EECE 2460
Introduction to Data Communication Networks
(Chapter 3)

Dr. Tricia Chigan
Tricia_Chigan@uml.edu
http://faculty.uml.edu/Tricia_Chigan

Chapter 3 - Transport layer: overview

Our goal:

- 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

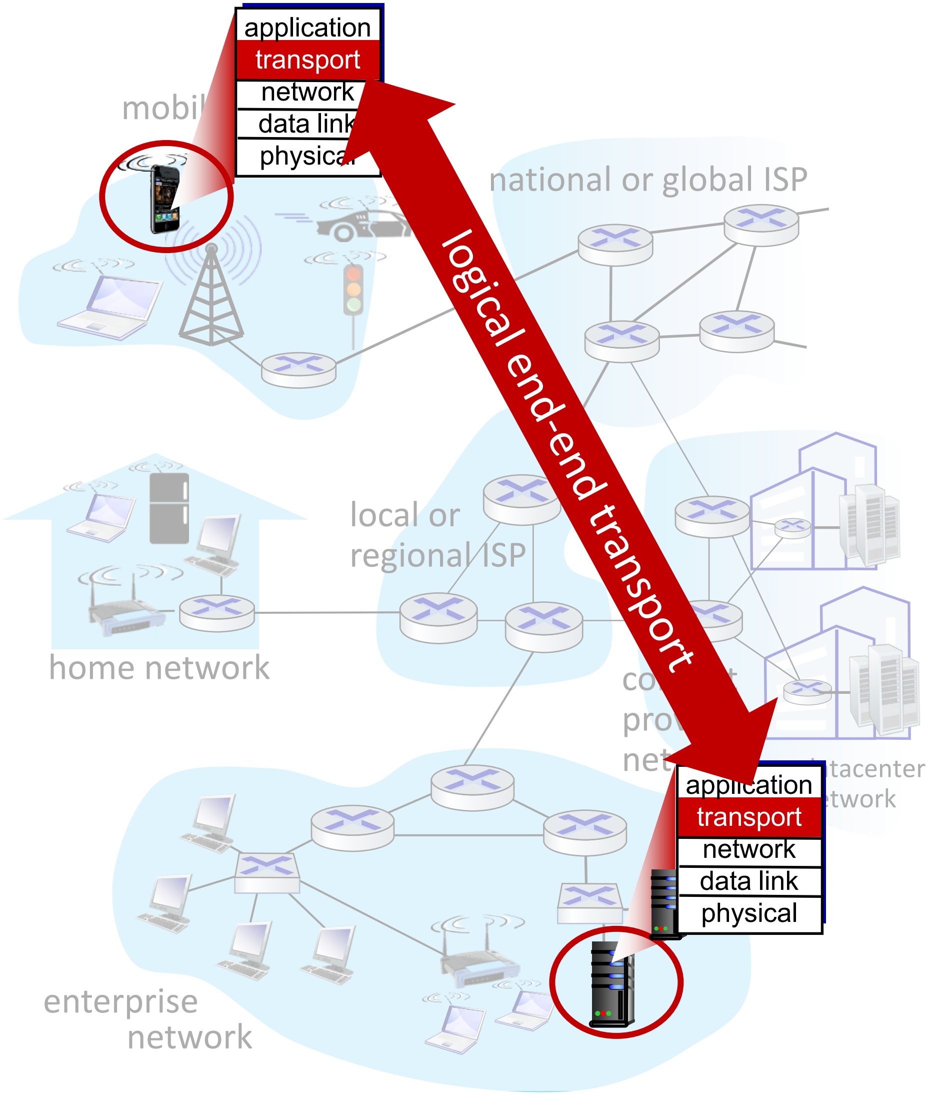
Lecture slides modified using textbook authors' version.

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

- provide ***logical communication*** between application processes running on different **hosts**
- transport protocols actions in **end systems**:
  - **sender**: breaks application messages into ***segments***, passes to network layer
  - **receiver**: reassembles segments into messages, passes to application layer
- two **transport** protocols available to Internet applications
  - TCP, UDP



Transport vs. network layer services and protocols

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 vs. network layer services and protocols

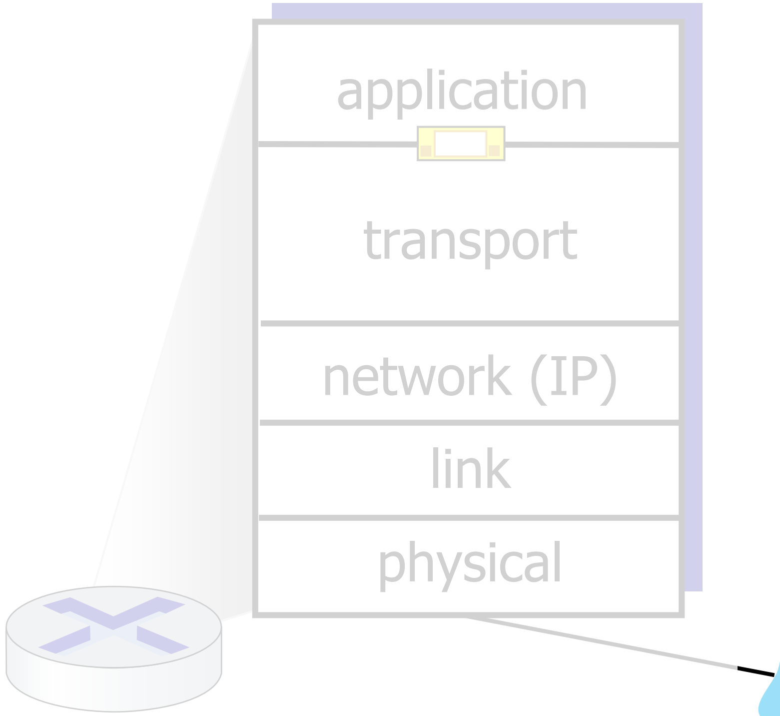
- network layer: logical communication between ***hosts***
- transport layer*:* logical communication between ***processes***
  - relies on, enhances, network layer services

household analogy:

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

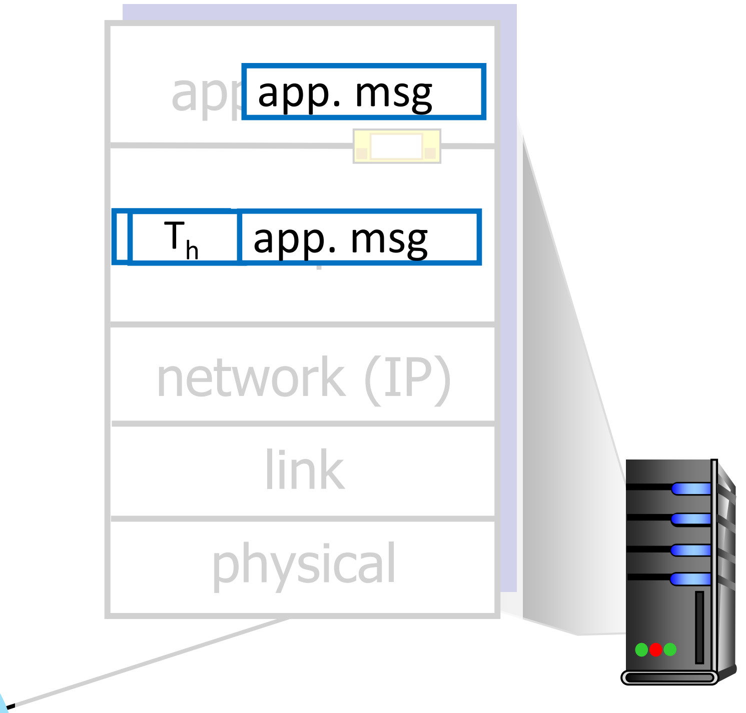
- hosts = houses
- processes = kids
- app messages = letters in envelopes

Transport Layer Actions

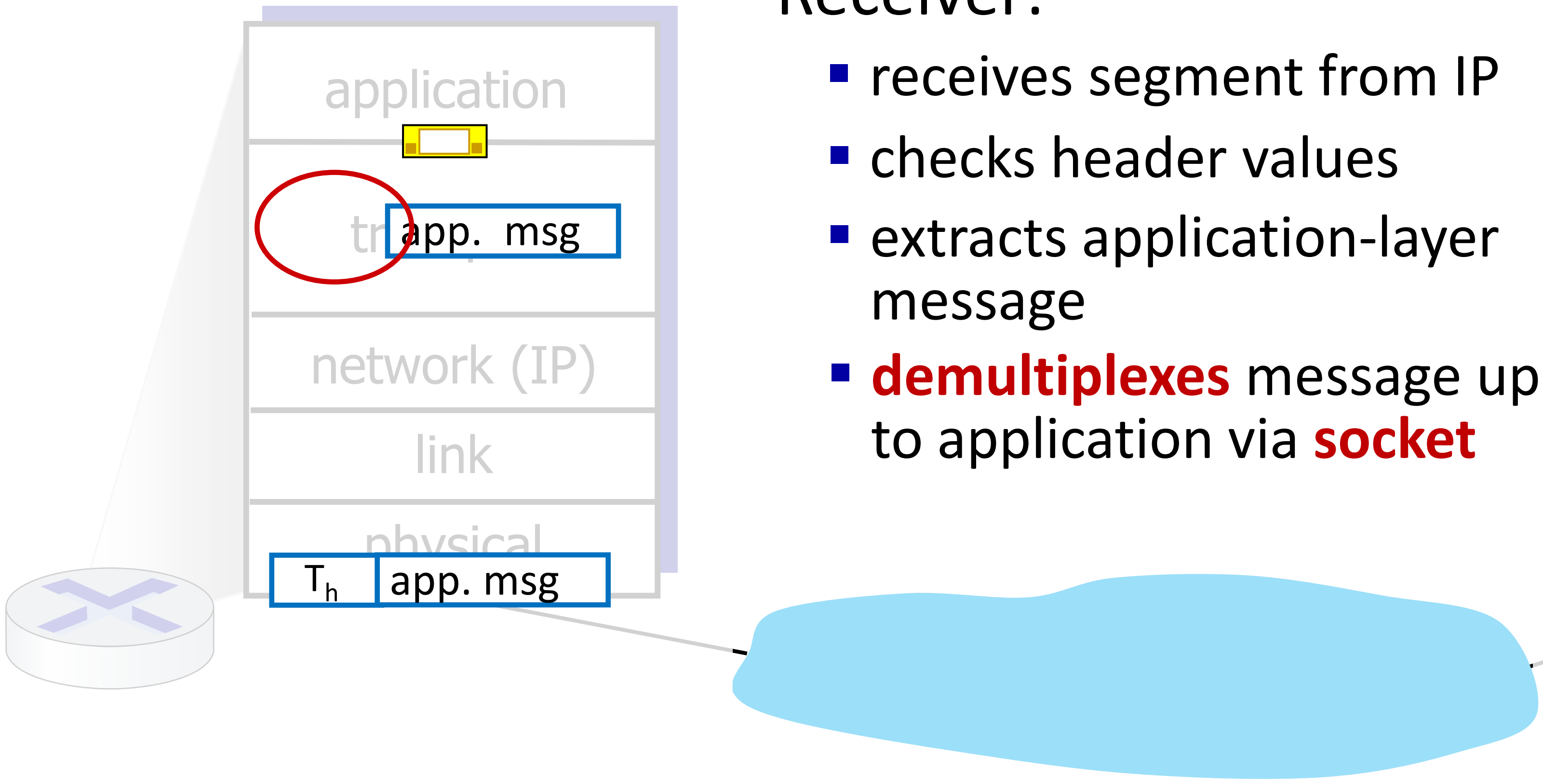


Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

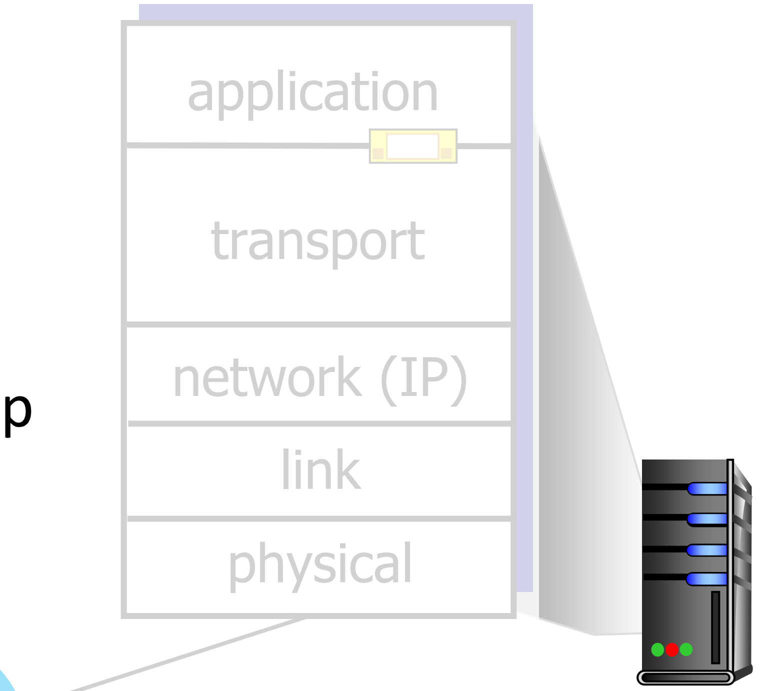


Transport Layer Actions



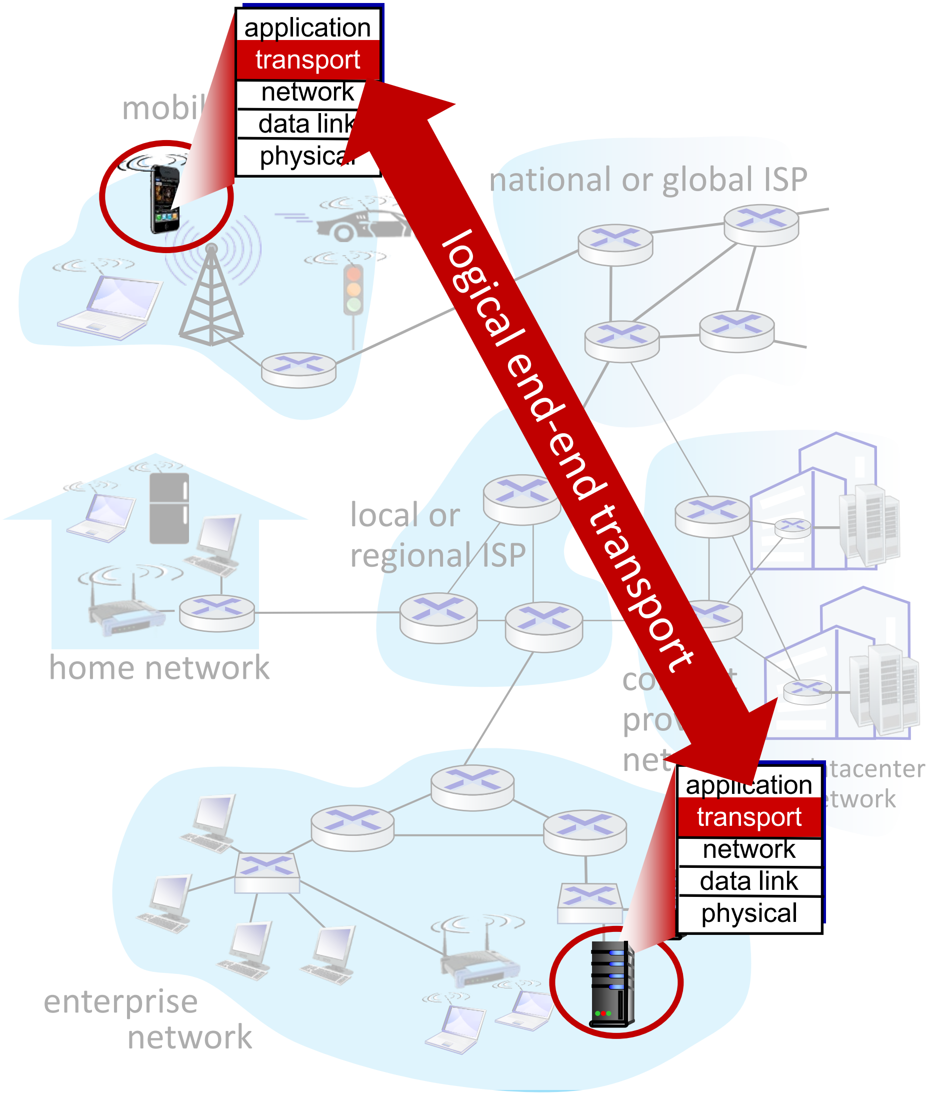
Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- **demultiplexes** message up to application via **socket**



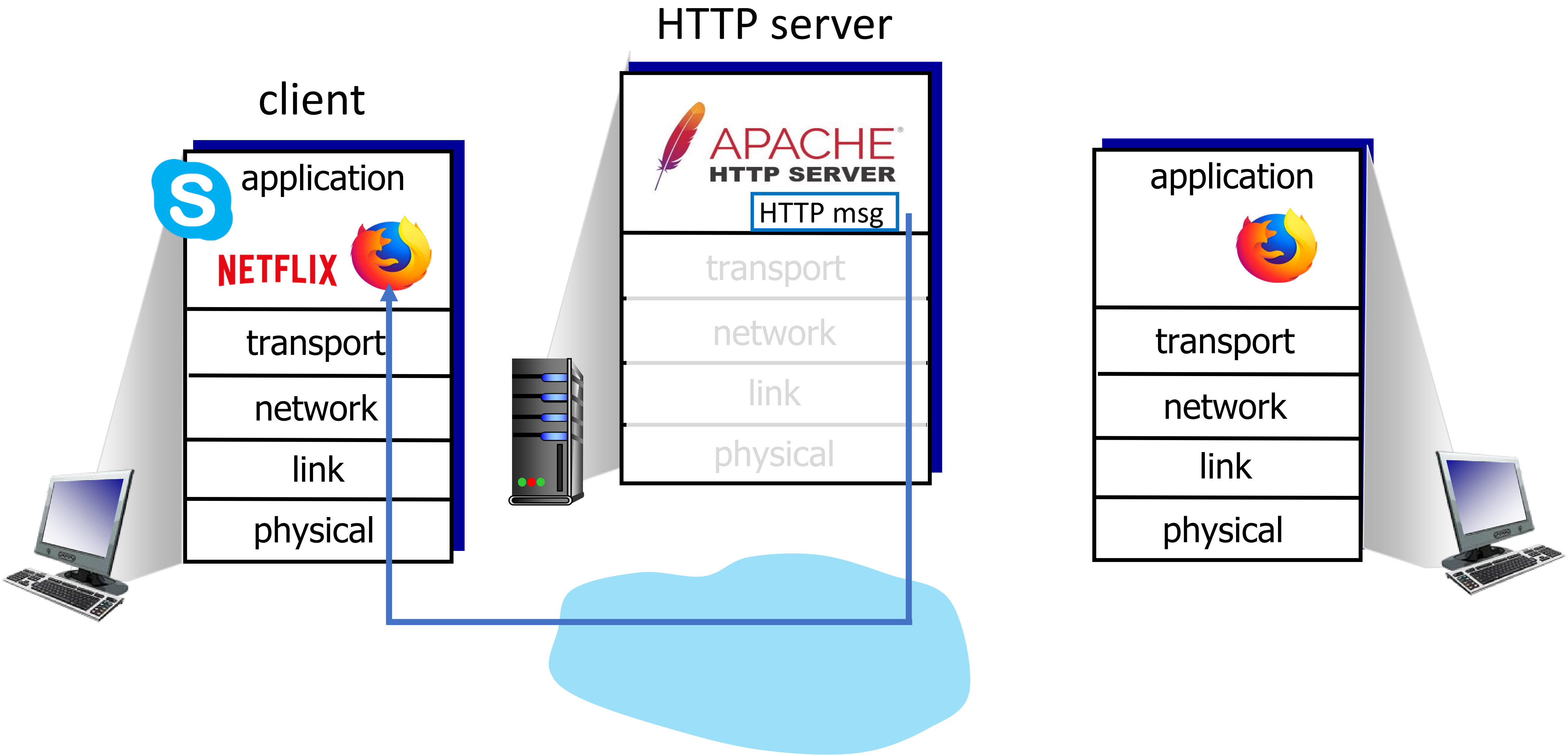
Two principal Internet transport protocols

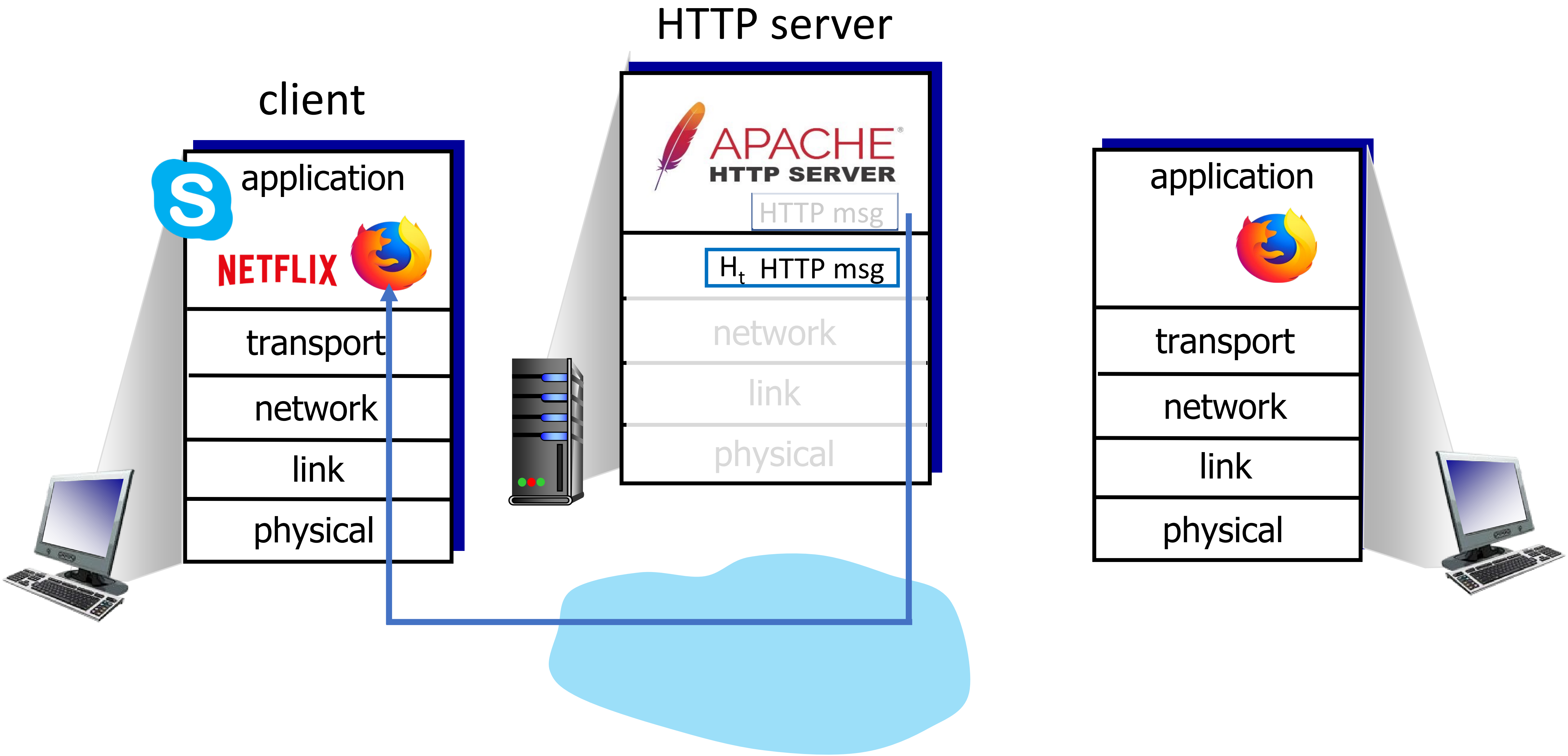
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

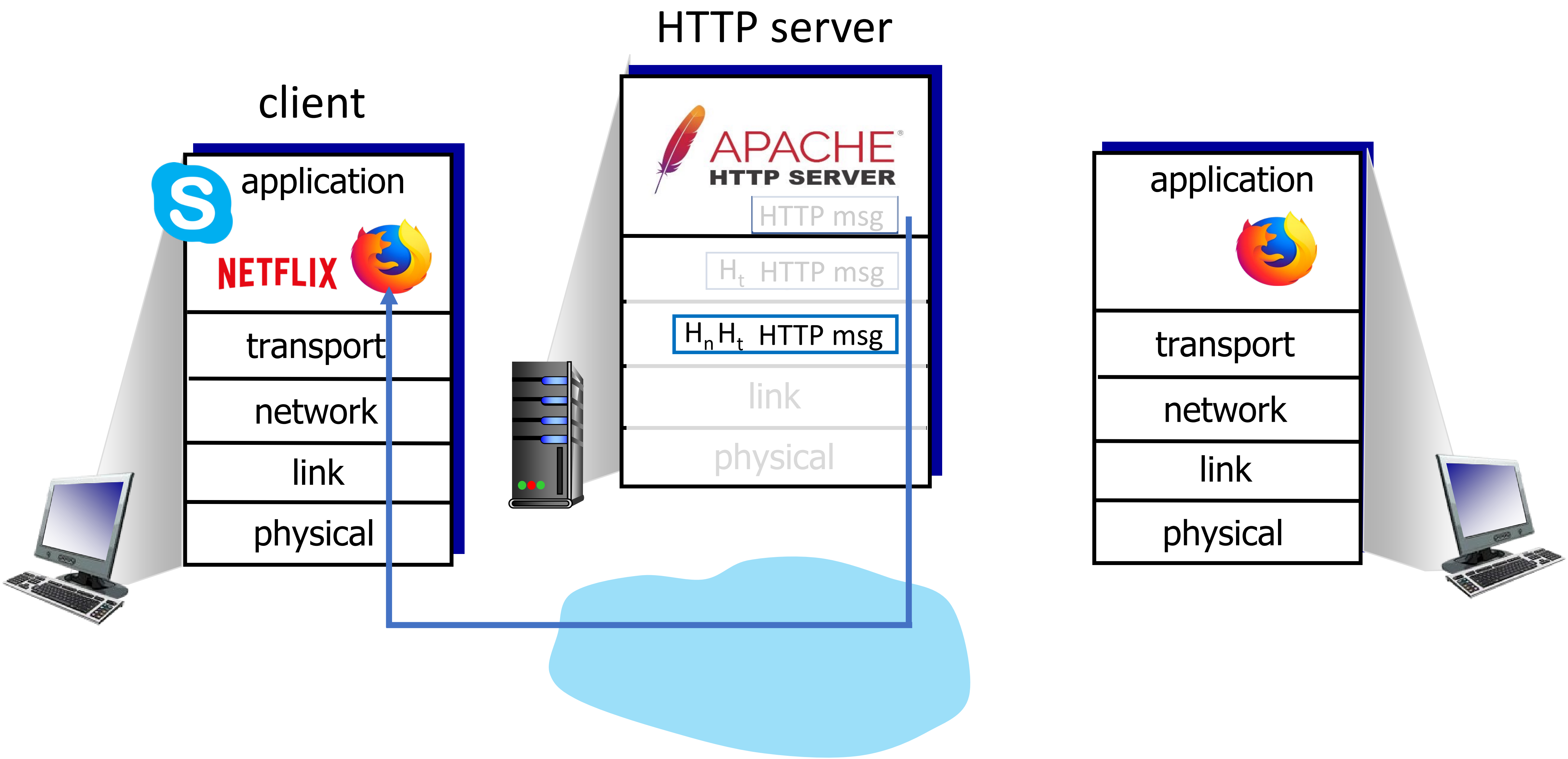


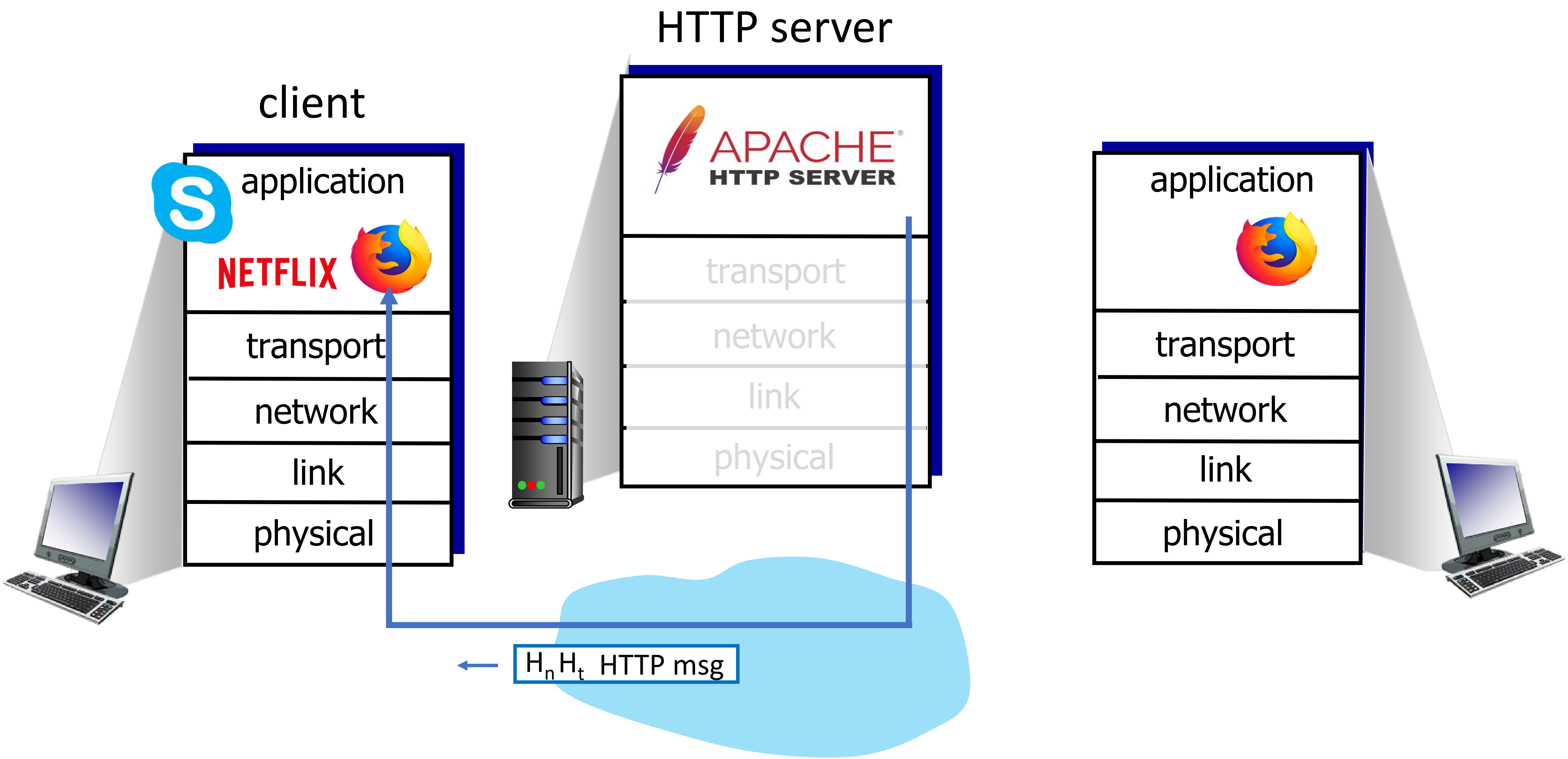
Chapter 3: roadmap

- 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

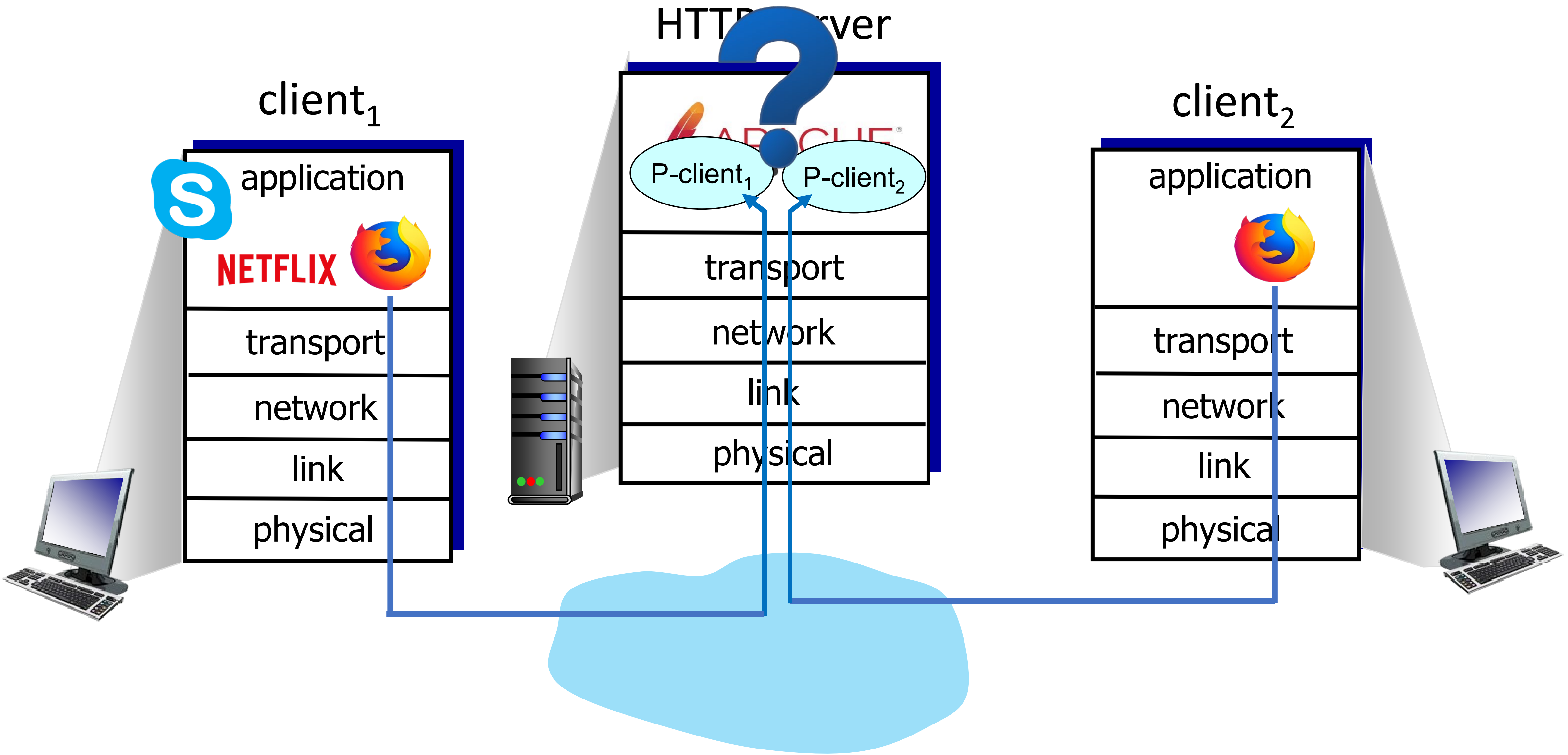

HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
transport
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
$H_n H_t$ HTTP msg
link
physical
application
transport
network
link
physical



HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
transport
network
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg


