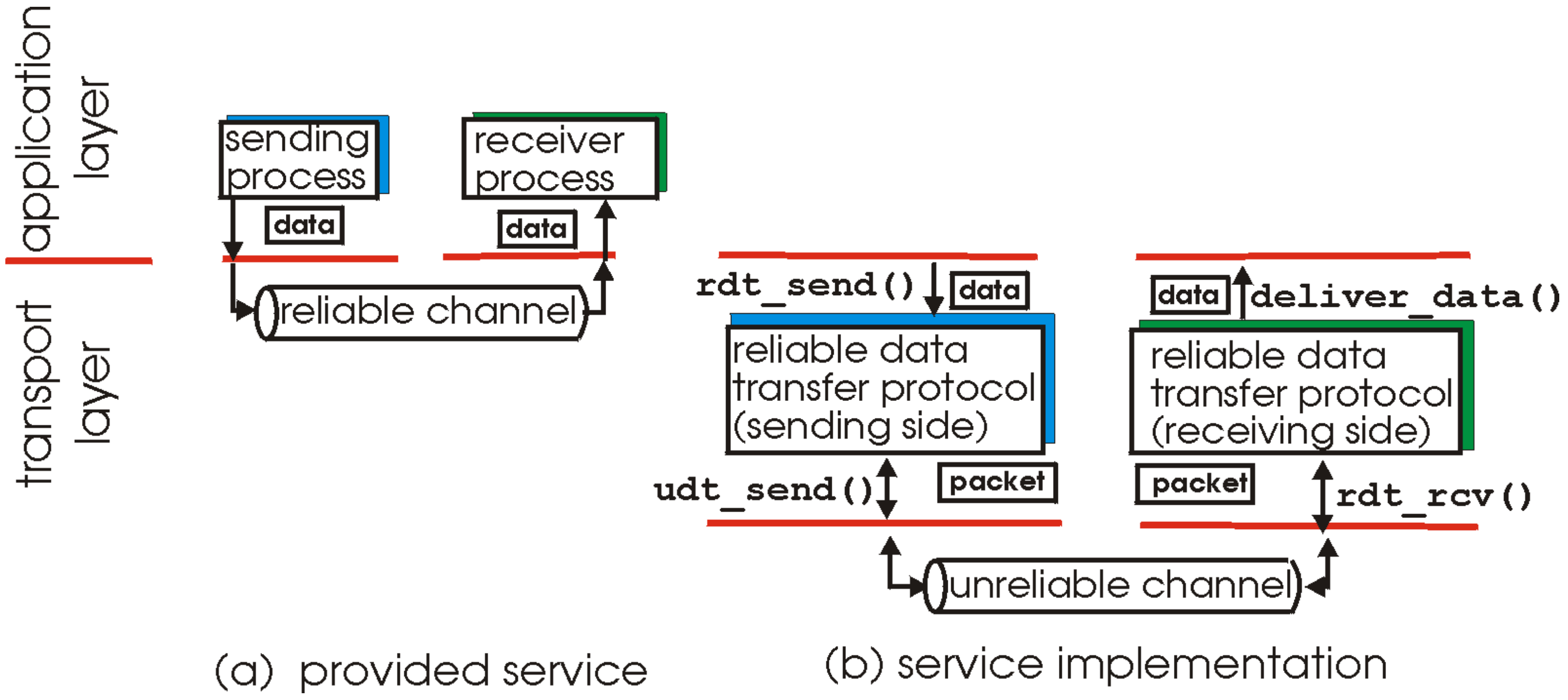
- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

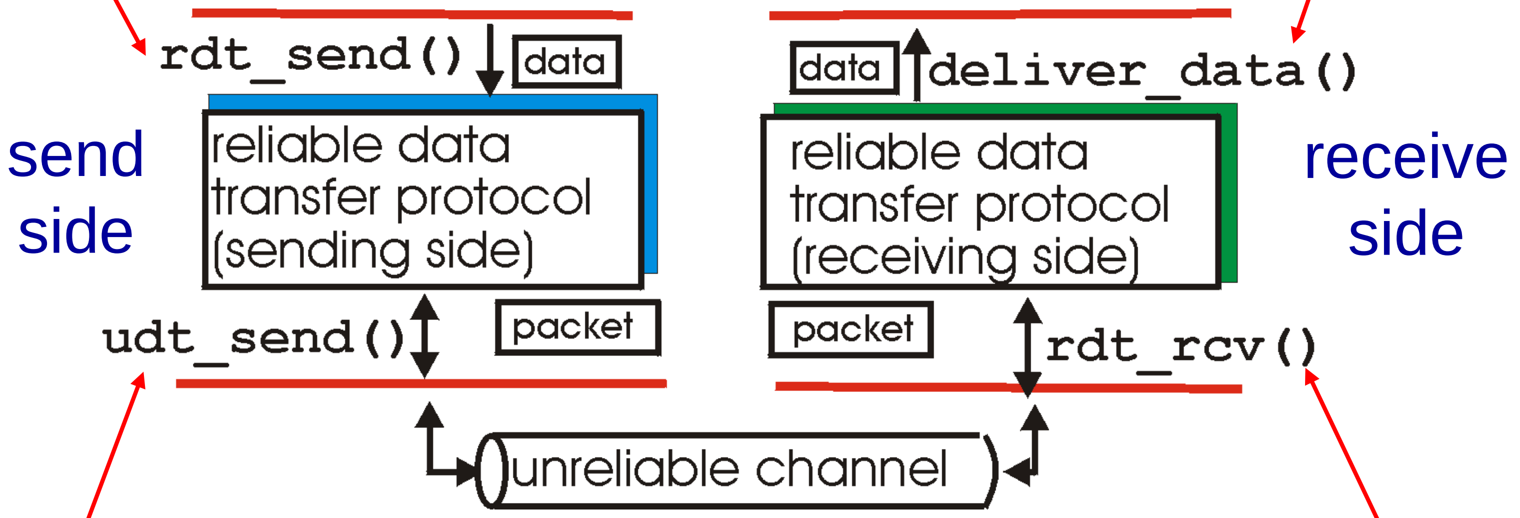


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

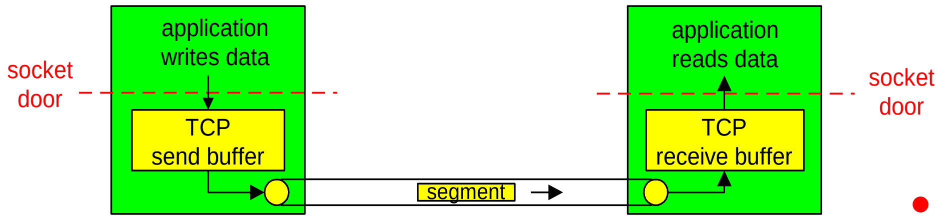
deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

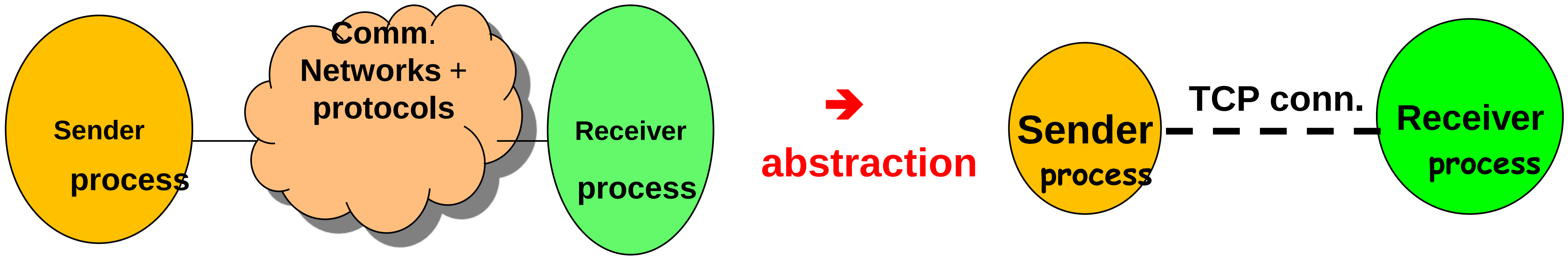
rdt_rcv(): called when packet arrives on rcv-side of channel

TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581



- point-to-point
  - one sender, one receiver
- reliable *byte stream*
  - no "message boundaries"
- pipelined
  - TCP **congestion** and **flow control** set **window size**
- full duplex data
  - bi-directional data flow over same connection
- connection-oriented
  - handshaking (exchange of control msgs) initializes sender and receiver states before data exchange
- flow controlled
  - sender will not overwhelm receiver

What is a TCP **connection**?



A connection is identified by (Src Port + Src IP, Dst Port + Dst IP).

A connection has well-defined start and finish events.

Comm. parameters are exchanged to establish a conn.: **ISN** (Init. Seq. #), **RWND** (receive window), **MSS** (Max Segment Size)

Receiver discards data associated with an old connection (say, estd. 0.5s back and reset)

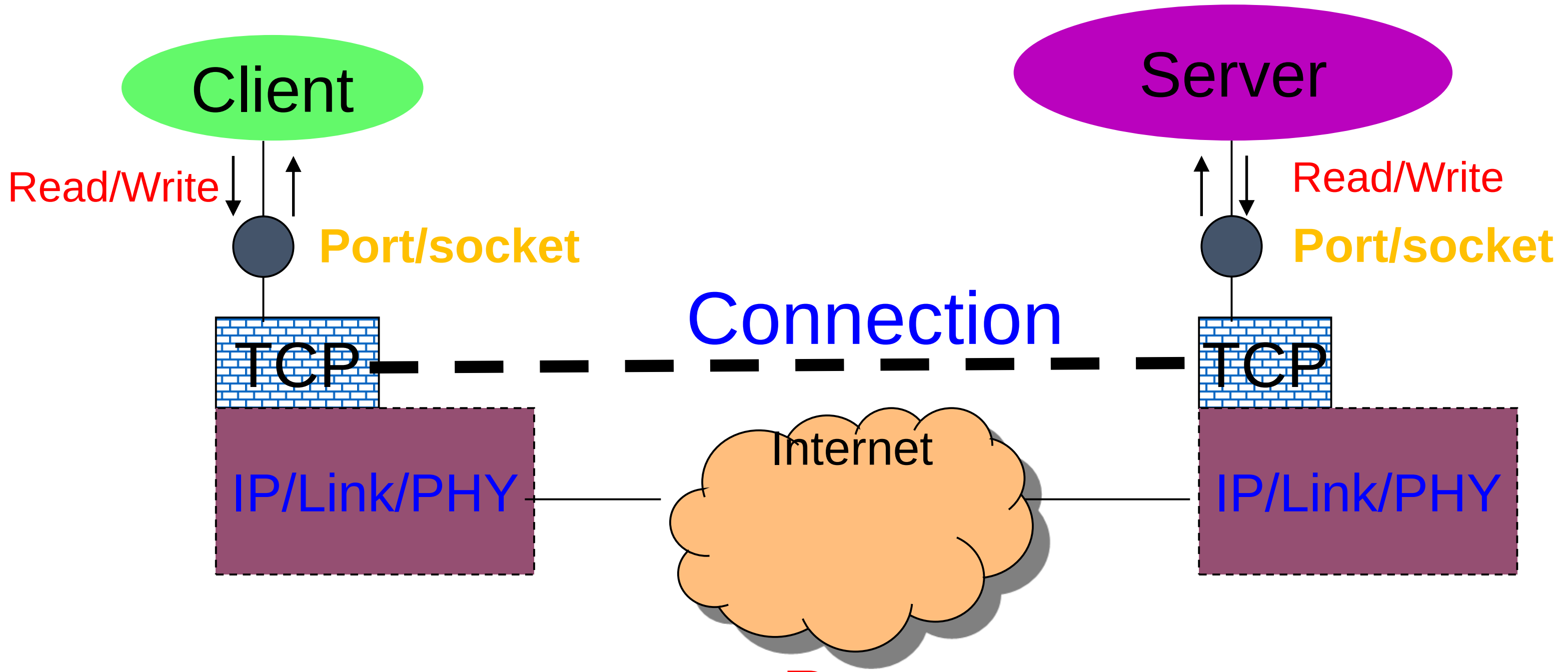
TCP Sender gets a *confirmation* of delivery via an ACK.

