- Reliability
- Flow control
  - Two sides (sender and receiver) have the same data rate handling capability
  - Sliding window protocol for reliable transmission – it automatically handles flow control
    - Send one window worth of packets and then wait for acknowledgement
    - As a result of this there is no overflow on the other side
- Connection Management
- Congestion Management (traffic)

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

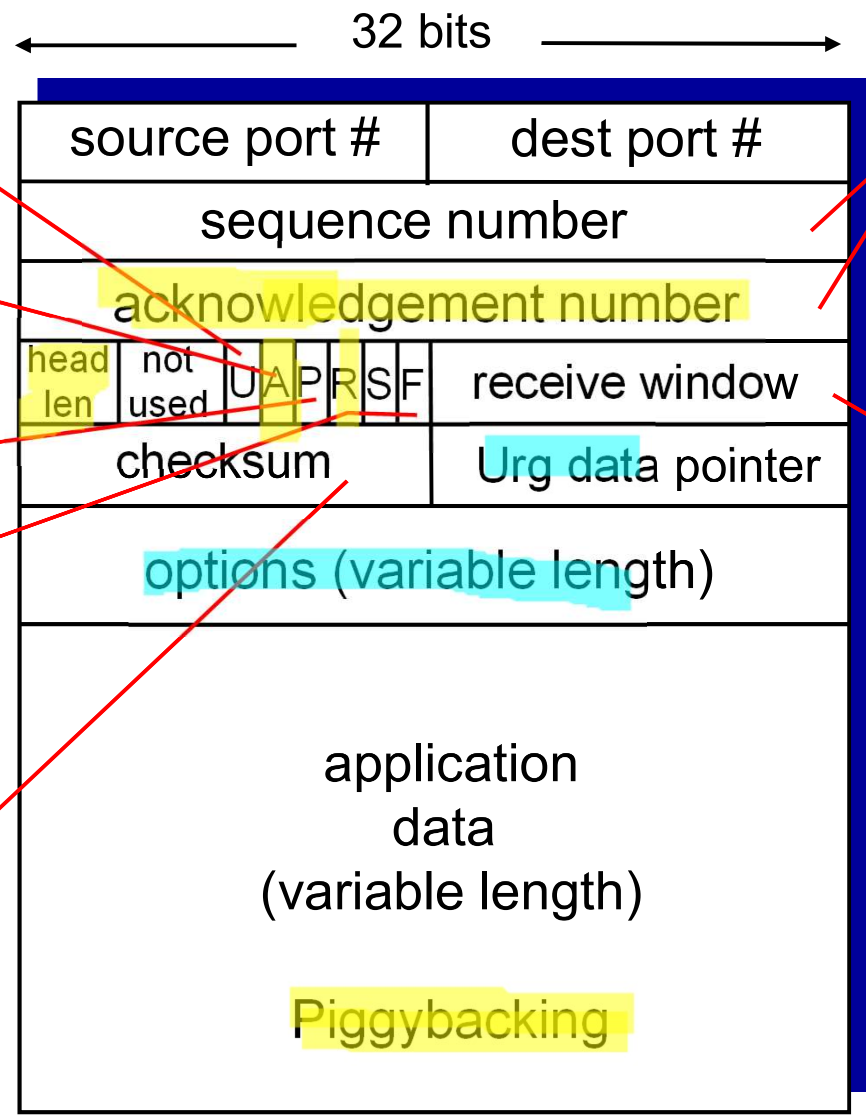
URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

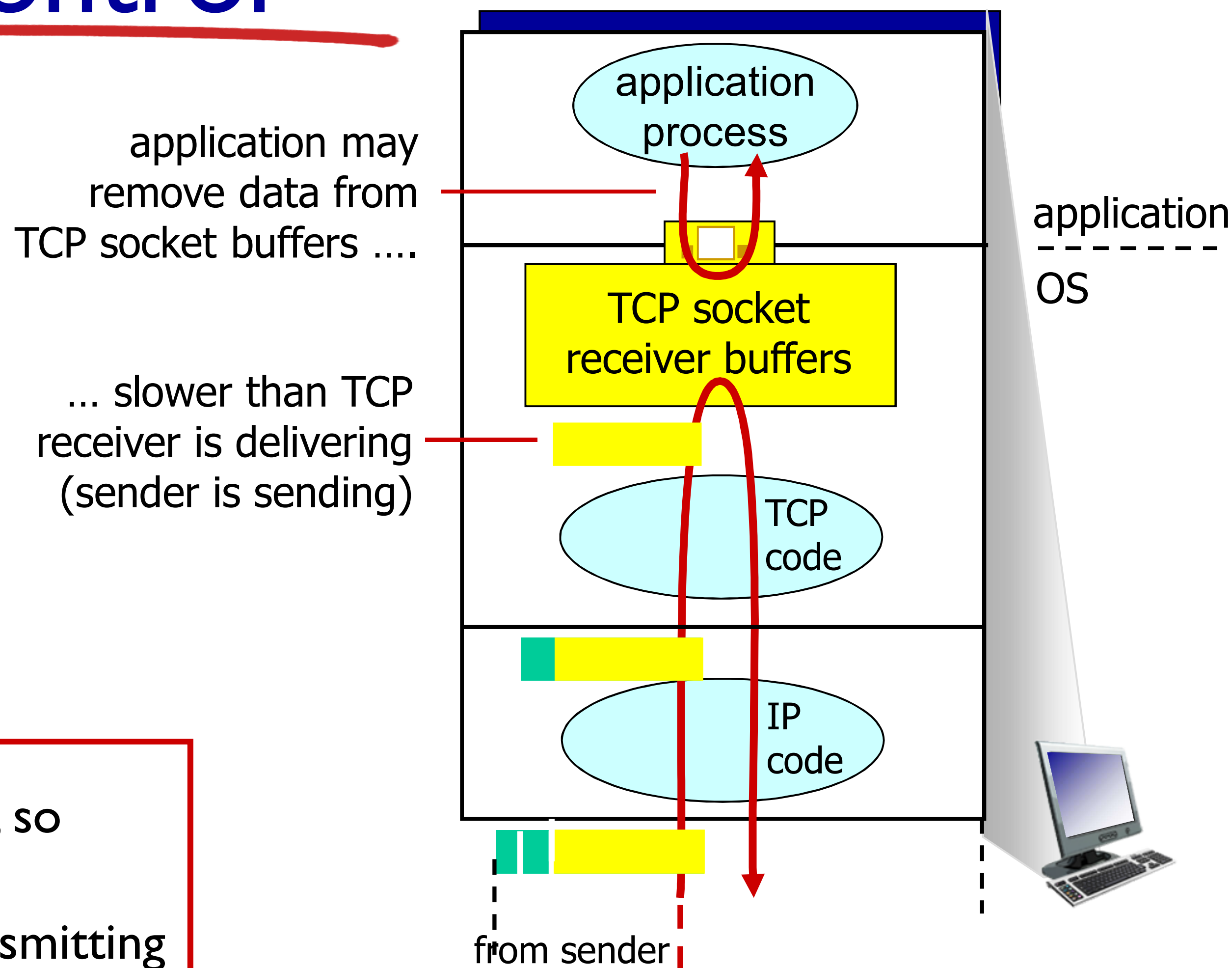
Internet checksum (as in UDP)



counting by bytes of data (not segments!)