HTTP server
client$_1$
client$_2$
application
NETFLIX
transport
network
link
physical
P-client$_1$
P-client$_2$
application
transport
network
link
physical
application
transport
network
link
physical

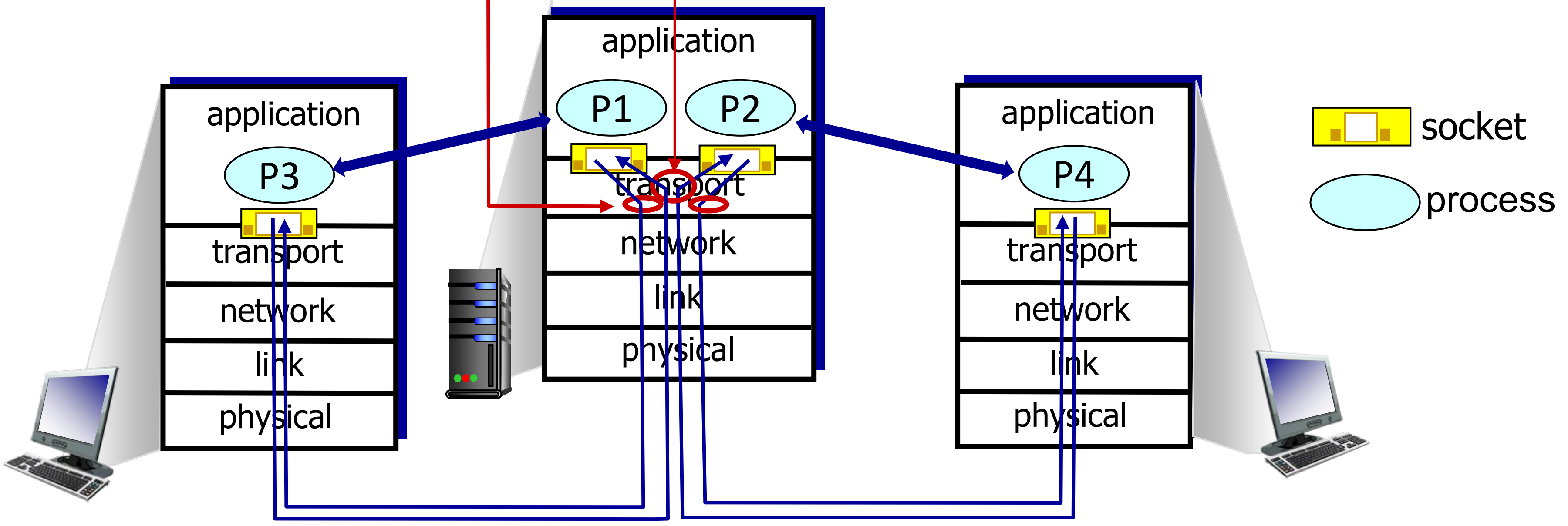
Multiplexing/demultiplexing

multiplexing at sender:

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

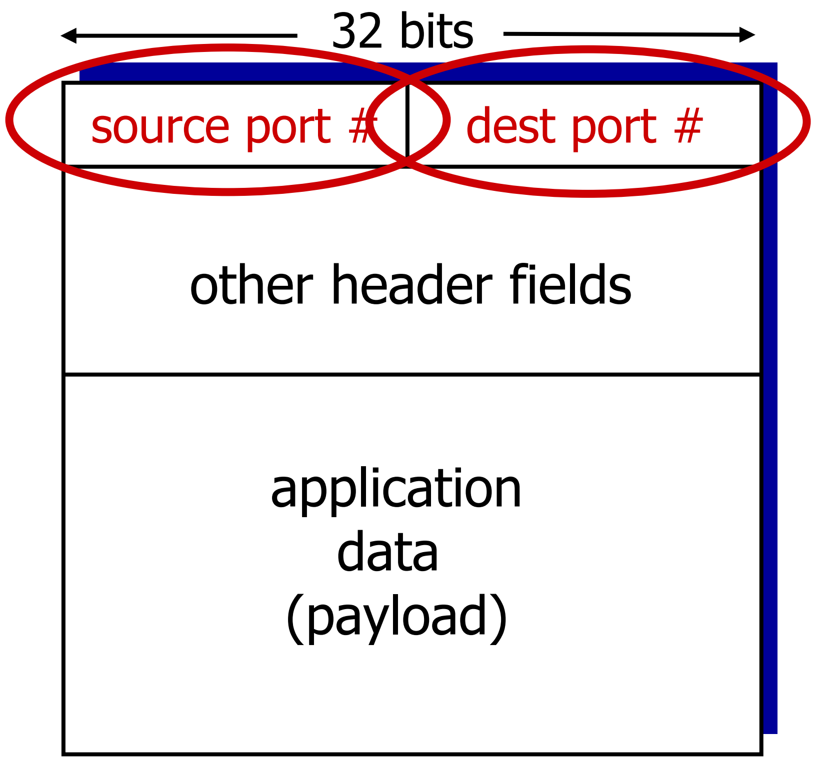
demultiplexing at receiver:

use header info to deliver received segments to **correct socket**



How demultiplexing works

- host **receives** IP **datagrams**
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer **segment**
  - each segment has source, destination **port number**
- host uses *IP addresses & port numbers* to direct segment to appropriate **socket**



TCP/UDP segment format

Connectionless demultiplexing (e.g., UDP)

Recall:

- when creating socket, must specify *host-local* port #:

```
DatagramSocket mySocket1
 = new DatagramSocket(12534);
```

- when creating **datagram** to send into UDP socket, must specify
  - destination IP address
  - destination port #

when **receiving host** receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

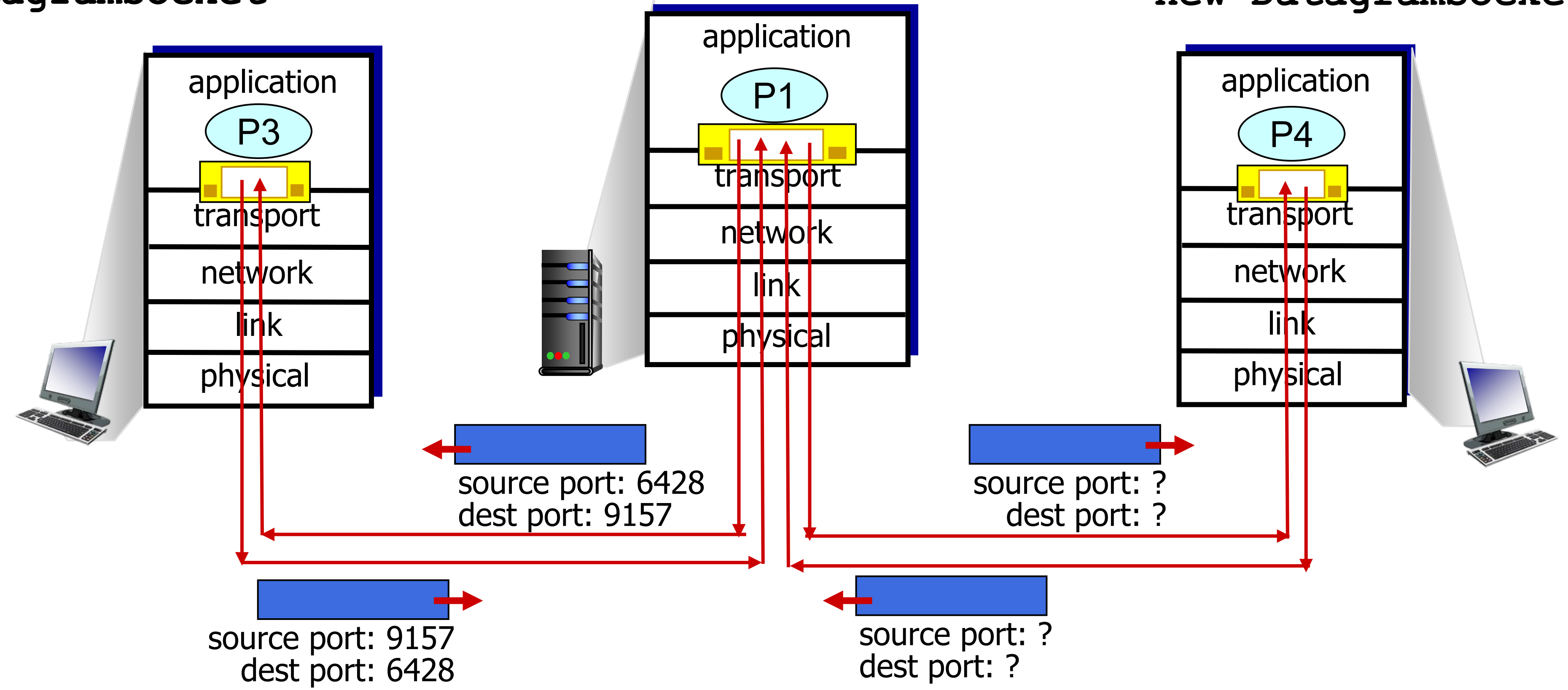
IP/UDP datagrams with ***same dest. port #,*** but different source IP addresses and/or source port numbers will be directed to ***same socket*** at **receiving host**

Connectionless demultiplexing: an example

```
DatagramSocket serverSocket = new DatagramSocket (6428);
```

```
DatagramSocket mySocket2 = new DatagramSocket (9157);
```

```
DatagramSocket mySocket1 = new DatagramSocket (5775);
```

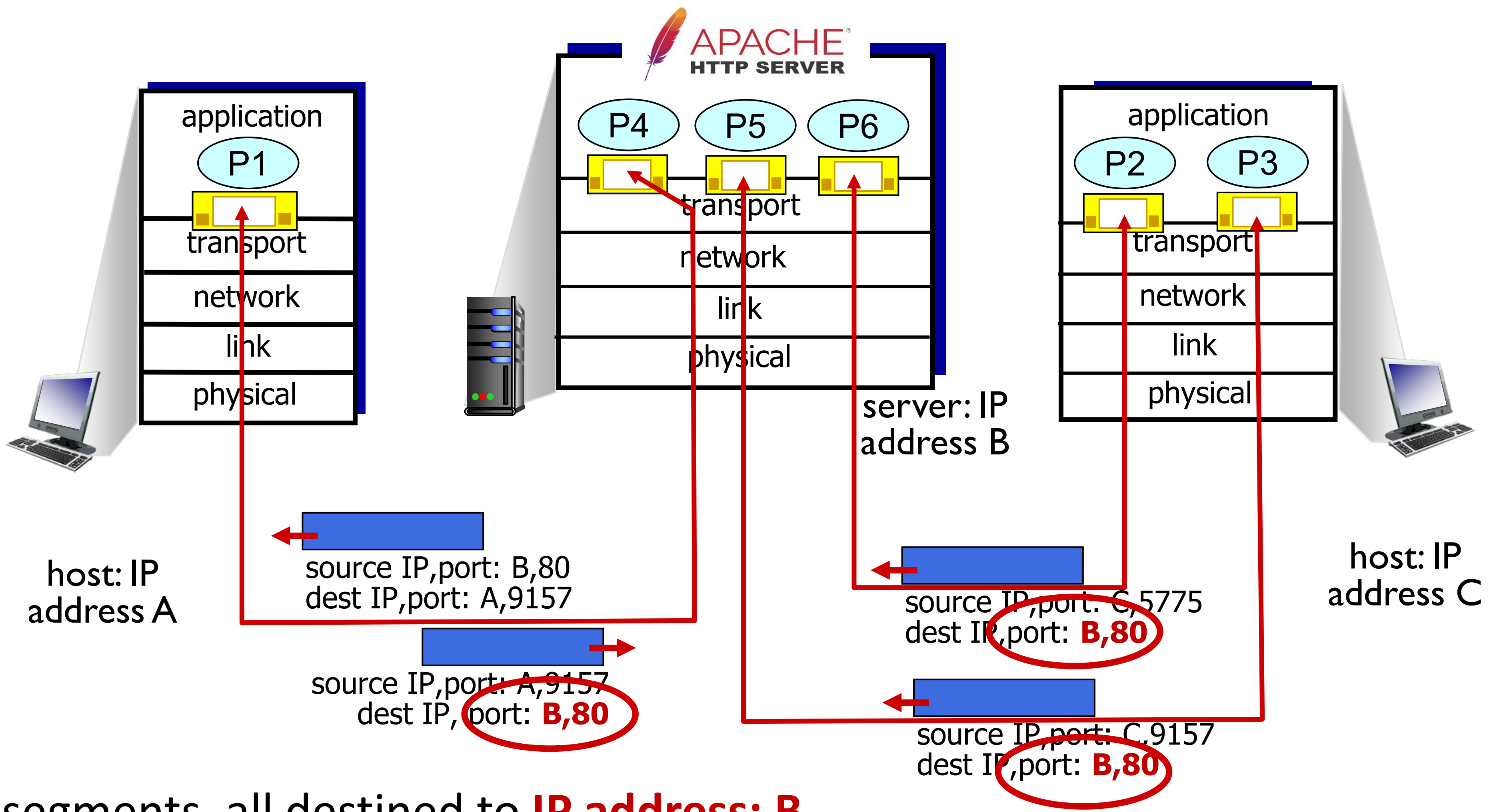


Connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

- server may support **many** simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a **different connecting client**

Connection-oriented demultiplexing: example



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

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using **2-tuple**: *destination IP and port numbers*
- **TCP:** demultiplexing using **4-tuple**: *source and destination IP addresses, and port numbers*
- Multiplexing/demultiplexing happen at ***all* layers**

Chapter 3: 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

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, 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 use:
  - 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

https://zoom.us/docs/doc/Zoom%20Connection%20Process%20Whitepaper.pdf

"Each of these media connections attempt to use **Zoom's** own protocol and connect **via UDP on port 8801**. If that connection can not be established, Zoom will also try **connecting using TCP on port 8801**, followed by SSL (port 443). "

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768 J. Postel
ISI
28 August 1980

User Datagram Protocol

Introduction

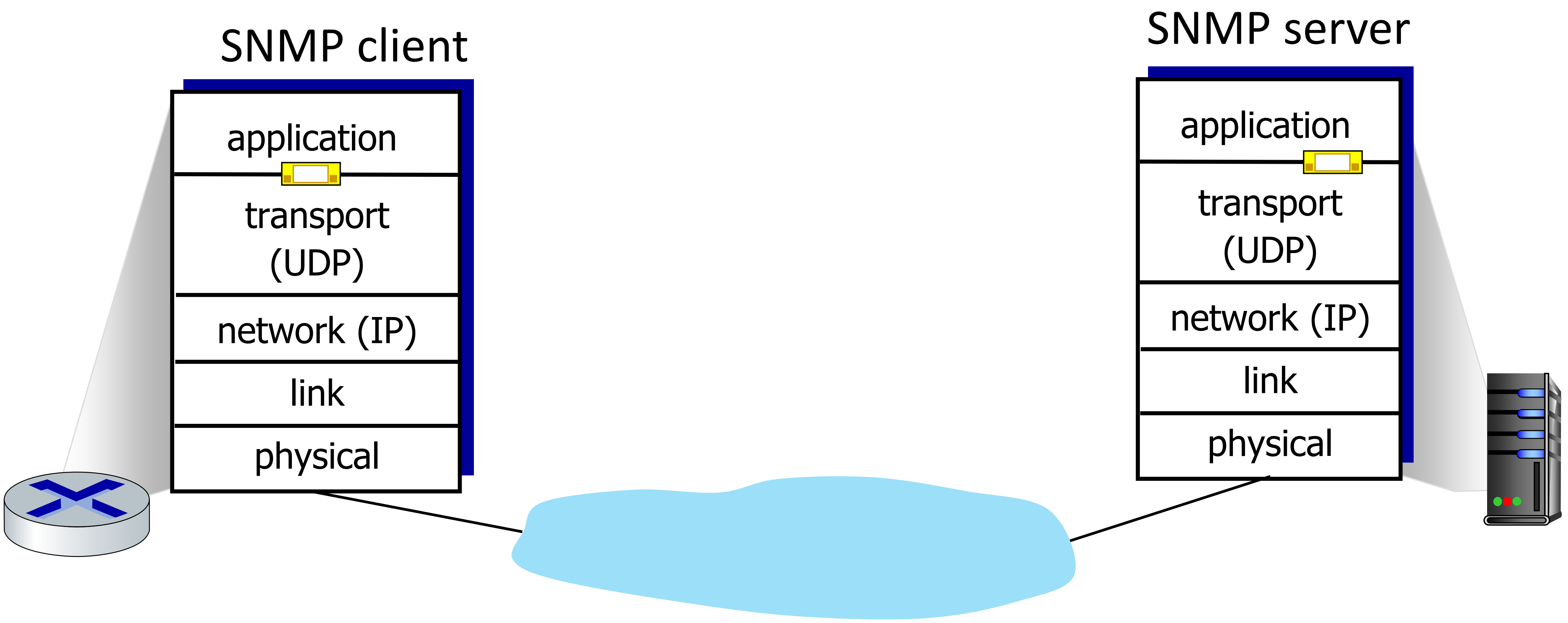
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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