TCP Receiver delivers *exactly one copy* of sender’s data by means of timeout, retransmission, ACK, sequence #s, and buffering mechanisms.

Flow control: receiver controls its recv. window size.

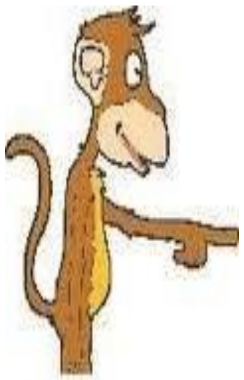
Congestion control: Timeouts trigger congestion control.

TCP: Application Context



Ports

- Reserved for well-known services
- Telnet/23, SMTP/25, FTP/20,21, HTTP/80, BGP/179, lp/515
- Free ports

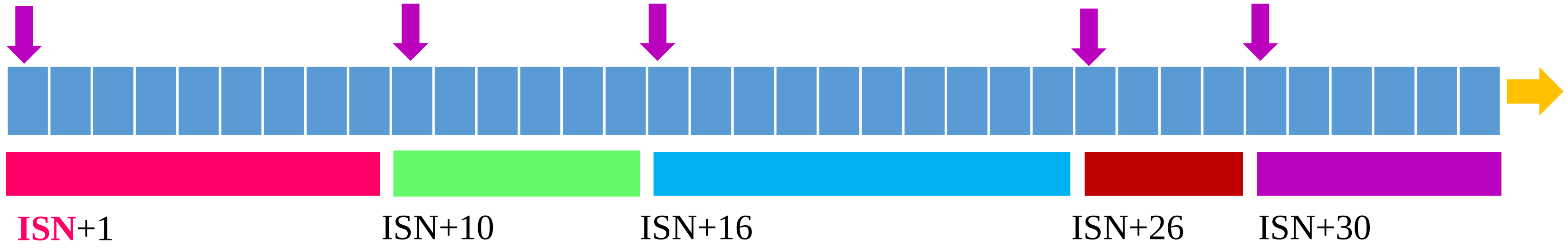


TCP is a **byte stream** protocol

Example: A file is viewed as a stream of bytes.

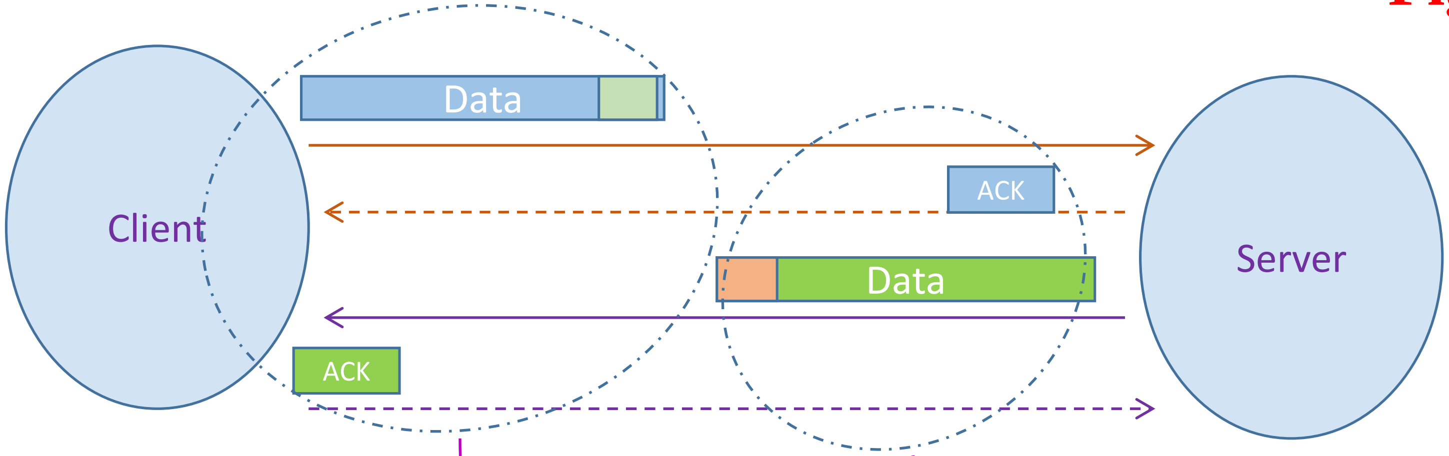
In fact, **data** produced by any source is considered as a stream of bytes.

Bytes have individual IDs. → Bytes are individually numbered.



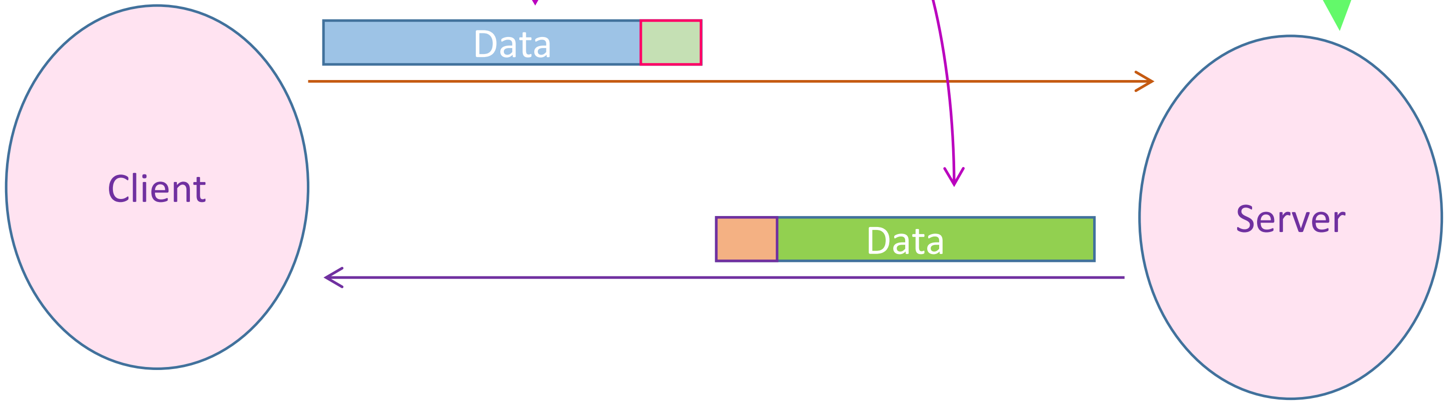
ISN: Initial Sequence Number

Piggybacking

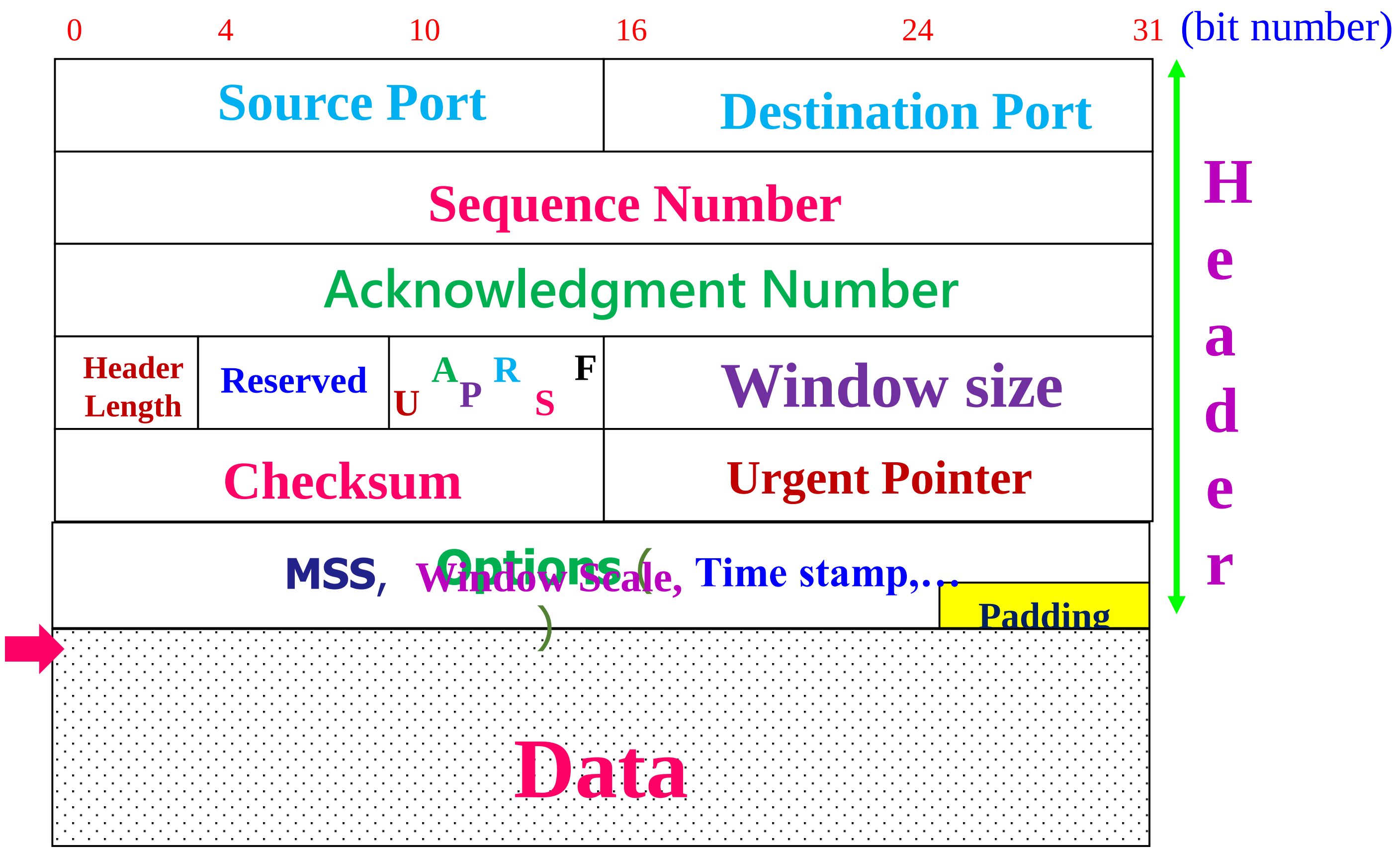


Small segments produce extra overhead: **transmission** and **processing** at routers.