bytes rcvr willing to accept

TCP flow control



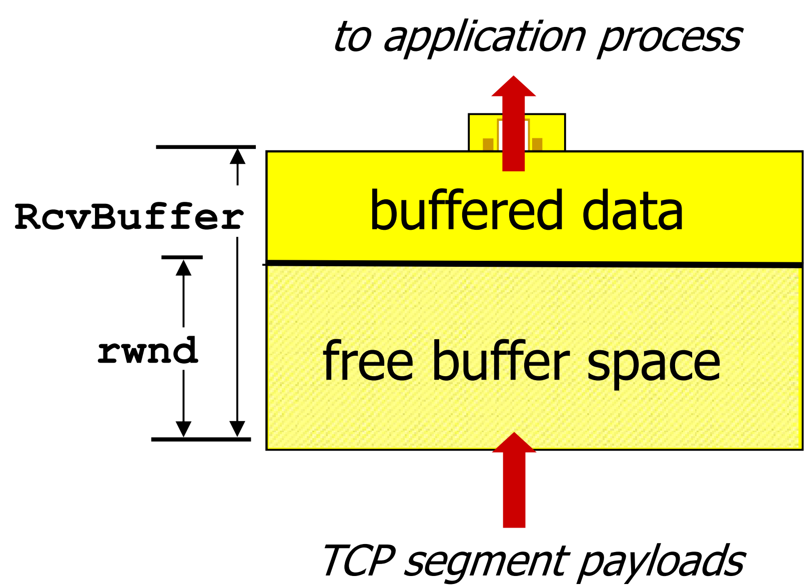
receiver protocol stack

flow control

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

TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`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 receiver's **`rwnd`** value
- guarantees receive buffer will not overflow



receiver-side buffering

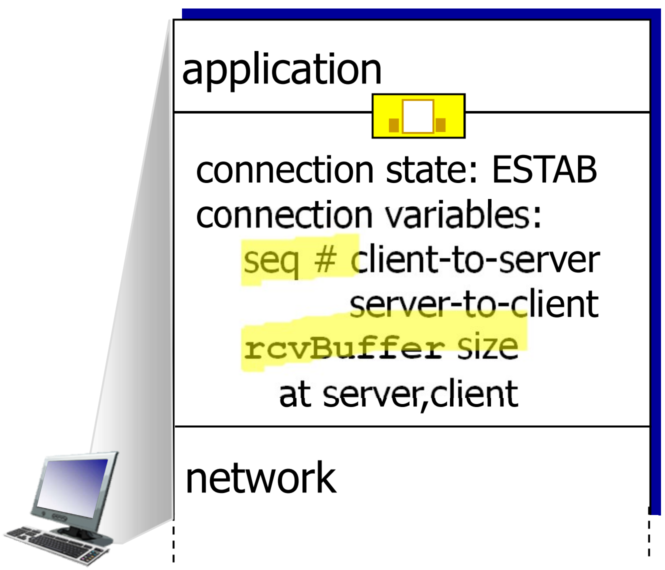
sockets

- Create socket | create socket
- Fill up the addr | bind()
- Connect() | listen()
- | accept()
- ------------------------
- Underlying network is connection-less
- TCP creates a table that has details of the port# of the corresponding node
- Entry – starting seq no, src port#, dest #. Src IP, dest IP.
- Whenever you have a packet received, you search this table for the port#s and map it to the connection-id (socket id)

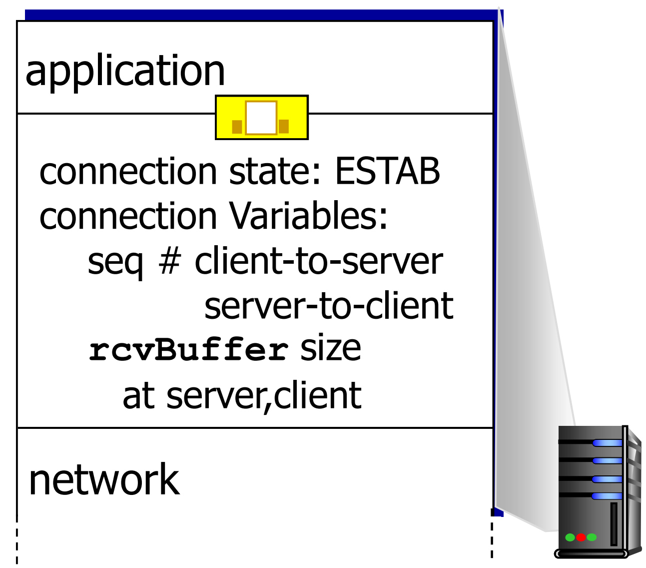
Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