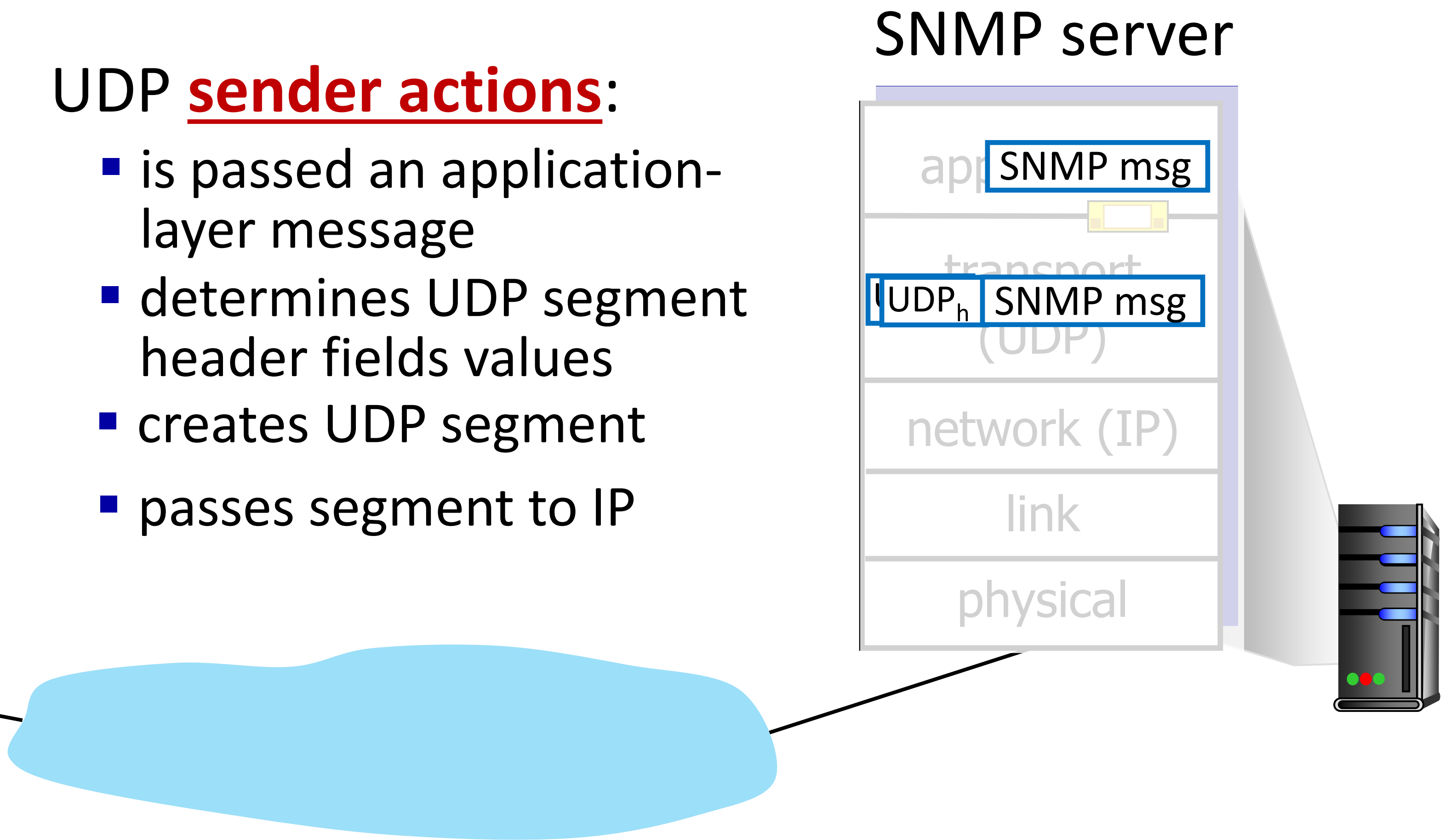
```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+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

UDP: Transport Layer Actions

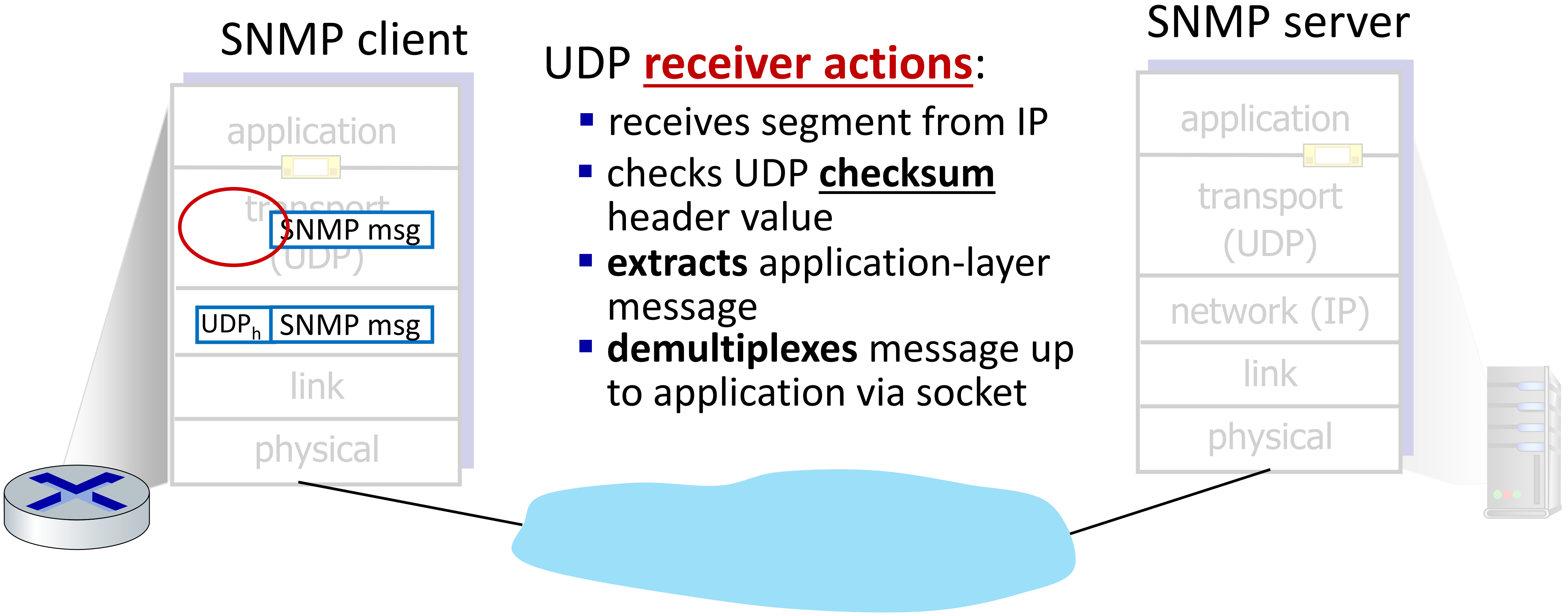


UDP: Transport Layer Actions

UDP **sender actions**:
- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP



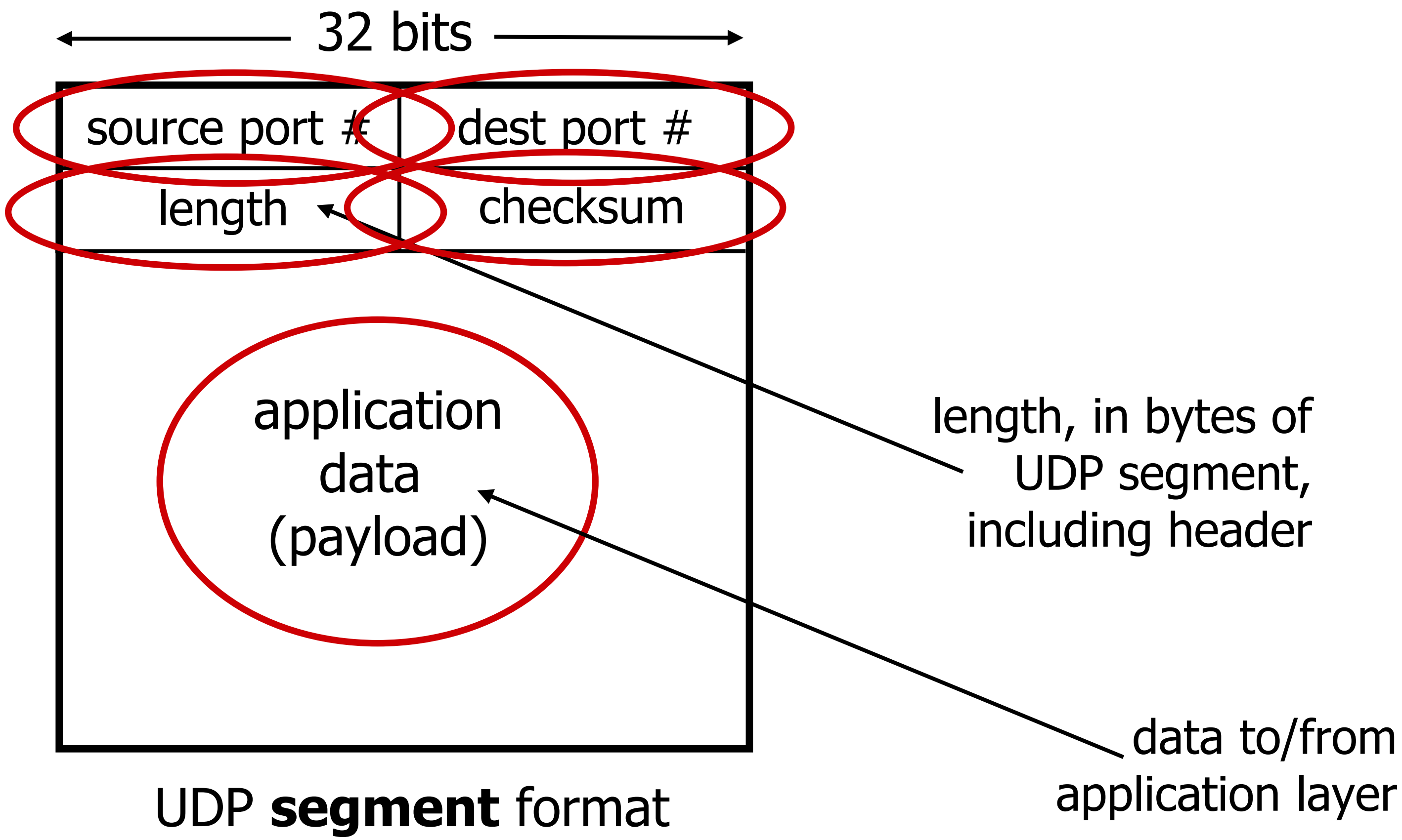
UDP: Transport Layer Actions



UDP **receiver actions**:

- receives segment from IP
- checks UDP **checksum** header value
- **extracts** application-layer message
- **demultiplexes** message up to application via socket

UDP segment header



UDP **segment** format

UDP checksum

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

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

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

Internet checksum

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

sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as **<u>sequence of 16-bit integers</u>**
- checksum: addition (**one's complement sum**) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - not equal - error detected
  - equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an 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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Internet checksum: weak protection!

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

0 1
1 0

Even though numbers have changed (bit flips), *no* change in checksum!

Summary: UDP

- "no frills" protocol:
  - segments may be **lost**, delivered **out of order**
  - **best effort** service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (**<u>no RTT incurred</u>**)
  - can function when network service is compromised
  - helps with reliability (**checksum**)
- build **additional functionality** on top of UDP in application layer (e.g., HTTP/3)

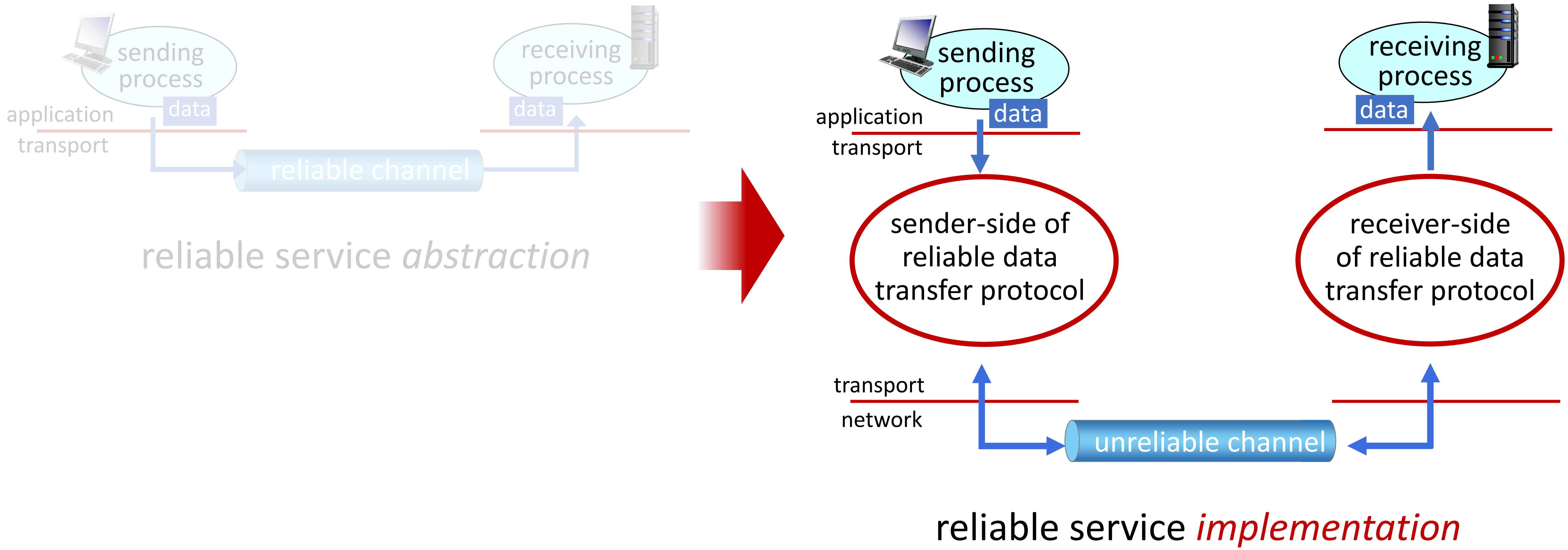
Chapter 3: roadmap

- 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

Principles of reliable data transfer

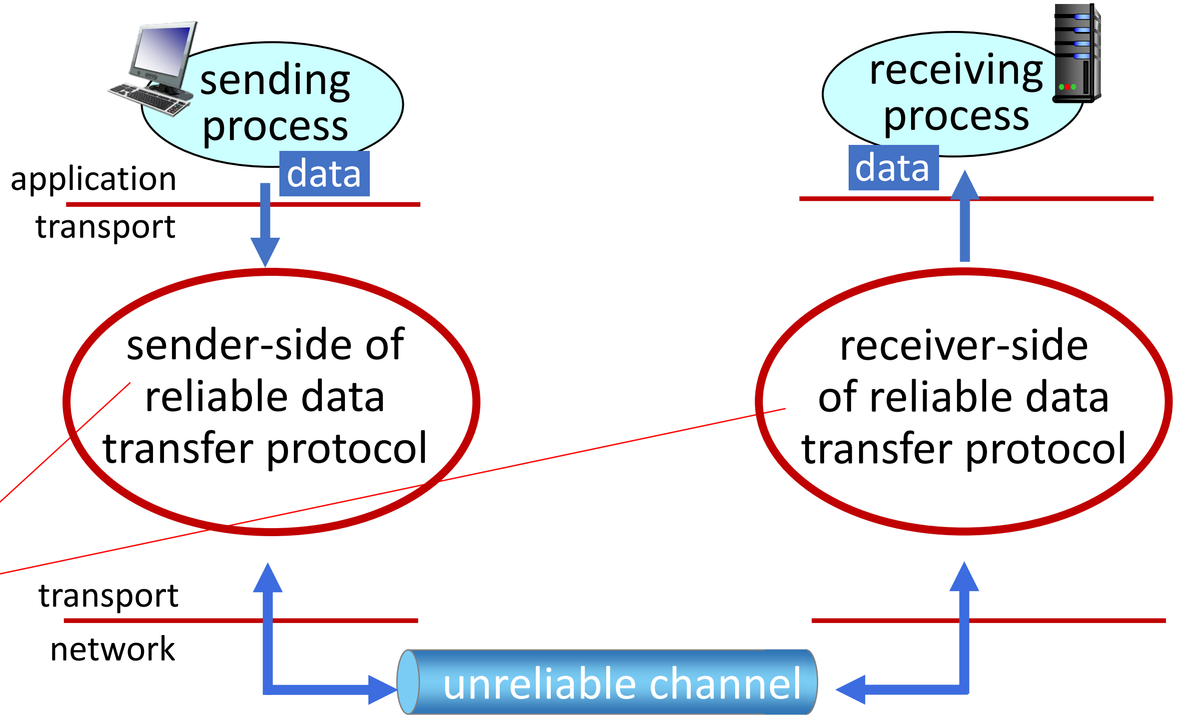
reliable service *abstraction*

Principles of reliable data transfer



Principles of reliable data transfer

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

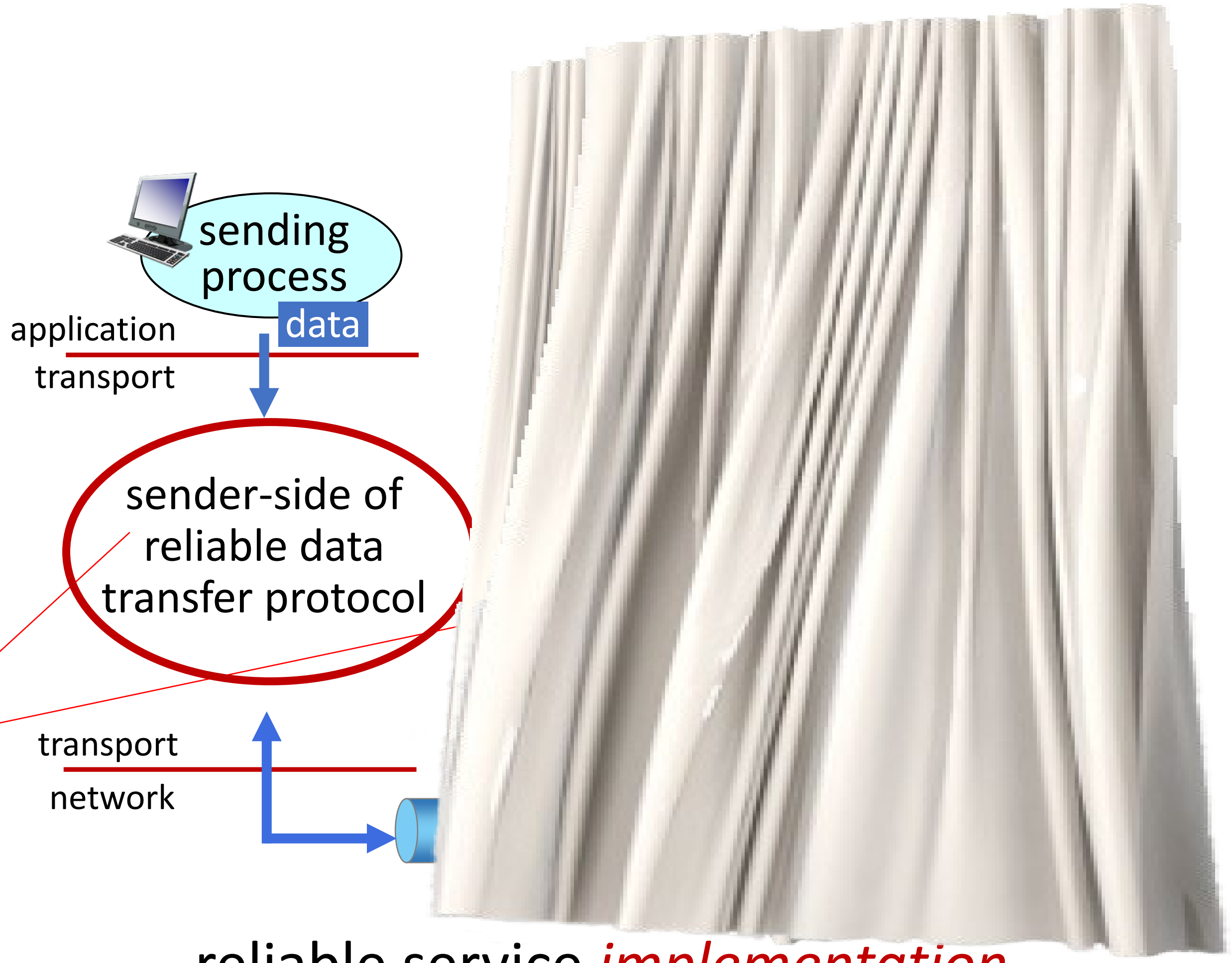


reliable service *implementation*

Principles of reliable data transfer

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

- unless communicated via a message

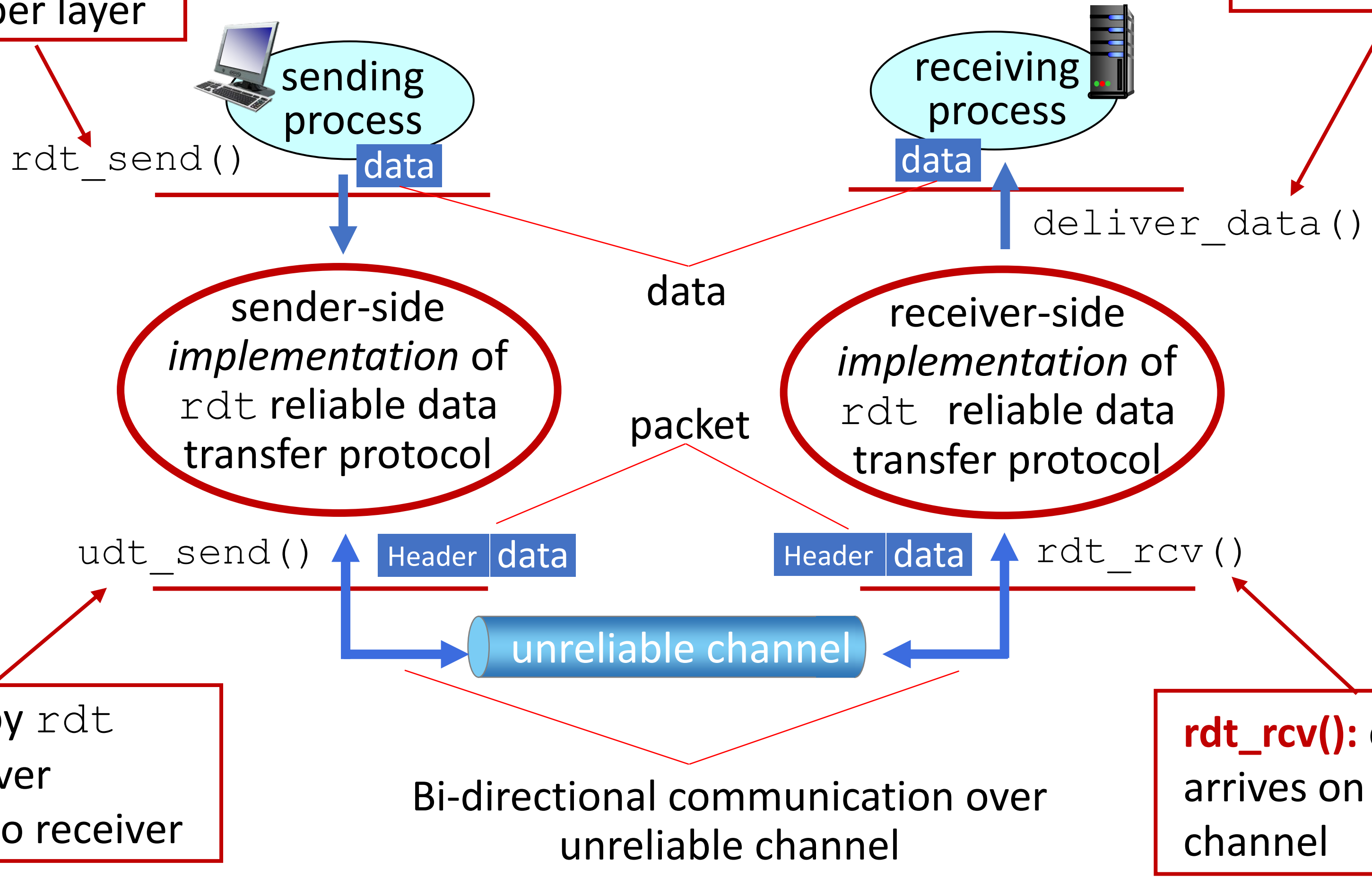


reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces

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 layer



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

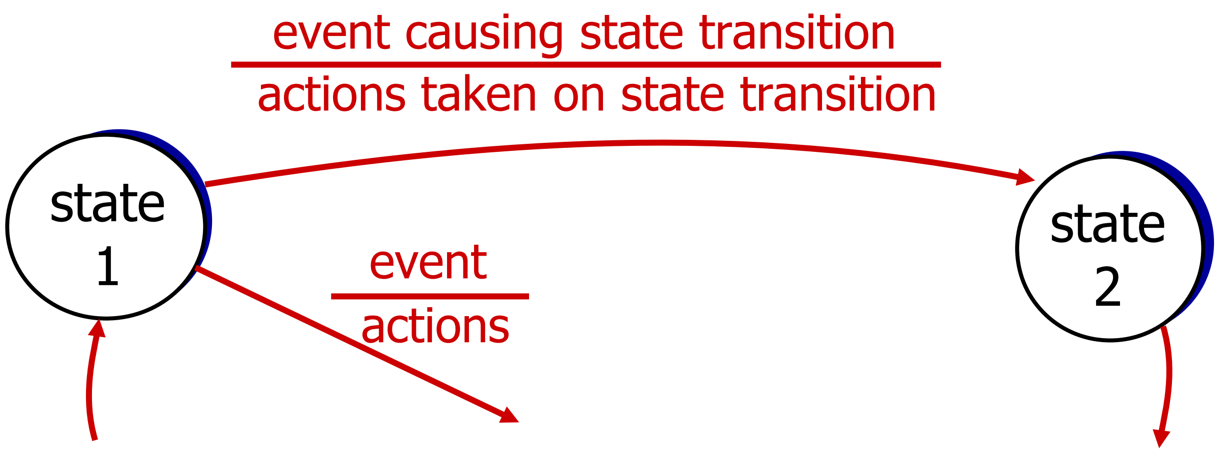
rdt_rcv(): called when packet arrives on receiver side of channel

Reliable data transfer: getting started

We will:

- **incrementally develop** sender, 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, receiver

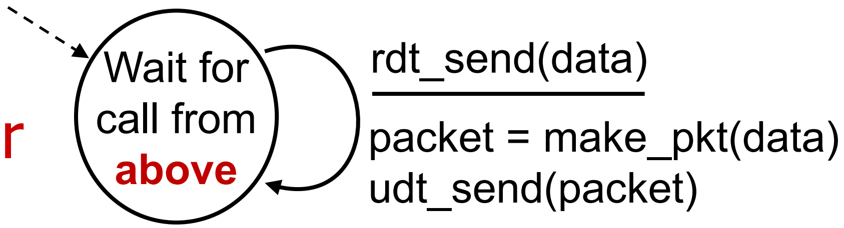
state: when in this "state" next state uniquely determined by next event



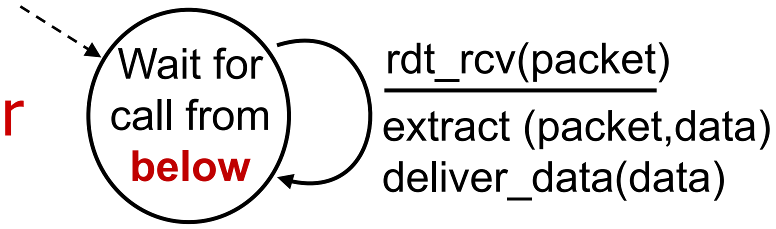
rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- *separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender



receiver



rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to ***detect*** bit errors
- *the* question: how to ***recover*** from errors?

How do humans recover from "errors" during conversation?

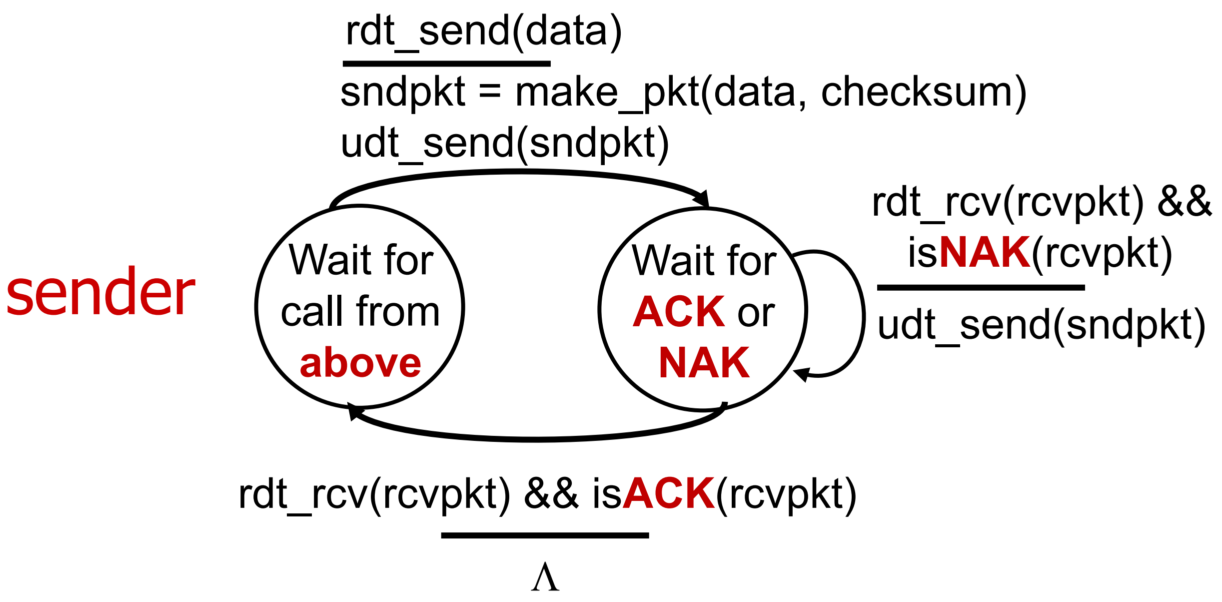
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: 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