Piggybacking: To eliminate separate ACK segments



TCP Segment Header



U: URG (**Urgent**)
A: ACK
P: PSH (**Push**)
R: RST (**Reset**)
S: SYN (**Sync.**)
F: FIN (**Finish**)

S=1 ➔ Seq. num. field carries ISN to be used
S=0 ➔ Seq. num. = Seq. # of the first data byte in seg.

MSS: Maximum Segment Size

TCP: Header

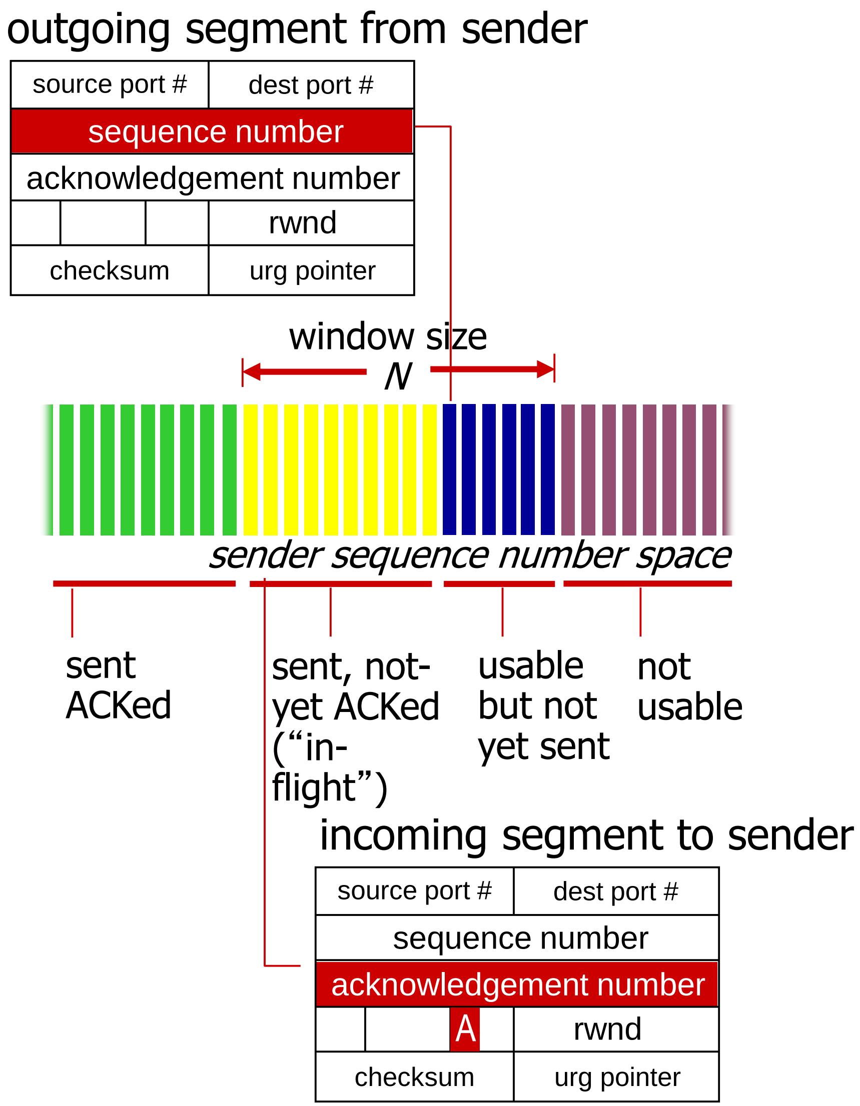
- Source/destination Ports
  - Port: A 16 bit local unique number on the host
  - Port + Host IP => Unique end point of an application
  - (Src Port + IP, Dst Port + IP): Unique connection ID
  - Source and destination IP: NOT part of a TCP segment
- 32-bit seq. number
  - SYN = 0 (DATA segment)
    - Position of the first data byte of this segment in the sender’s data stream
  - SYN = 1
    - ISN to be used in the sender’s byte stream. (in fact, ISN+1)
    - Different each time a host requests a connection

TCP seq. 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: Header

- 32-bit ACK number
  - Valid if ACK = 1
  - Identifies the sequence number of the NEXT data byte that the sender of the ACK expects to receive.
- Header length in 4-byte units
  - Lets the receiver know the beginning of the data area due to the variable length of the *Option* field.
- Reserved (6 bits)
  - For future use. All 0’s.

TCP: Header

- URG: ‘1’ => Urgent Pointer is valid
- ACK: ‘1’ => ACK Seq# is valid
- PSH:
    - ‘1’: The receiving TCP module passes the data to the application immediately
    - ‘0’: The receiving TCP module may delay the data
- RST: ‘1’ => Tells the receiver to abort the conn.
- SYN: This bit requests a connection
- FIN
    - ‘1’: Sender has no more data to send, but is ready to receive.

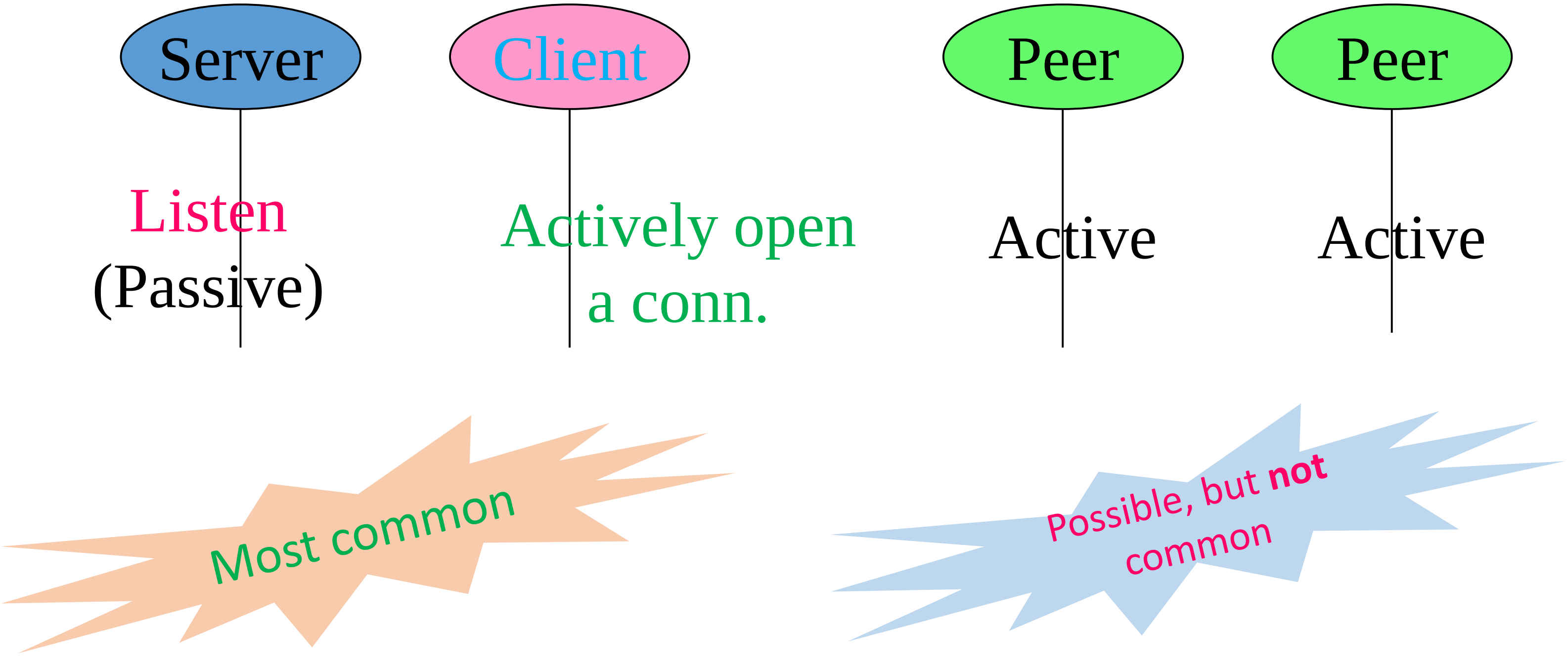
TCP: Header

- Window Size
  - The number of bytes the sender is willing to receive.
    - Used in flow control and congestion control
- Checksum: For error detection; scope: complete seg.
- Urgent Pointer: Valid if URG = ‘1’
  - Urgent data
    - Start byte is not specified, but it is considered to be the start of the seg.
    - Final byte in receiver’s buffer: Seq# + Urgent Ptr.
  - The sender can send “control” information to the receiver to be processed on a priority basis.

TCP: Header

- Options
  - MSS
    - The Max Segment Size accepted by the sender
    - Specified during connection set up
  - Window Scale
    - Allows the use of a larger advertised Window Size
  - Time Stamp
    - Used in Round-Trip Time (RTT) calculation
    - Intended to be used on high-speed connection
      - Sequence number may wrap around during a connection.
      - New segments are distinguished from old segments by means of time stamps