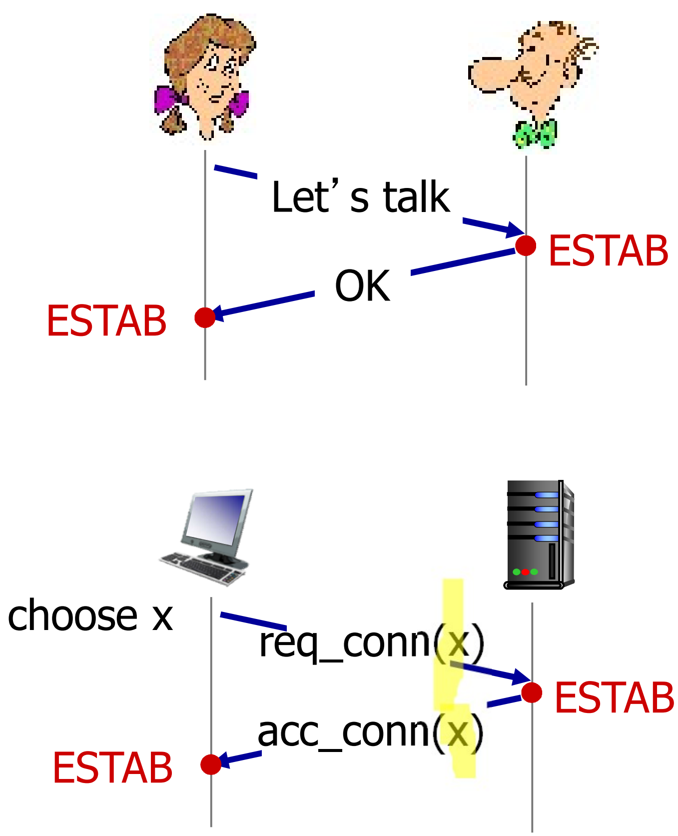

```
Socket clientSocket =
  newSocket("hostname","port
  number");
```




```
Socket connectionSocket =
  welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:

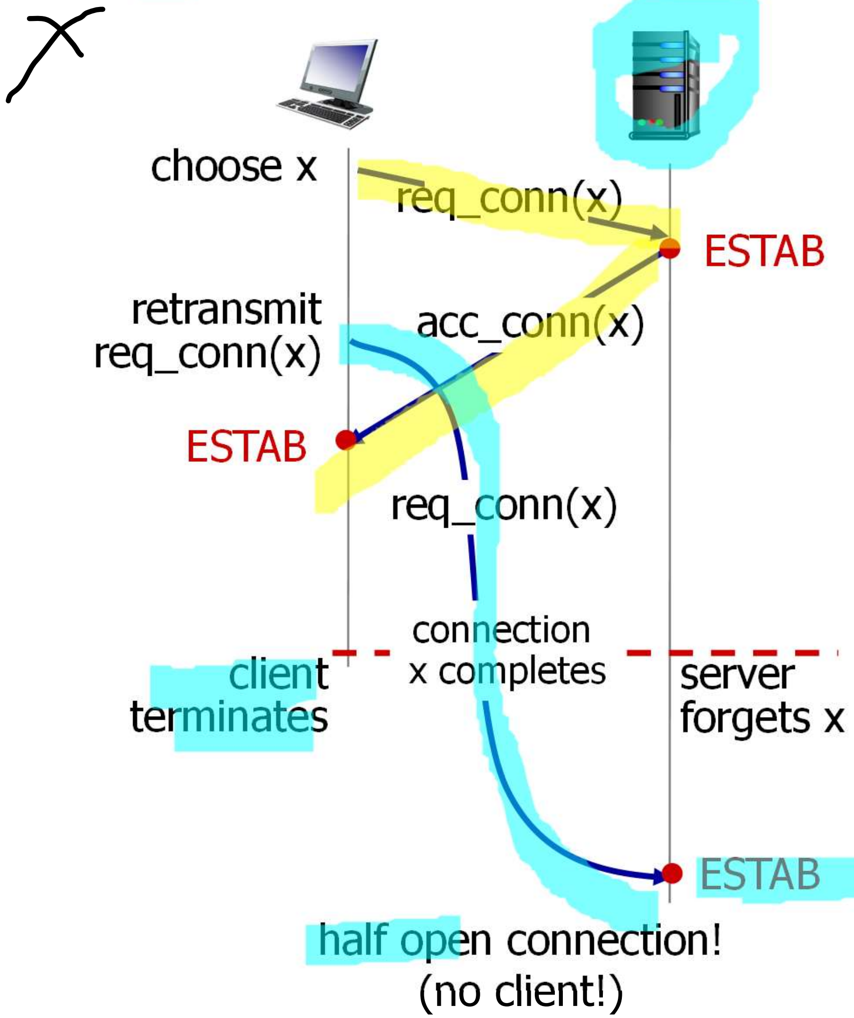


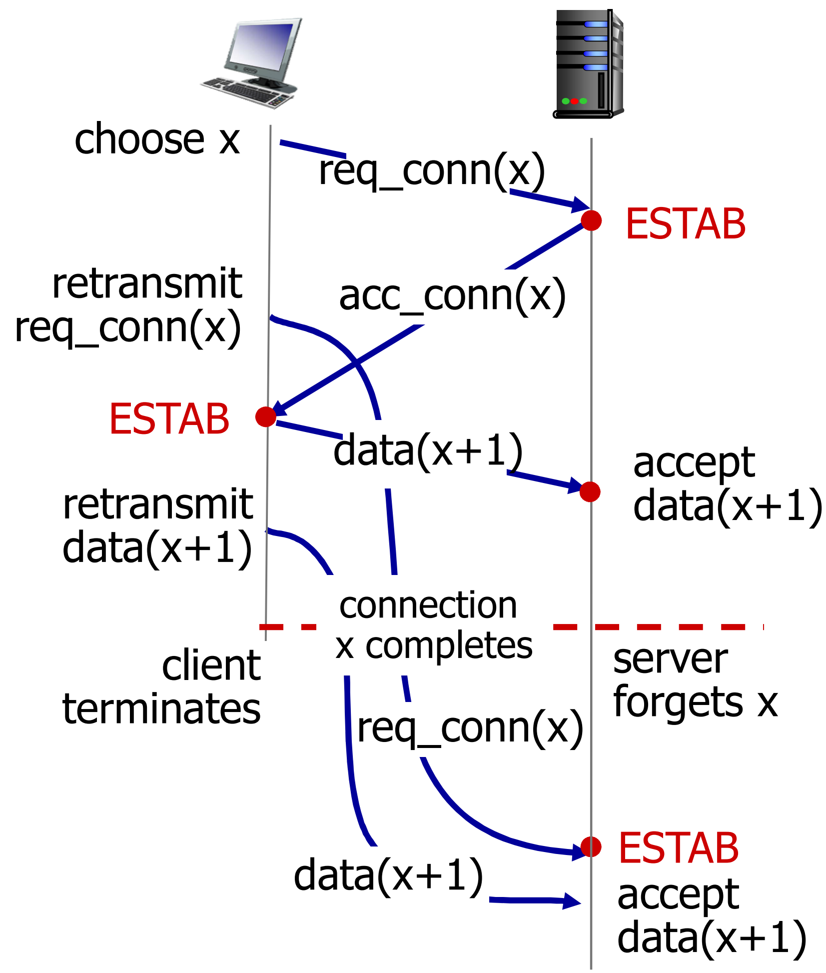
Q: will 2-way handshake always work in network?

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

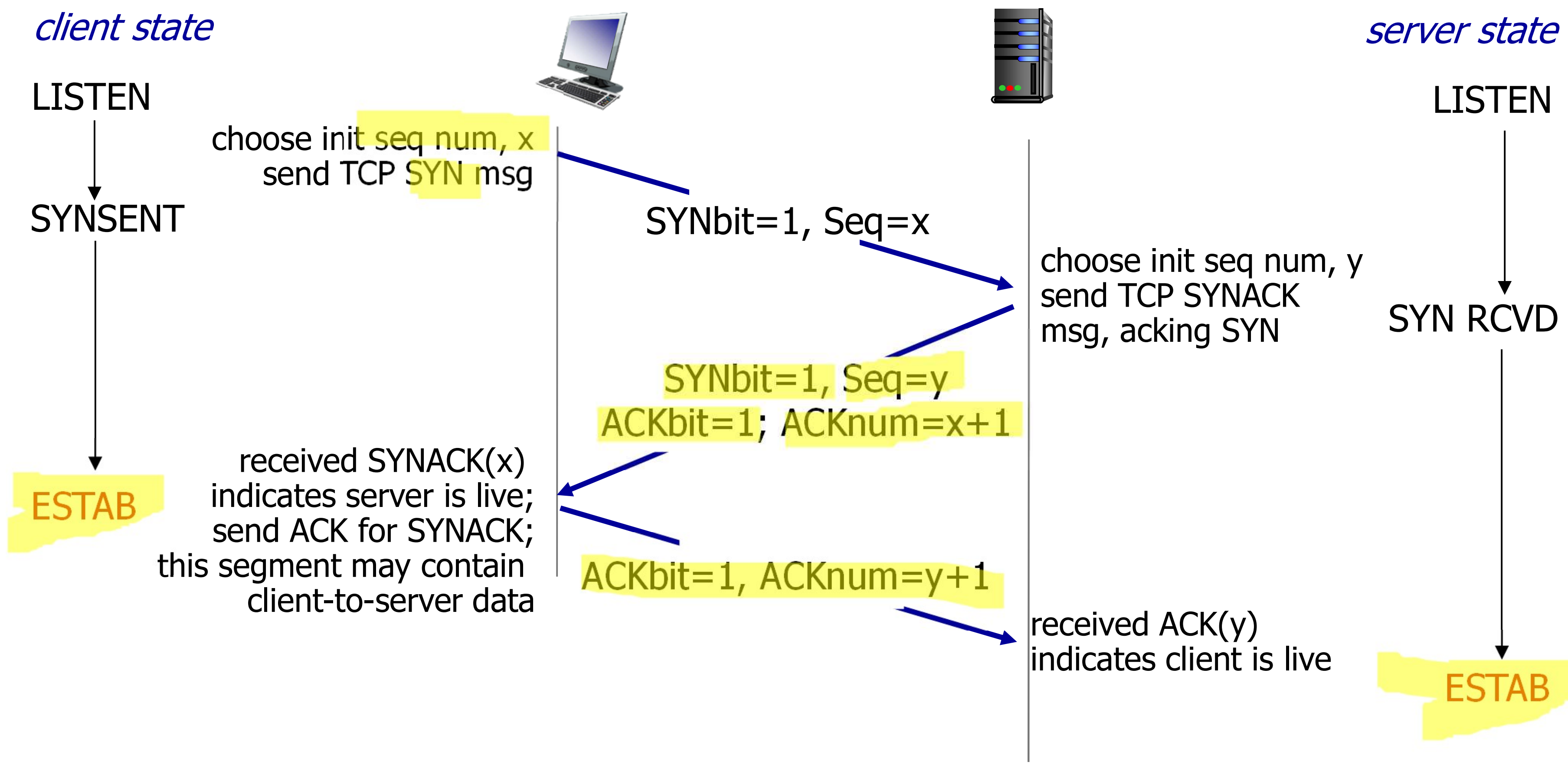
Agreeing to establish a connection

2-way handshake failure scenarios:

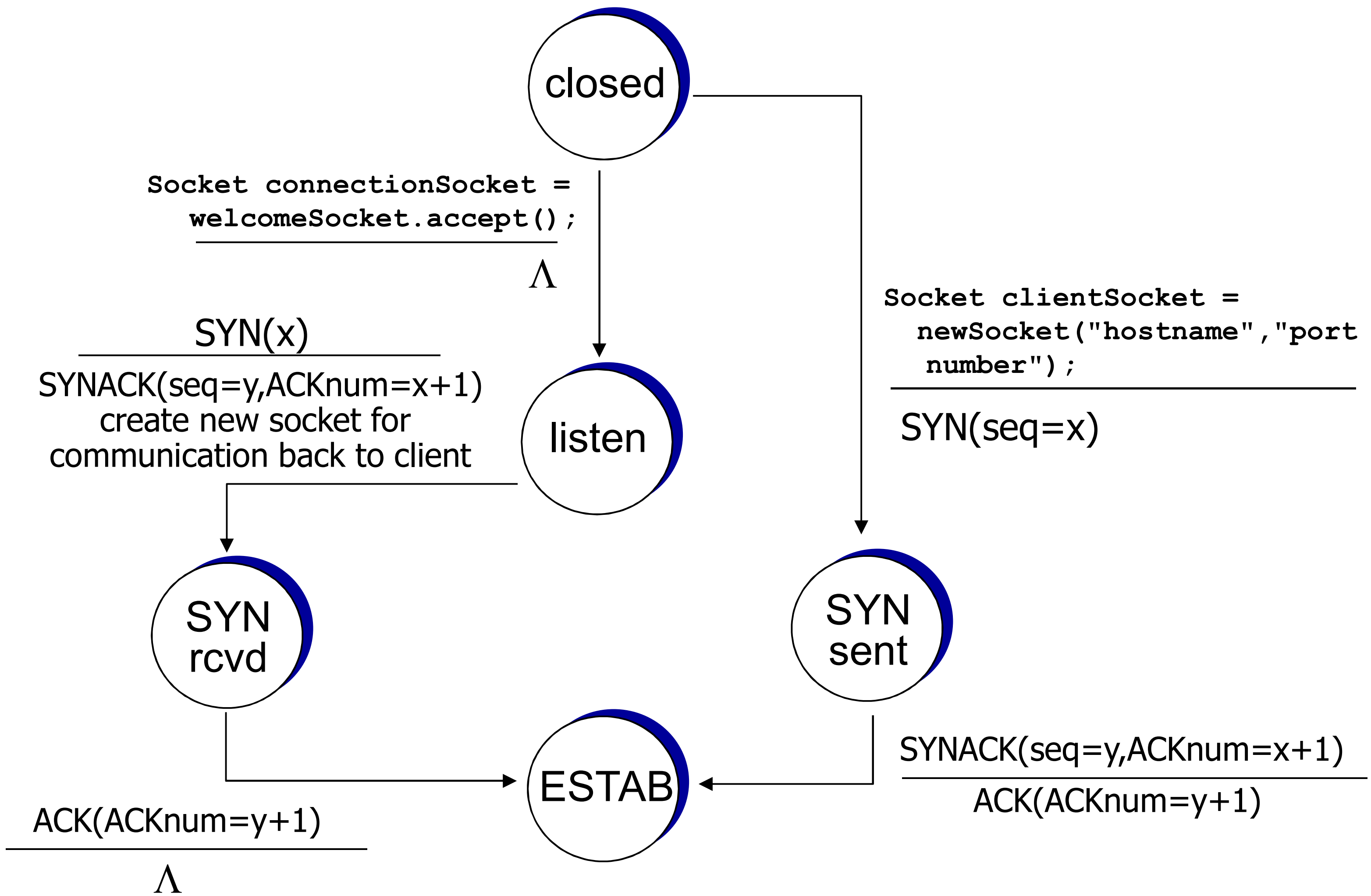




TCP 3-way handshake



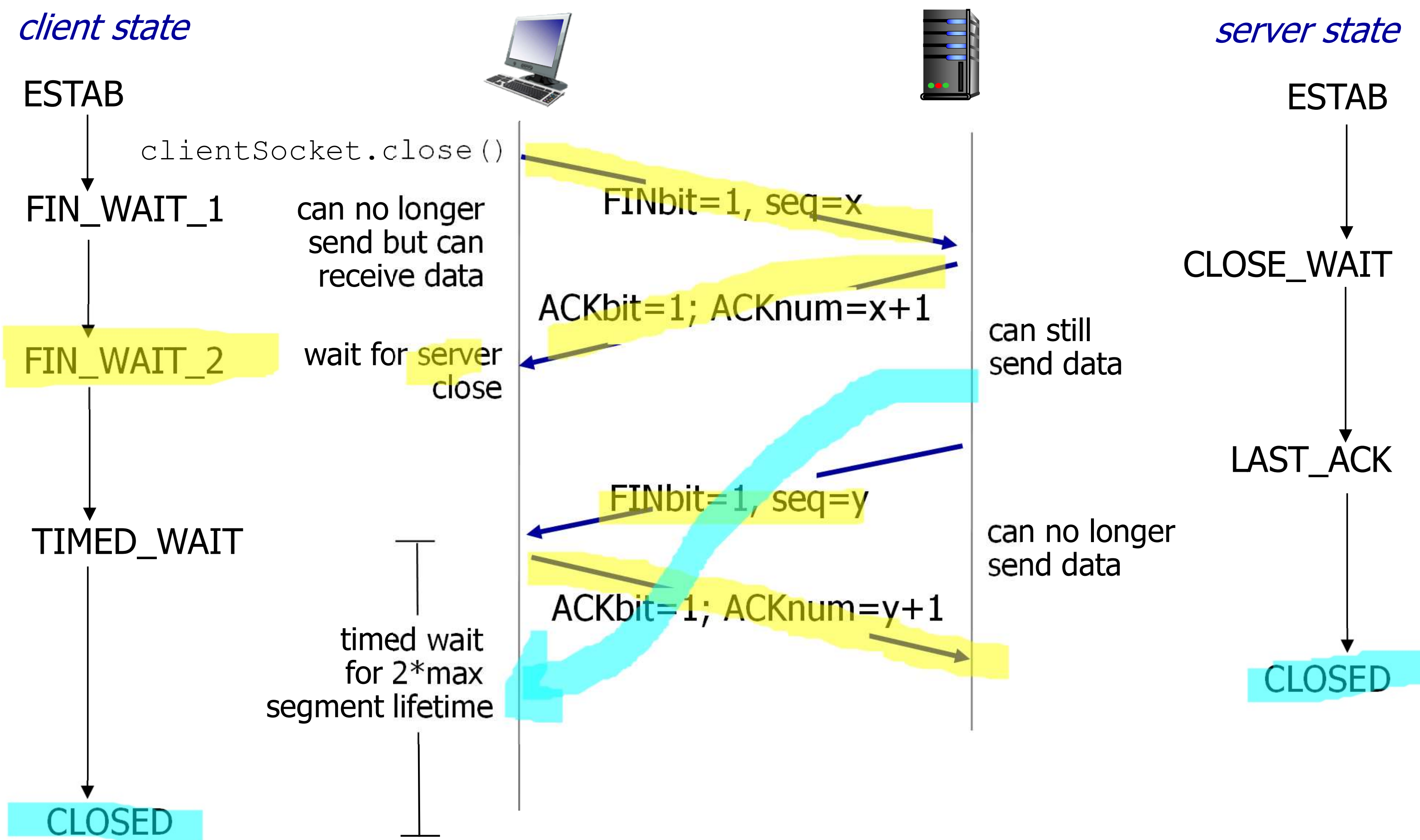
TCP 3-way handshake: FSM