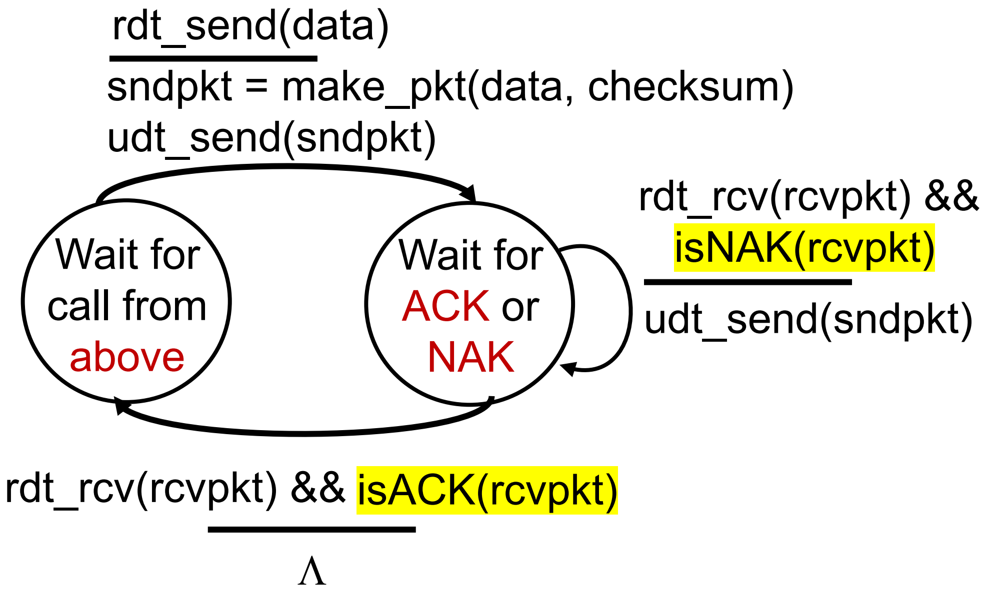
sender sends one packet, then waits for **receiver response**

rdt2.0: FSM specifications



rdt2.0: FSM specification

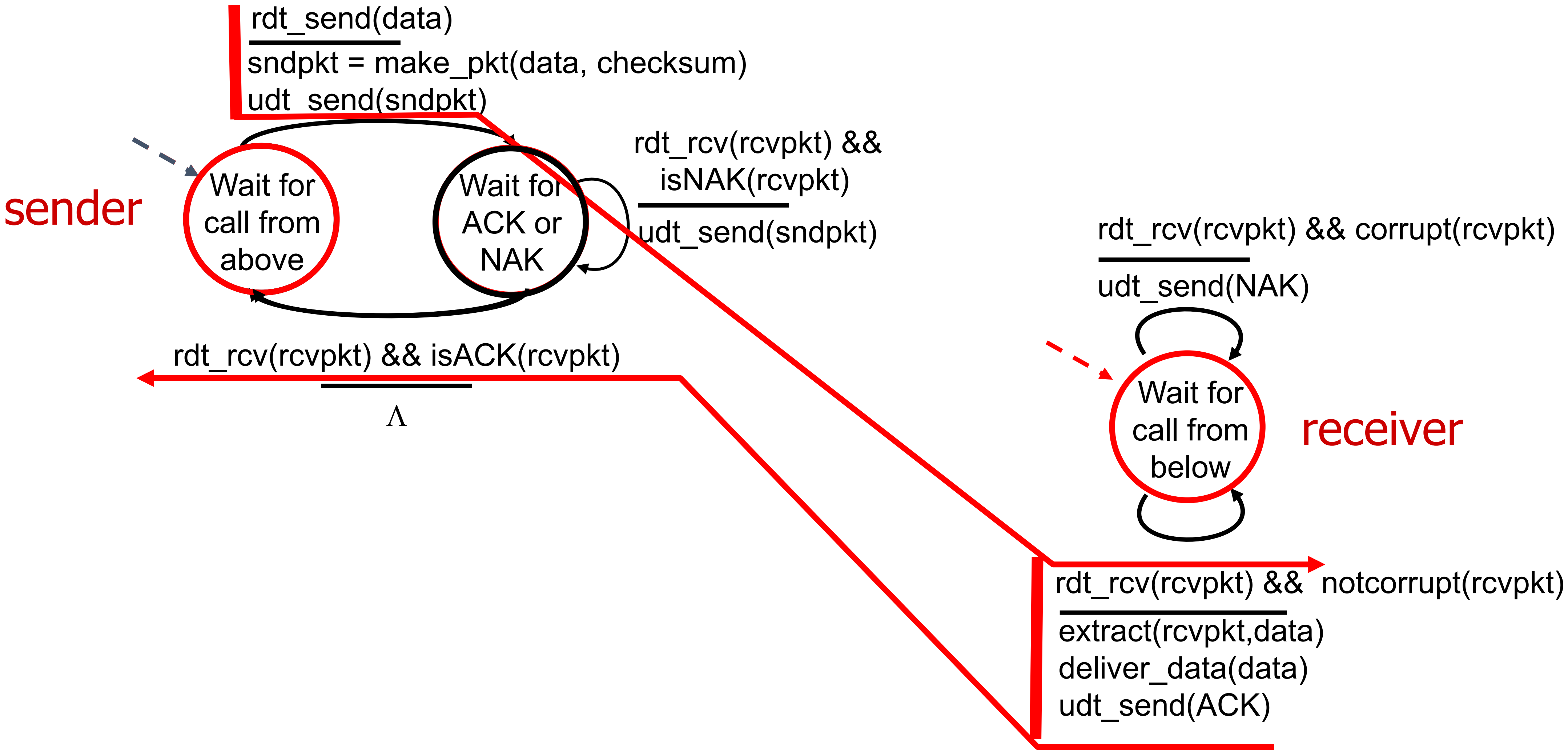
sender



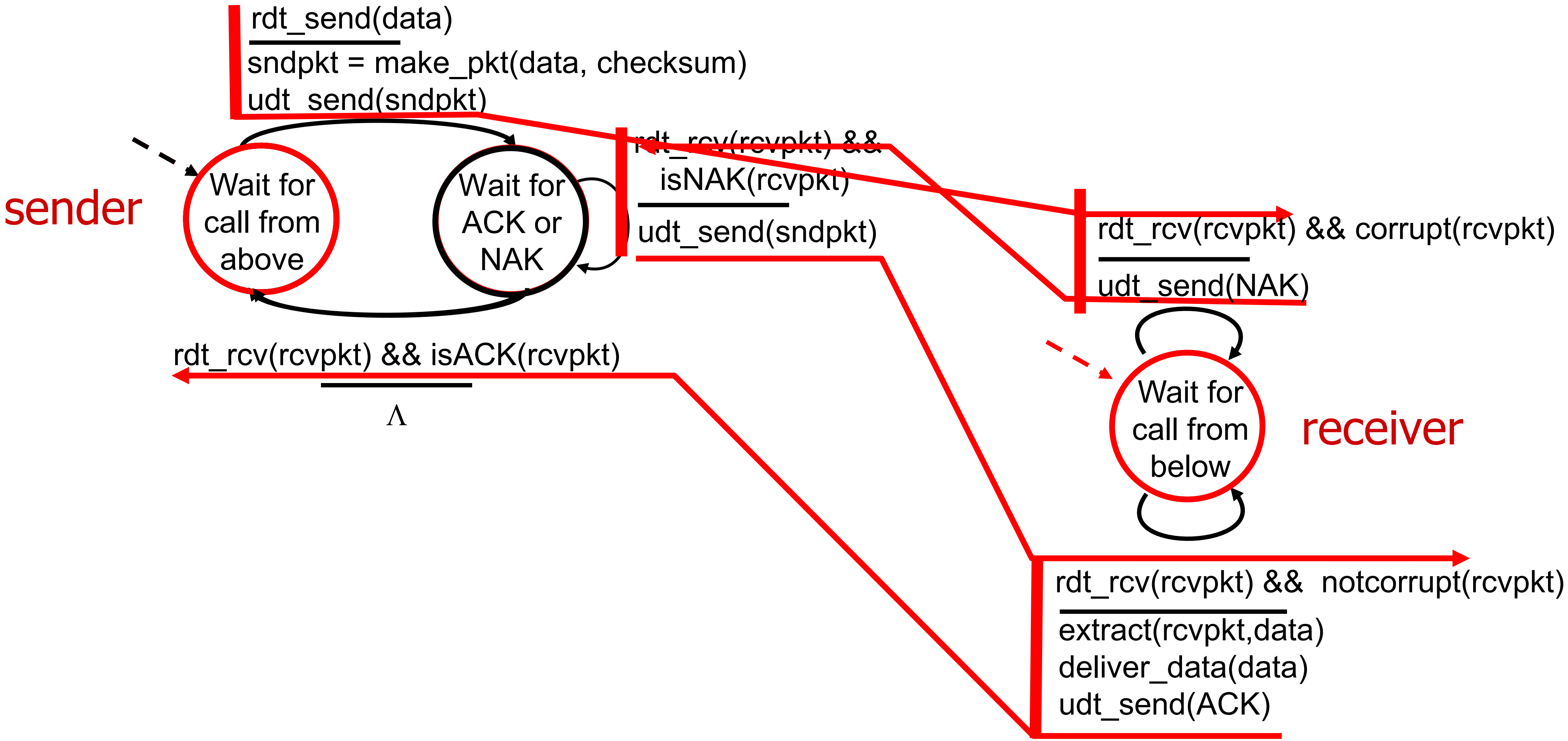
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow **communicated from receiver** to sender

- that's why we need a **protocol**!

rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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

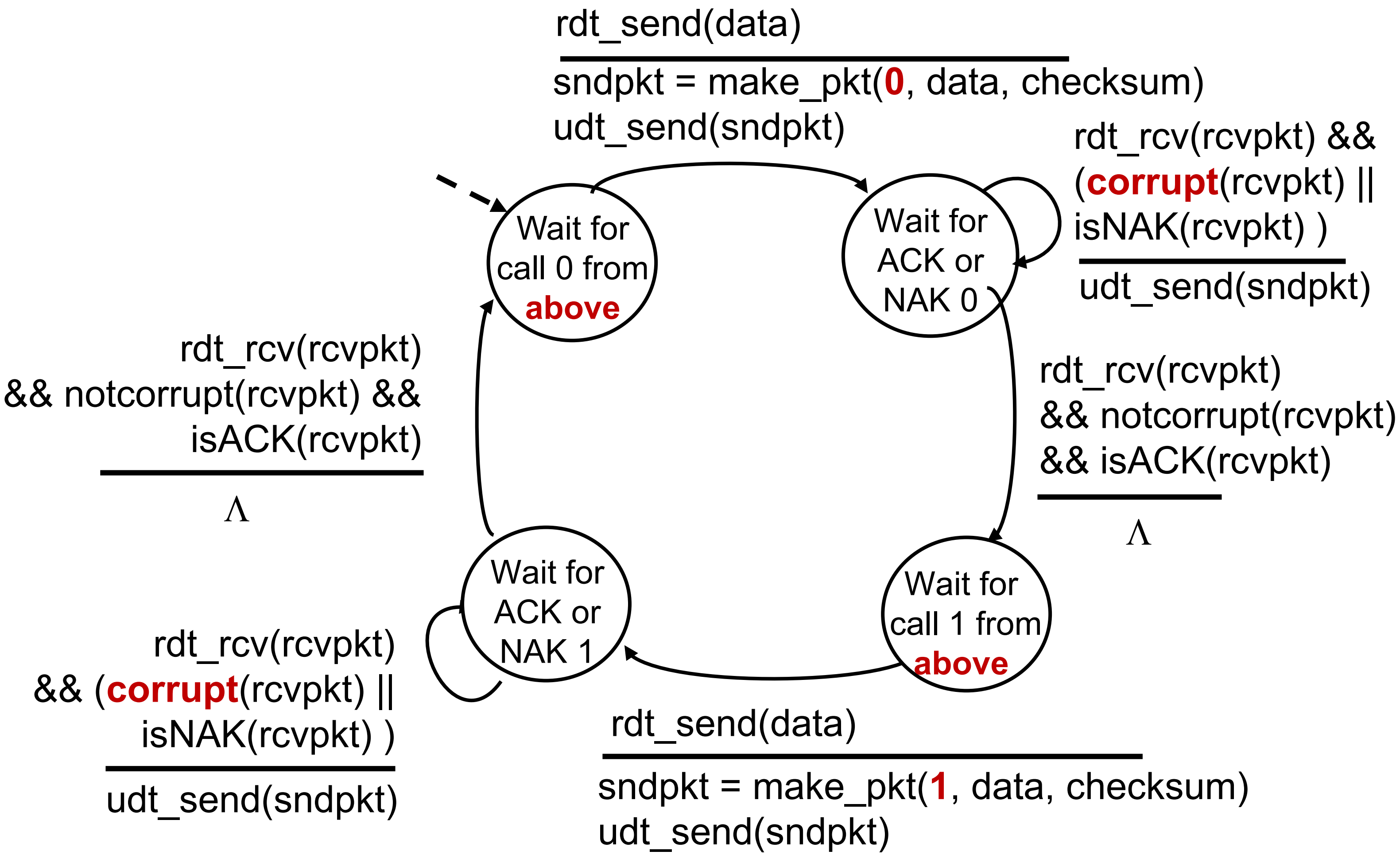
handling duplicates:

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

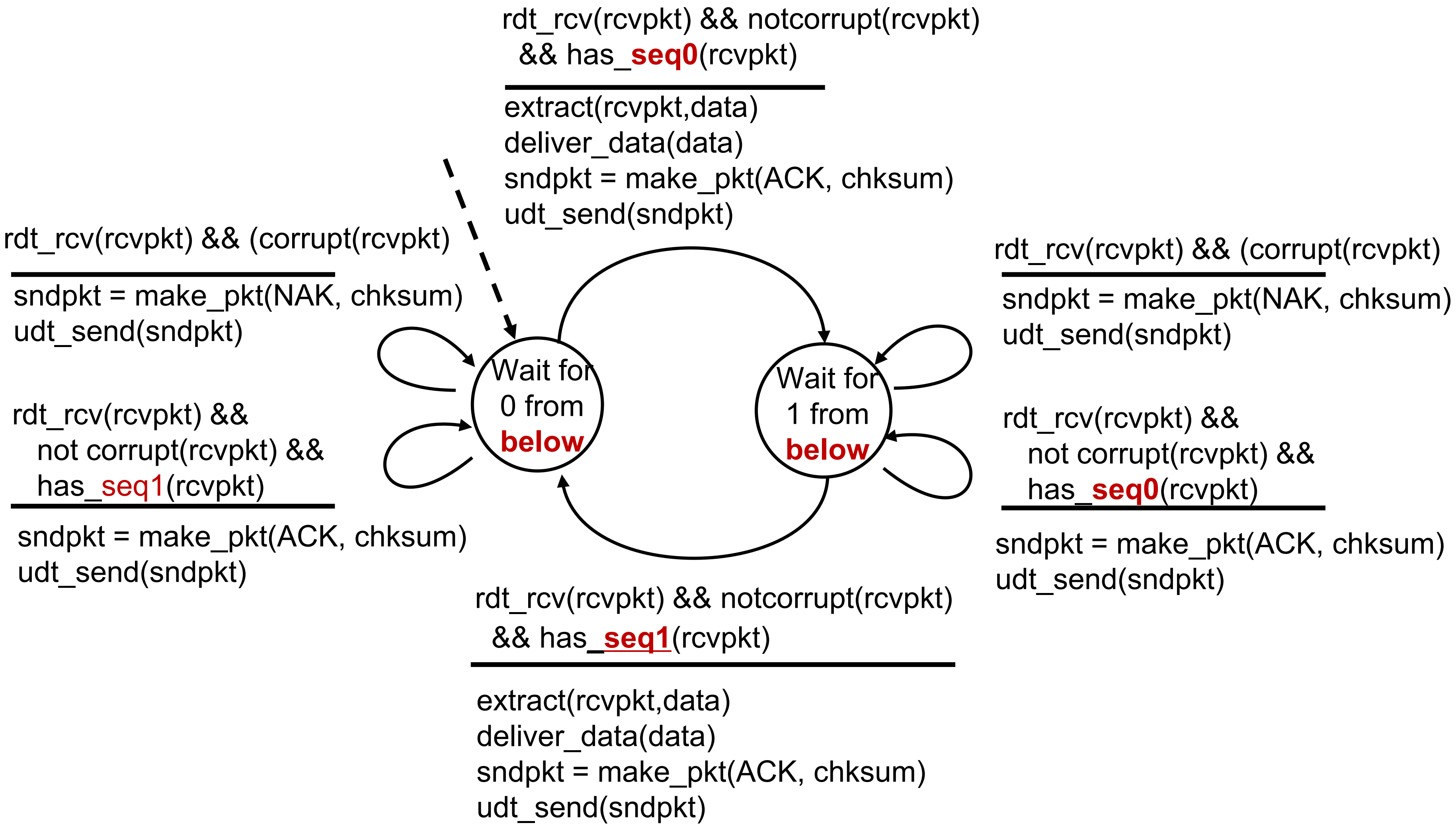
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling garbled ACK/NAKs



rdt2.1: 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:

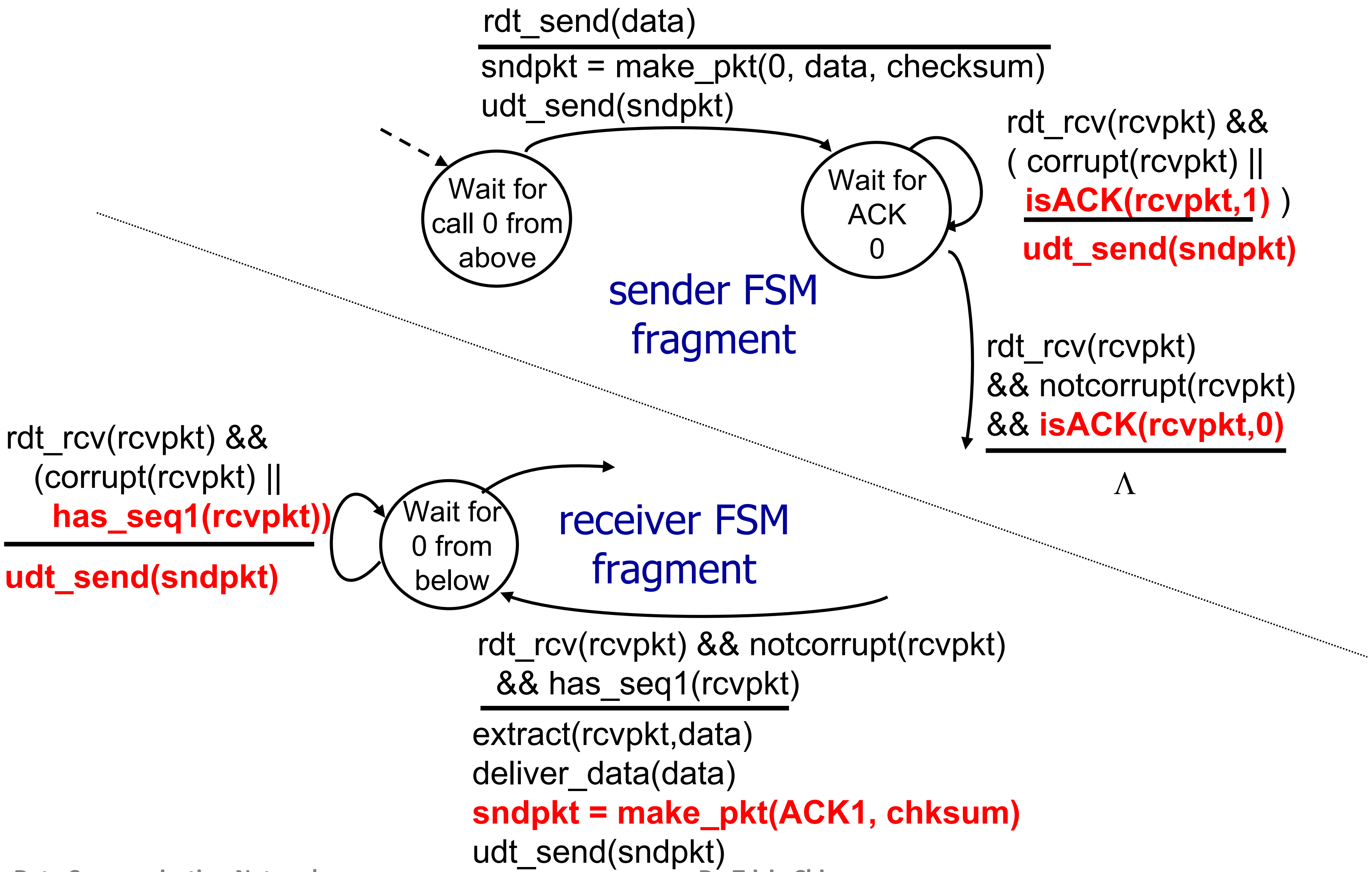
- must check if received packet is **duplicate**
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can not know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - 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

rdt2.2: sender, receiver fragments



rdt3.0: 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?

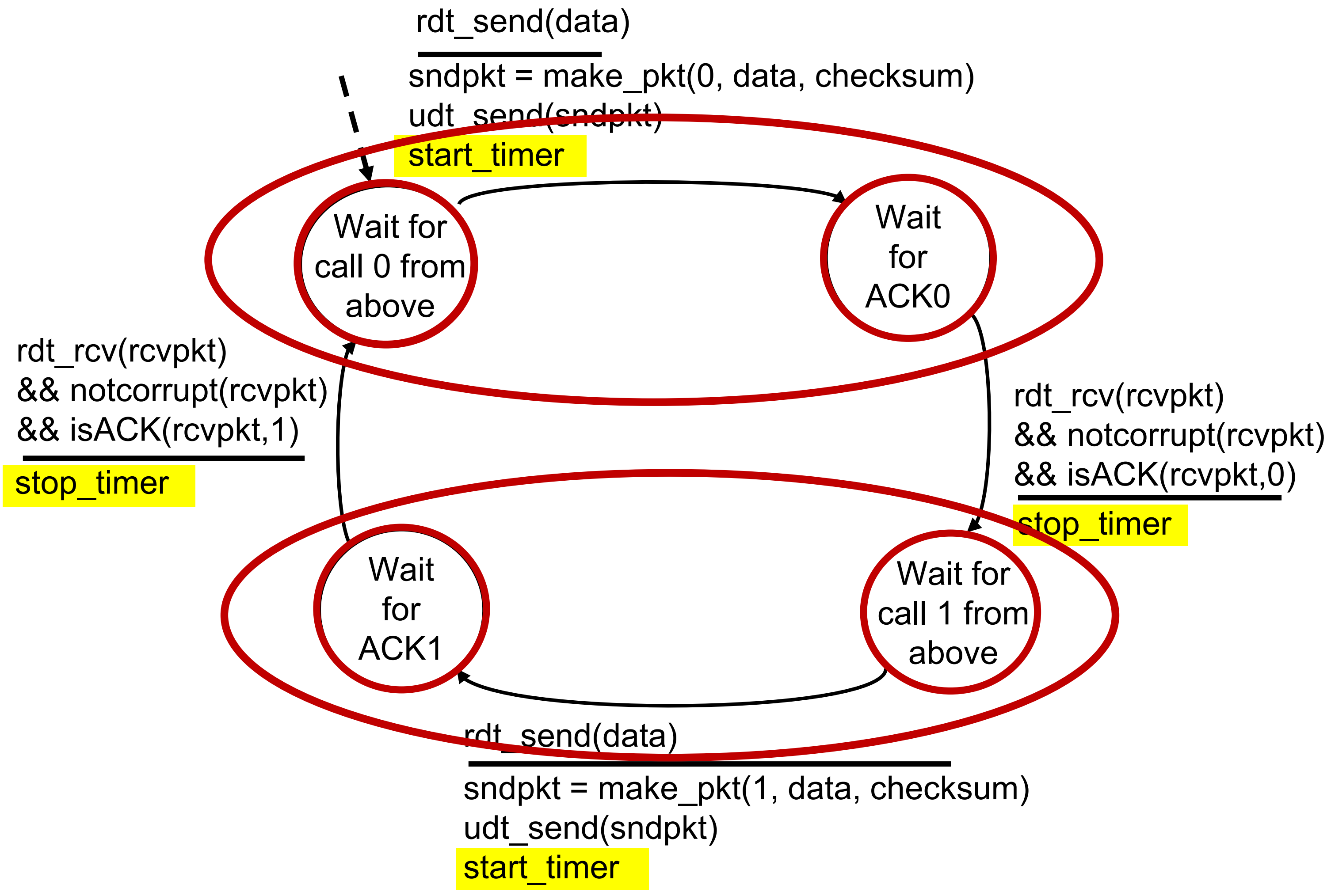
rdt3.0: channels with errors *and* loss

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

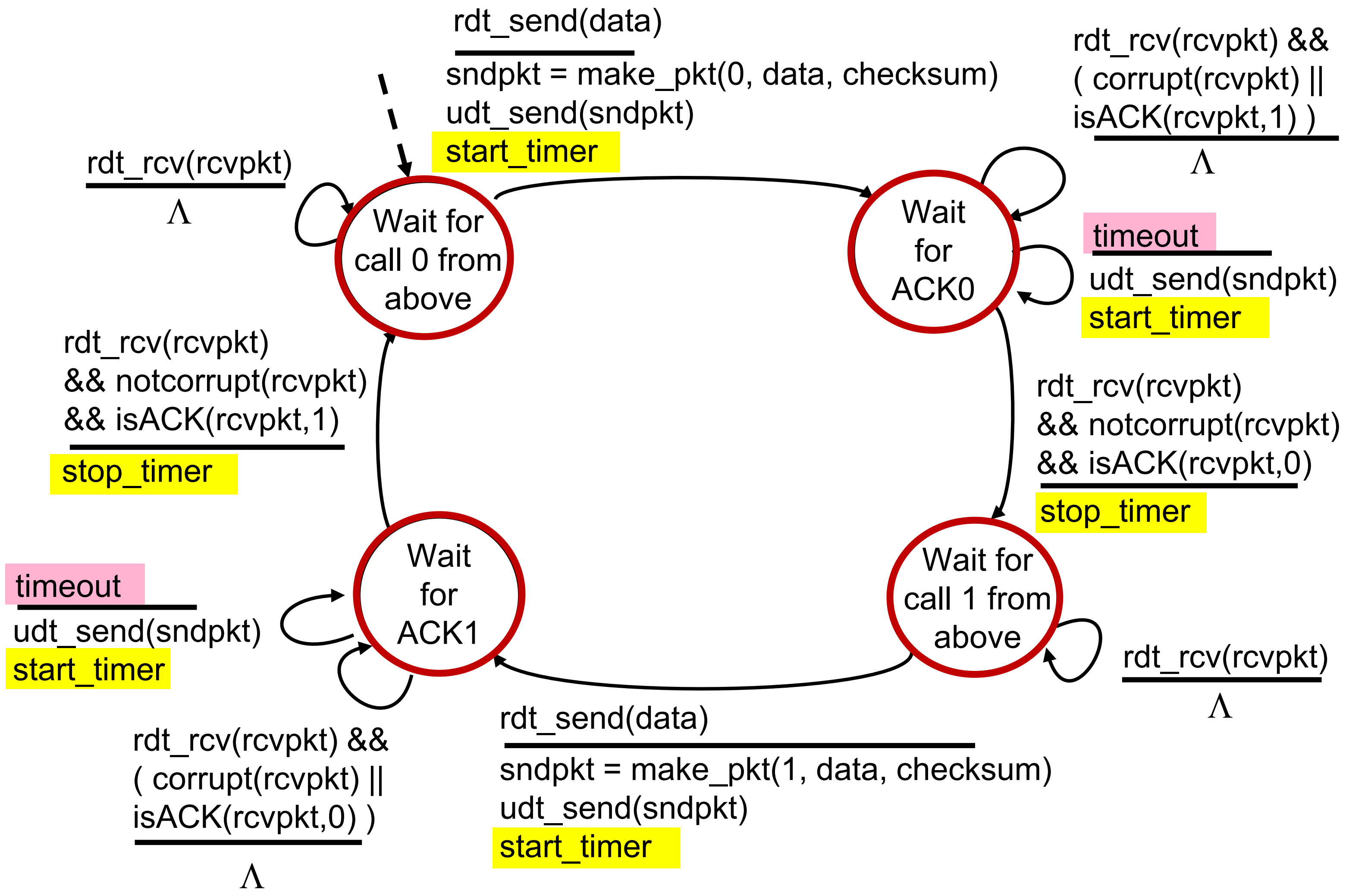
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but **seq #s already handles this**!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

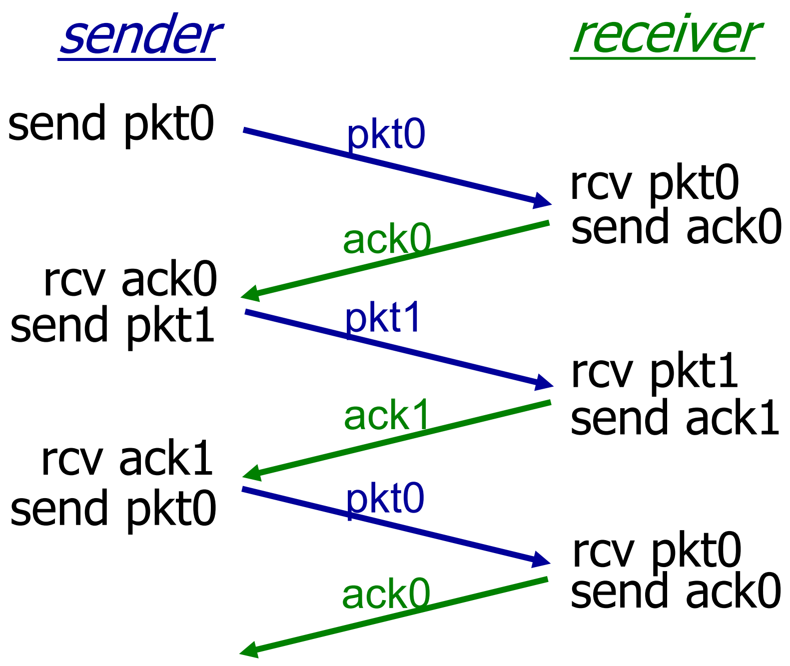
rdt3.0 sender (not required)



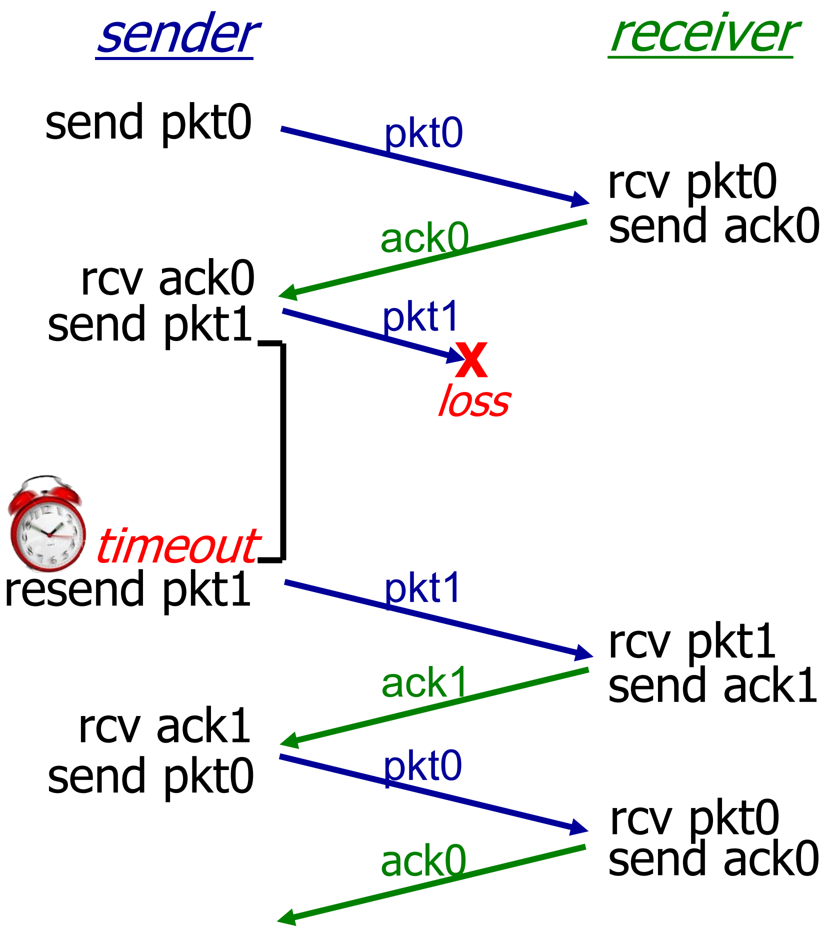
rdt3.0 sender (not required)



rdt3.0 in action

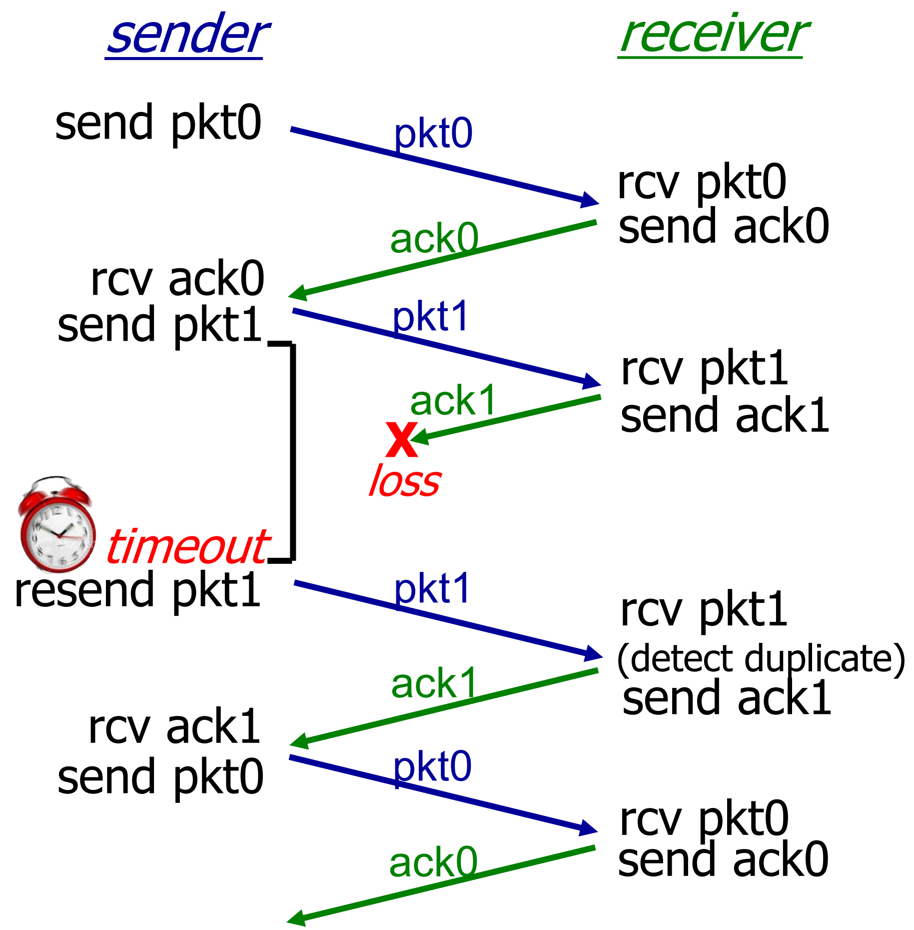


(a) no loss

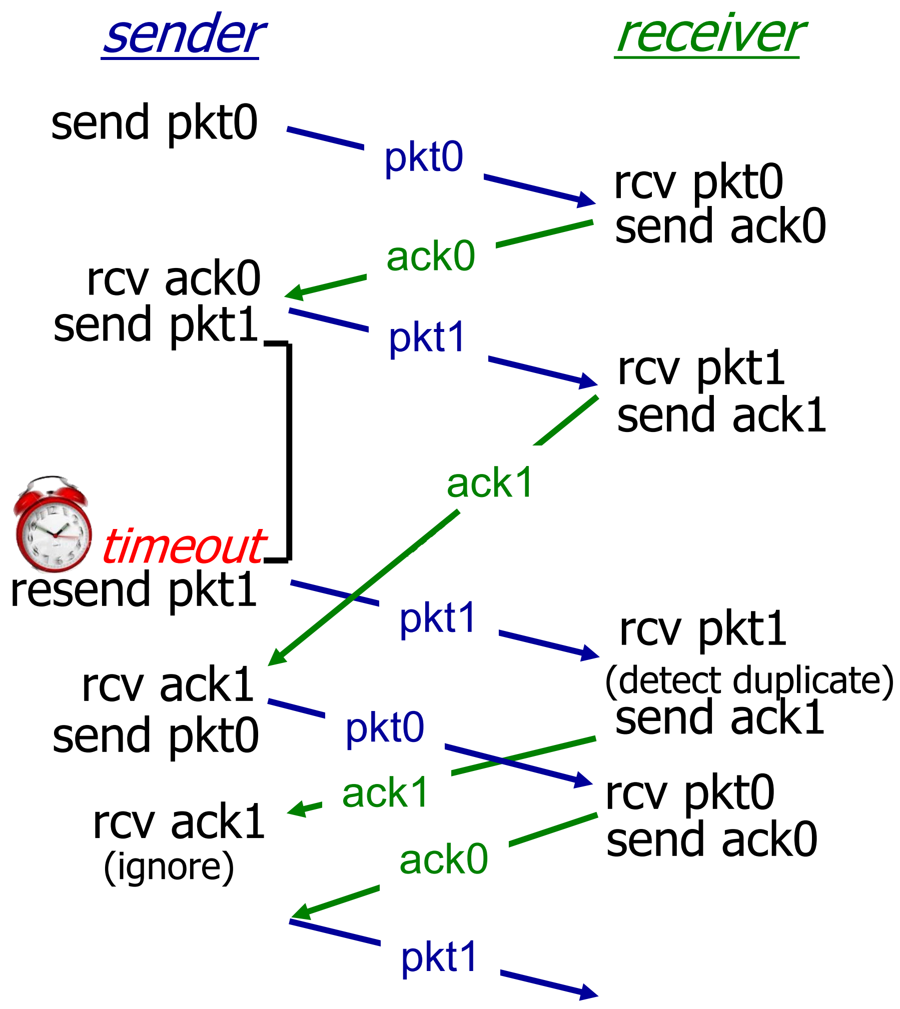


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

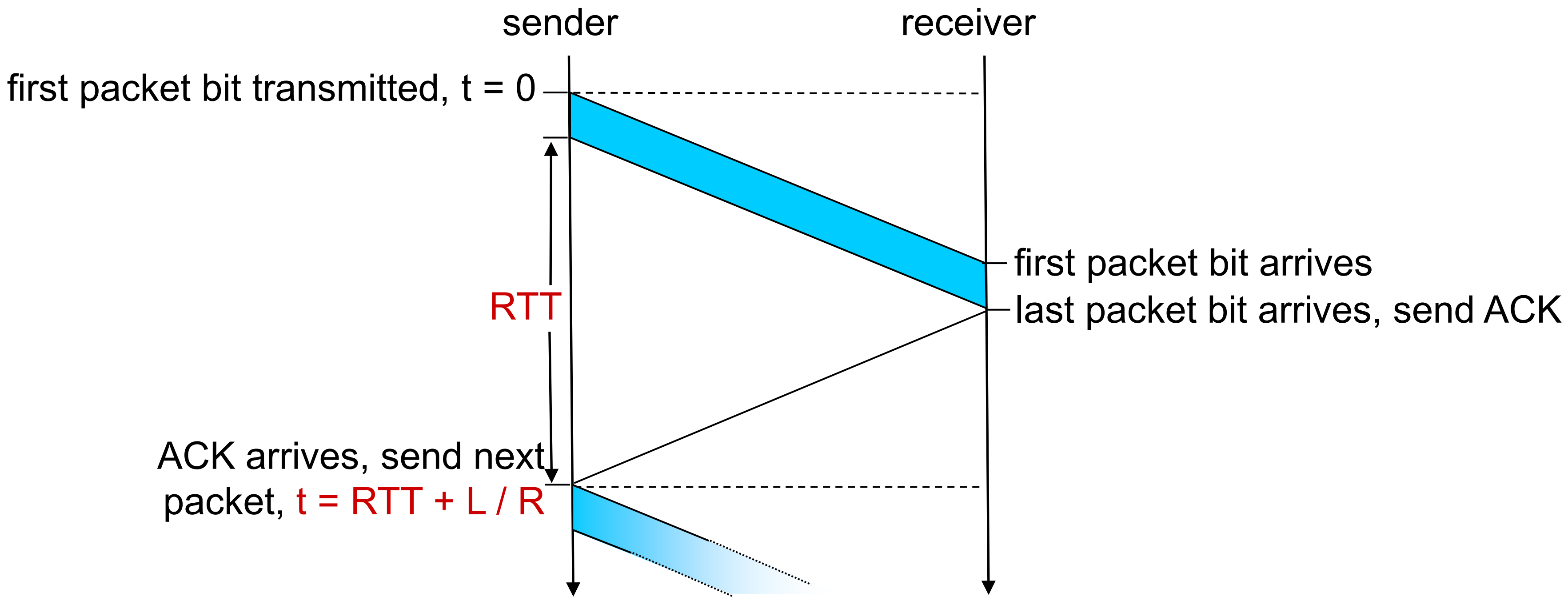
Performance of rdt3.0 (stop-and-wait)

- U_{sender}: *utilization* – fraction of time sender busy sending

- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

rdt3.0: stop-and-wait operation

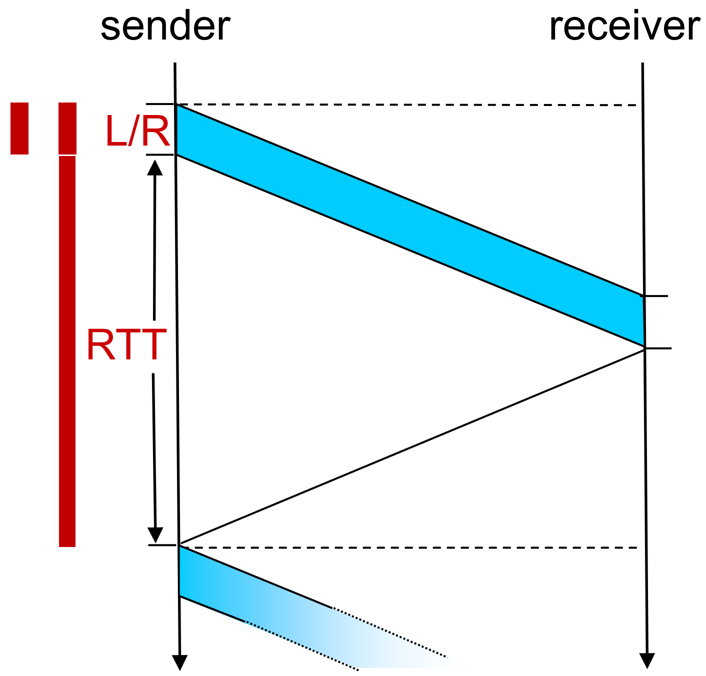


rdt3.0: stop-and-wait operation

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$

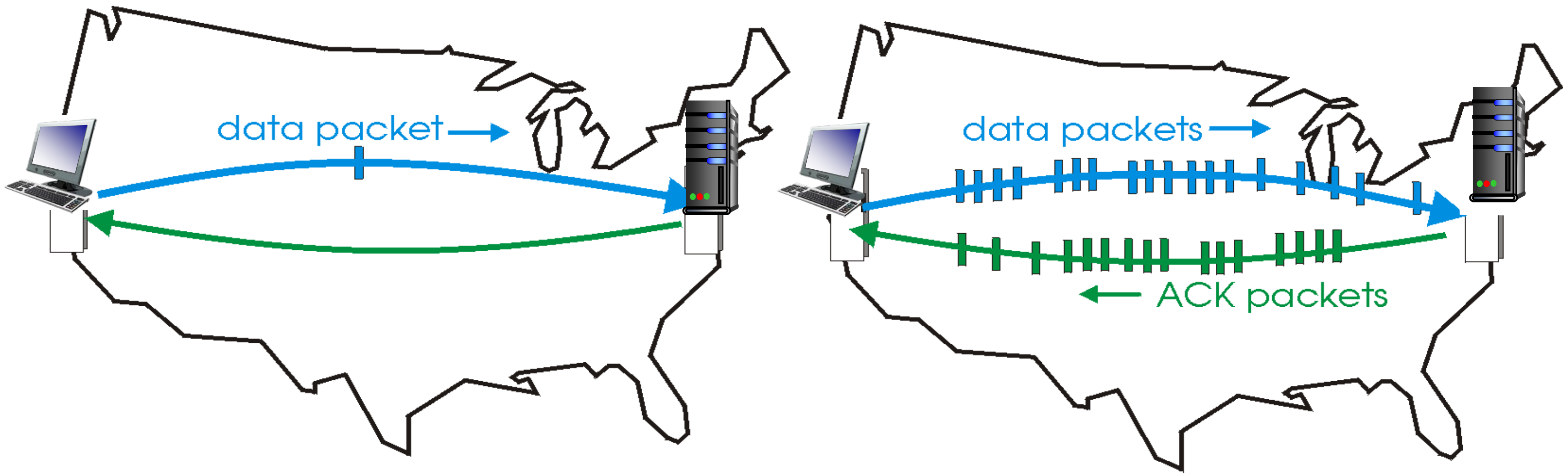


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

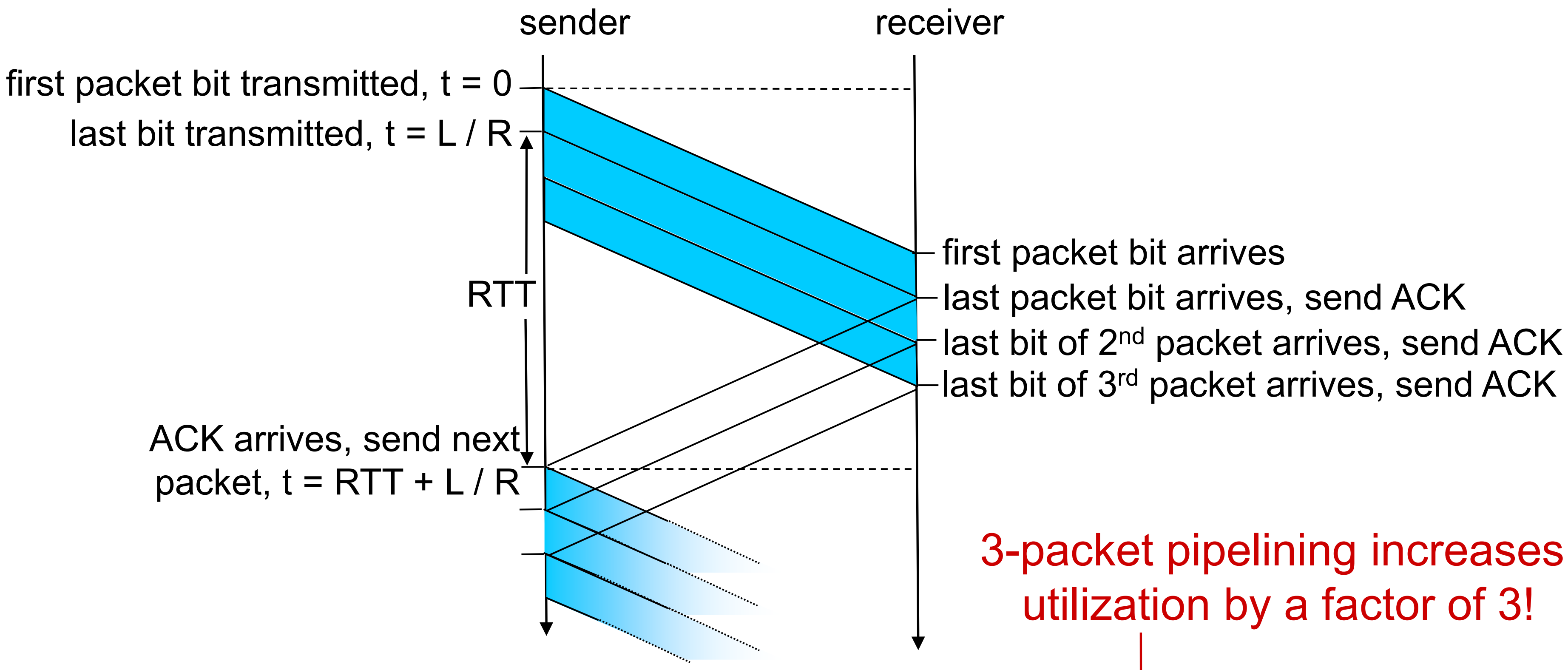
- range of **sequence numbers** must be increased
- **buffering** at sender and/or receiver



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Pipelining: increased utilization

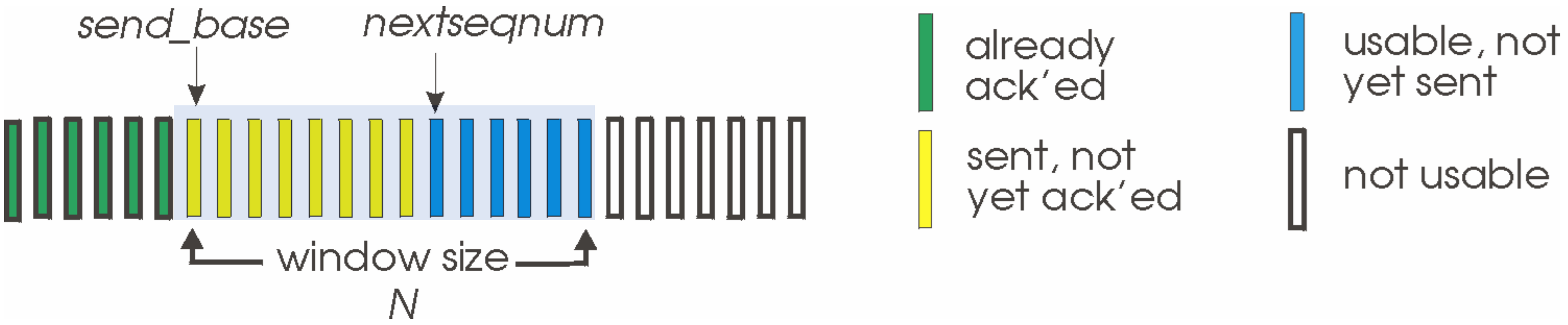


3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: sender

- sender: "**window**" of up to N, consecutive transmitted but unACKed pkts
  - **k-bit seq #** in pkt header (**$N = 2^k$**)



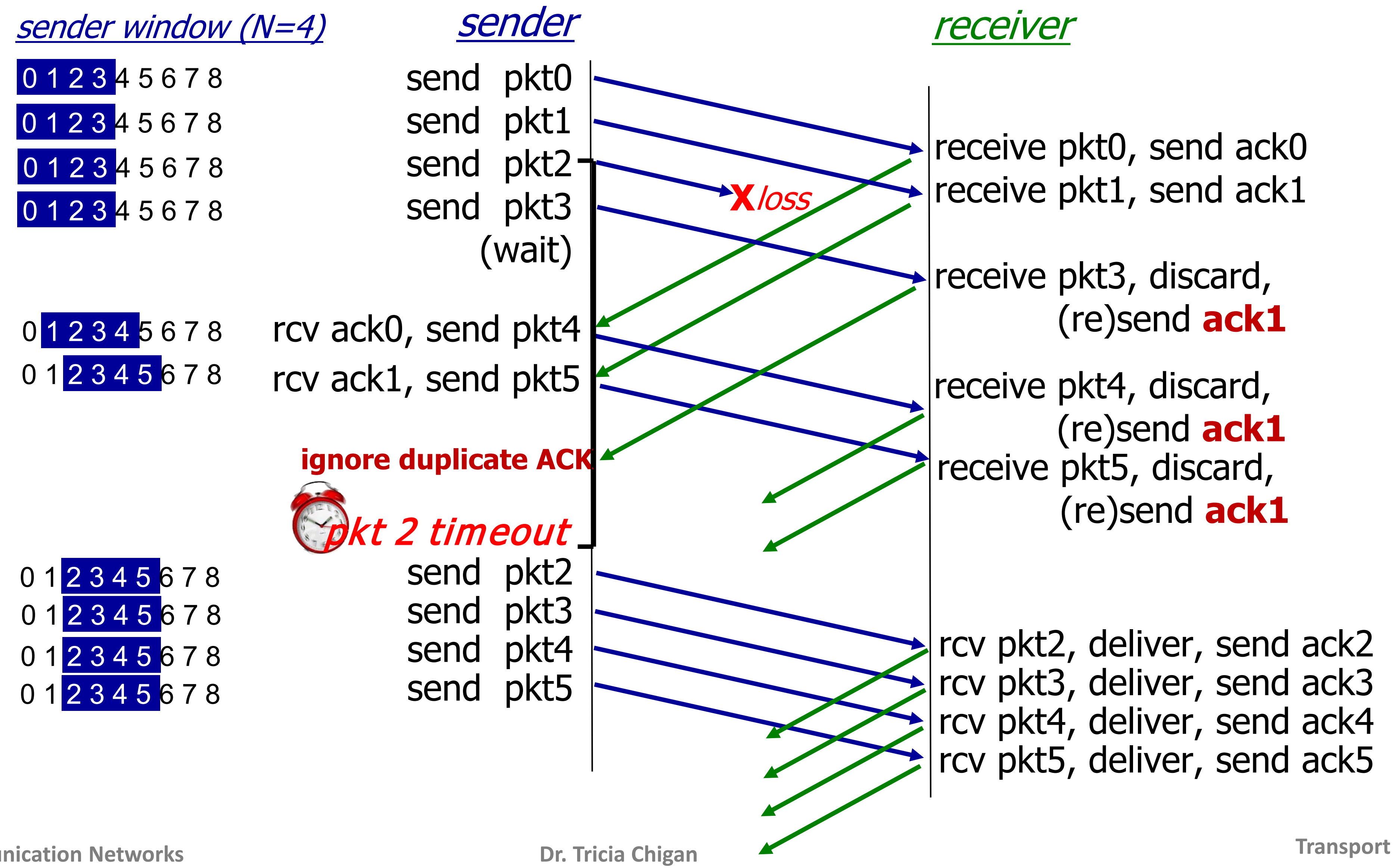
- ***cumulative ACK:*** ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for **oldest in-flight packet**
- ***timeout(n):*** retransmit packet n and all higher seq # packets in window

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** out-of-order packets
  - **re-ACK** pkt with highest in-order seq # (***cumulative ACK***)

Receiver view of sequence number space:

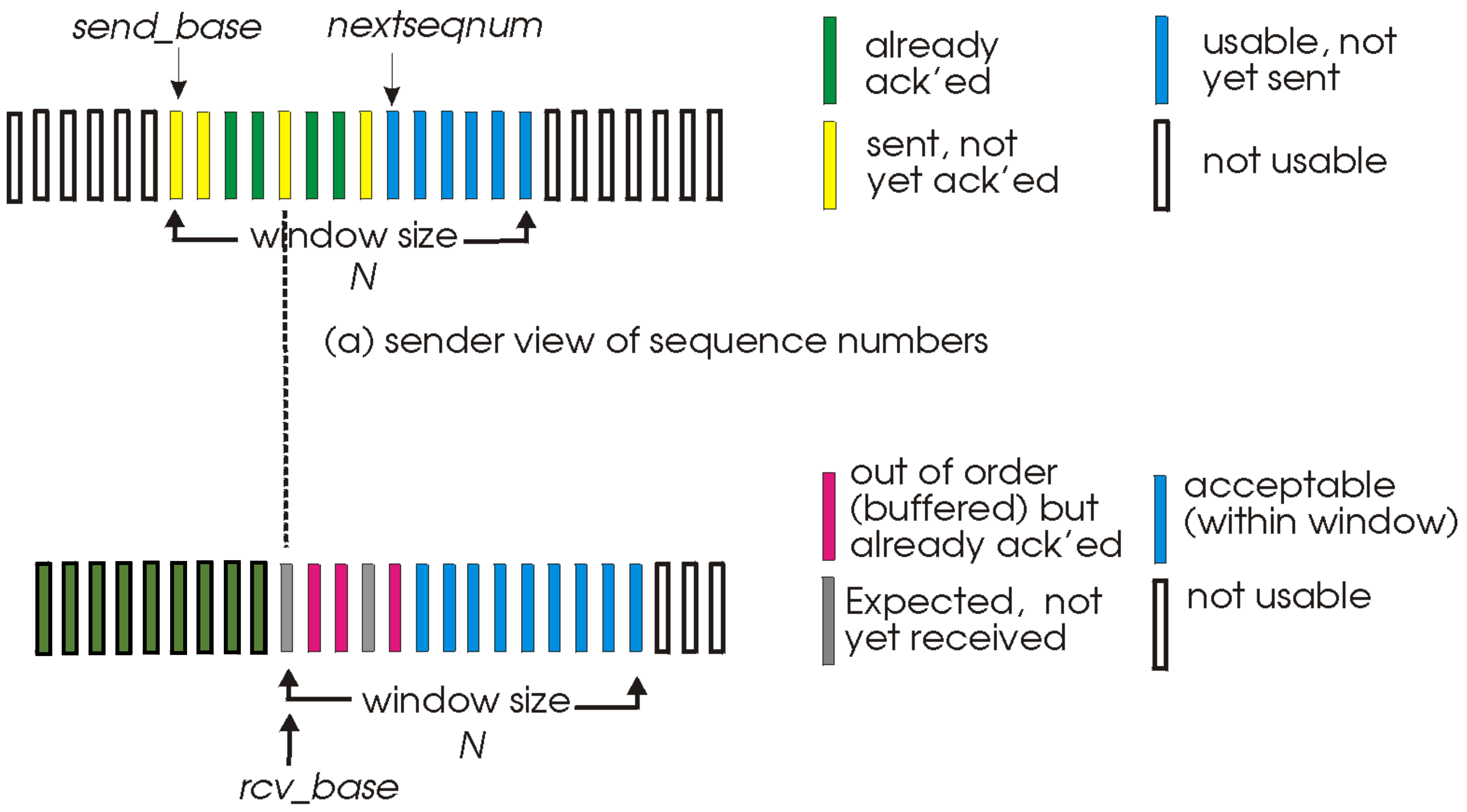
Go-Back-N in action



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



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

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 [rcv_base, rcv_base+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

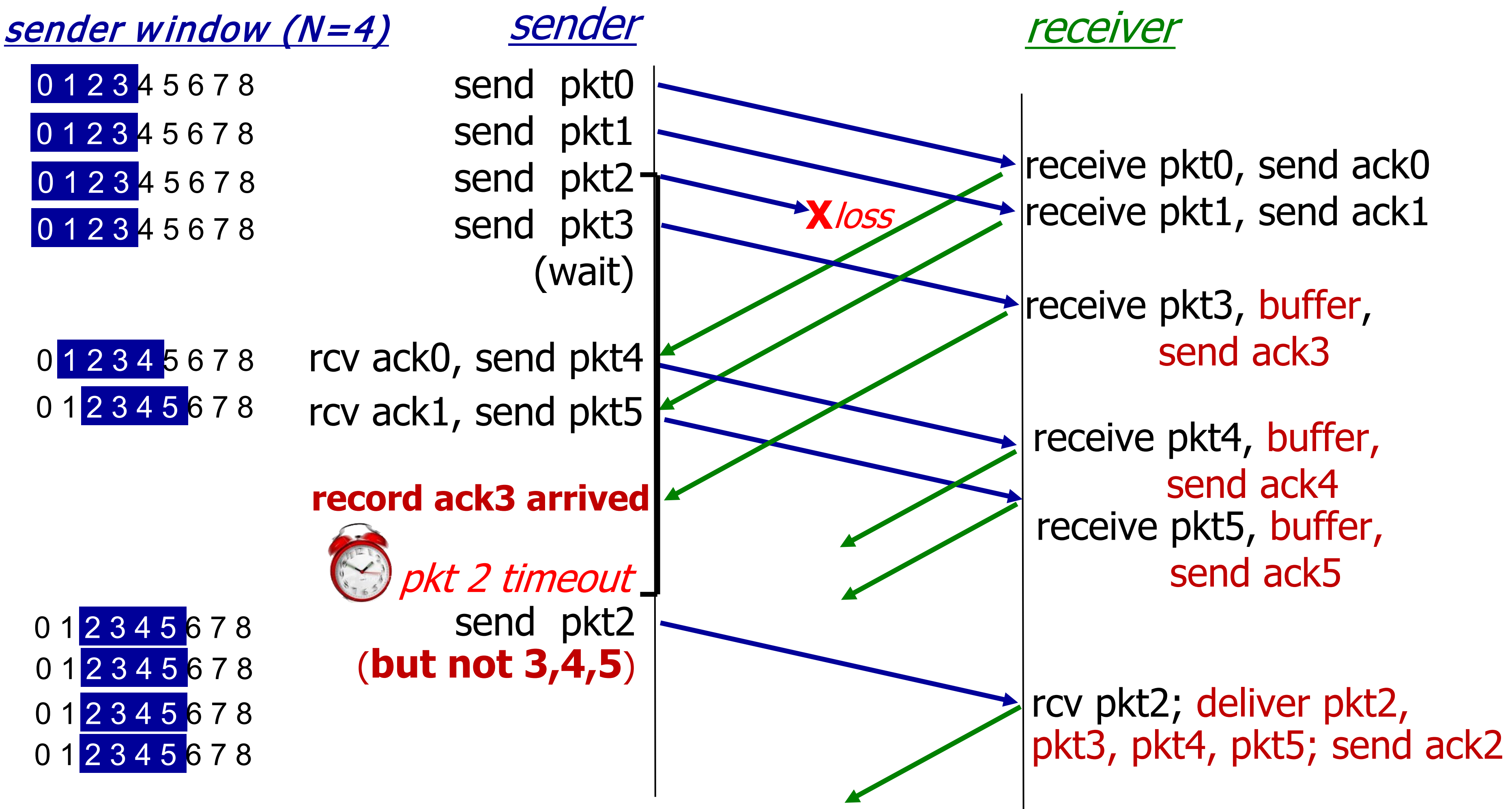