TCP Conn.: Established in two ways



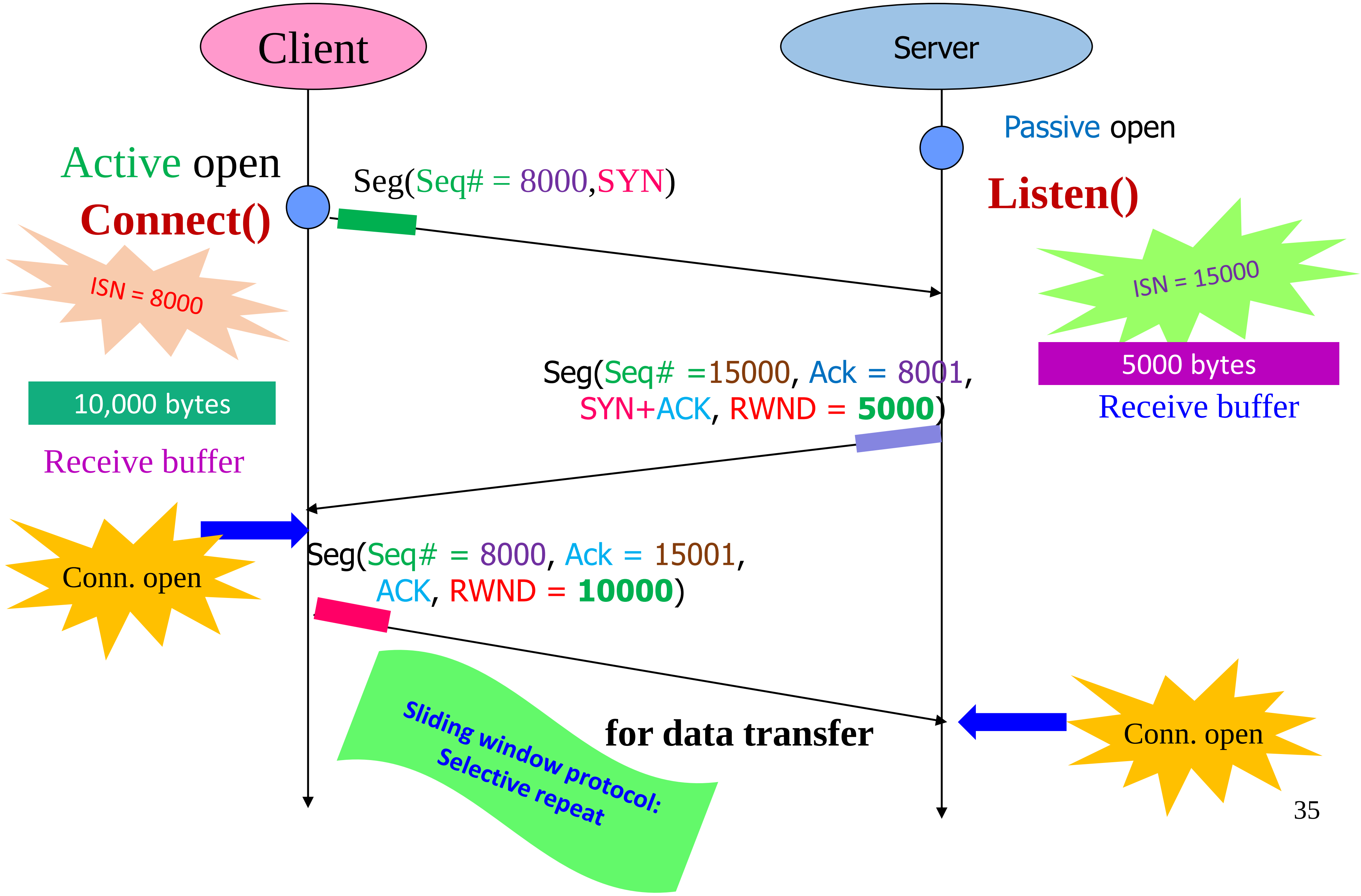
The server must be running, and attached to a known port.

Example: An HTTP server is attached to TCP at port #80.

TCP Connection: 3-way handshake

- Use these fields to understand the opening of a conn.
    - Connection request (SYN)
    - Sequence number
    - Acknowledgement (ACK)
    - Receive window size

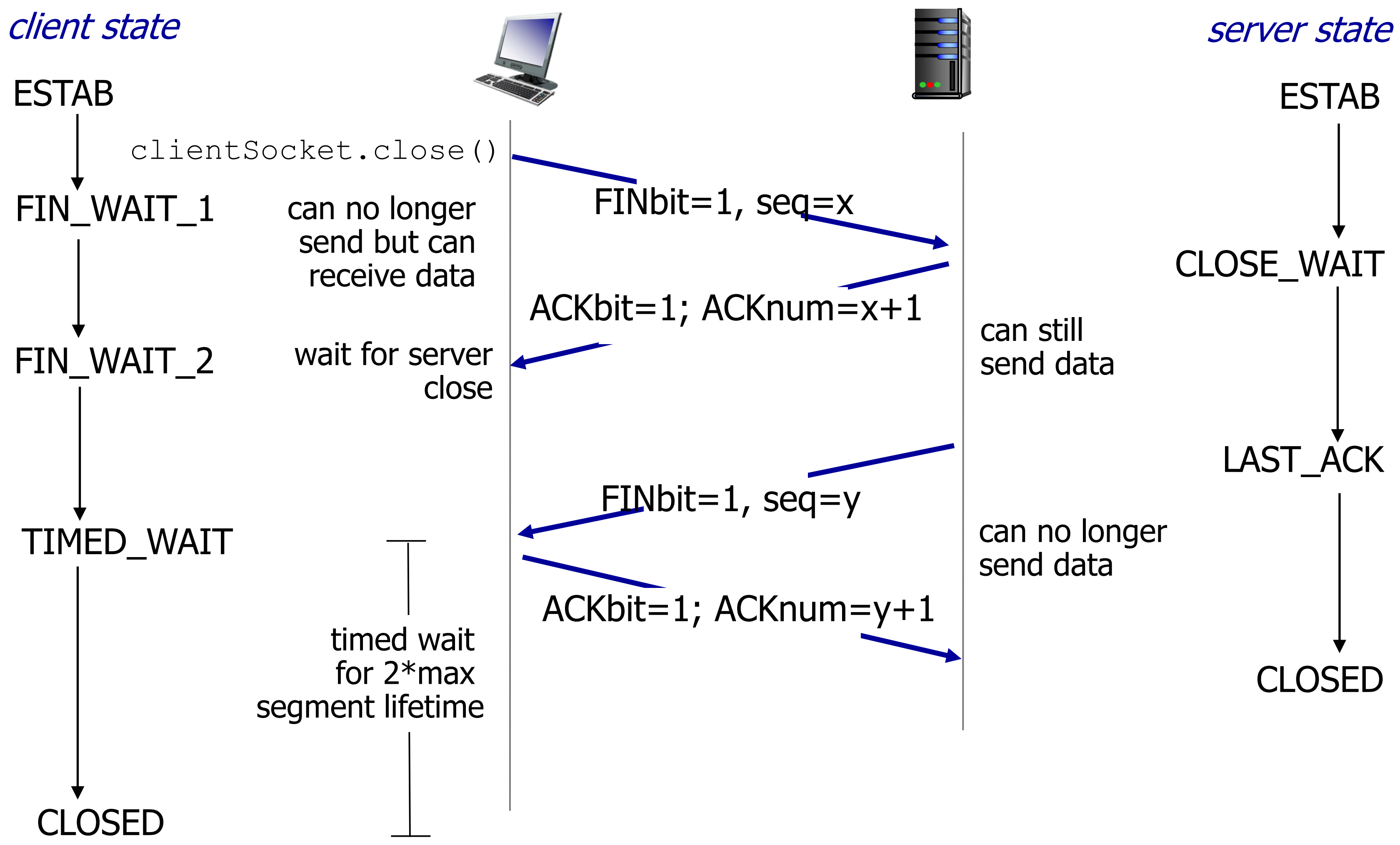
TCP Connection: 3-way handshake



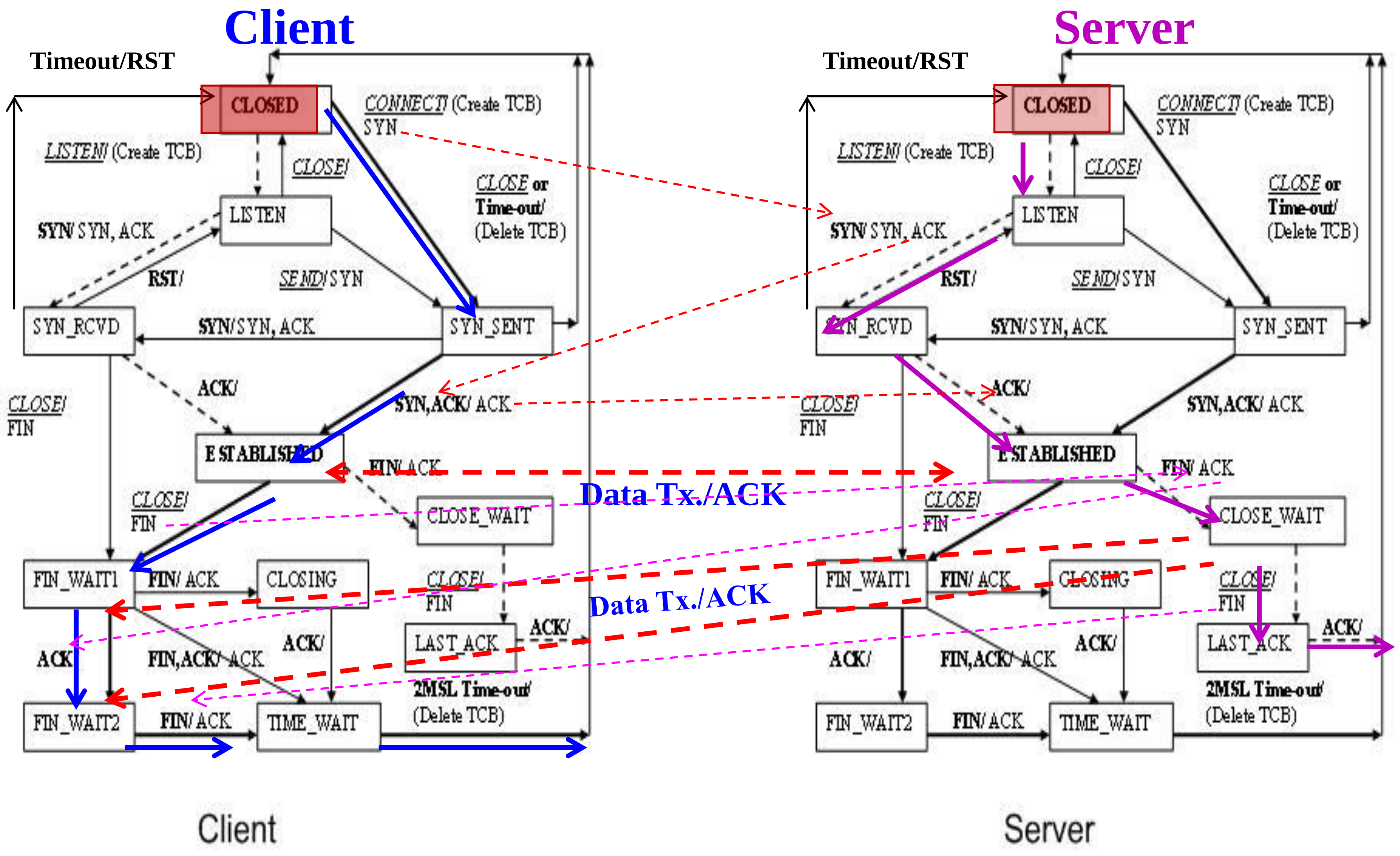
TCP Connection: 3-way handshake

- SYN segment from client to server
    - SYN = 1
    - A random initial Seq# (ISN)
    - RWND is undefined (defined later …)
    - Options
- SYN+ACK segment from server to client
    - SYN = 1
    - A random initial Seq# (ISN)
    - ACK = 1 (server acks the received SYN segment)
    - Ack Seq.#: The sequence # of first data byte to be received
    - RWND: Receive window size
- ACK from client to server
    - ACKs the second SYN segment
    - RWND