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
- ------------------------------
- Half open / half closed socket
- Close(send-id) and recv-id is still open

TCP: closing a connection



- best effort service, store and forward. (queue- finite buffer)
- Sender (TCP) – flow control
- Sender (UDP) – no reliability, no flow control
- Buffer can overflow and packets can get lost.

- Road network –
  - There are more vehicles than the capacity of the road they we say that road is congested.
  - Not desirable. Same thing happens in Internet as well.
- ------------------
- Reduce the vehicles coming on the road.
- Expressway – controlled entry
- *> -- *>
- Entry point and exit point communicate with each other and when one vehicle/packet leaves exit, another vehicle/packet is allowed to enter.
- Ensure that total number of packets remain constant and there is no congestion.
- In TCP, we already have a connected sender/recv and we have ACK.

- Work in steady state
- If your network is already congested, it will remain congested.
- If the network is underutilized, it remains under-utilized.
- -------------------
- If congestion, tcp must slow down!
- Congestion -> buffer overflow -> packet loss
- -> no ack -> timeout.
- Timeout -> assume that network is congested -> slow down.
- -------------------

- As long as the ack keep coming, increase the rate of transmission (takes care of under-utilization)

- SLOW_START_ALGORITHM – we are going to study next.

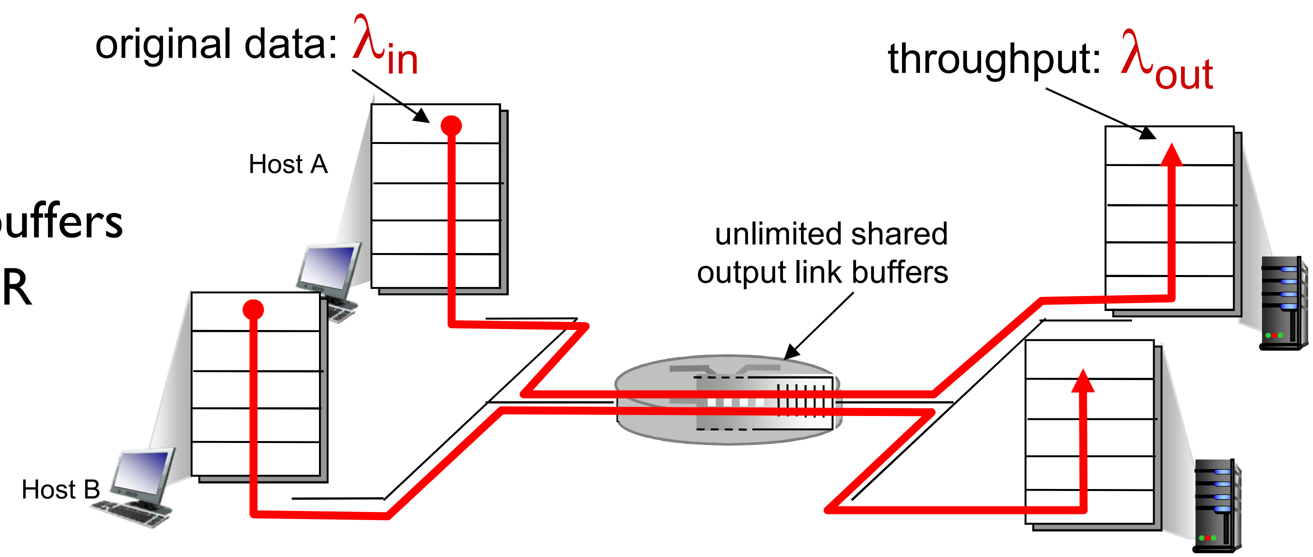
Principles of congestion control

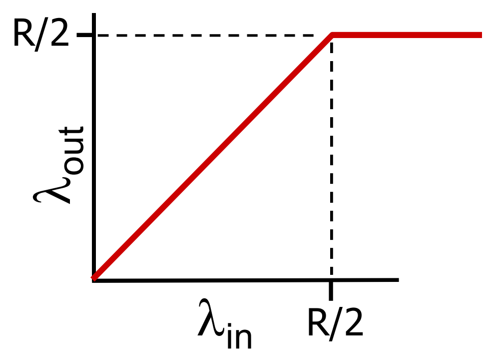
congestion:

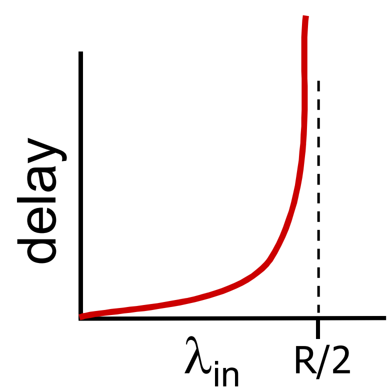
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission



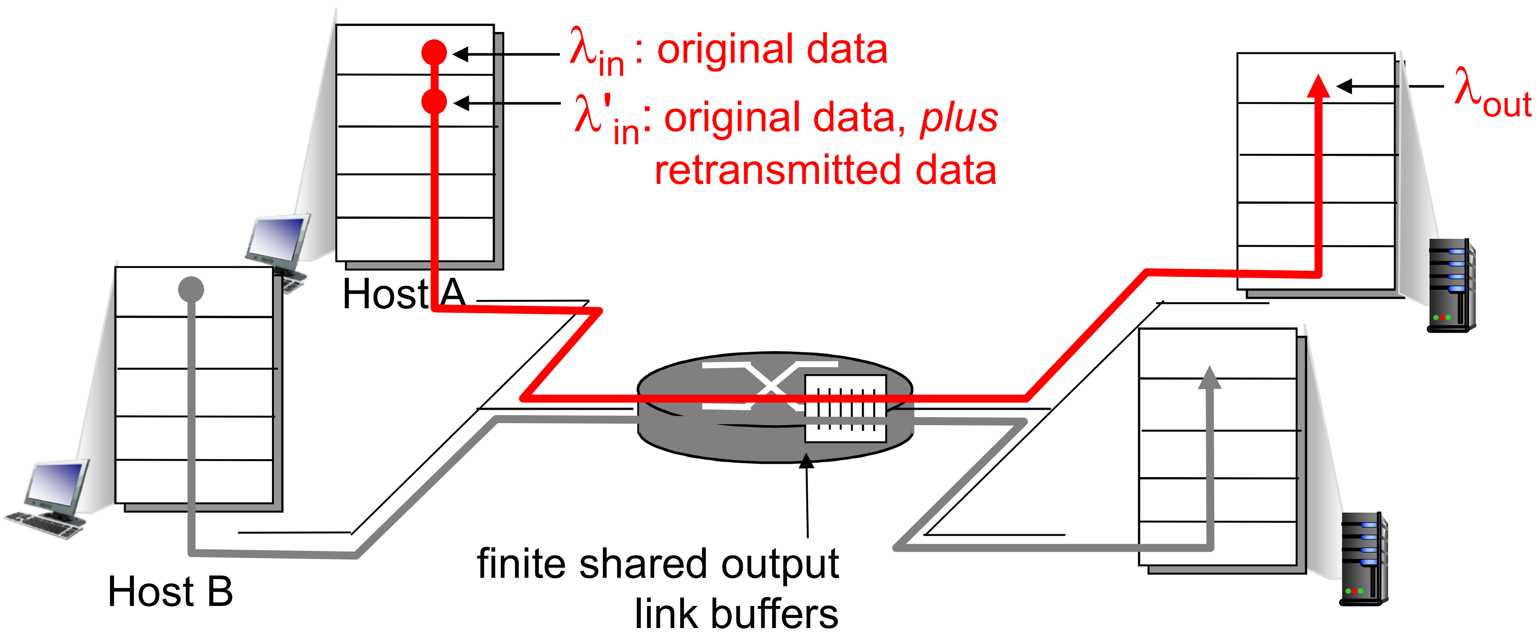




- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, approaches capacity

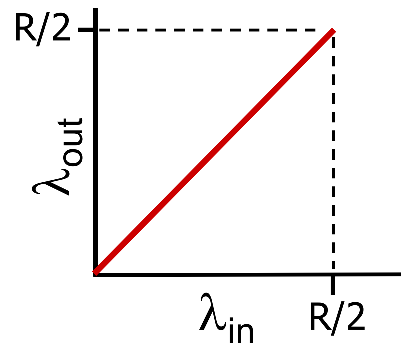
Causes/costs of congestion: scenario 2

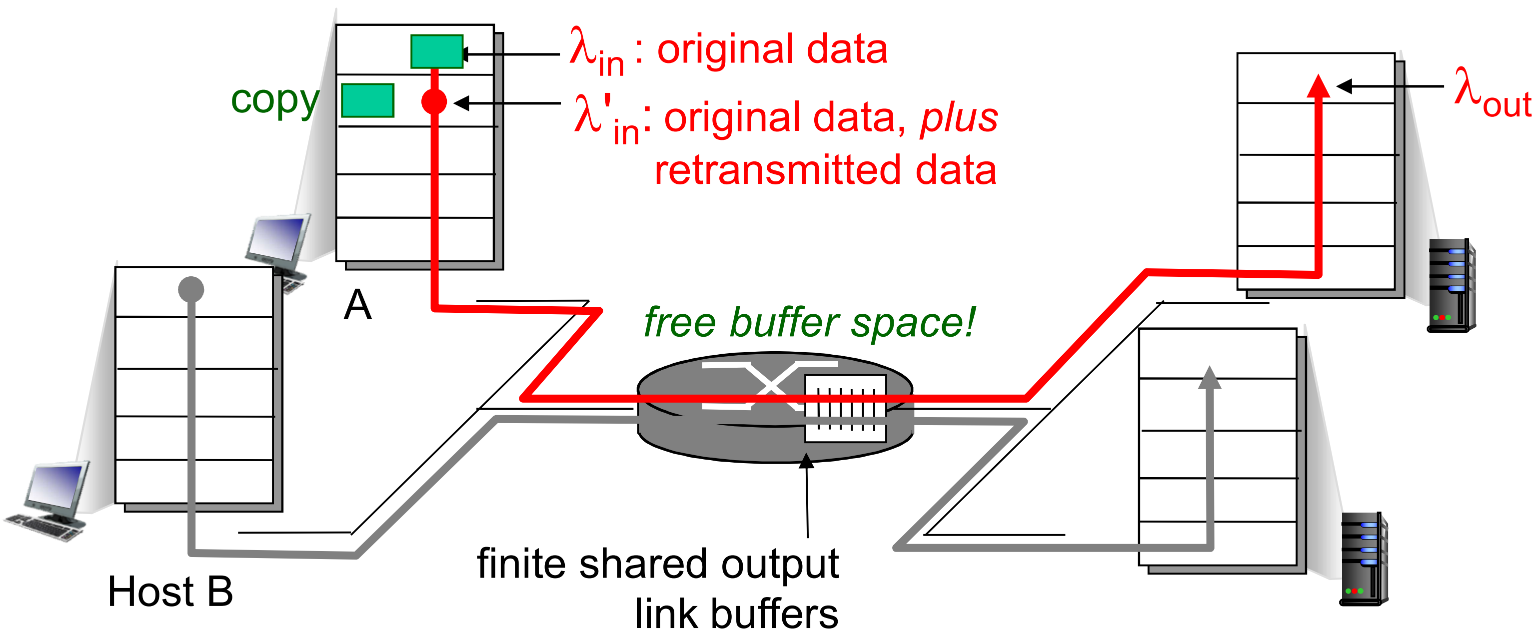