packet *n* in [rcv_base-N,rcv_base-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action



Q: what happens when ack2 arrives?

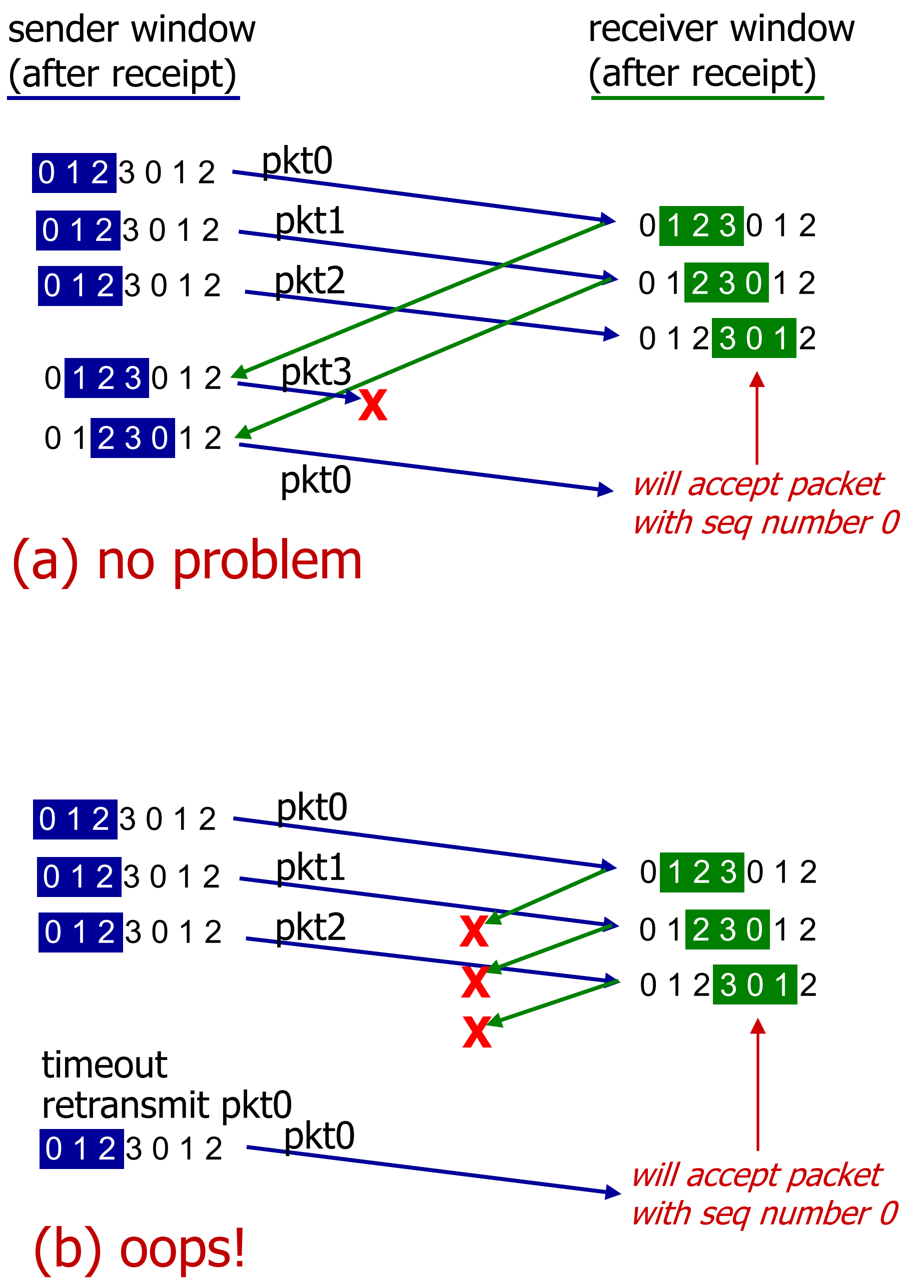
Sequence Numbers vs. Window Size

- How large do sequence numbers need to be?
  - **SeqNum** field is **finite**; sequence numbers wrap around (**reuse**)
  - Must be able to **detect** wrap-around
  - Sequence number space must be **larger** than the number of **outstanding packets**
  - Depends on sender/receiver window size
- Example
  - Max seq = 7 (0, 1,...,7), send_win=recv_win=7
  - If pkts 0..6 are sent successfully and all **acks** lost
  - **Sender** retransmits **old** **0..6**
  - **Receiver** expects **7,0..5**, but receives them as **second incarnation** of **0..5**
- Max sequence must be $\geq$ send window + recv window
  - For Go-Back-N: Max seq $\geq$ send window + 1
  - For Repeated Select: Max seq $\geq$ 2 x sender window (often = receiver window)

Selective repeat: a dilemma! (reference)

example:

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



Selective repeat: a dilemma! (reference)

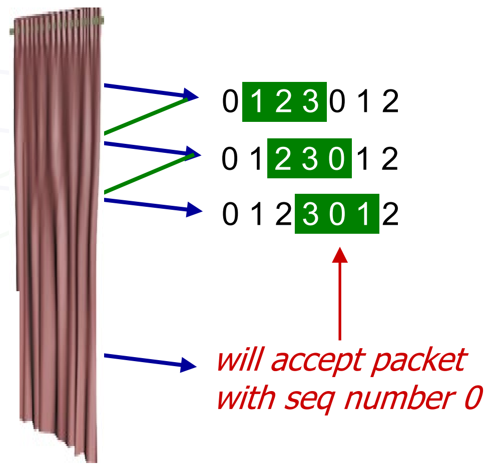
example:

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

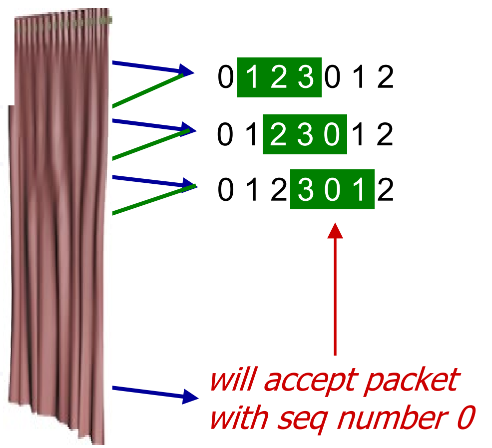
Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

sender window (after receipt)

receiver window (after receipt)



- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*



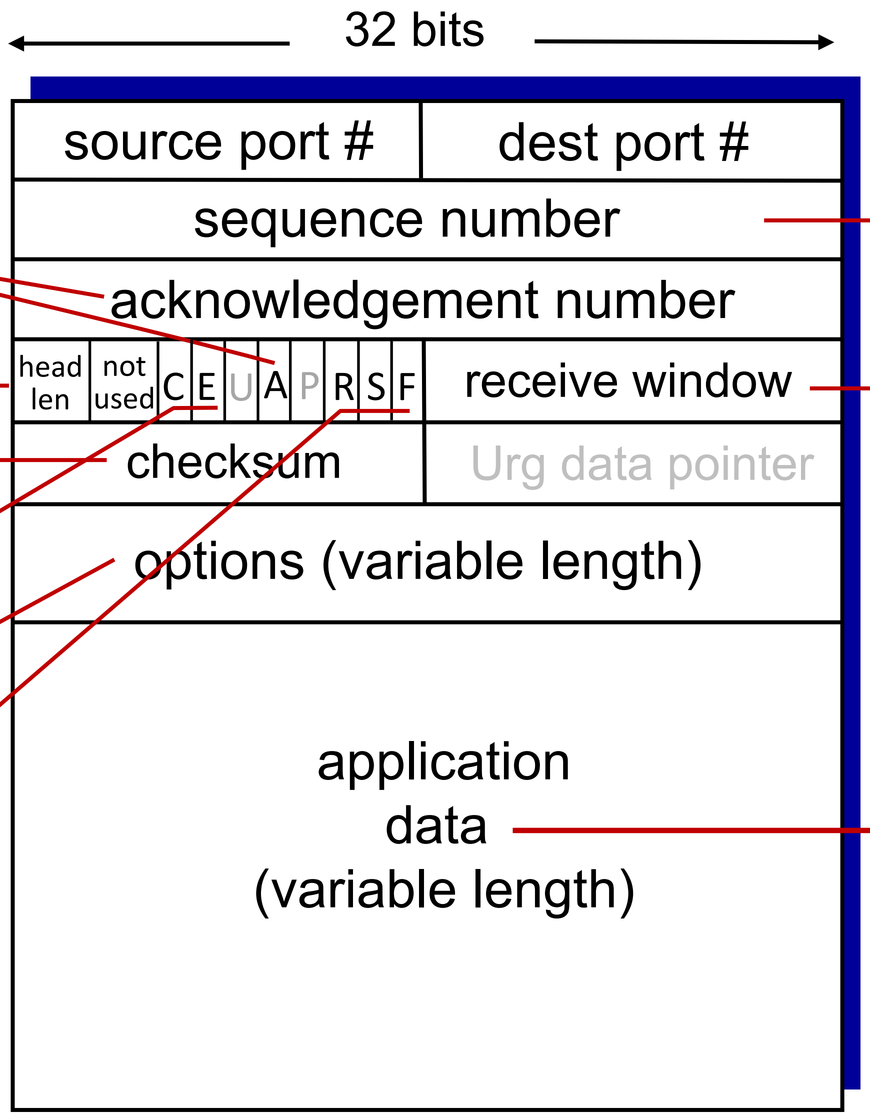
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order ***byte steam***:
  - no "message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size 1460 bytes
- cumulative ACKs
- pipelining:
  - TCP congestion and **flow control** set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

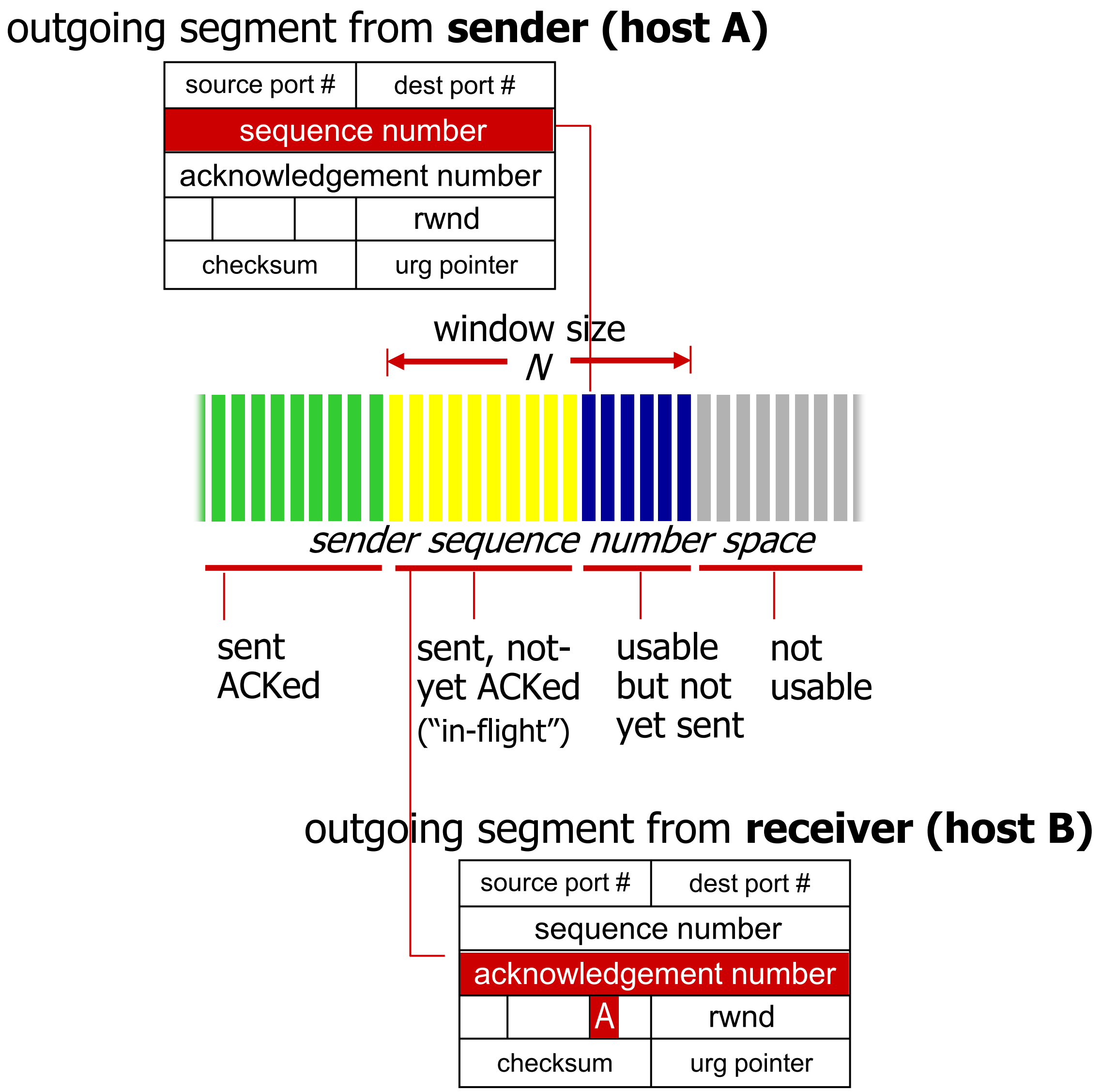
- byte stream "number" of first byte in segment's data

Acknowledgements:

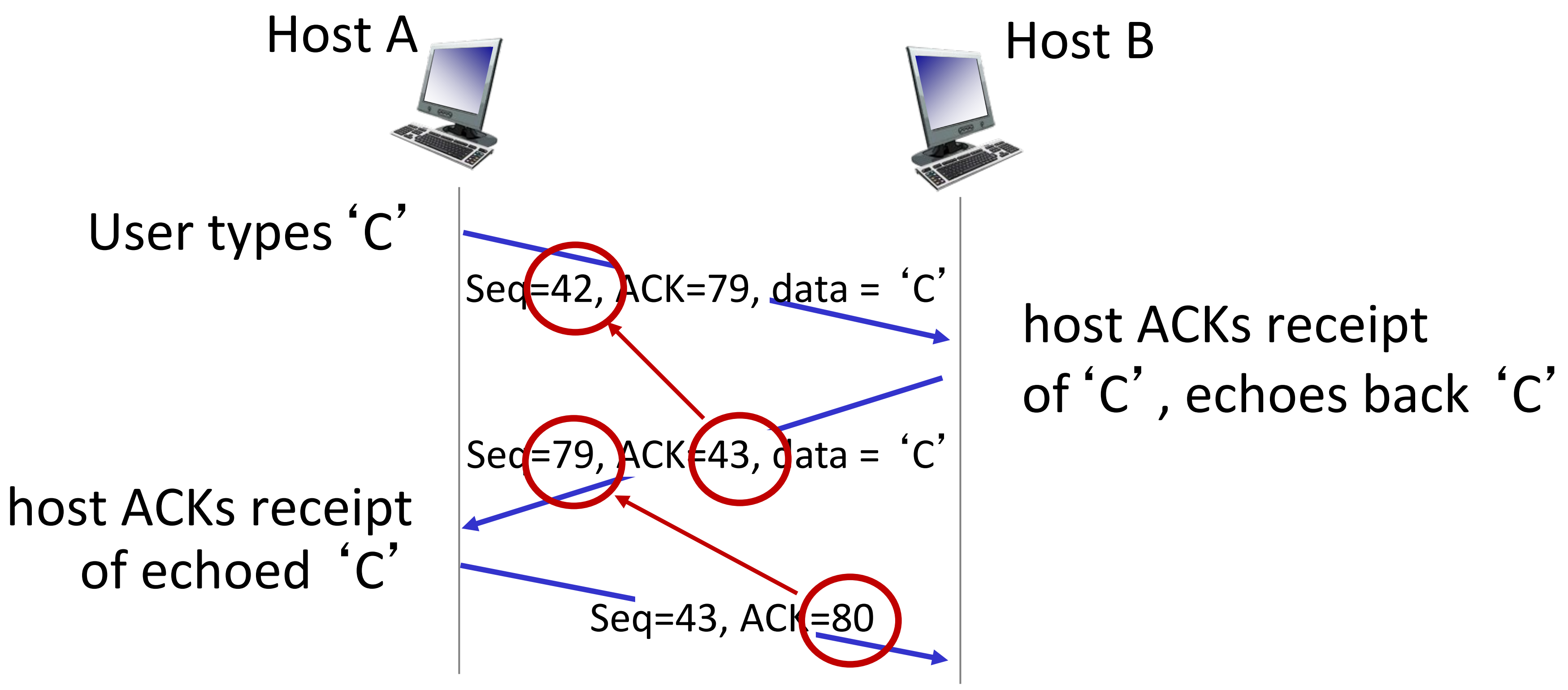
- seq # of next byte expected to **receive** from **other side**
- **cumulative ACK**

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 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

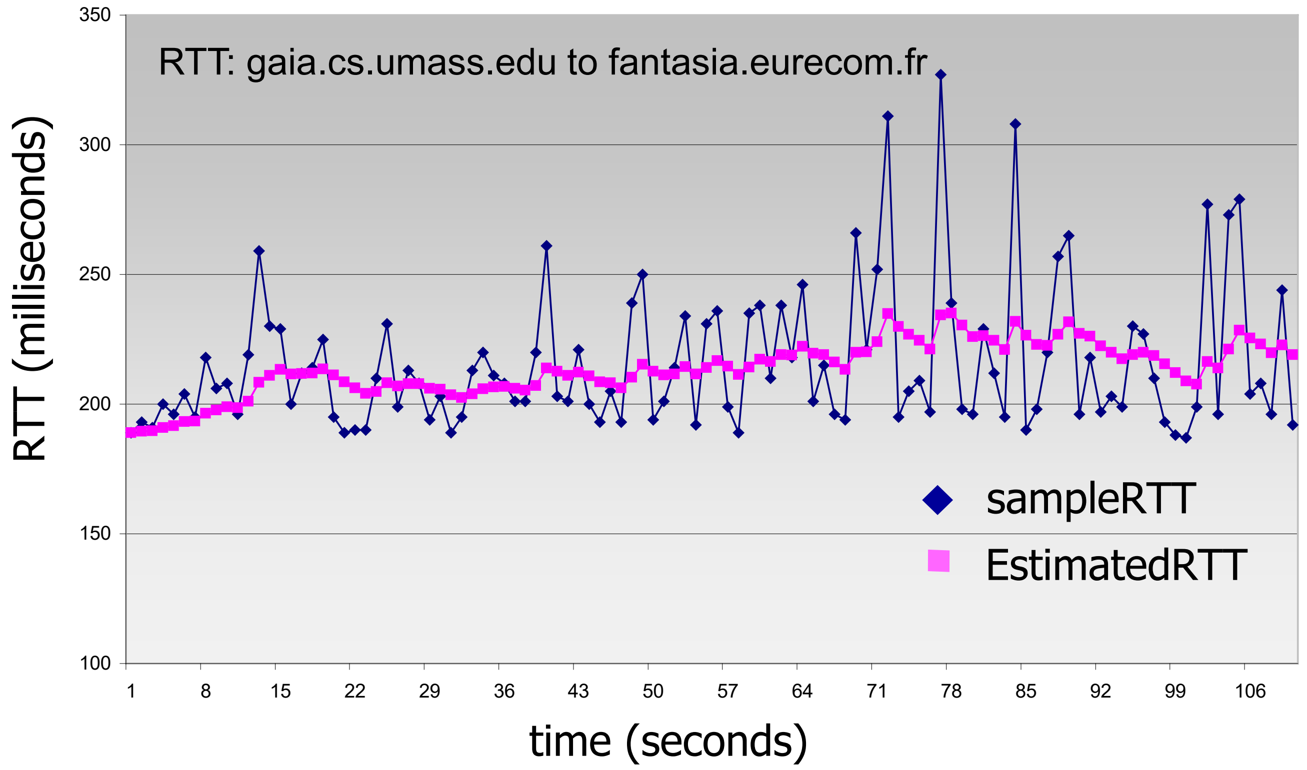
Q: how to estimate RTT?

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

TCP round trip time, timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- **timeout** interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`**: want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT (for EstimatedRTT)

"safety margin" (for 4*DevRTT)

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP Sender (simplified): transmission, retransmission

Event : data received from application
- create segment with seq #
- **seq #** is byte-stream number of first data byte in segment
- **start timer** if not already running
  - think of timer as for oldest unACKed segment
  - expiration interval: **`TimeOutInterval`**

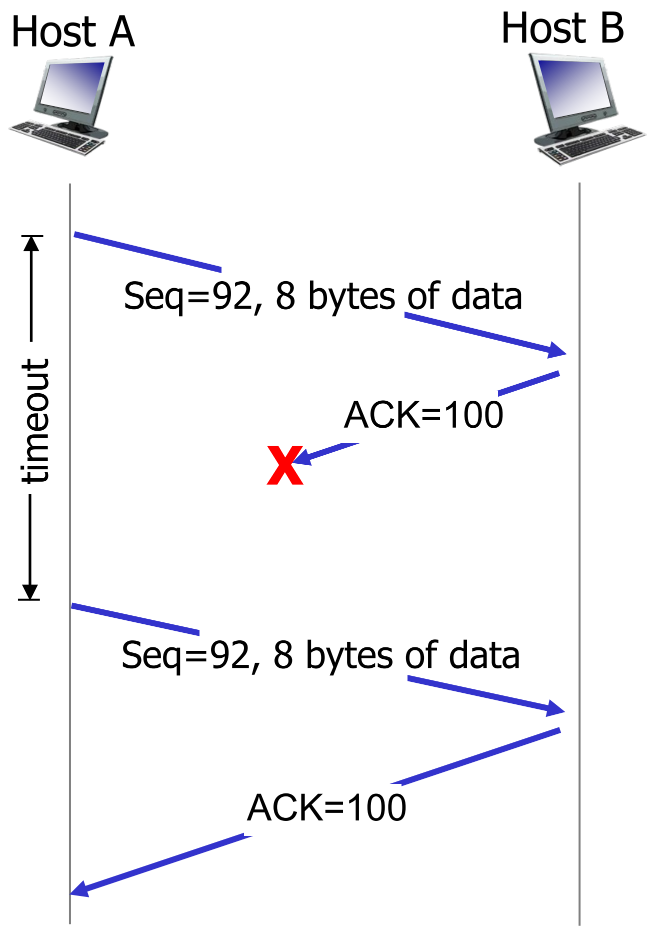
Event: *timeout*
- **retransmit** segment that caused timeout
- restart timer

Event: *ACK received*
- if ACK acknowledges previously unACKed segments
  - **updat**e what is known to be ACKed
  - **start timer** if there are still unACKed segments

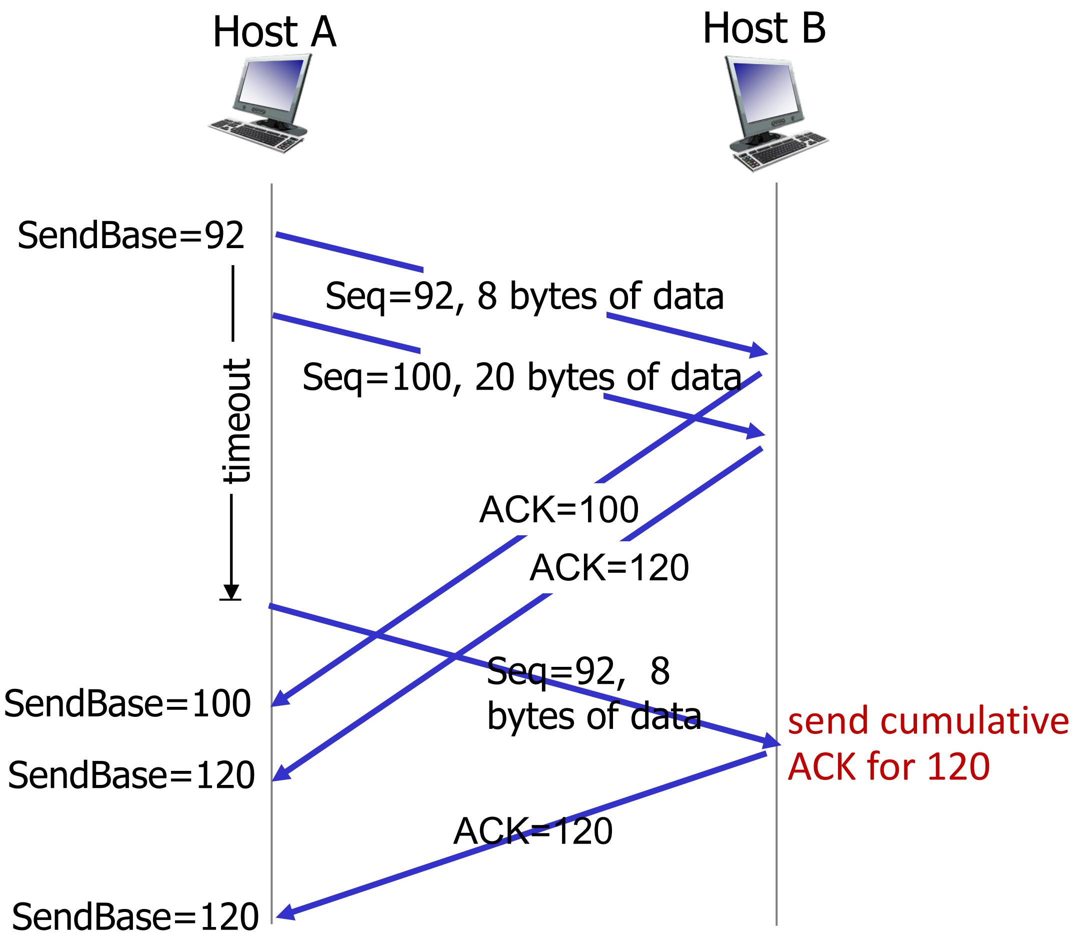
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP: retransmission scenarios

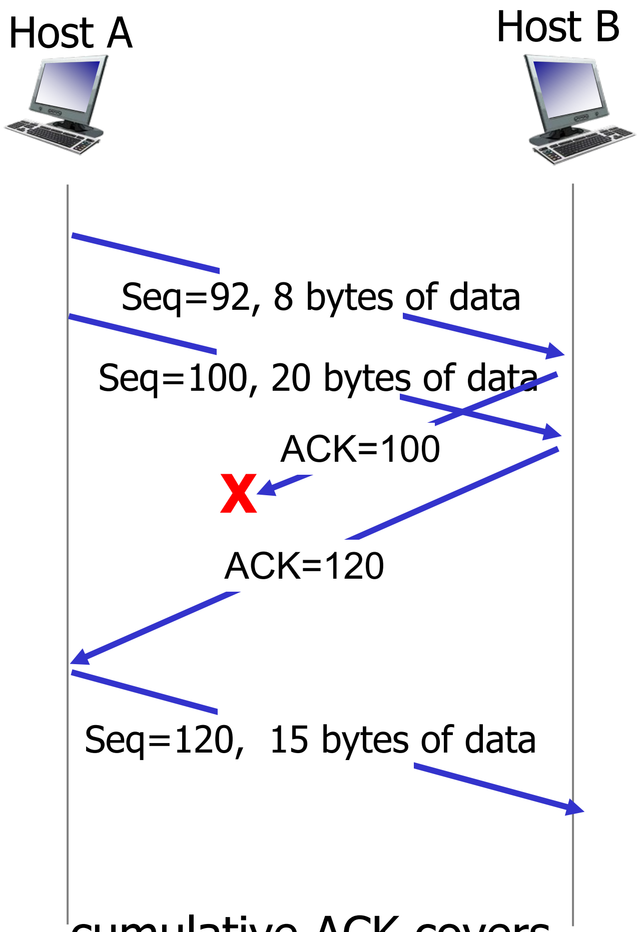


lost ACK scenario



premature timeout

TCP: retransmission scenarios



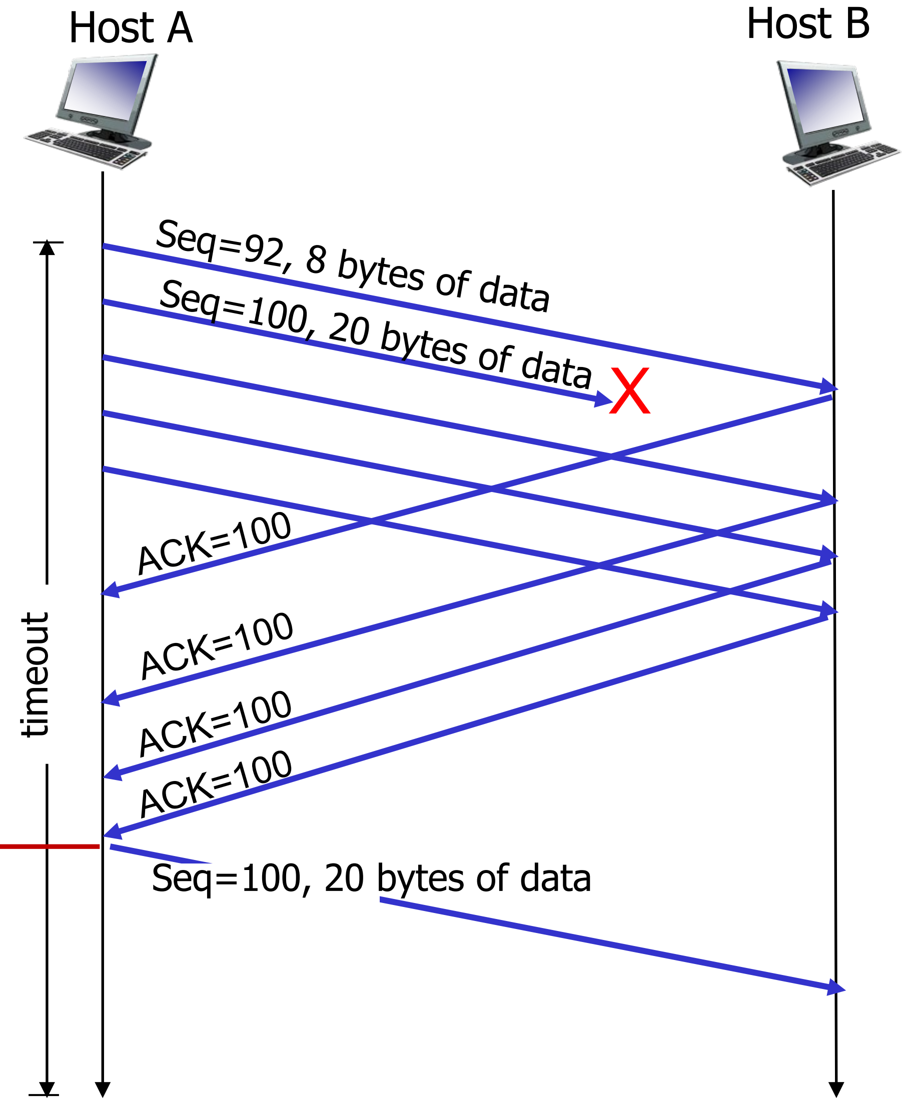
cumulative ACK covers for earlier lost ACK

TCP fast retransmit

TCP fast retransmit