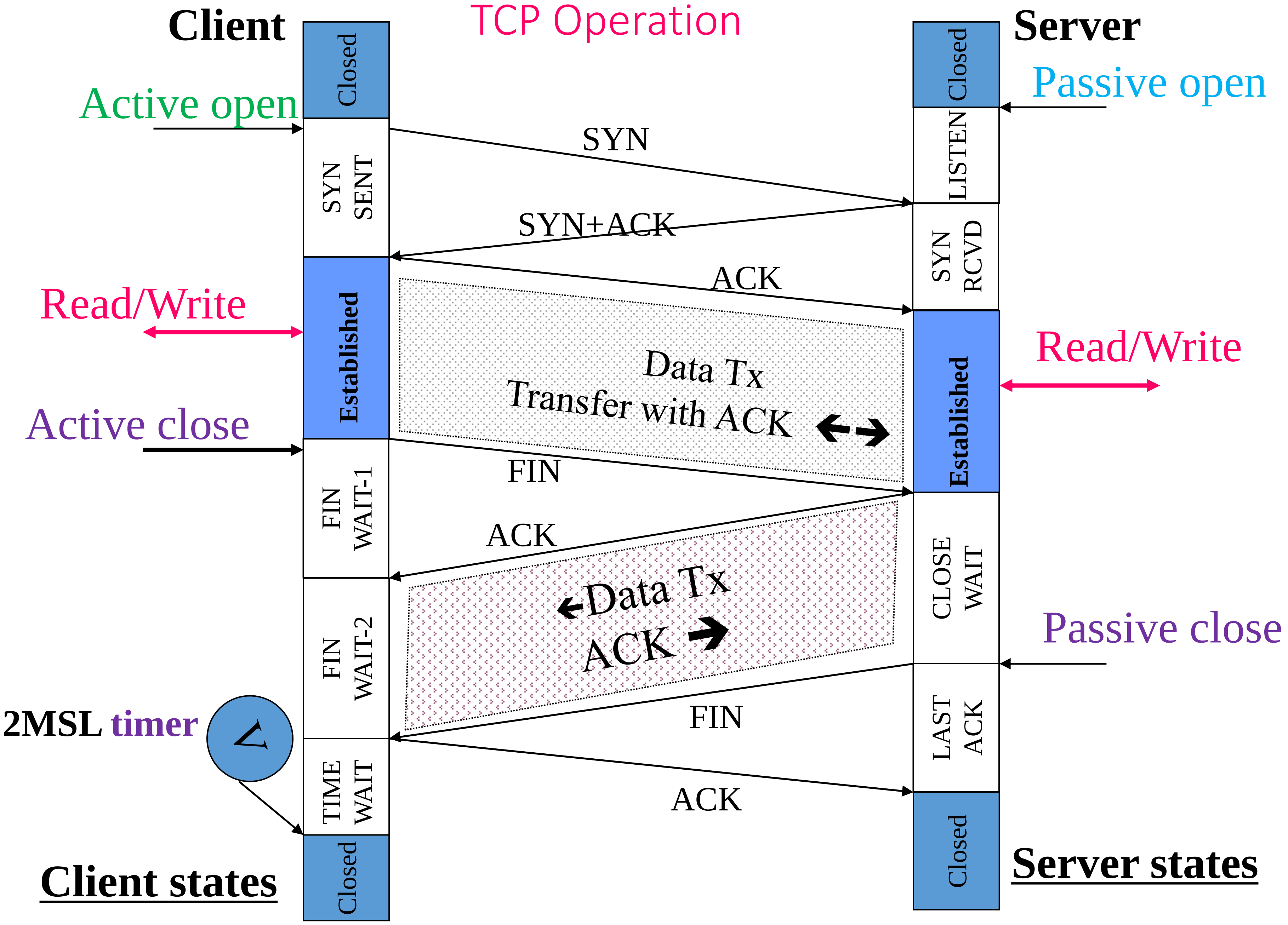
TCP: closing a connection



Client/Server Communication and State Transitions
(TCB: Transmission Control Block)



TCP Operation



ACK Generation Rules

- When an in-order data segment is received, delay the ACK until
  - another data segment is received, OR
  - 500 ms has elapsed.

Reduce the # of ACKs;
Apply piggybacking

- When an out of sequence segment with a higher sequence # arrives
  - Send an ACK with the expected seq#

Gap in byte stream detected.

- When a missing segment arrives
  - Send an ACK to announce the next seq# expected.
- If a duplicate segment arrives, immediately send an ACK.

So that further timeouts do not occur….

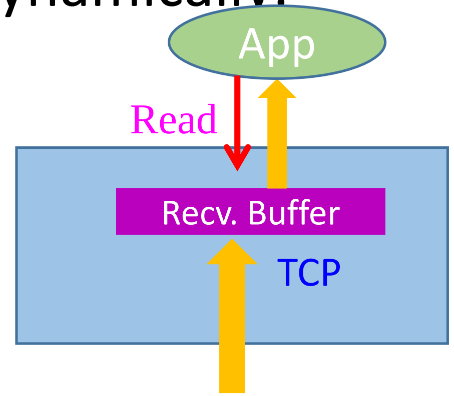
TCP: Flow Control (FC)

- FC: Regulates the amount of data a source can send before receiving an ACK.

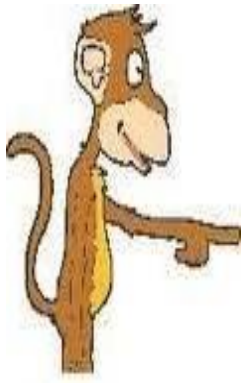
- **Sliding Window Protocol with selective repeat is used.**
  - The bytes within the window are the bytes that can be in transit.
    - There is a separate retransmission timeout (RTO) timer for each segment (except ACKs)

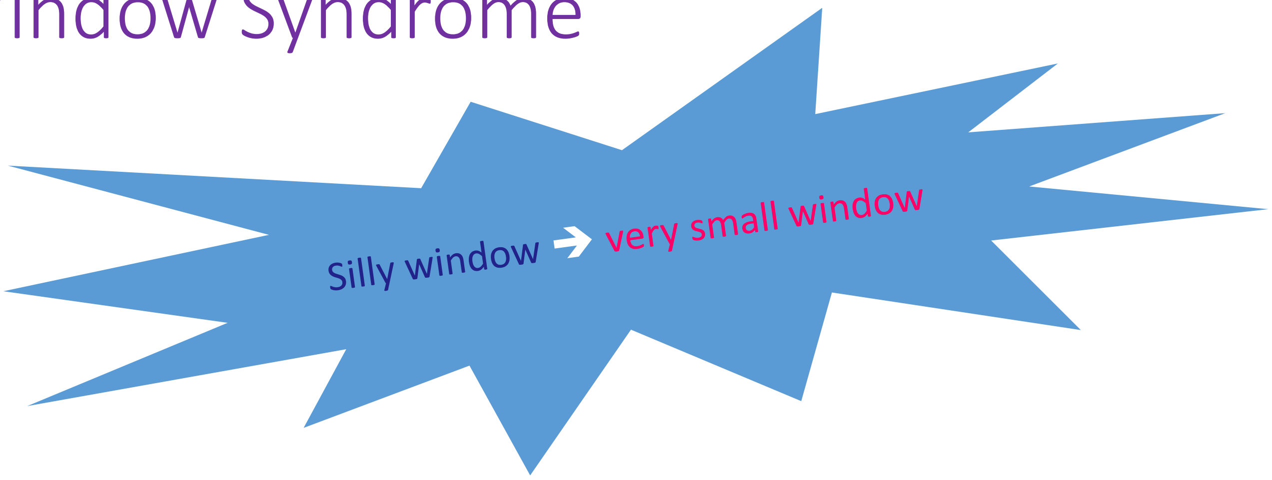
- The **receiver** can open/shrink/close its window, dynamically.

⑩ FC is performed by the receiver



TCP: Silly Window Syndrome



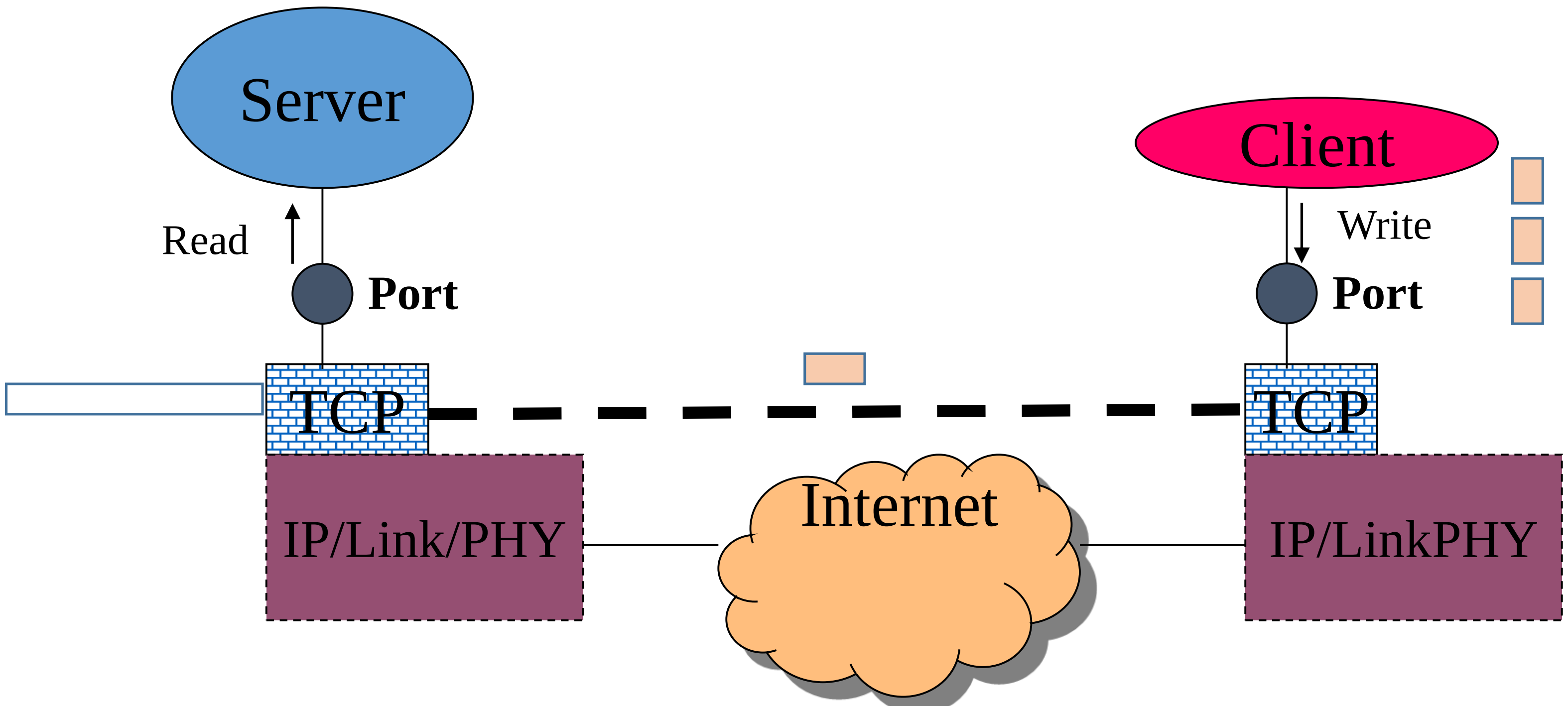


- Silly Window Syndrome (SWS)
  - ✓(#of data bytes in a segment/segment length) is too small

Example: 5 bytes of data; seg. length = 5 +20; ratio = 5/25 = 0.2
1000 bytes of data; seg. length = 1000 + 20; ratio = 1000/1020 = 0.98

- SWS occurs if
  - the sender and/or the receiver is very slow.