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



Causes/costs of congestion: scenario 2

idealization: perfect knowledge
- sender sends only when router buffers available

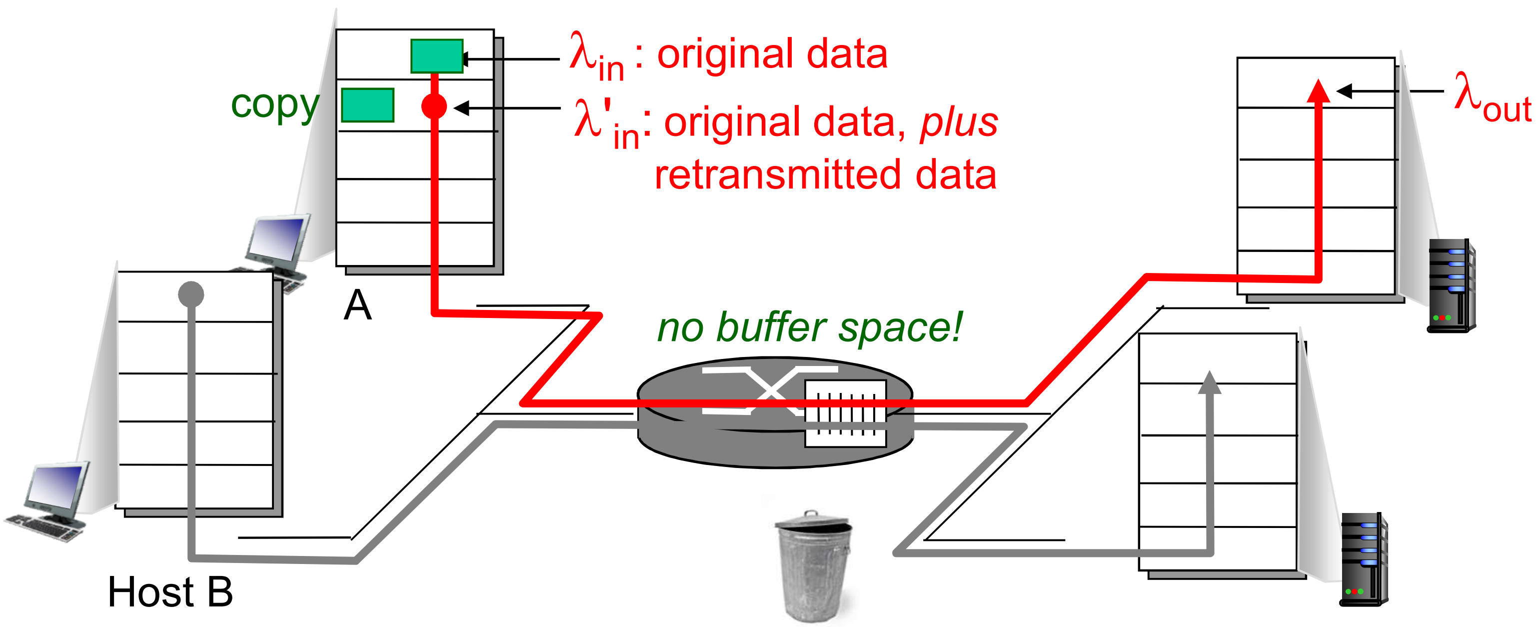




Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost, dropped at router due to full buffers

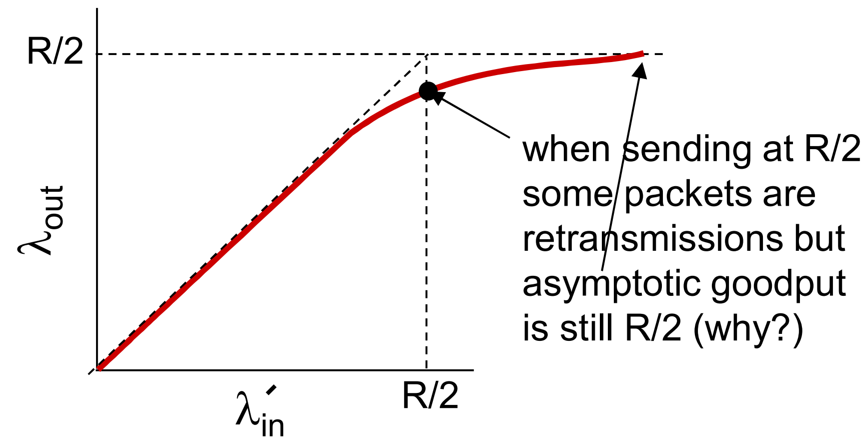
- sender only resends if packet *known* to be lost

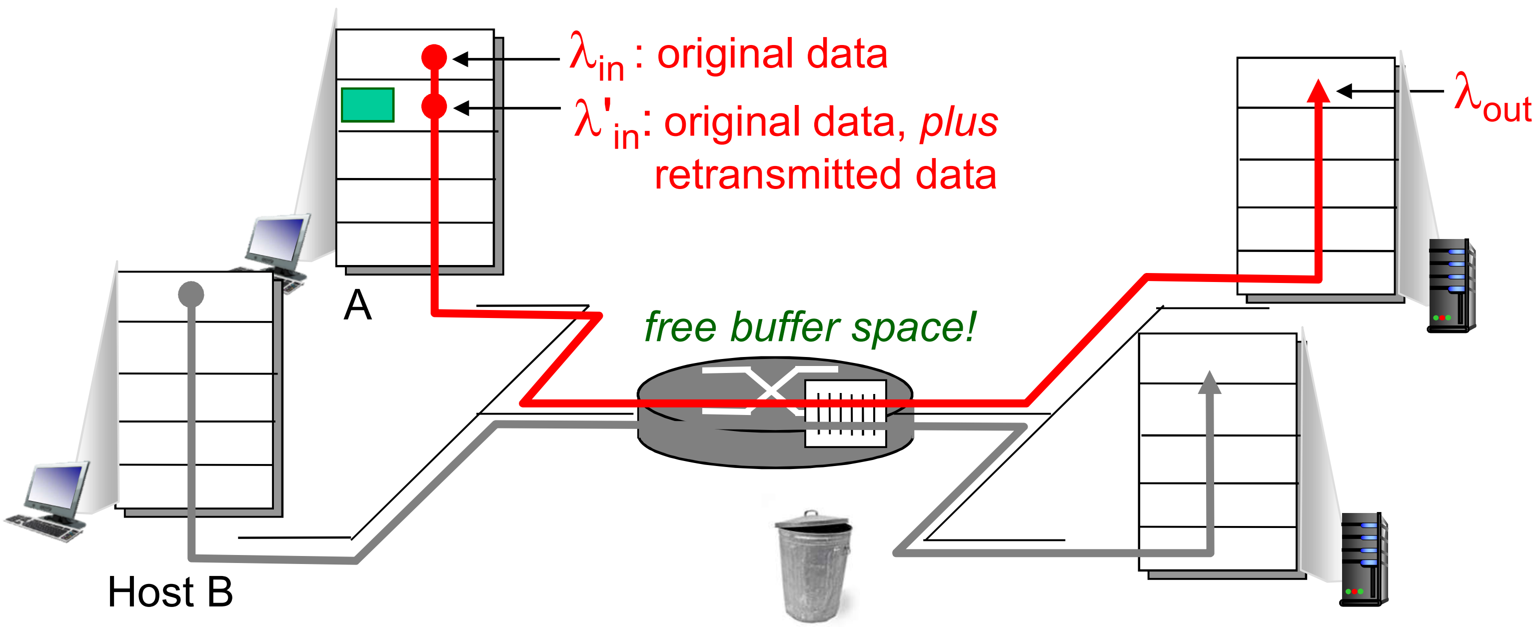


Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

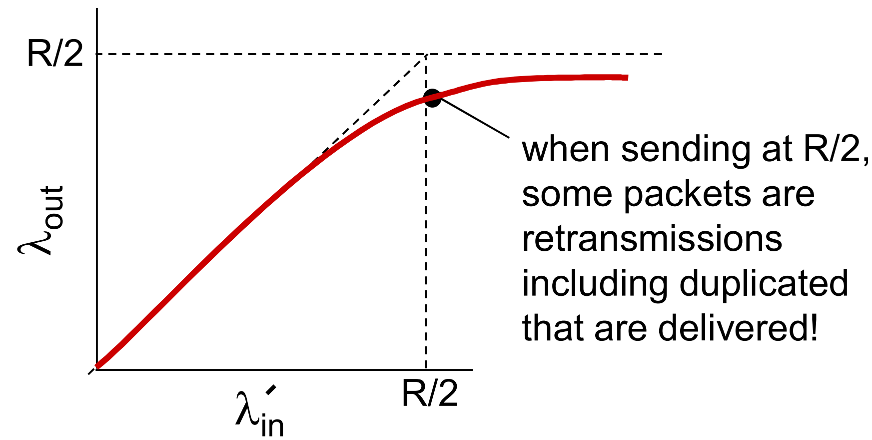


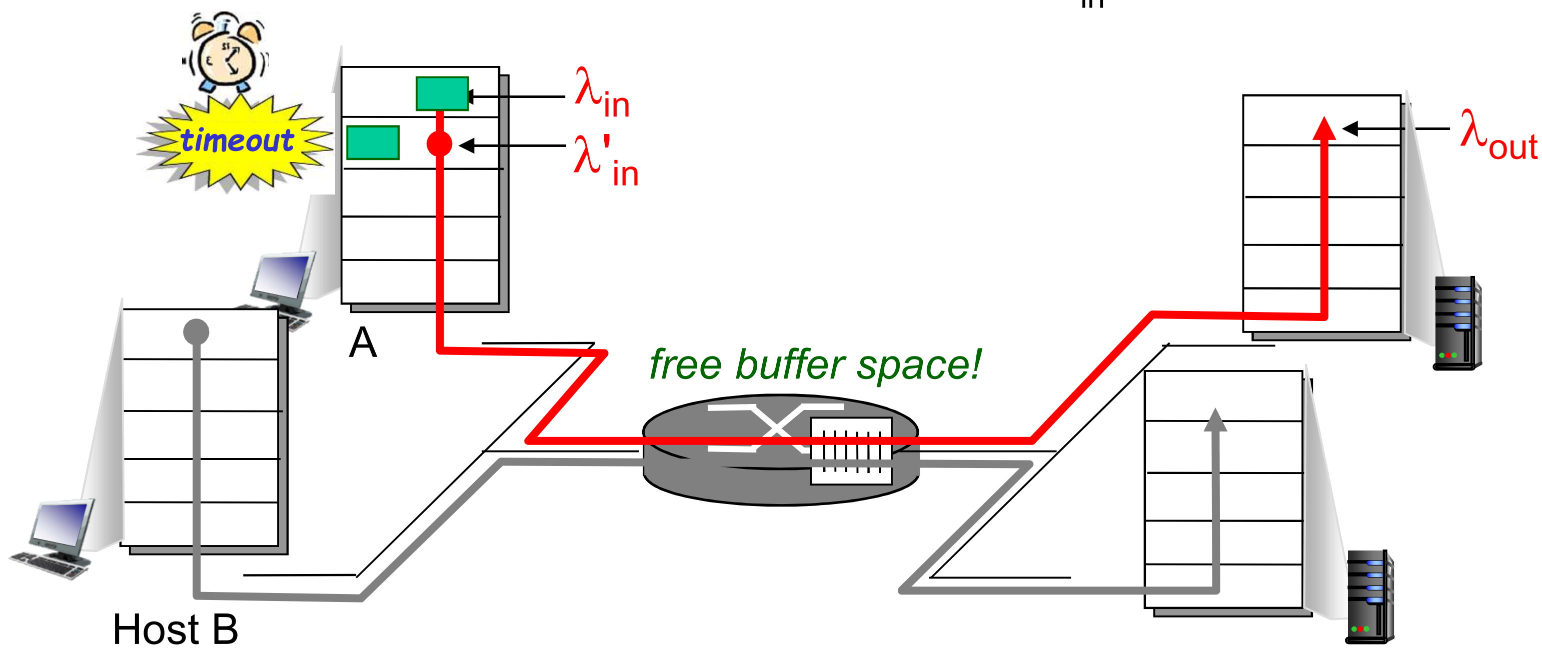


Causes/costs of congestion: scenario 2