if sender receives **3 additional** ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout



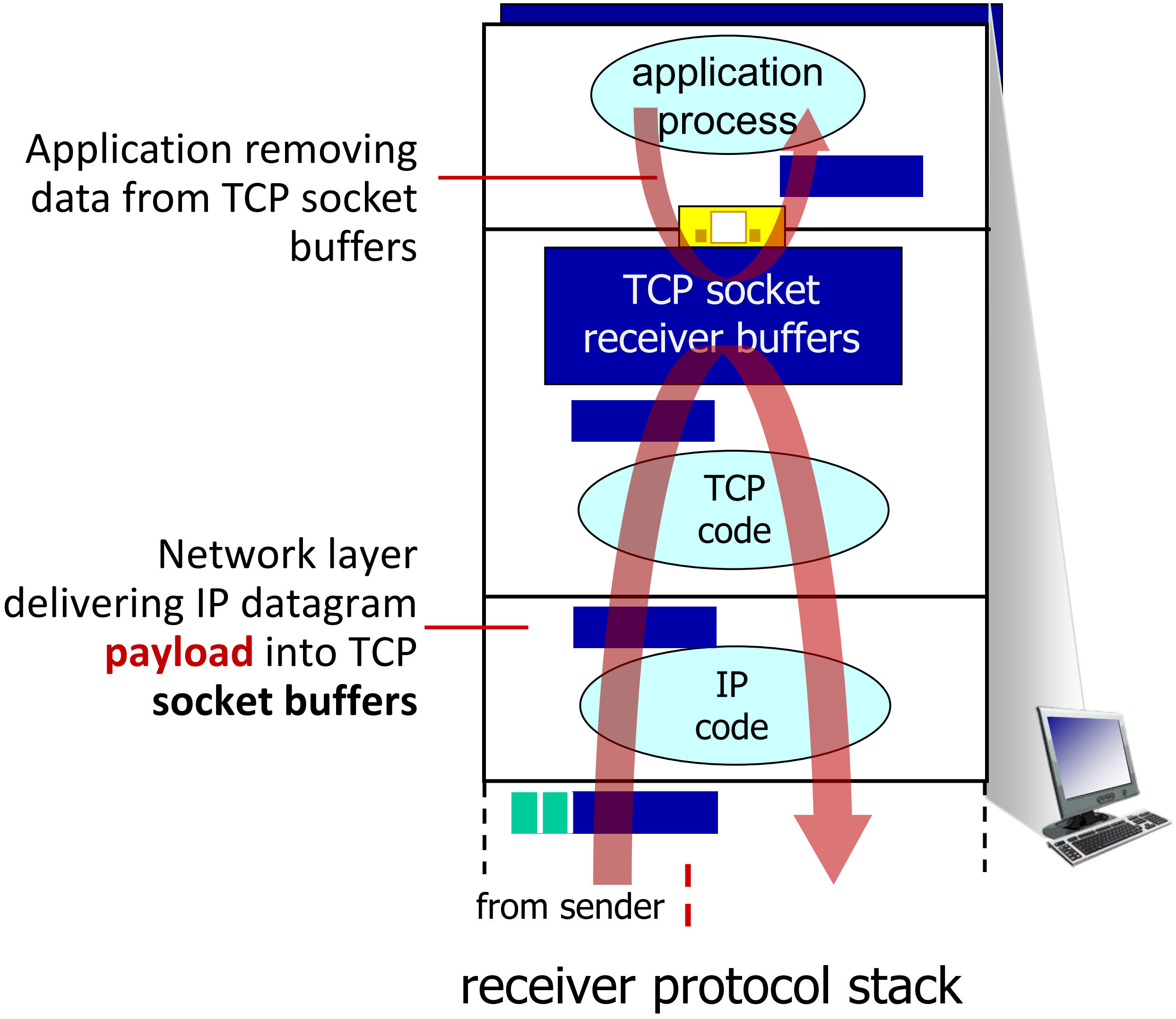
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP flow control

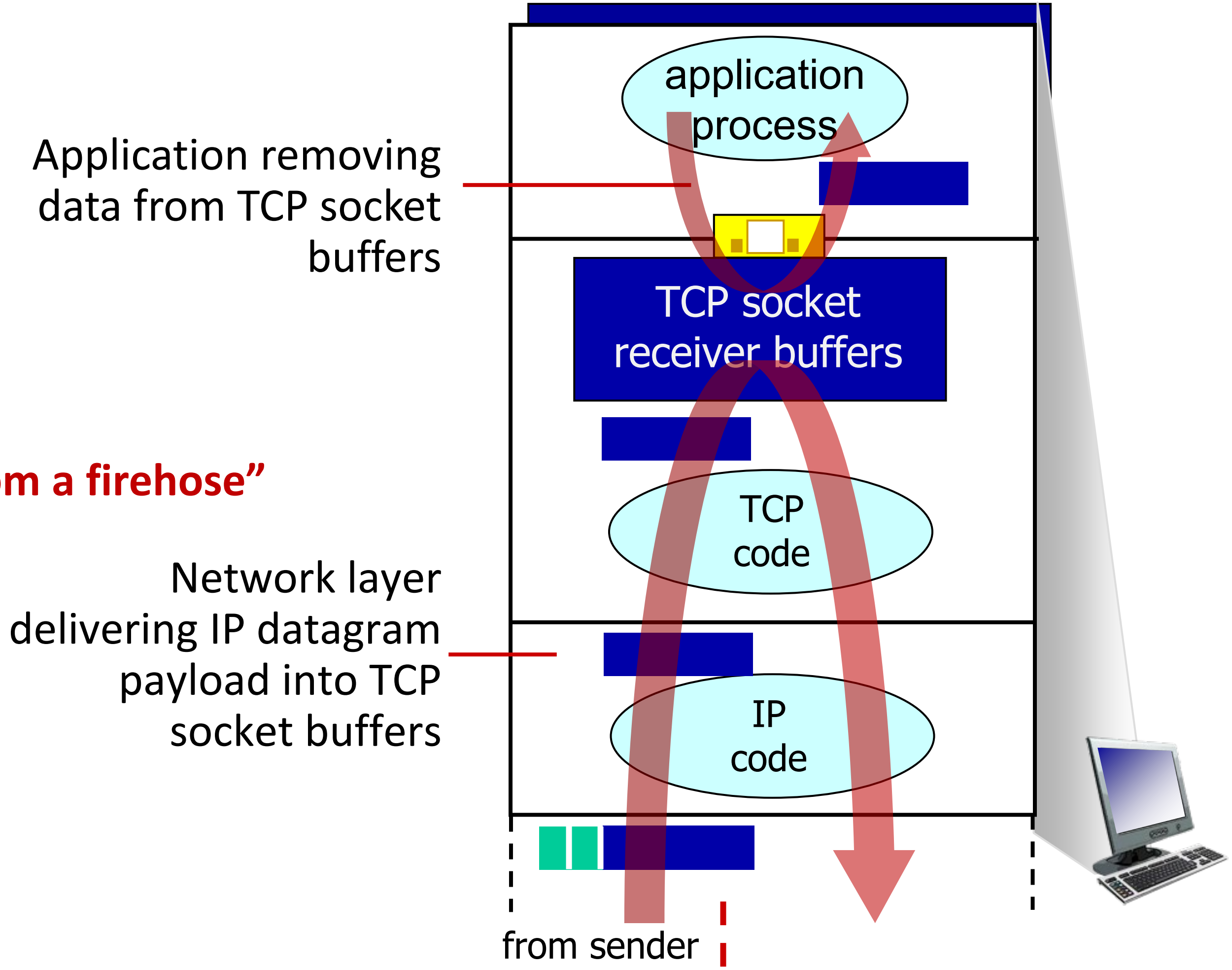
Q: What happens if **network layer** delivers data faster than **application layer** removes data from socket buffers?



TCP flow control

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

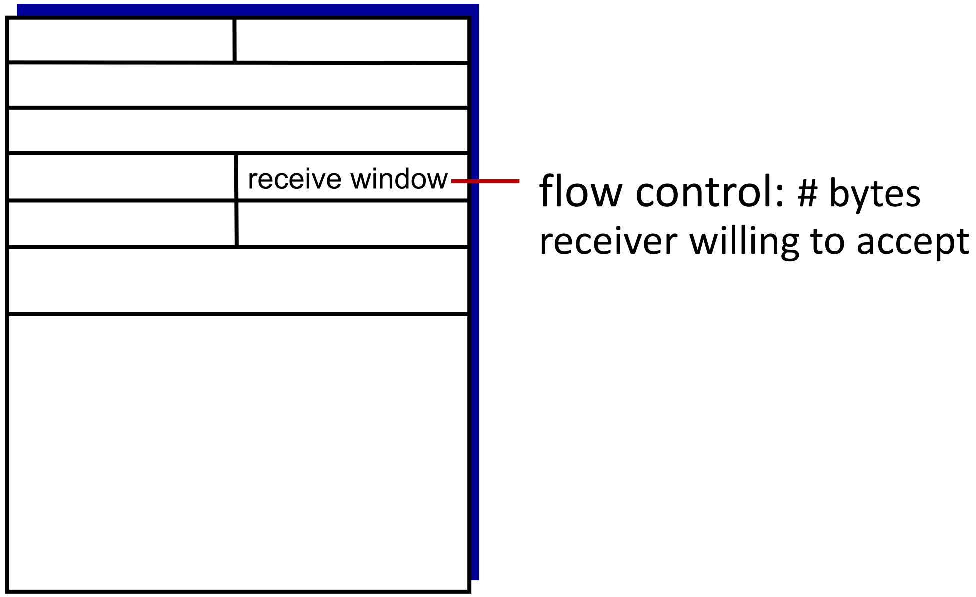
"no one can drink from a firehose"

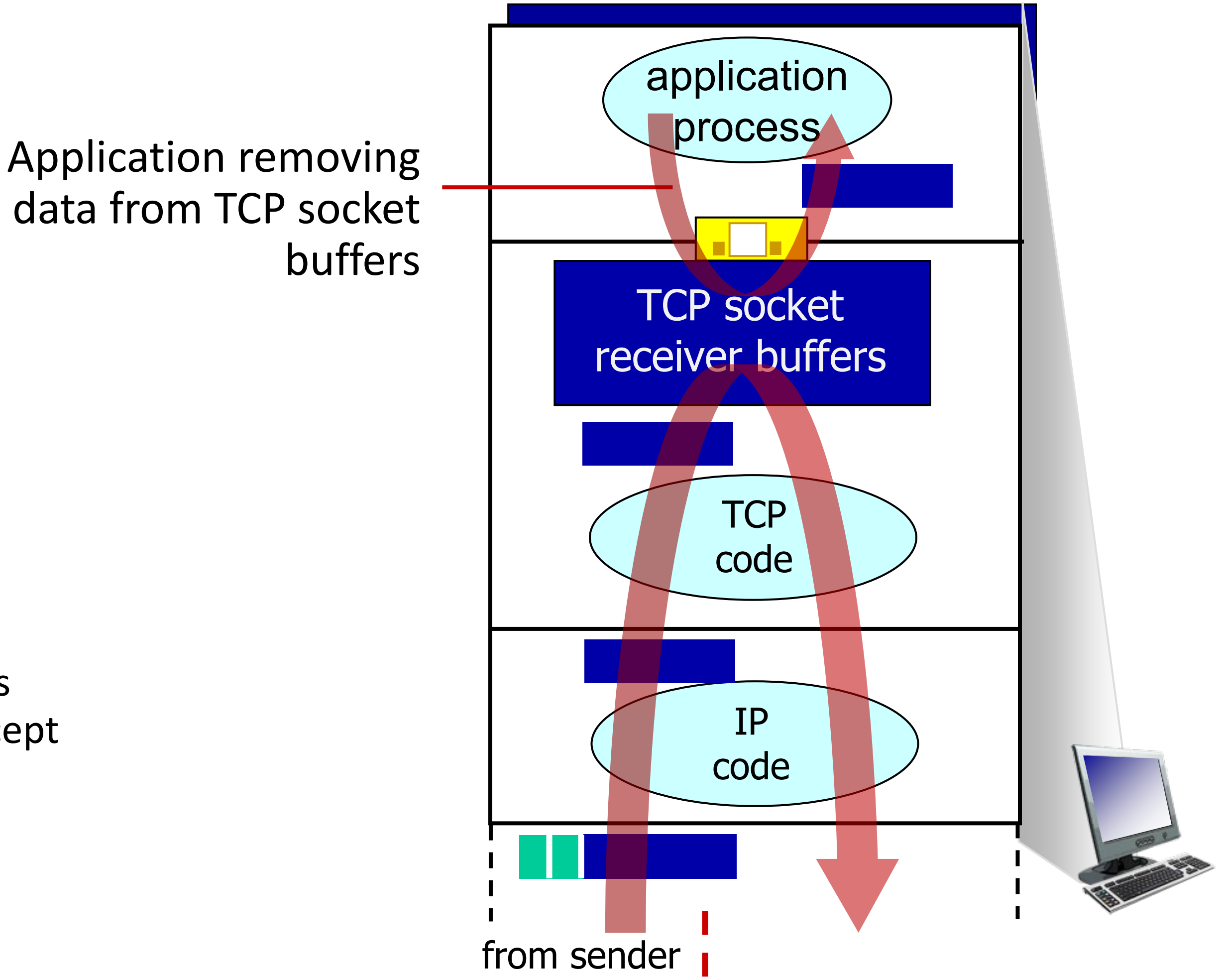


receiver protocol stack

TCP flow control

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

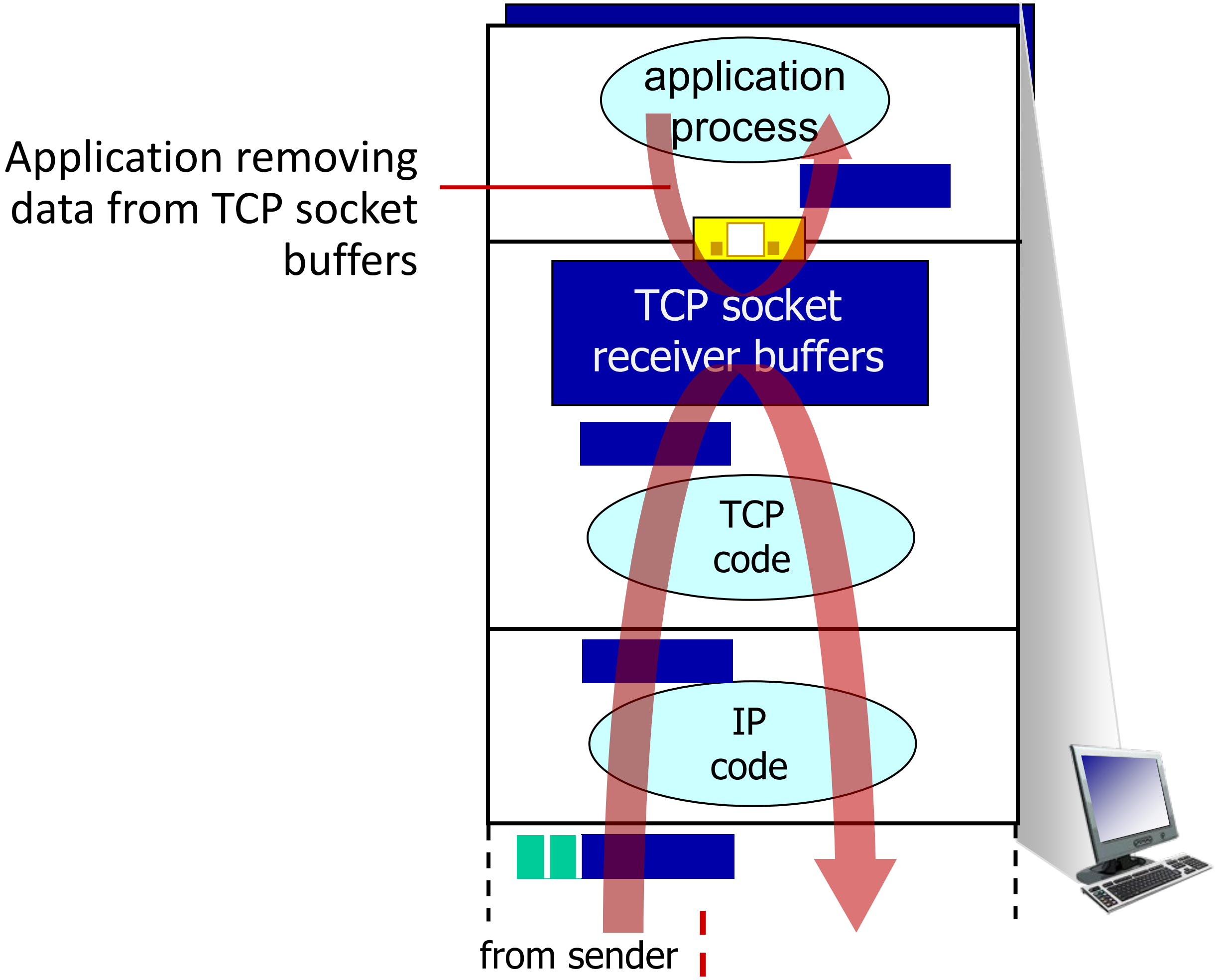




receiver protocol stack

TCP flow control

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



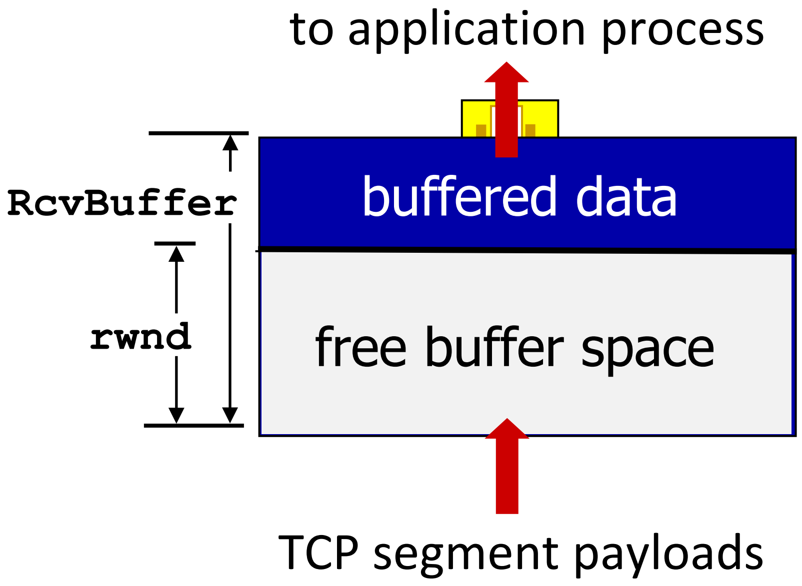
receiver protocol stack

flow control

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

TCP flow control

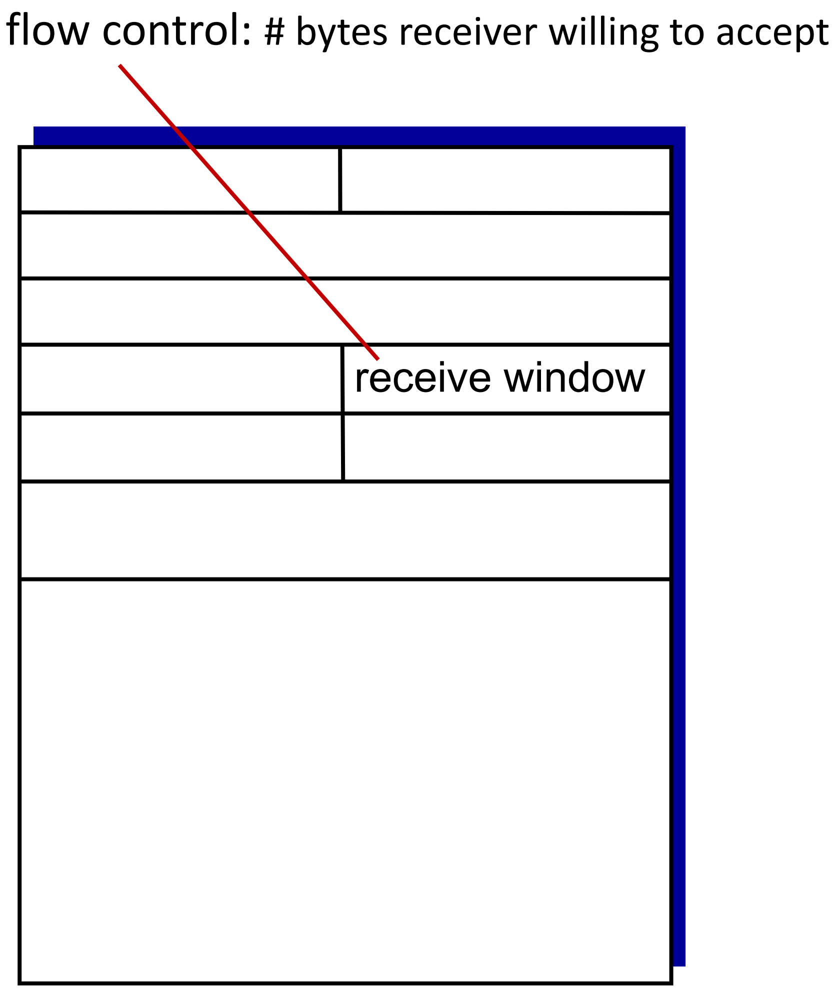
- TCP **receiver** “advertises” free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many **operating systems** autoadjust **`RcvBuffer`**
- **sender** limits amount of unACKed (“in-flight”) data to received **`rwnd`**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



TCP **receiver-side** buffering

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

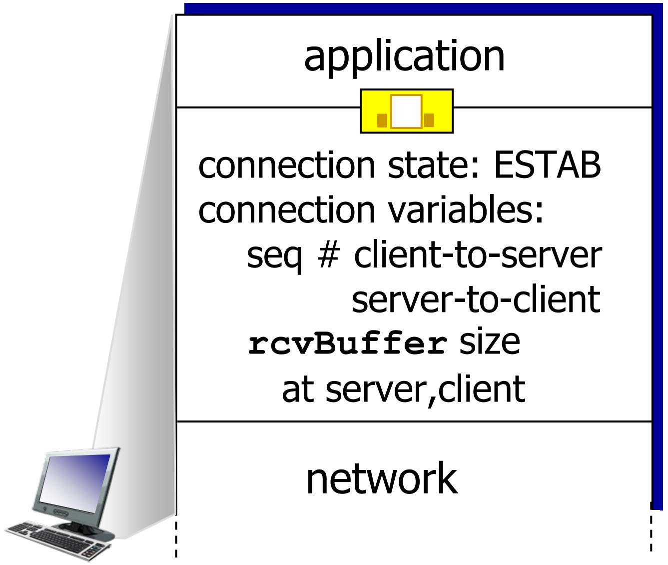


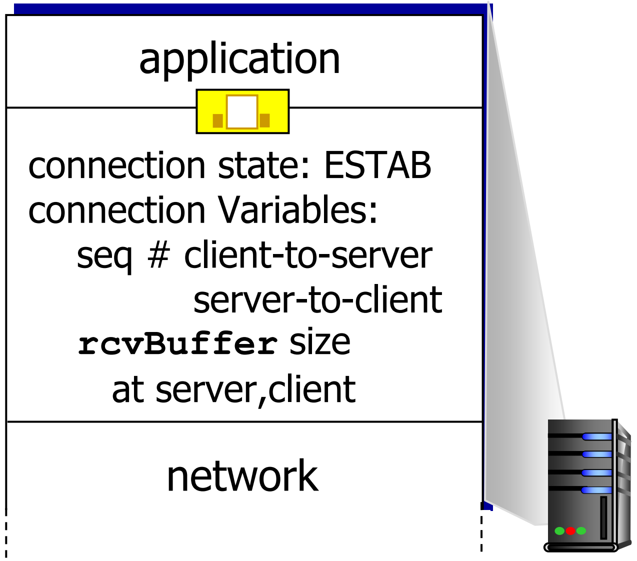
TCP segment format

TCP connection management

before exchanging data, sender/receiver "handshake":

- agree to **establish connection** (each knowing the other **willing** to establish connection)
- agree on **connection parameters** (e.g., starting seq #s)






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

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

A human 3-way handshake protocol

TCP 3-way handshake

Client state

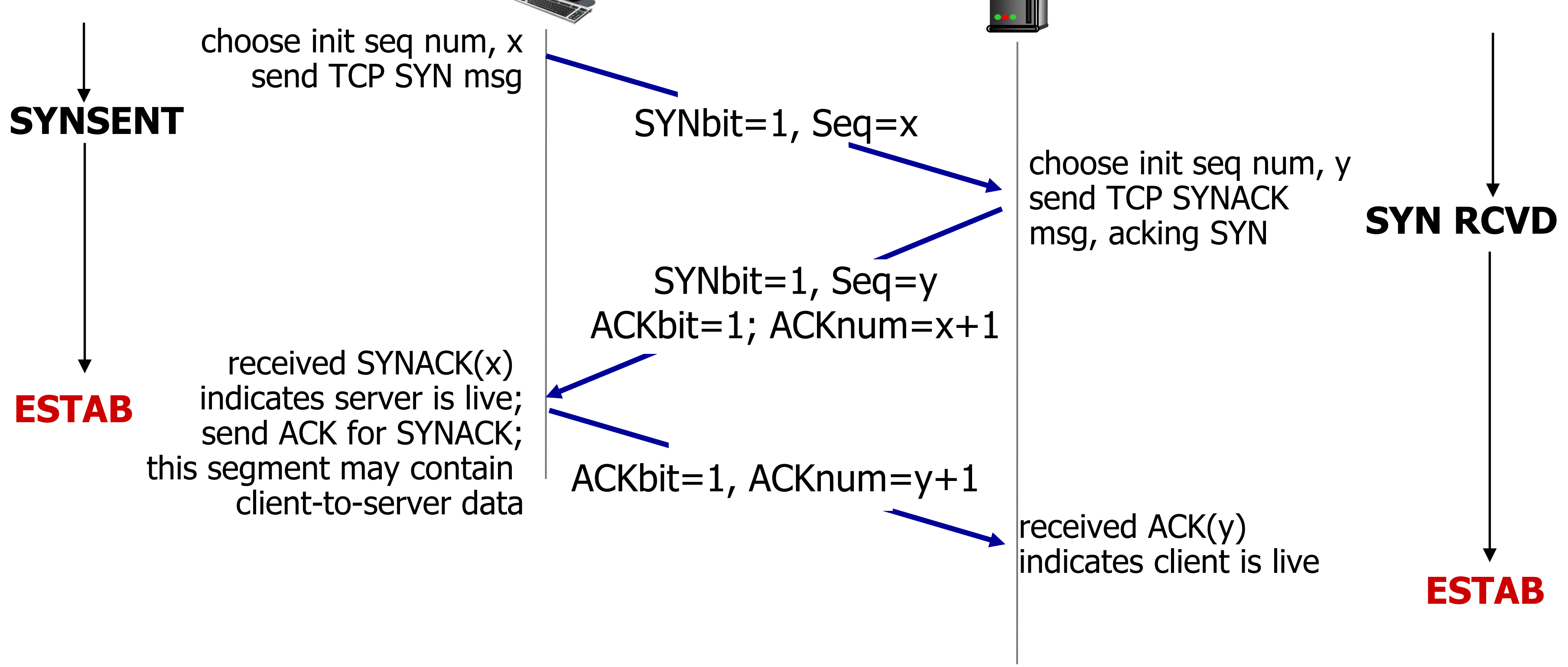
```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName,serverPort))
```

Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```



Closing a TCP 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

Chapter 3: 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

Congestion:

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

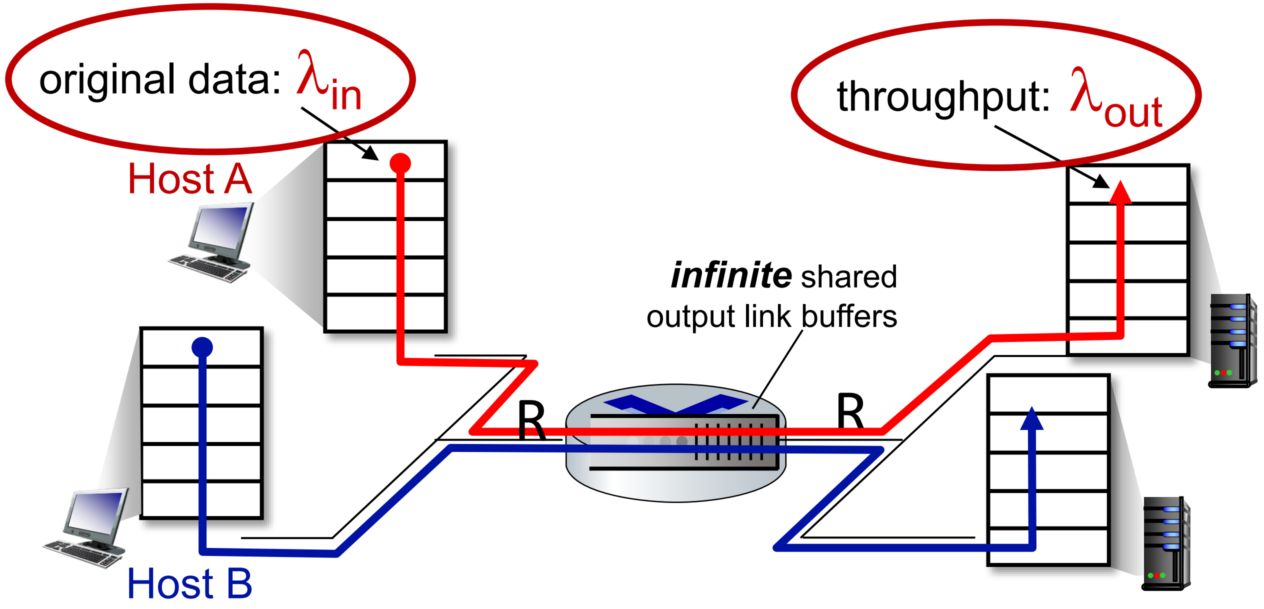
congestion control: **too many senders,** sending too fast

flow control: **one sender** too fast for **one receiver**

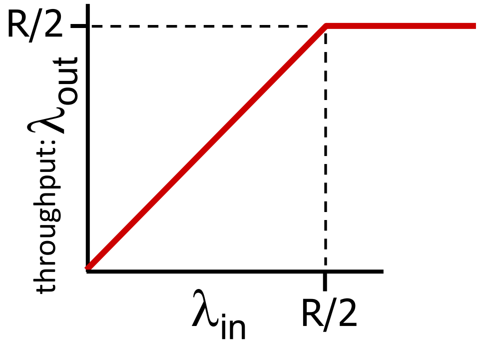
Causes/costs of congestion: scenario 1

Simplest scenario:

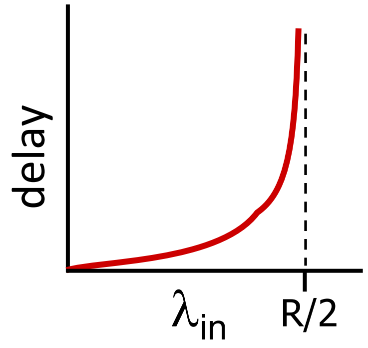
- one router, infinite buffers
- input, output link capacity: R
- two flows
- **no retransmissions** needed



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



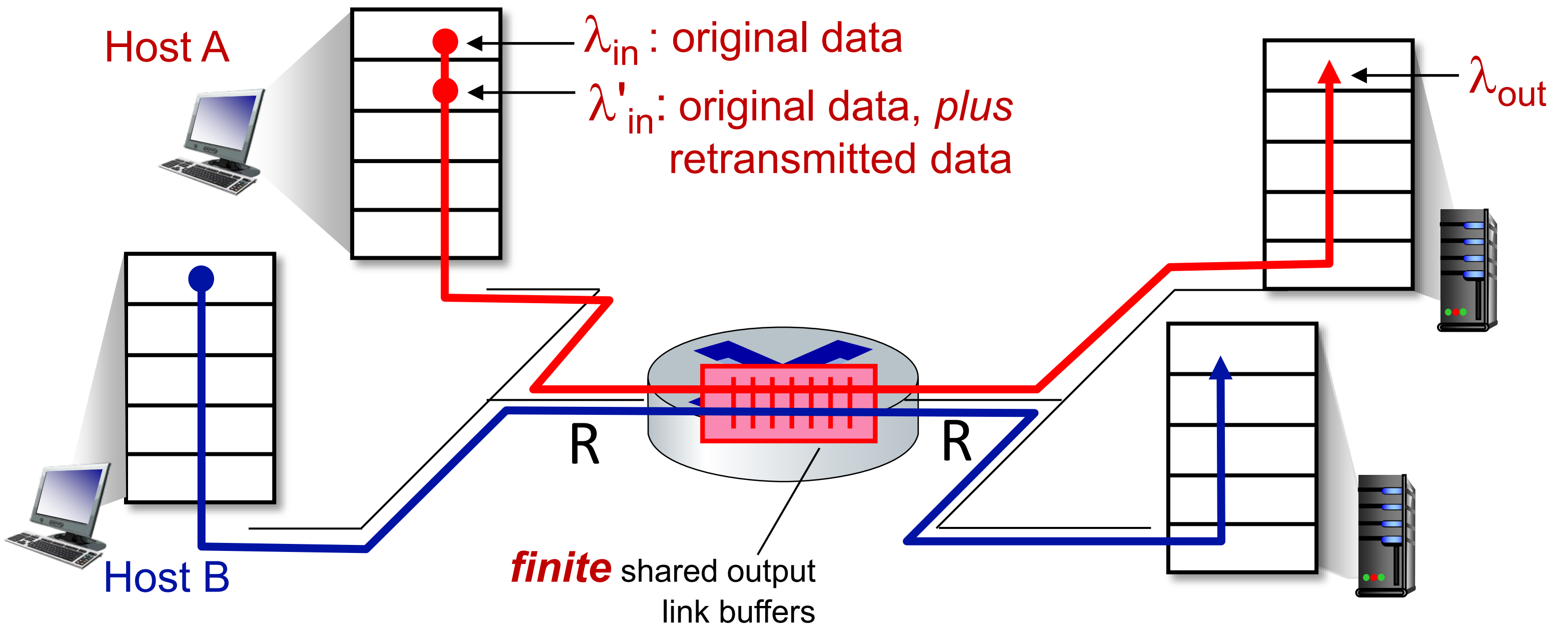
maximum per-connection **throughput**: R/2



large **delays** as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

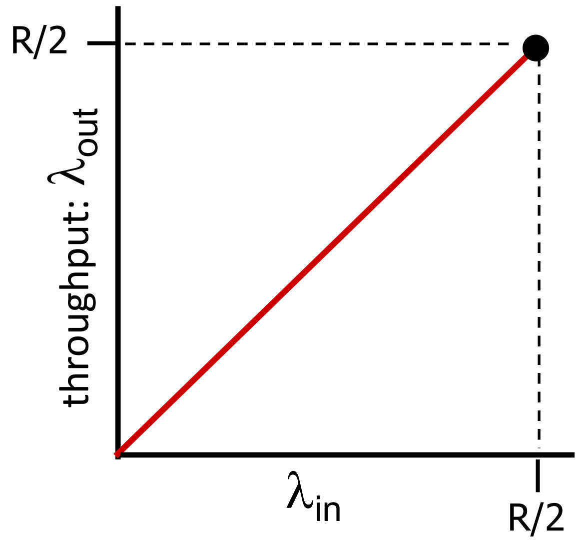
- one router, ***finite*** buffers
- sender **retransmits** lost, timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

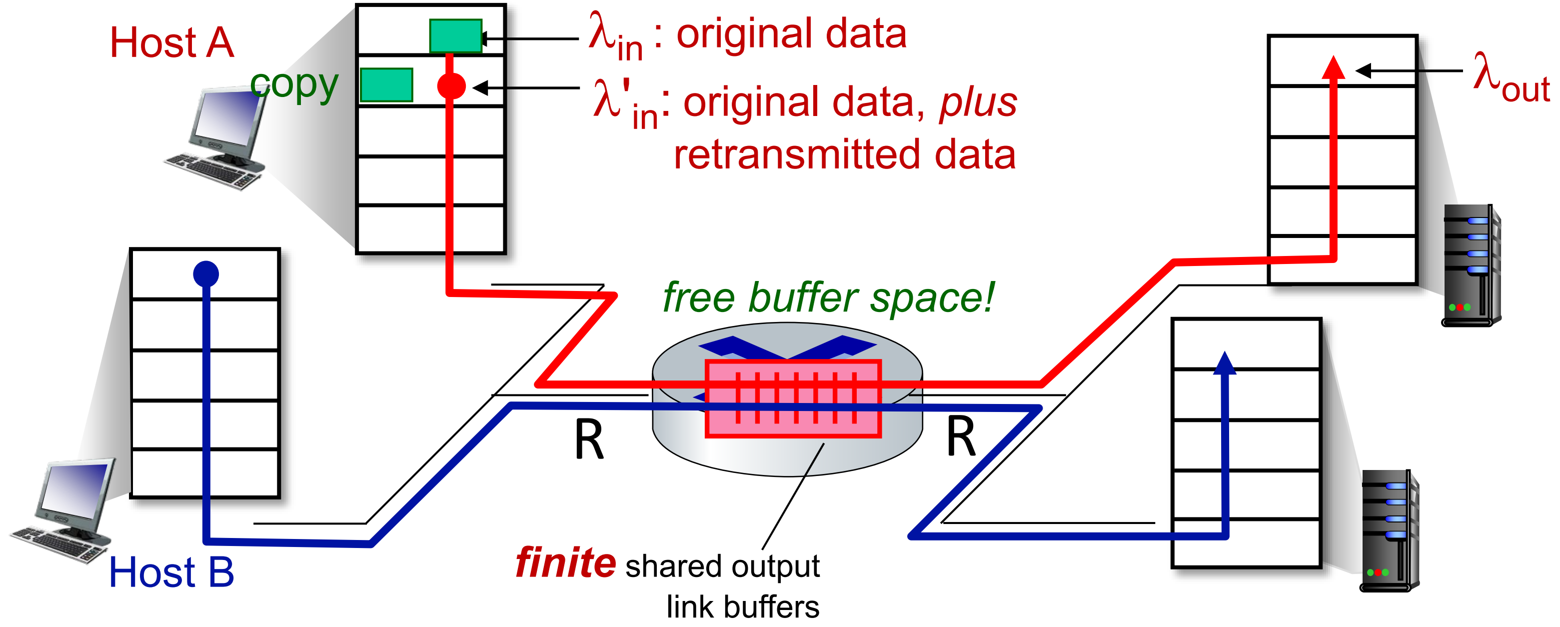


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

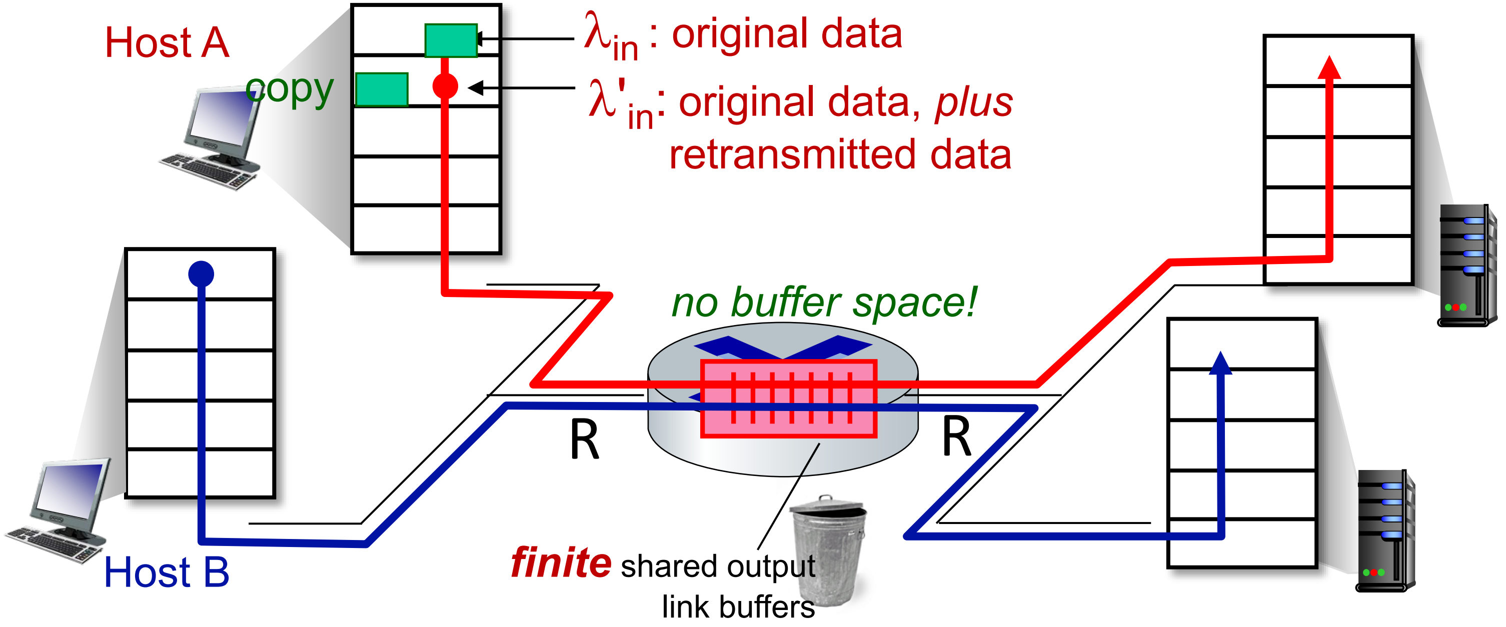




Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

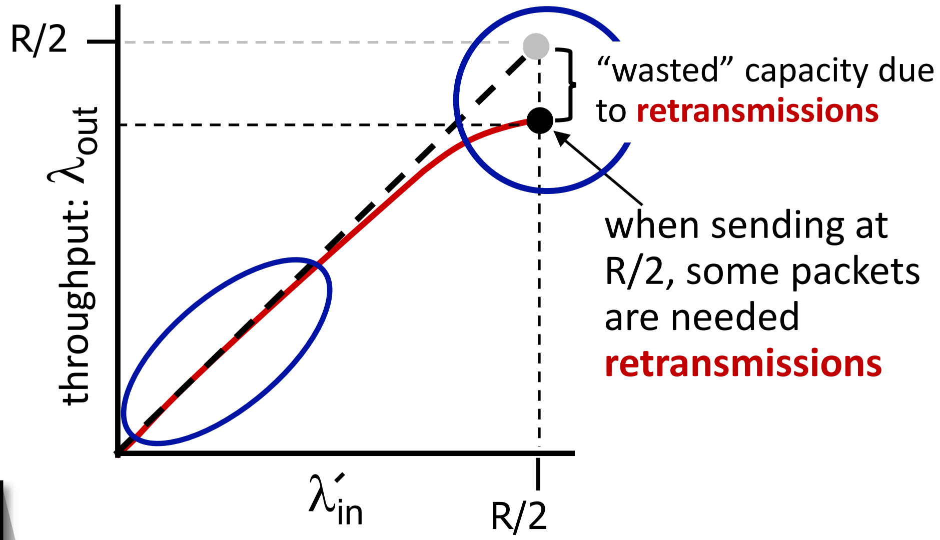
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost

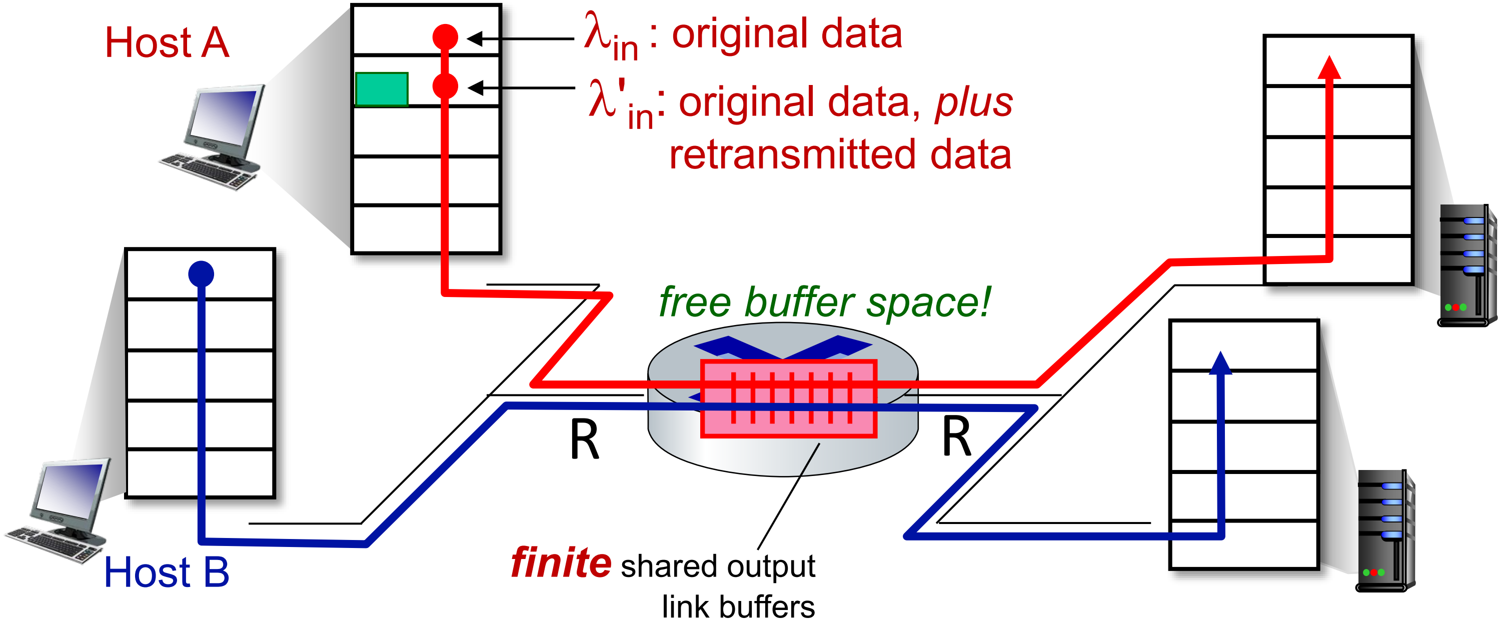


Causes/costs of congestion: scenario 2

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

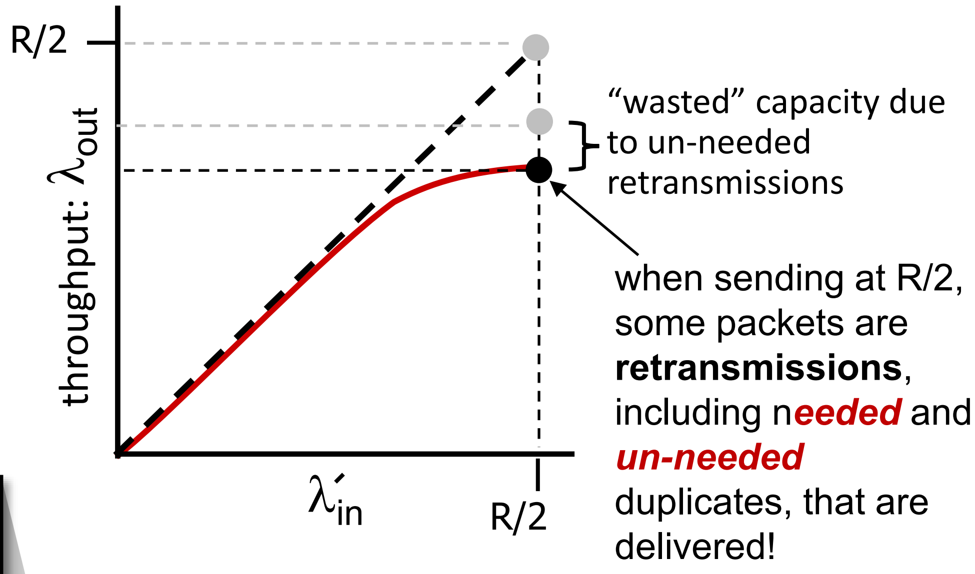