TCP: Silly Window Syndrome (Sender produces small data blocks)



Nagle’s solution

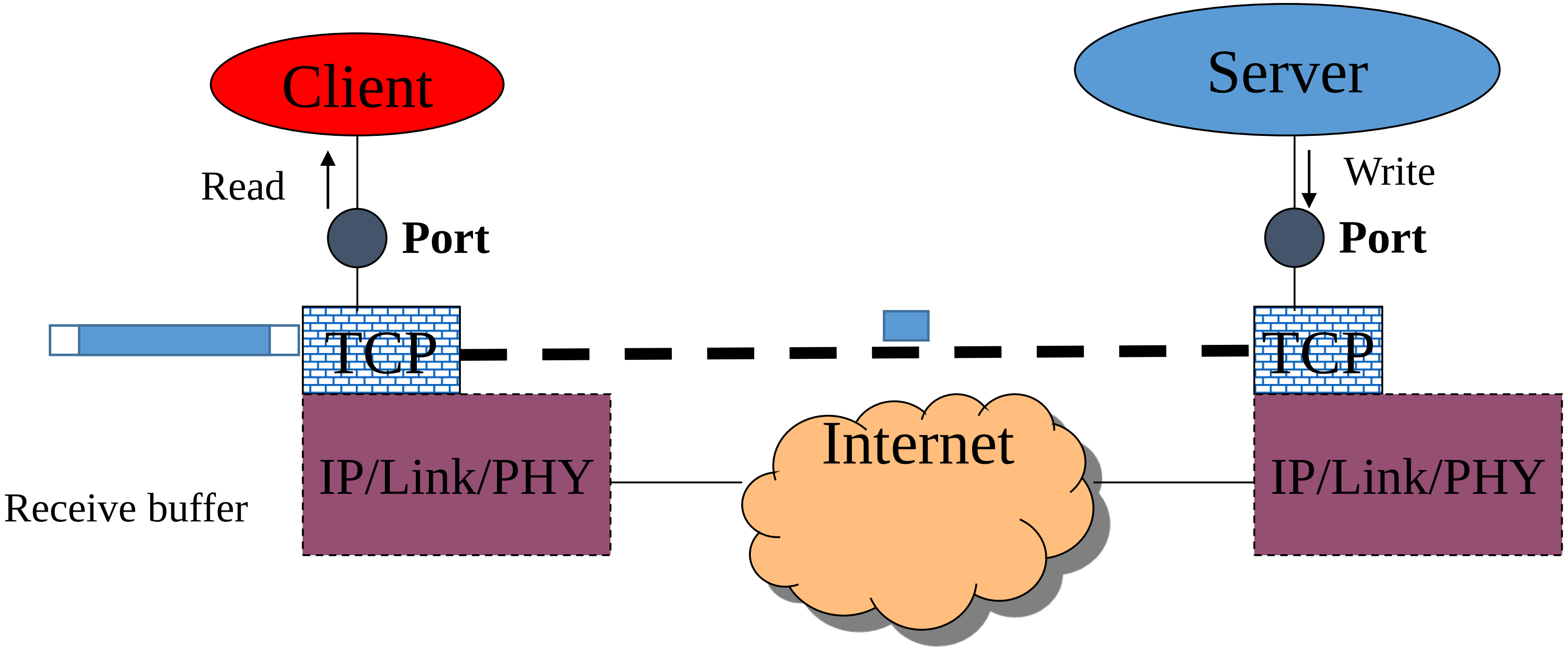
Sender sends the **first segment** even if it is a small one.

Next, **wait** until an ACK is received OR a maximum-size segment is accumulated **before sending the next segment**

…… and repeat “Next” ...

TCP: Silly Window Syndrome (Slow Receiver)

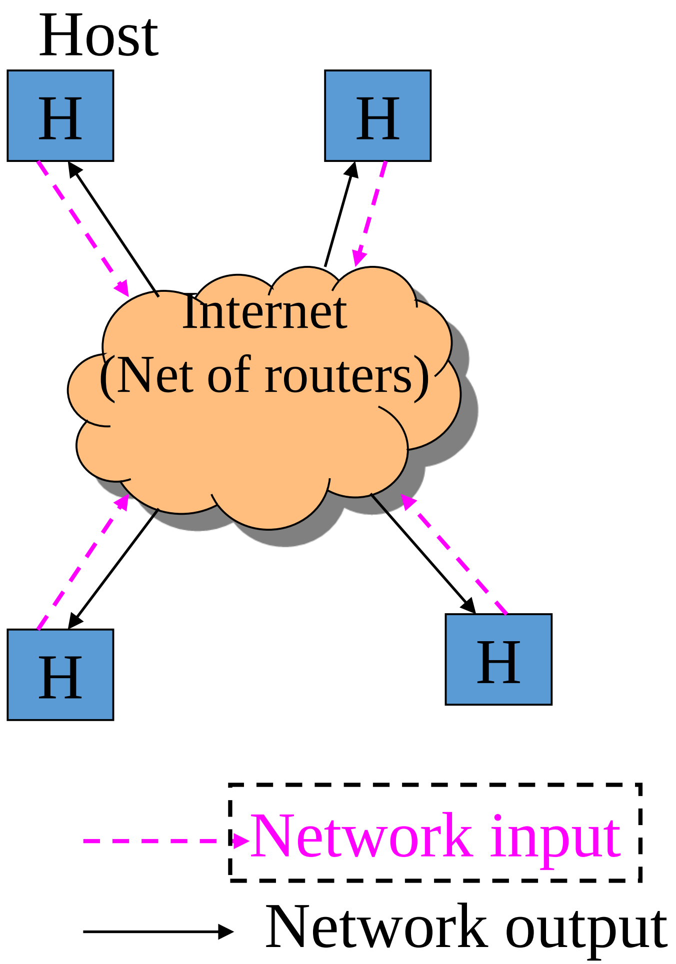
Client is emptying the buffer slowly ➔ RWND is small

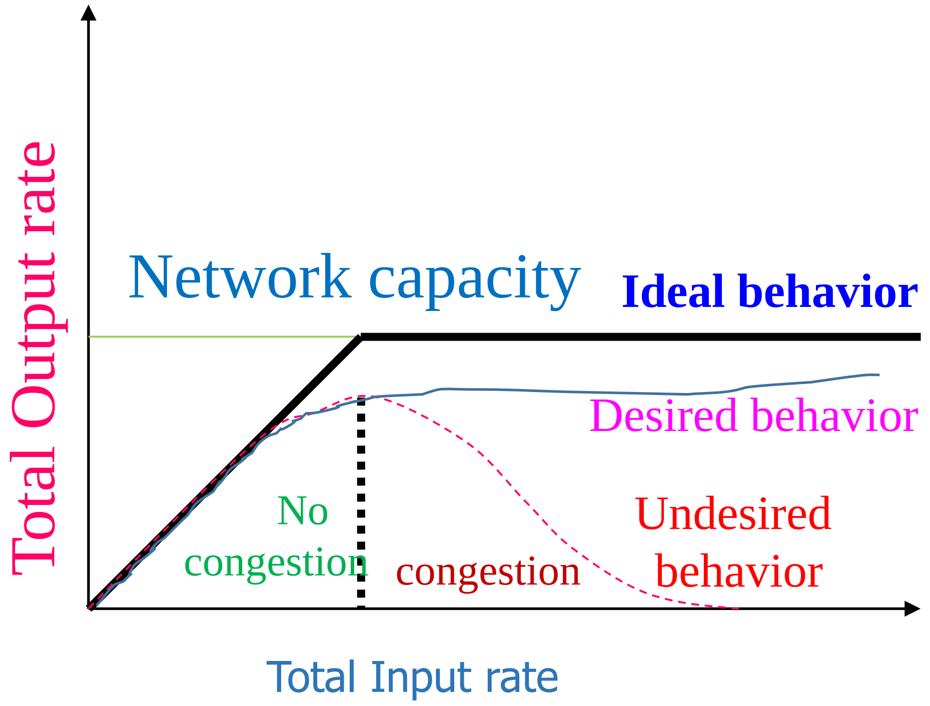


Clarke's solution

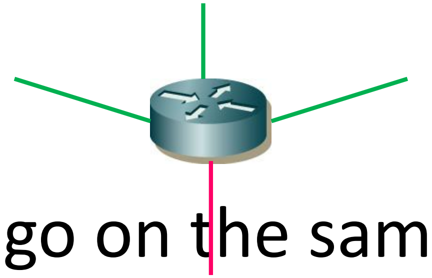
Send an **ACK** and **close** the window until another segment can be received or buffer is ½ empty.

TCP: Congestion Control





Causes of congestion



- Packets arriving on many input links want to go on the same output link
  - Queue builds up for the outgoing link.
    - Router starts dropping packets.

- Slow routers
  - Queues build up if computing tasks take too much time.
    - Buffer mngmt., updating RT, running routing p'cols, looking up RT

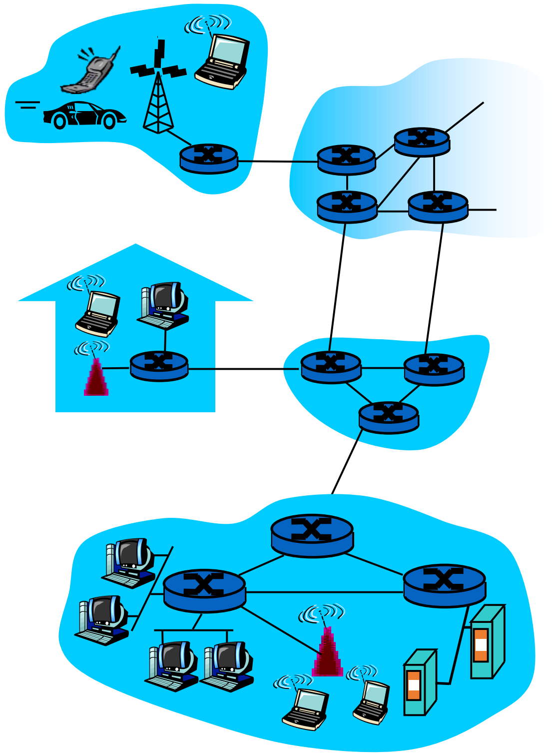
- Hosts produce/download too much …

General Principles of Congestion Control

- **Monitor** the system to know when and where congestion is happening.

- **Communicate** this information to where actions can be taken.

- **Adjust** system operation to correct the problem.



General Principles of Congestion Control

Monitor: A variety of **metrics** can be monitored.

Fraction of all packets discarded due to lack of buffer

Average queue length

Number of retransmitted packets

Average packet delay

Communicate: **Notify** the entities that need to take actions.