Realistic: duplicates

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

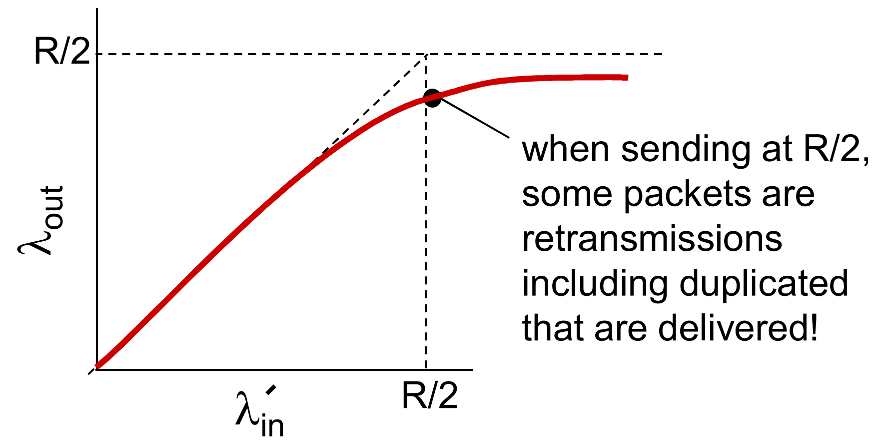




Causes/costs of congestion: scenario 2

Realistic: duplicates

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



"costs" of congestion:

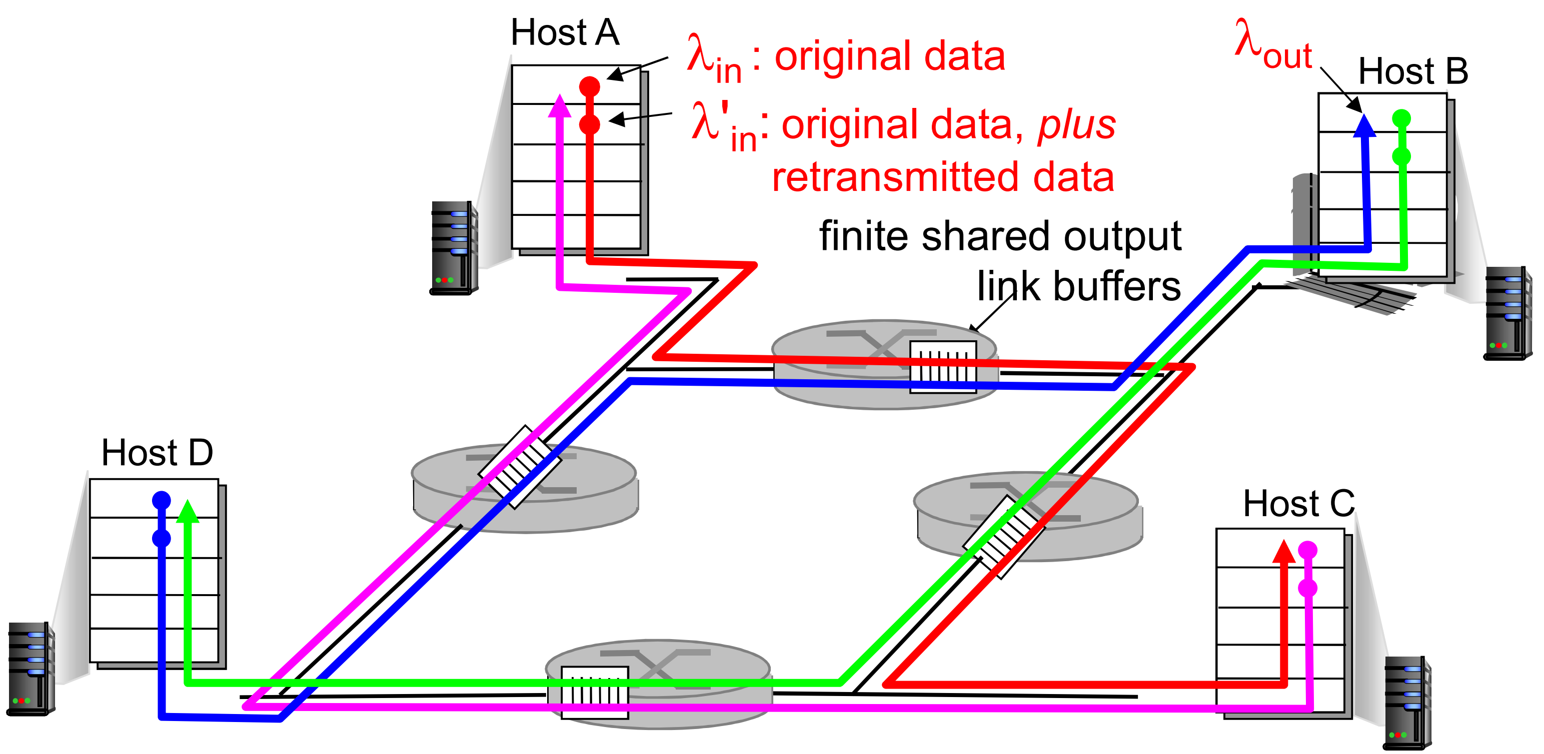
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

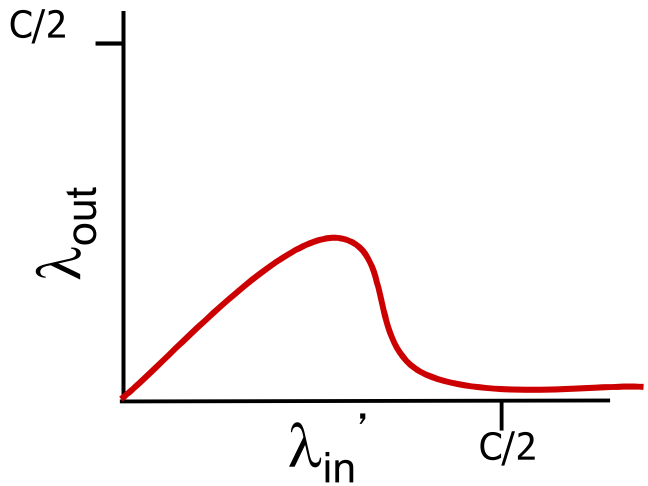
- four senders
- multihop paths
- timeout/retransmit

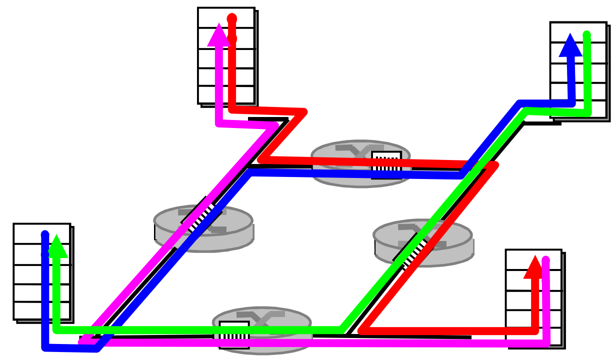
Q: what happens as λ_{in} and λ_{in}' increase ?

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 used for that packet was wasted!