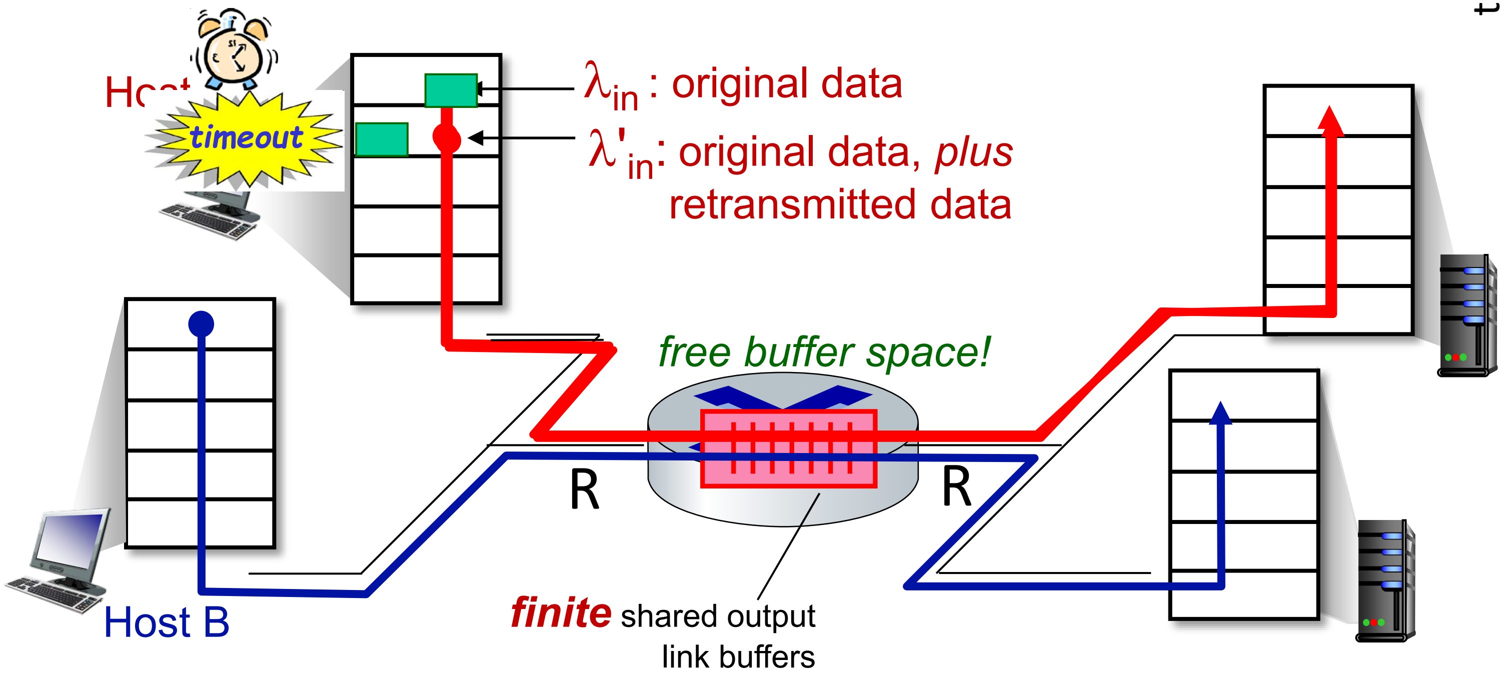


Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

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

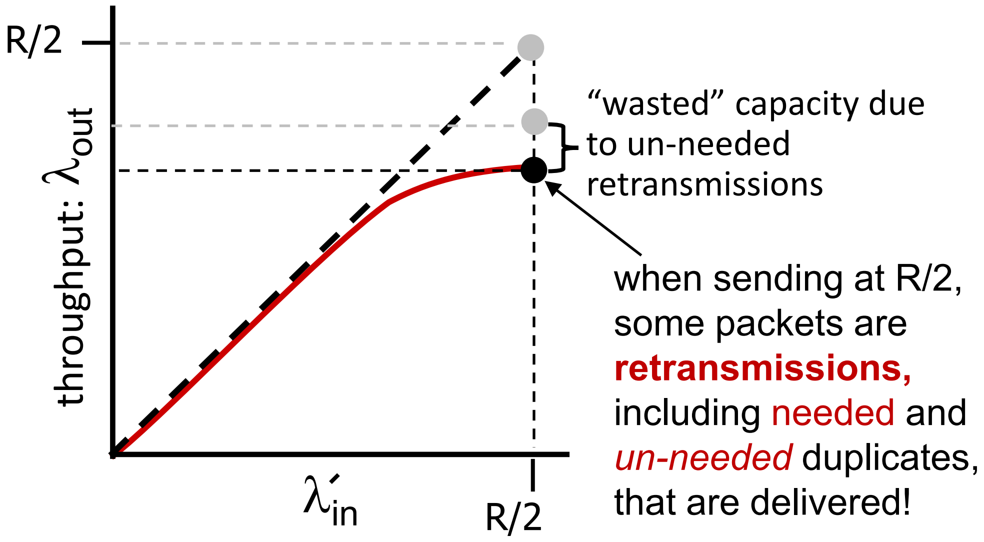




Causes/costs of congestion: scenario 2

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:

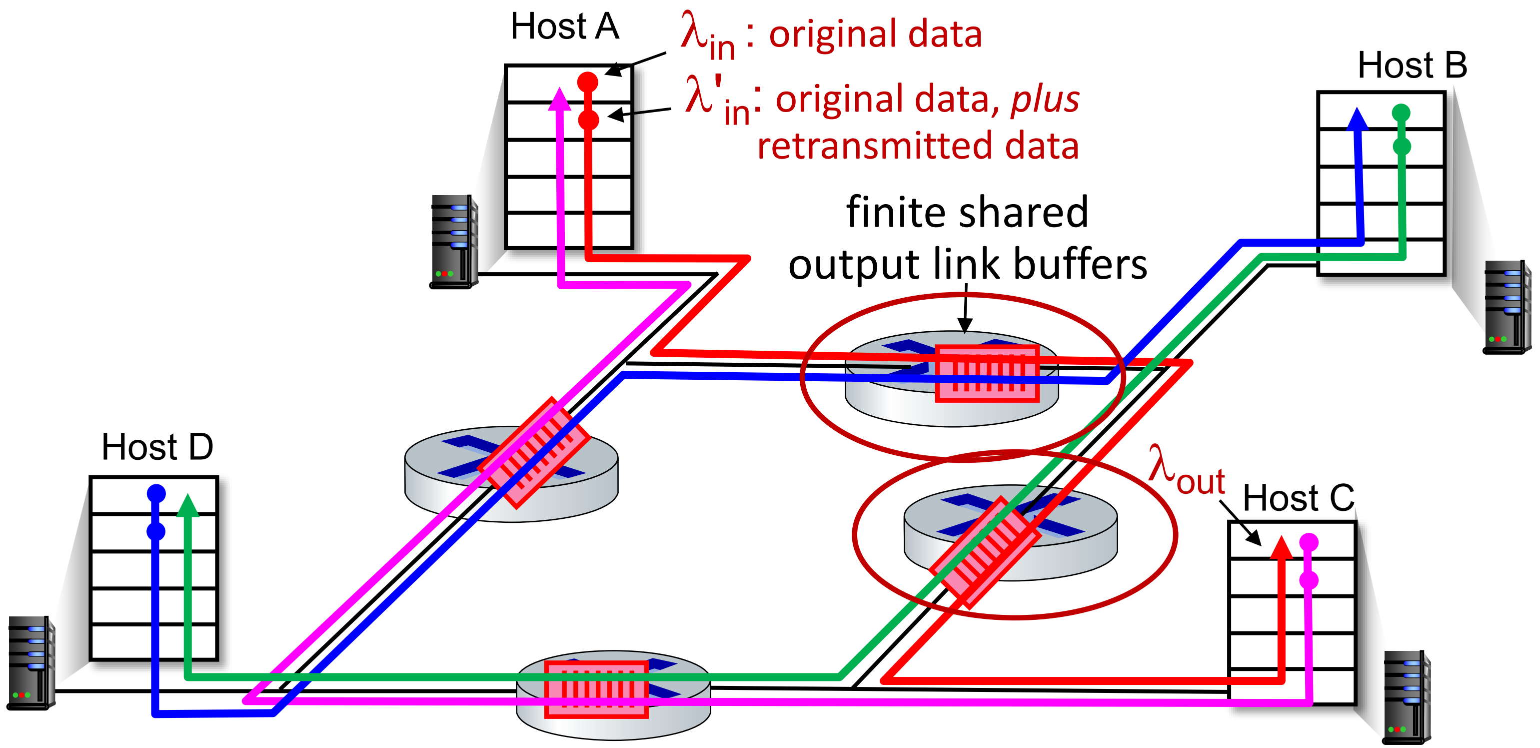
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3 (skipped)

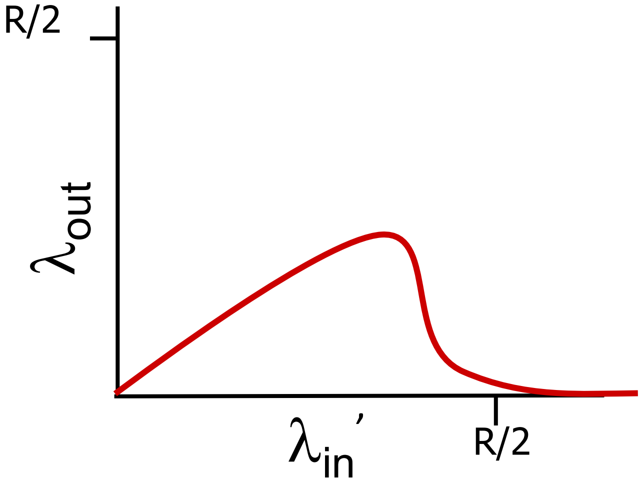
- *four* senders
- *multi-hop* paths
- timeout/retransmit

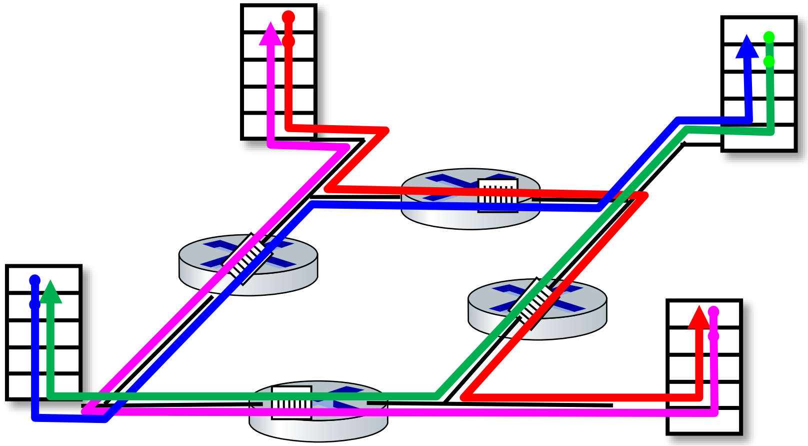
Q: what happens as λ_{in} and λ_{in}' increase ?

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



Causes/costs of congestion: scenario 3



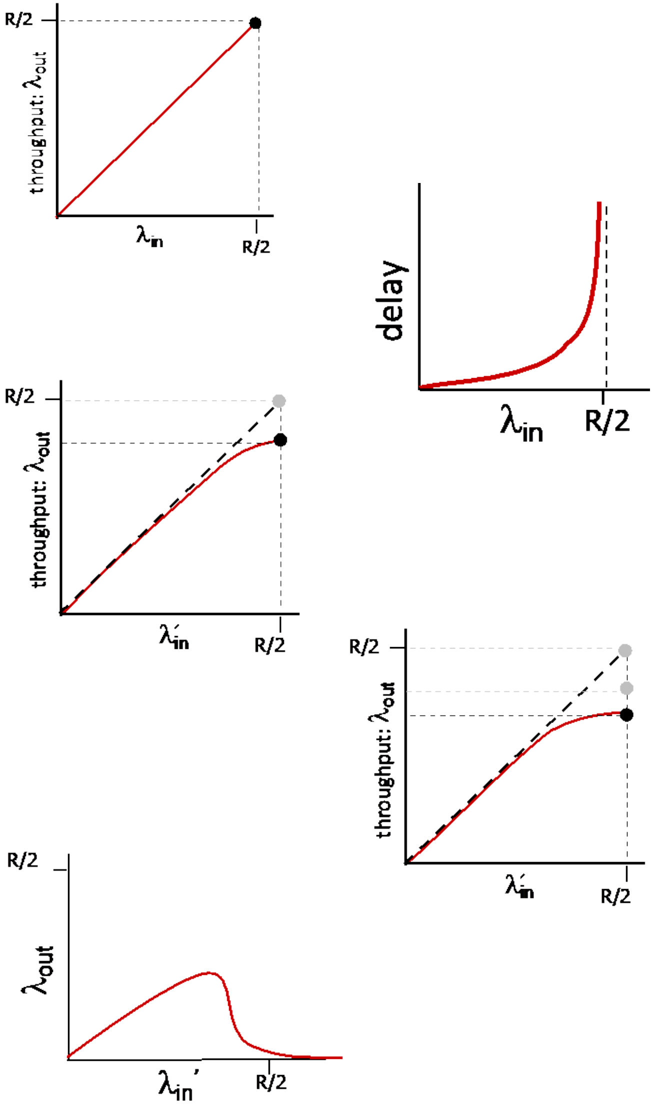


another "cost" of congestion:

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

Causes/costs of congestion: insights

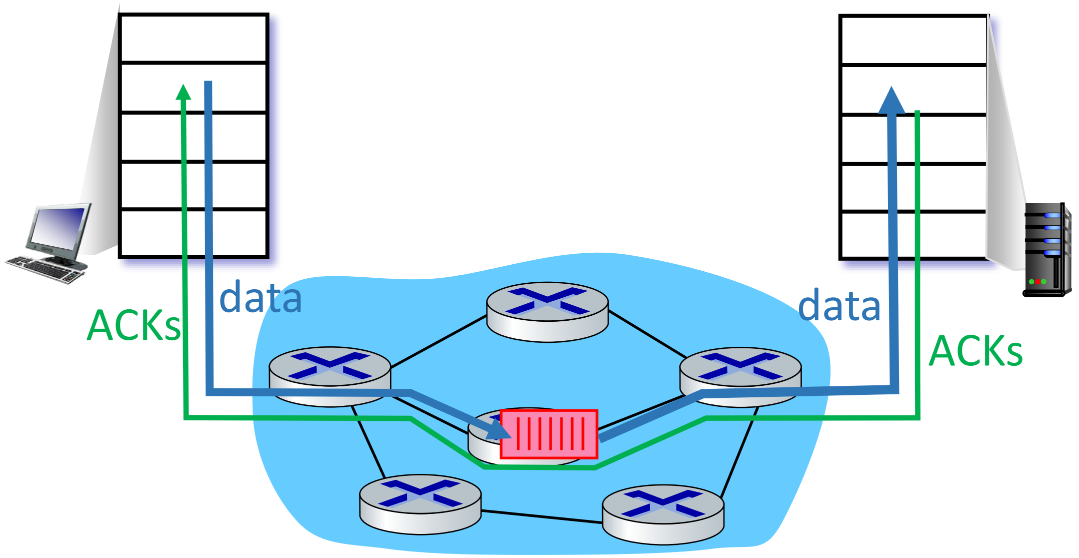
- **throughput** can never exceed capacity
- **delay** increases as capacity approached
- **loss/retransmission** decreases effective throughput
- **un-needed** duplicates further decreases effective throughput
- **upstream** transmission capacity / buffering wasted for packets lost downstream



Approaches towards congestion control

End-end congestion control:

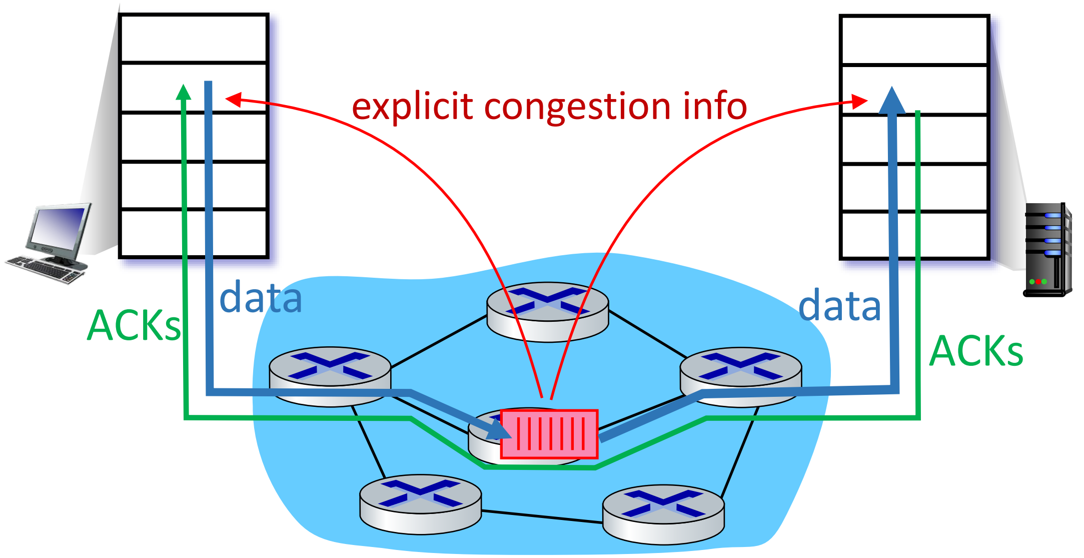
- no explicit feedback from network
- congestion ***inferred*** from observed loss, delay
- approach taken by **TCP**



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**
- TCP ECN, ATM, DECbit protocols



Chapter 3: 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

TCP congestion control: AIMD

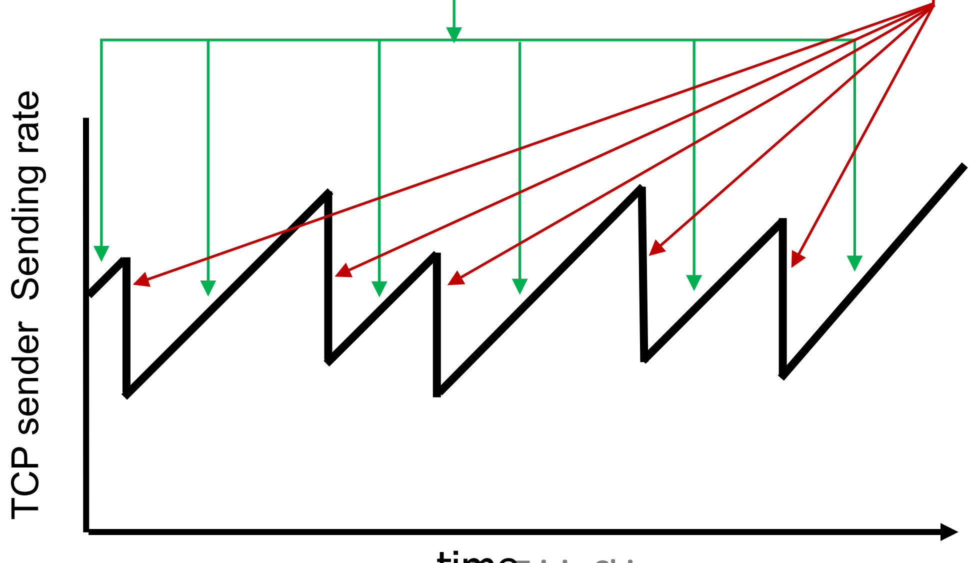
- *approach:* senders can increase **sending rate** until packet loss (congestion) occurs, then decrease sending rate on loss event

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

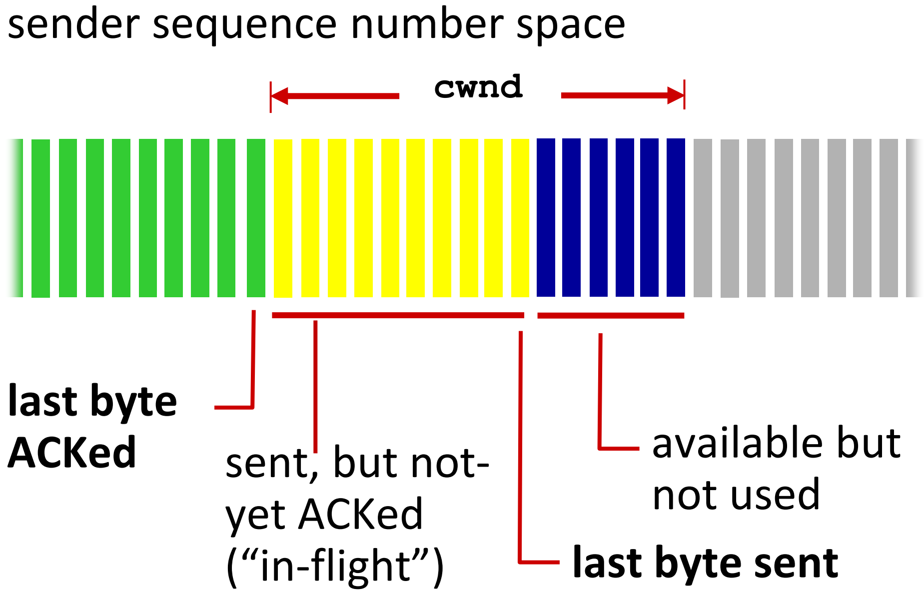
Multiplicative decrease detail: sending rate is

- 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**:

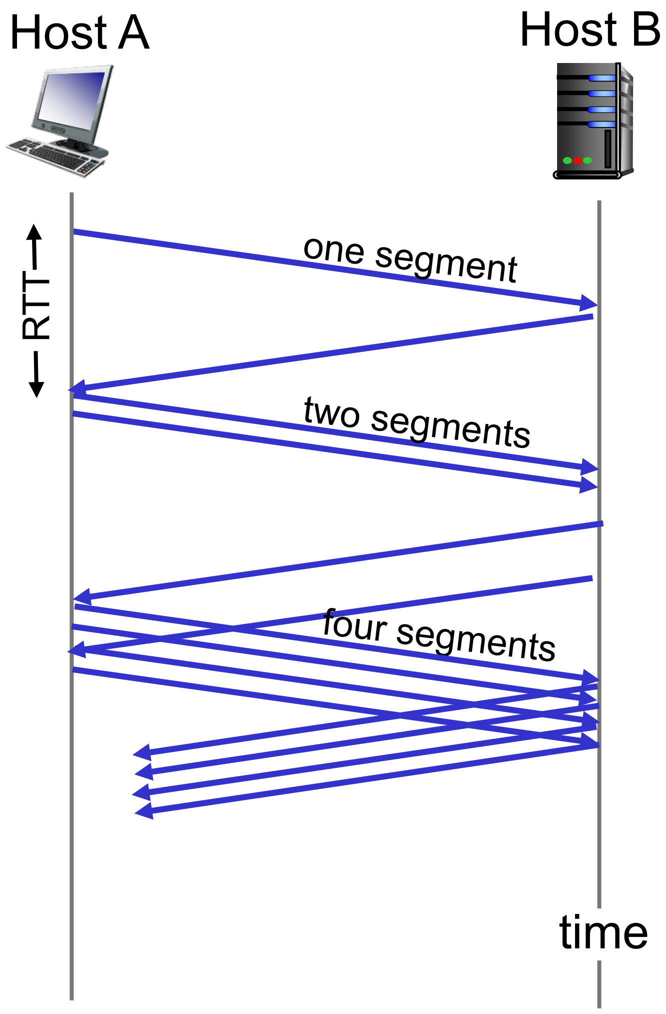
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission: `LastByteSent- LastByteAcked` $\leq$ `cwnd`
- `cwnd` is **dynamically adjusted** in response to **observed** network congestion (implementing TCP congestion control)

TCP slow start

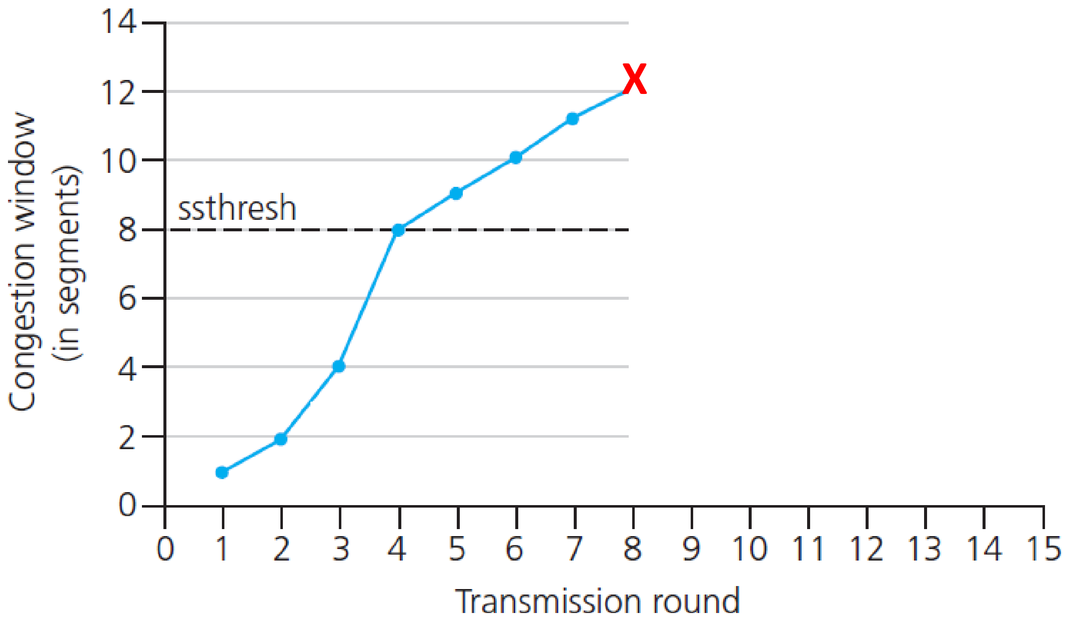
- when connection begins, increase rate **exponentially** until **first loss event**:
  - initially **`cwnd`** = 1 MSS
  - **double `cwnd` every RTT**
  - done by incrementing **`cwnd`** for **every ACK received**
- *summary:* initial rate is slow, but ramps up **exponentially** fast



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 **last timeout**.



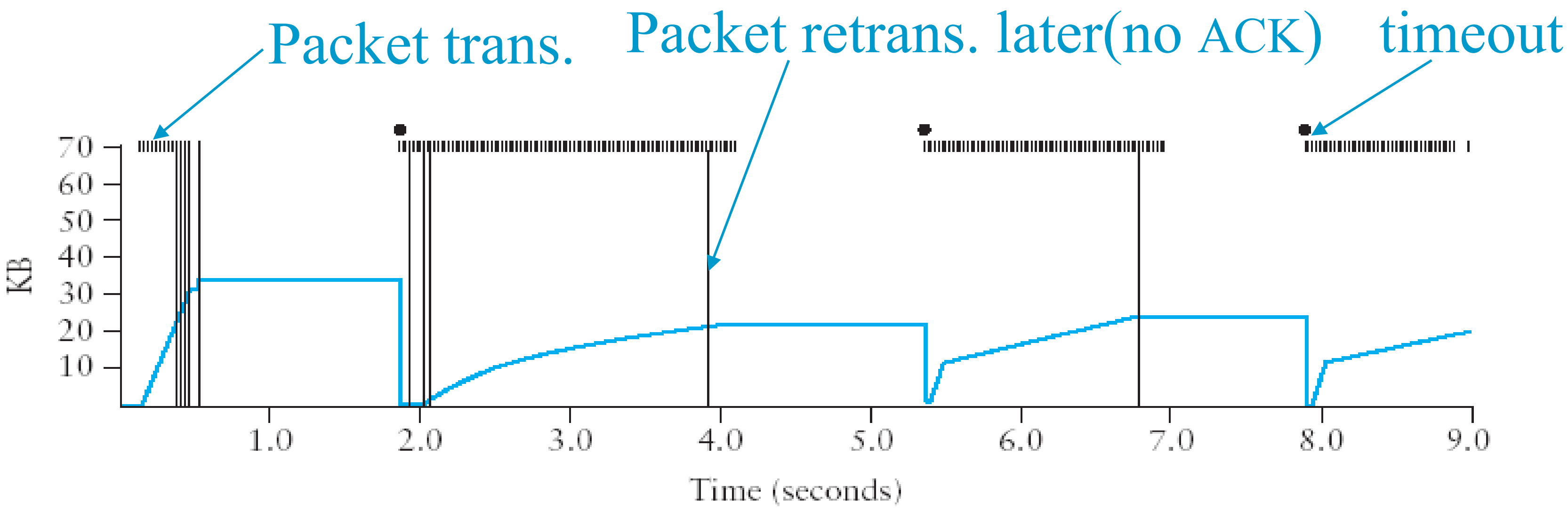
Implementation:

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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Slow Start (cont., Reference)

- **Exponential** growth, slower than **all at once** (original TCP)
- Used… (https://book.systemsapproach.org/congestion/tcpcc.html#)
  - when **first starting** connection
  - when connection goes **dead** waiting for timeout (more knowledge)
- Trace of TCP CongestionWindow: **interplay** of "slow start" & "AIMD"

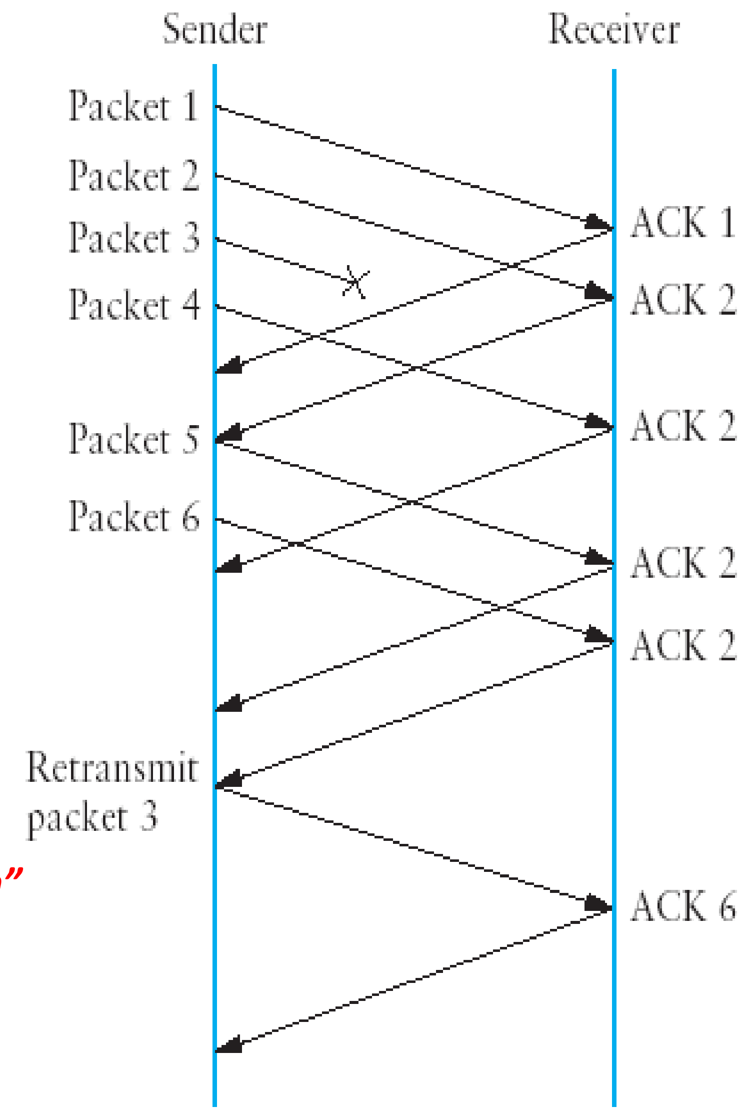


Courtesy of "*Computer Network: A System Approach*" by Larry Peterson and Bruce Davie (Chapter 6.3)

- Problem: lose up to **half** a `CongestionWindow`'s worth of data

Fast Retransmit and Fast Recovery (Reference)

- Problem:
  - **coarse-grain** TCP timeouts lead (flat part in previous figure) to **idle periods**