Fields in packet headers can be reserved to carry this info.

Hosts and routers can send probe packets to enquire.

Adjust system operation: **Take actions.**

Deny service to some users.

Degrade service to some users.

Have users schedule their demand in a more predictable manner.

All protocol layers contribute to congestion "prevention"

- Link layer

Don't discard out-of-sequence packets.
(Selective-Repeat is better than Go-back-N.)

Reduce the # of smaller packets (e.g. piggyback ACKs).

- Network layer

Apply load balancing: Spread traffic over many paths.

Use good discard policies.

File transfer: Drop <u>new</u> packets.

Real-time: Drop <u>old</u> packets.

- TCP layer

Next …

TCP: Congestion Control (CC)

- CC is achieved by controlling the transmission rate at the sender after "detecting" congestion.
  - Tx rate is controlled by controlling the window size.
- Main idea in controlling CW (congestion window)

❖ Slow start (CW = 1 MSS)
but quickly speed up to congestion threshold (CT): 1,2,4, 8, …CT

❖ Congestion avoidance
beyond threshold, increase linearly: CW++, CW++, …, RWND

v Congestion **detection**
Go back to slow start ….

TCP: Congestion Control

- **Slow start**
  - ✓ Initially, CW = 1: Tx 1 Seg. (MSS)
  - ✓ If ACK received before TO
    CW = 2 (= CW x 2): Tx 2 Segs.
  - ✓ If ACKs received before TO
    CW = 4 (= CW x 2): Tx 4 Segs.
  - ✓ If ACKs received before TO
    CW = 8 (= CW x 2): Tx 8 Segs.

    :
  - ✓ Continue **until** you hit a **threshold**:
    Congestion Threshold (CT)

Normally, CT = 64 KBytes

TO: Timeout

- **Congestion Avoidance:** Additive Inc.
  - ✓ Each time the whole window of segs. is ACKed
    CW = CW + 1
    (CWmax = RWND)

- **Congestion Detection**

RTO timer goes off **Tahoe** + Reno

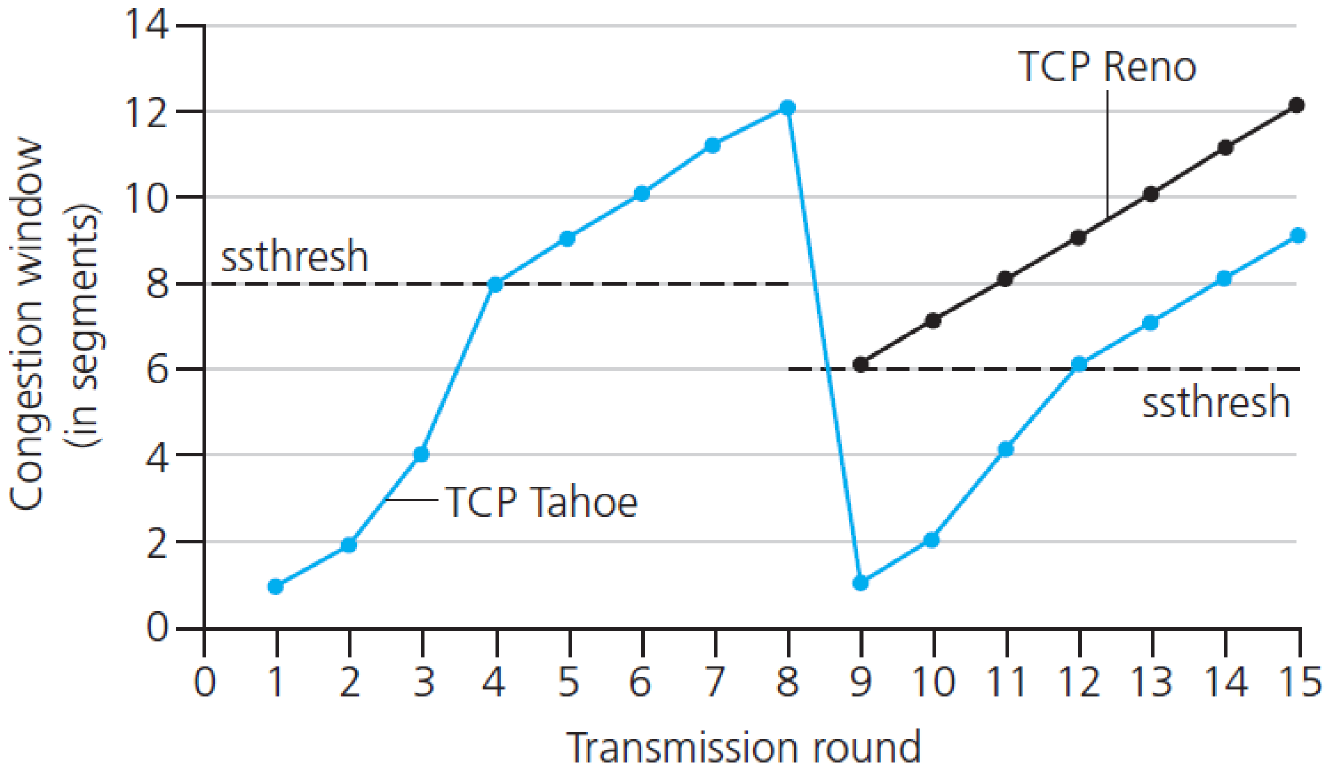
CT = CW/2 and **CW = 1**

3 duplicate ACKs received
(AAAA)

Tahoe CT = CW/2 and **CW = 1**

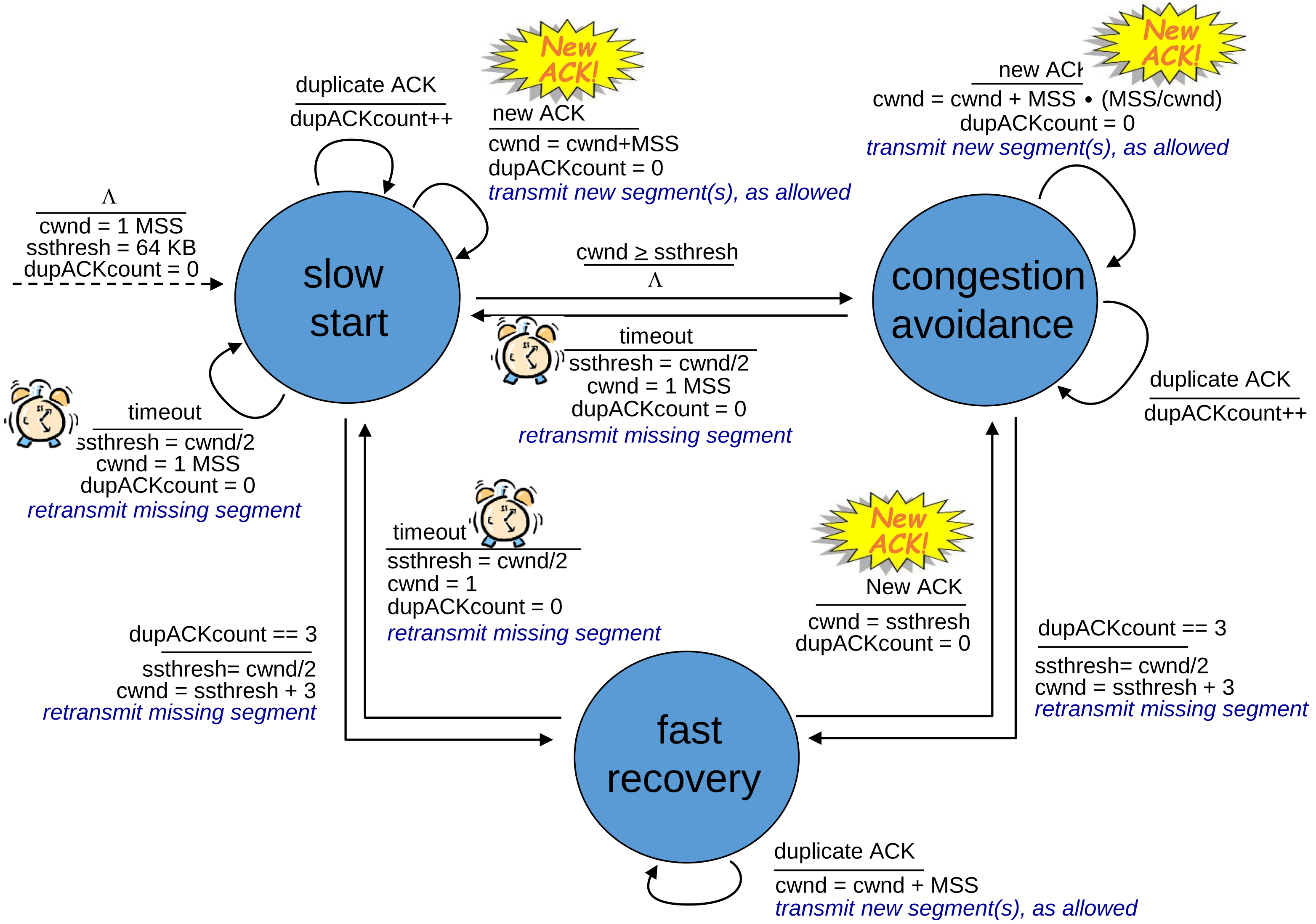
Reno CT = CW/2 and **CW = CT**

Congestion Control



- Variable **`CT`**
- Congestion Threshold is also known as **`ssthresh`**

Summary: TCP Congestion Control



TCP: Timers

Four kinds of timers

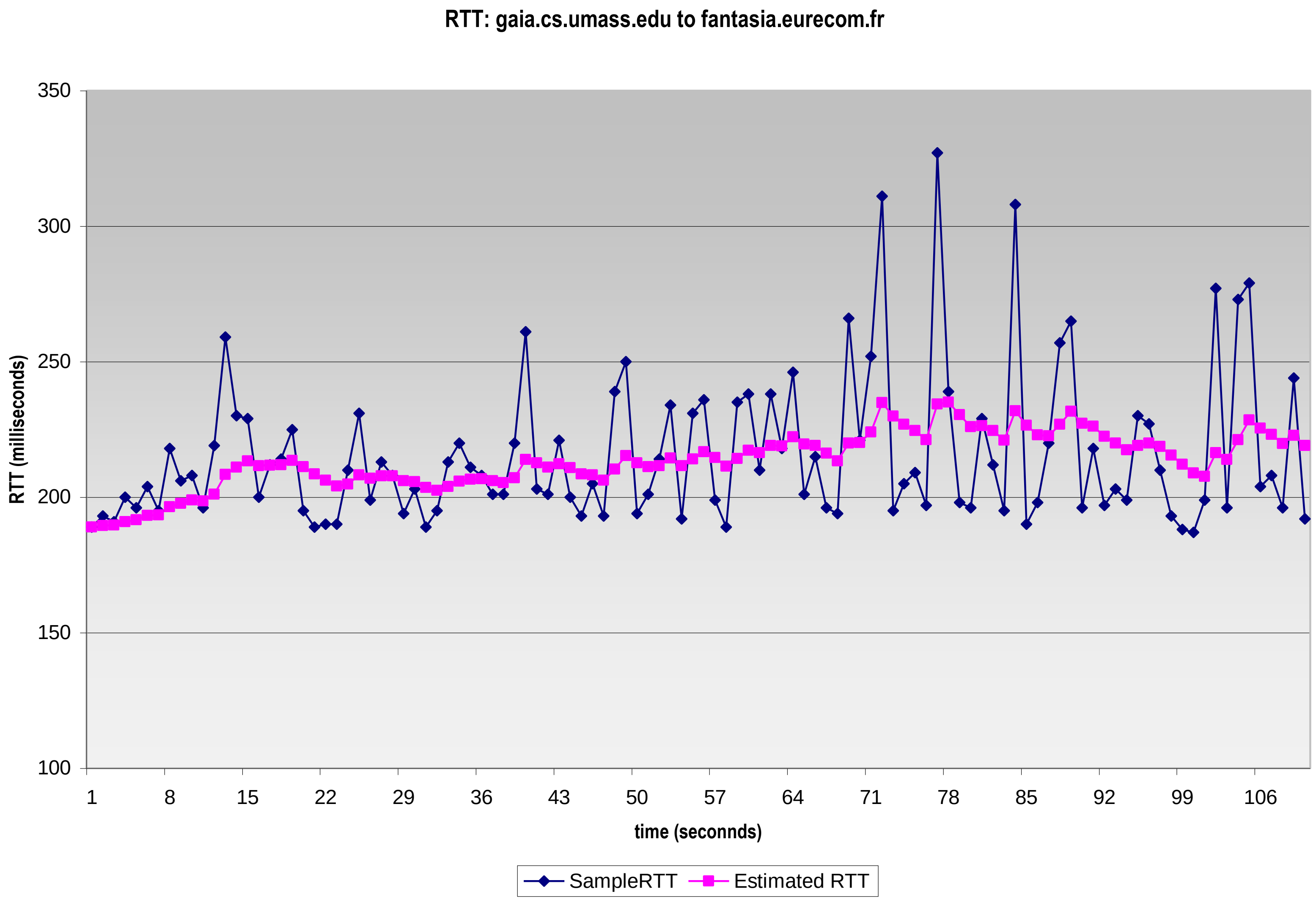
Retransmission Timeout (RTO) timer

Persistence Timer

Keep-Alive Timer

TIME-WAIT Timer (2*MSL timer)

Example RTT estimation:



TCP: Timers (RTO)

- Operation
  - For each segment transmitted (except ACK and RST), start an RTO
  - If RTO goes off, retransmit the segment and restart RTO
- RTO

Remember

Initially: Default value (60s)

After measurements (RTT_M): $RTO = RTT_S + 4 \cdot RTT_D$

RTT_S (RTT Smoothed): $\alpha = 0.125$ (typical value)

After first measurement $RTT_S = RTT_M$

After another measurement $RTT_S = (1 - \alpha) RTT_S + \alpha . RTT_M$

RTT_D (RTT Deviation): $\beta = 0.25$ (typical value)

After first measurement $RTT_D = RTT_M/2$

After another measurement $RTT_D = (1 - \beta) RTT_D + \beta . |RTT_S - RTT_M|$

TCP: Persistence Timer

Recall:
Clarke's solution to
silly window syn.

- A receiver can **close** the window and reopen it with an ACK
  - Problem: If the ACK is lost, there is deadlock.
    - Solution:

✓ When a sending TCP receives a segment with RWND = 0, **start a persistence timer.**

✓ Persistence timer goes off: Send a *<u>probe</u>* segment (1 byte data) to alert the receiver.

✓ Persistence timer value
Initially: Equal to RTO
Subsequently: **Doubled** with each Tx of the probe.
Saturates at 60 sec.

TCP: Timers (Keepalive and TIME-WAIT)

- Keepalive Timer

  ✓ To sustain mostly idle connections (as between BGP routers)

  ✓ Each time the server hears from a client
  Reset the timer: 2 hours.
  If the server does not hear from the client for **2 hours**
  Send a **probe** segment.
  If there is no response after 10 probes (75 sec apart)
  Assume that the client is down.

- TIME-WAIT Timer (2.MSL) Maximum Segment Lifetime

  ✓ Used during connection termination.

  ✓ Standard: MSL = 120 sec (implementations choose a smaller value)

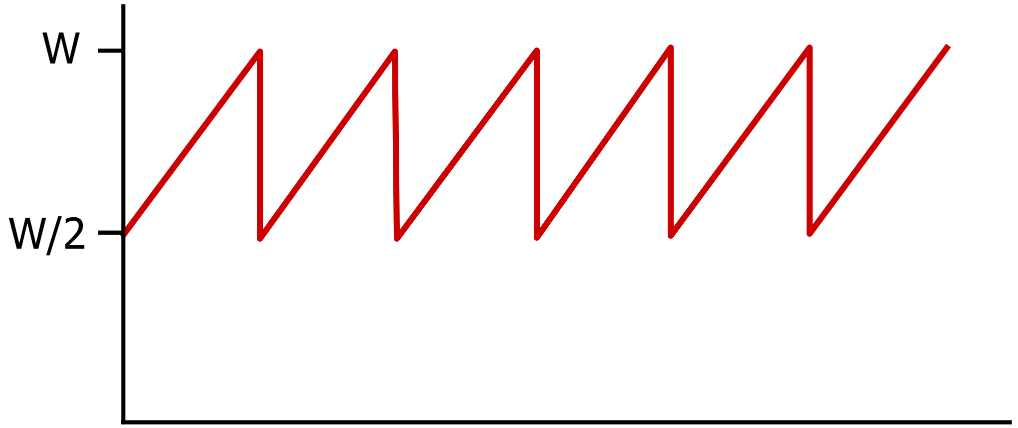
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP throughput

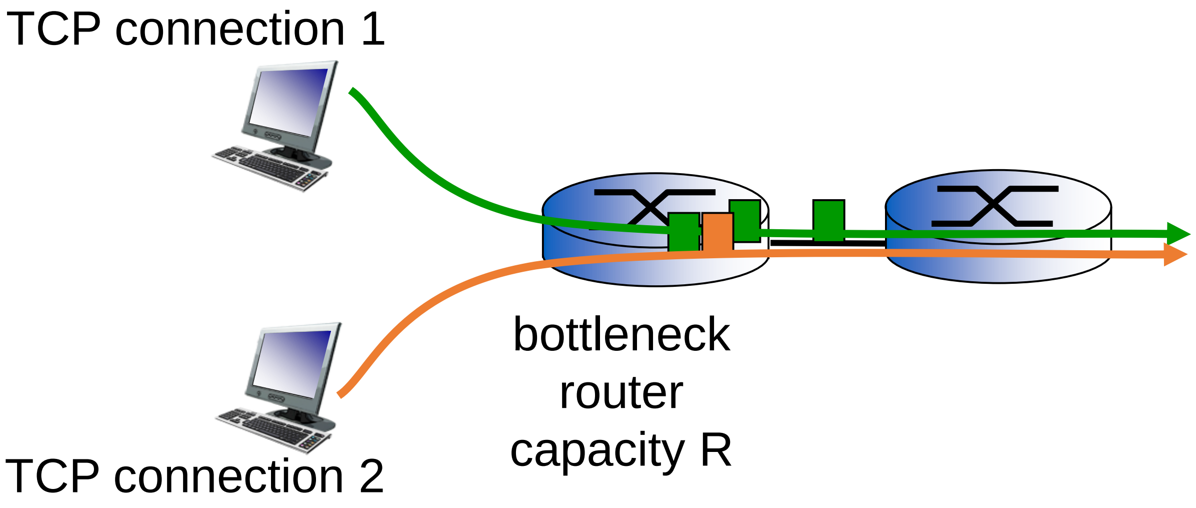
- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP Fairness

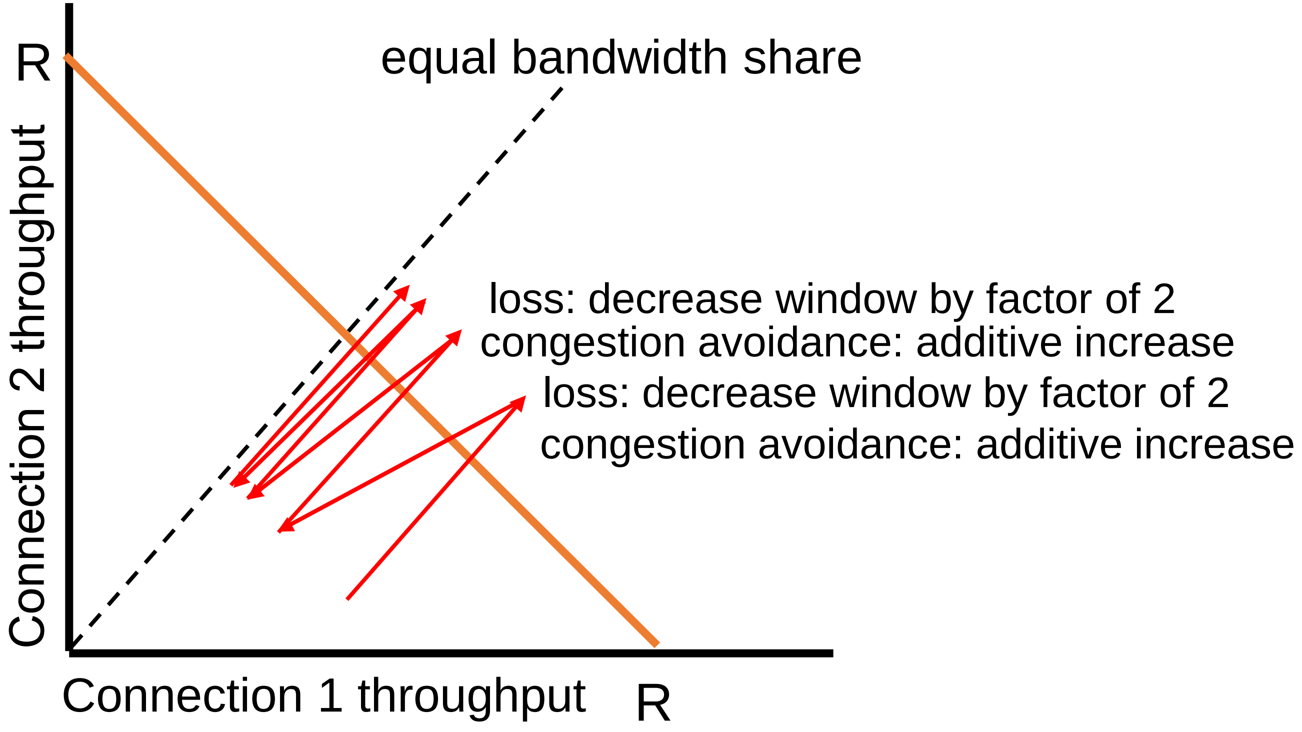
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions:

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



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"