    - **`EffWin = MaxWin - (LastByteSent - LastByteAcked)`**
- **Fast** retransmit:
  - use **duplicate** ACKs to trigger retransmission,
  - until the sender sees some # of duplicated ACKs, it then **retransmits** the missing packet.
  - In TCP, sender waits till **three** duplicated ACKs
- Fast Recovery

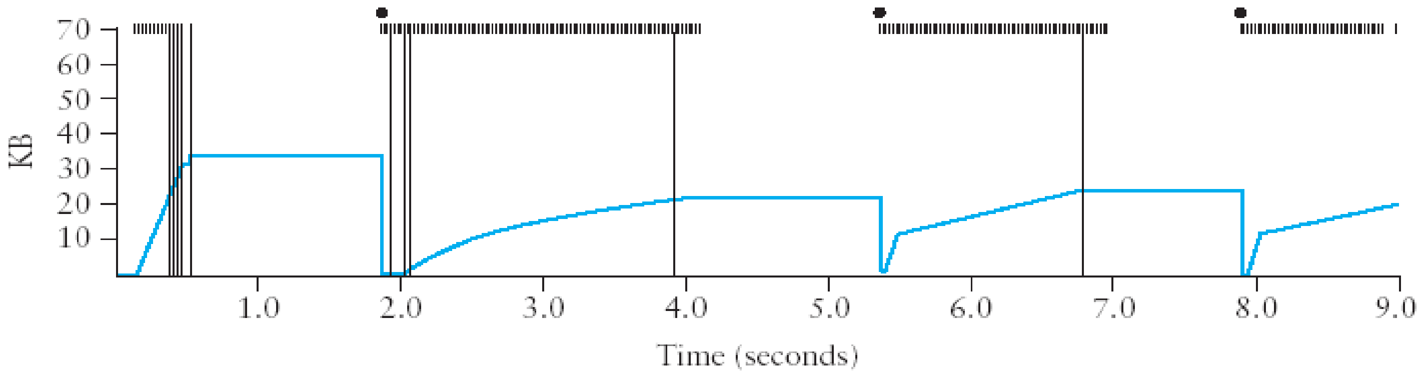


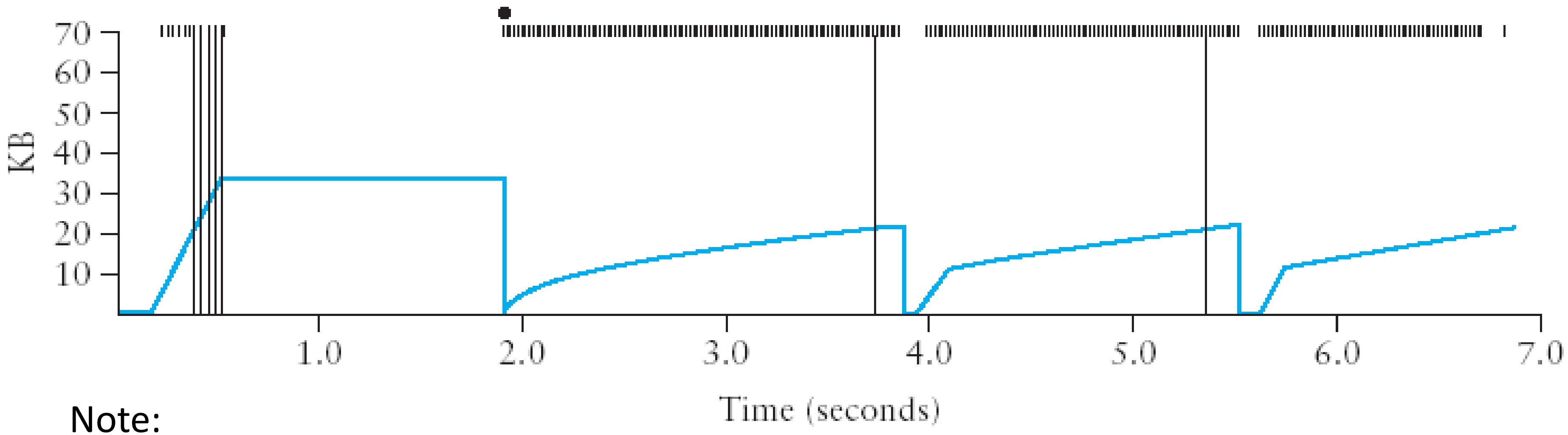
Courtesy of "*Computer Network: A System Approach*" by Larry Peterson and Bruce Davie (Chapter 6.3)

(https://book.systemsapproach.org/congestion/tcpcc.html#)

Results

Too aggressive, once lost, all lost, no enough duplicated ACKs to trigger fast retransmission



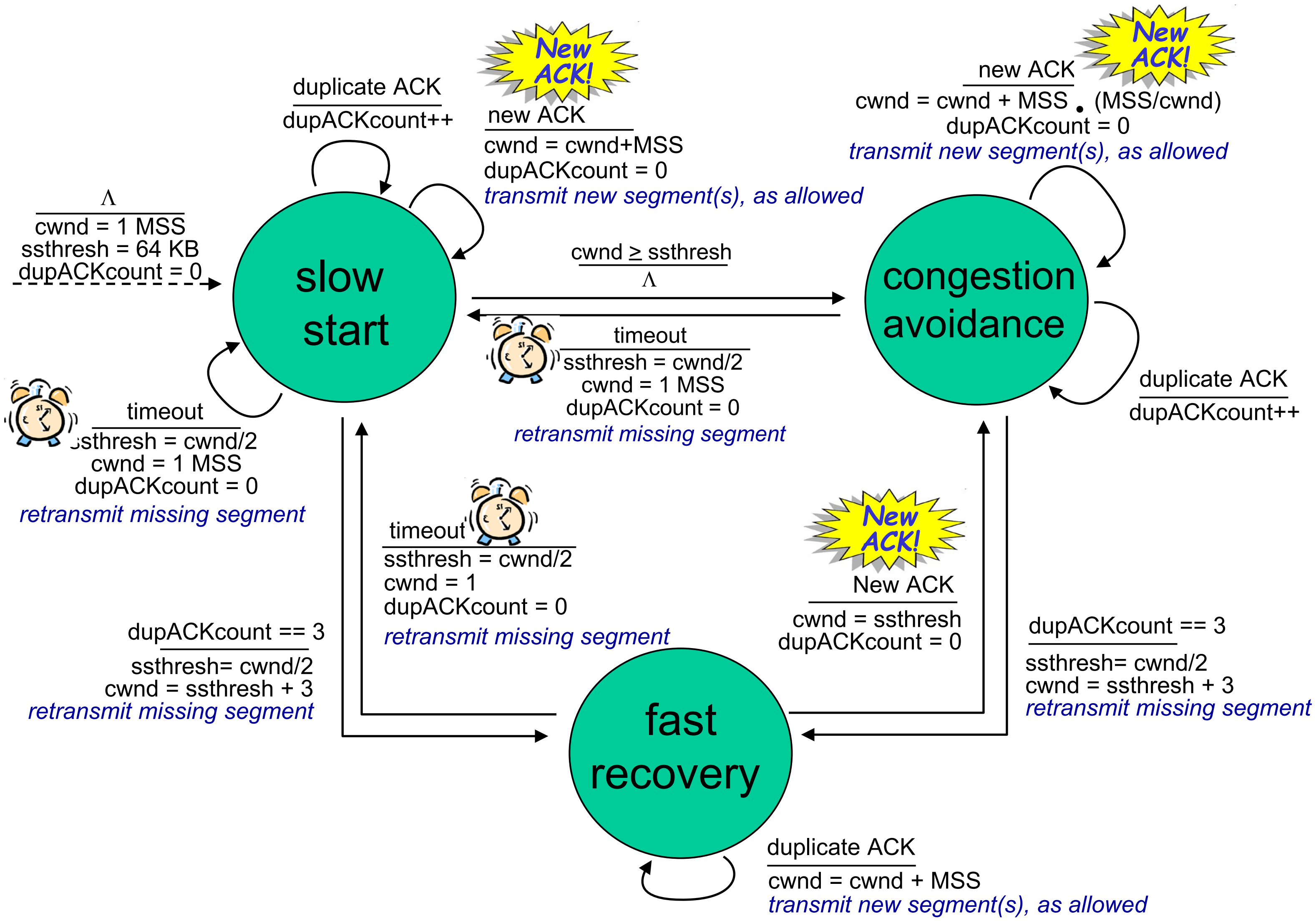


Courtesy of "*Computer Network: A System Approach*" by Larry Peterson and Bruce Davie (Chapter 6.3)

Note:

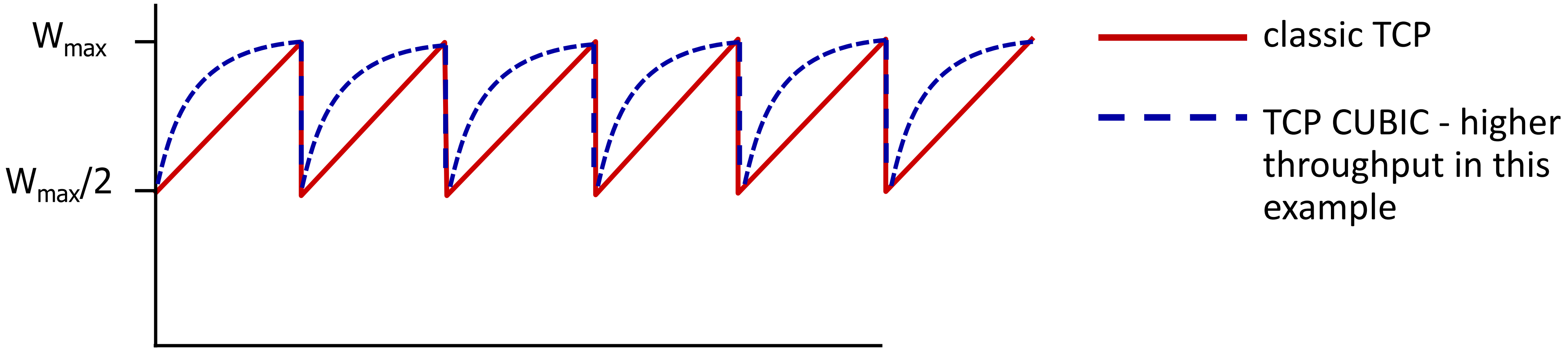
- For a **small window size**, there will not be enough packets in transit to cause enough duplicate ACKs to be delivered;
- Given the current 64KB maximum advertised window size, TCP's fast retransmit mechanism is able to detect up to **three** dropped packets per window in practice.
- **AIMD -> "Slow" Start -> Fast Retransmit Fast Recovery: improvement**

Summary: TCP congestion control (skipped)



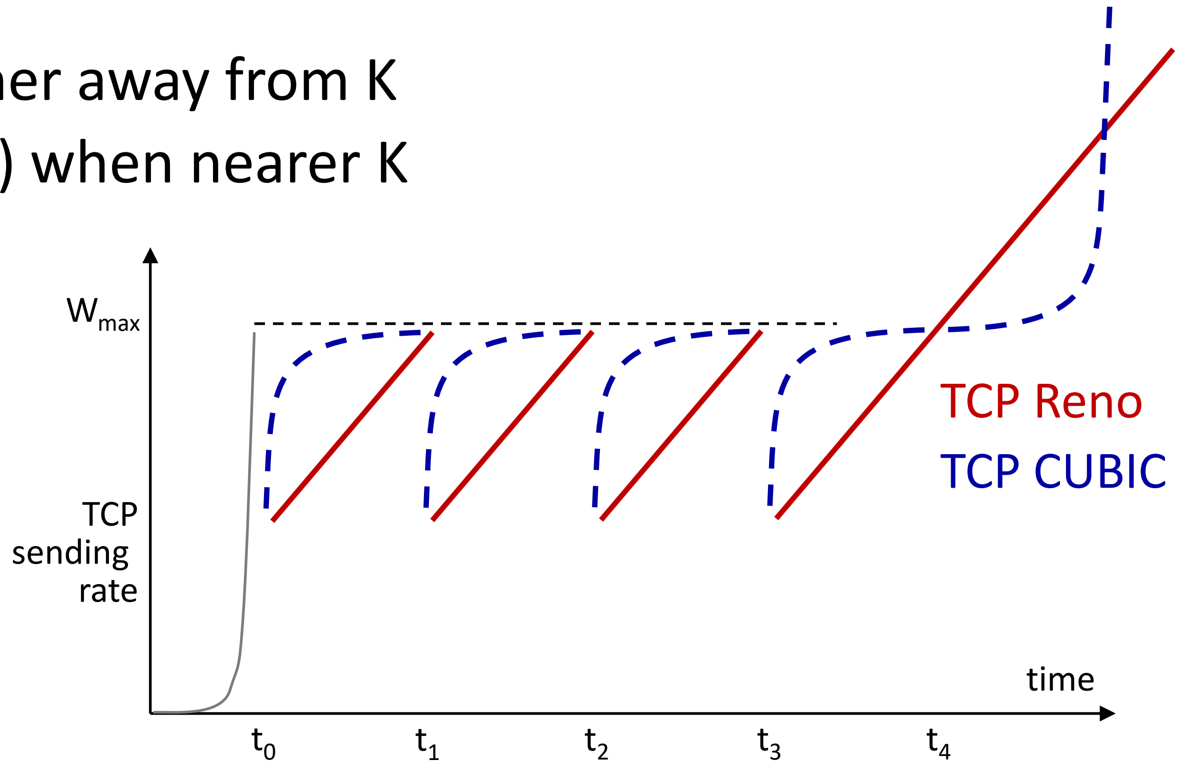
TCP CUBIC (not required)

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: **sending rate** at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to W_{max} *faster*, but then approach W_{max} more *slowly*



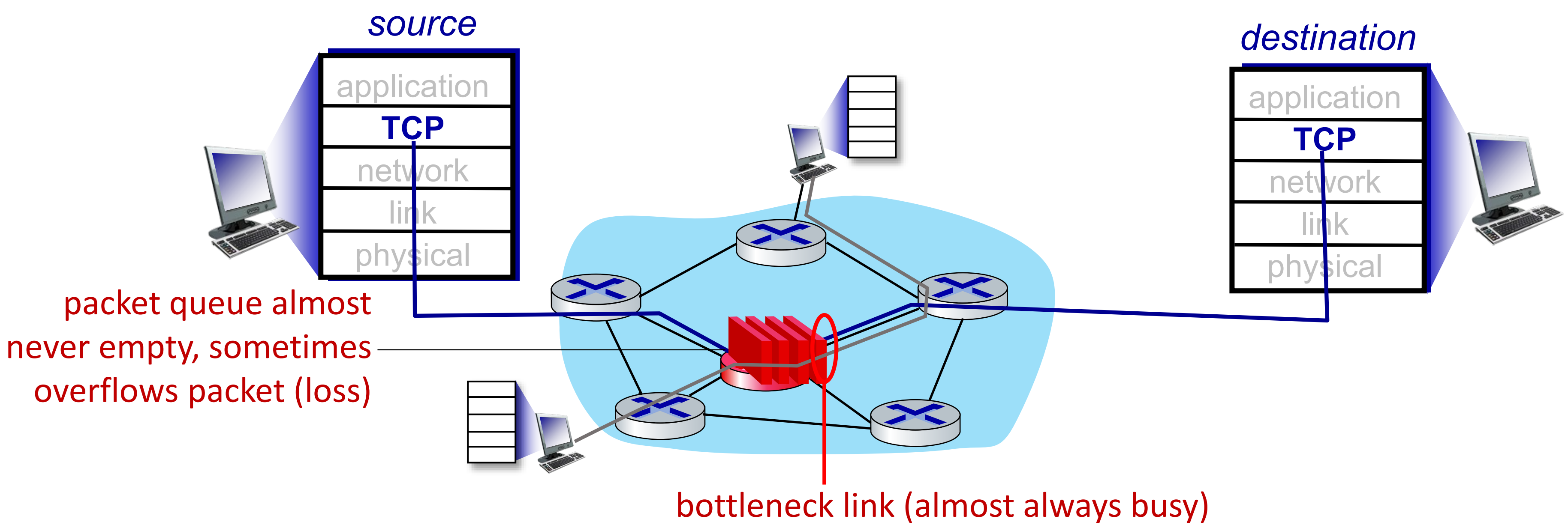
TCP CUBIC (not required)

- K: **point in time** when TCP window size will reach W_{max}
  - K itself is tuneable
- **increase W** as a function of the *cube* of the distance between **current time** and **K**
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in **Linux**, most popular TCP for popular Web servers



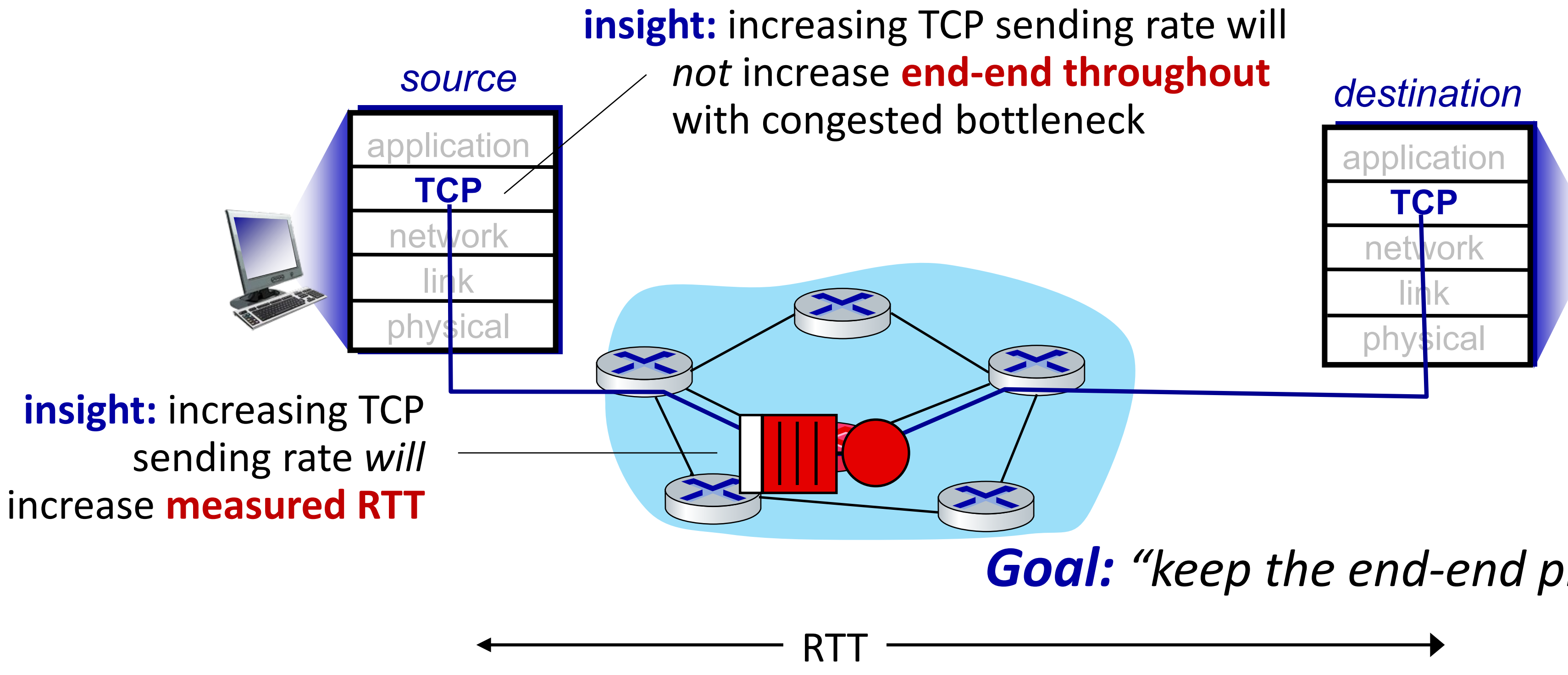
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's **sending rate** until packet loss occurs at some router's output: the *bottleneck link*



TCP and the congested "bottleneck link"

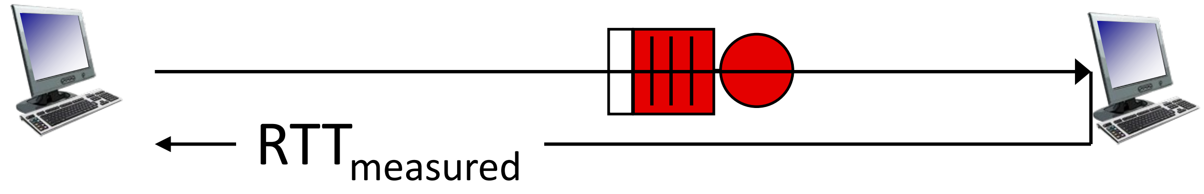
- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the ***bottleneck link***
- understanding congestion: useful to focus on congested bottleneck link



Goal: *"keep the end-end pipe just full, but not fuller"*

Delay-based TCP congestion control (TCP Vegas)

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but **avoid high delays/buffering**



$$\textbf{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

- RTT_{min} : minimum observed RTT (**uncongested path**)
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}

if measured throughput "very close" to **uncongested throughput**
    increase `cwnd` linearly /* since path not congested */
else if measured throughput "far below" **uncongested throughout**
    decrease `cwnd` linearly /* since path is congested */

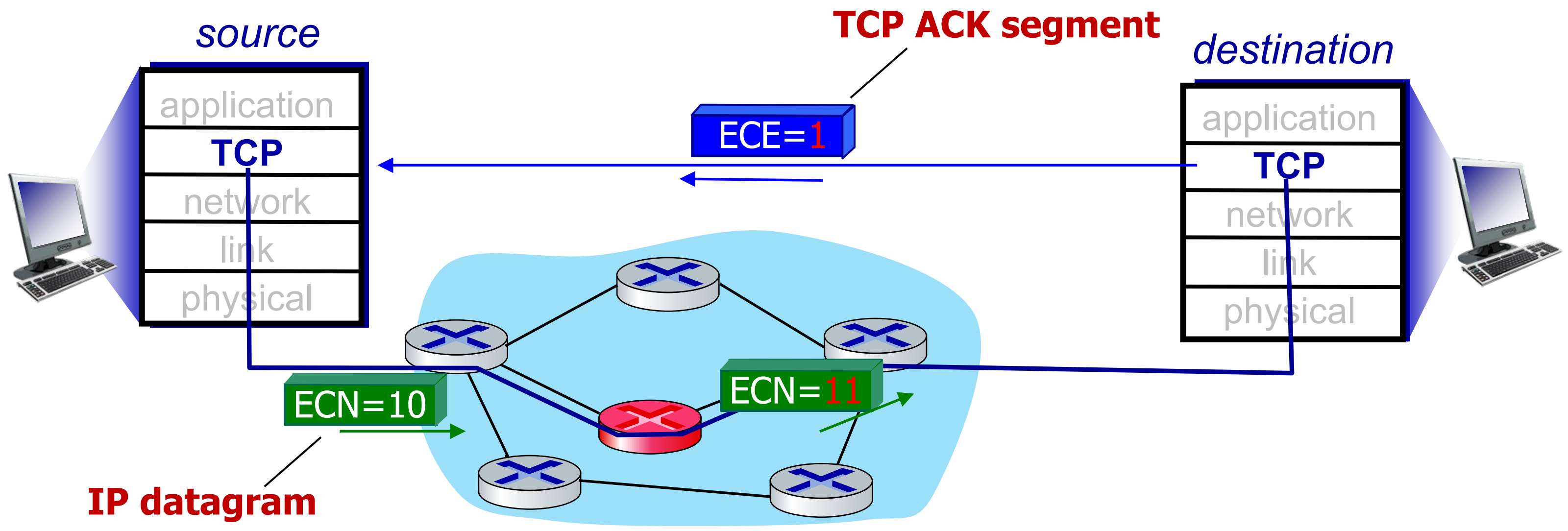
Delay-based TCP congestion control

- congestion control without inducing/forcing loss
- maximizing throughput ("keeping the pipe just full… ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - Bottleneck Bandwidth and Round-trip propagation time (**BBR**) deployed on **Google's** (internal) backbone network

Explicit congestion notification (ECN)

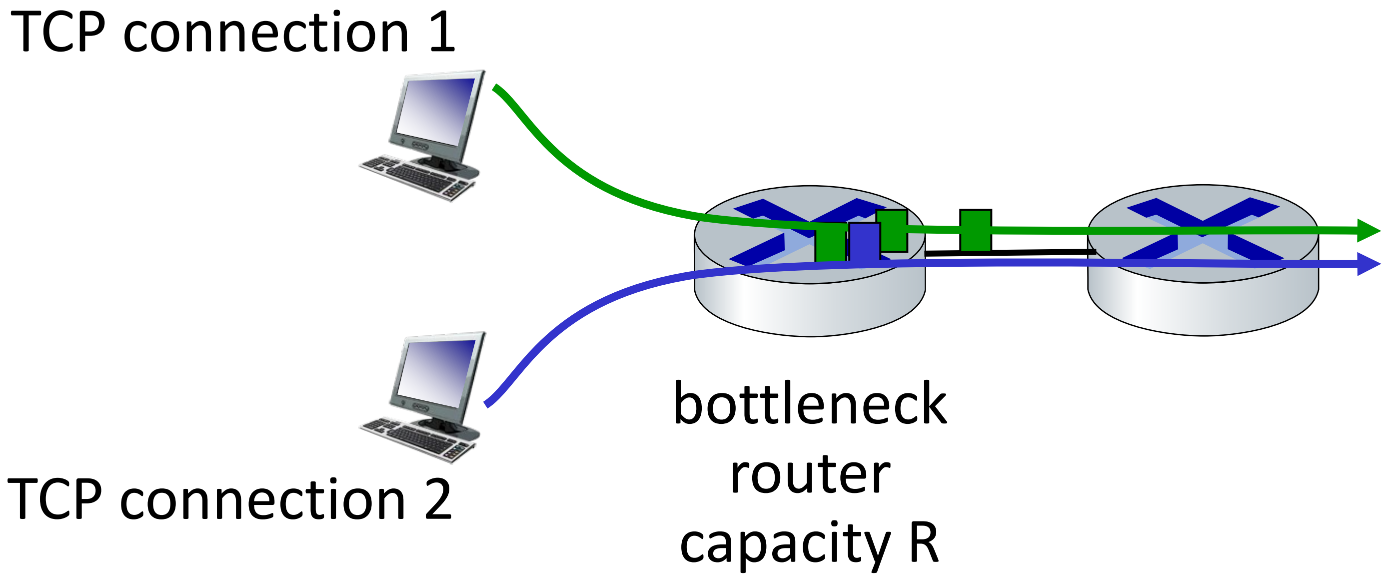
TCP deployments often implement *network-assisted* congestion control:

- 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 fairness

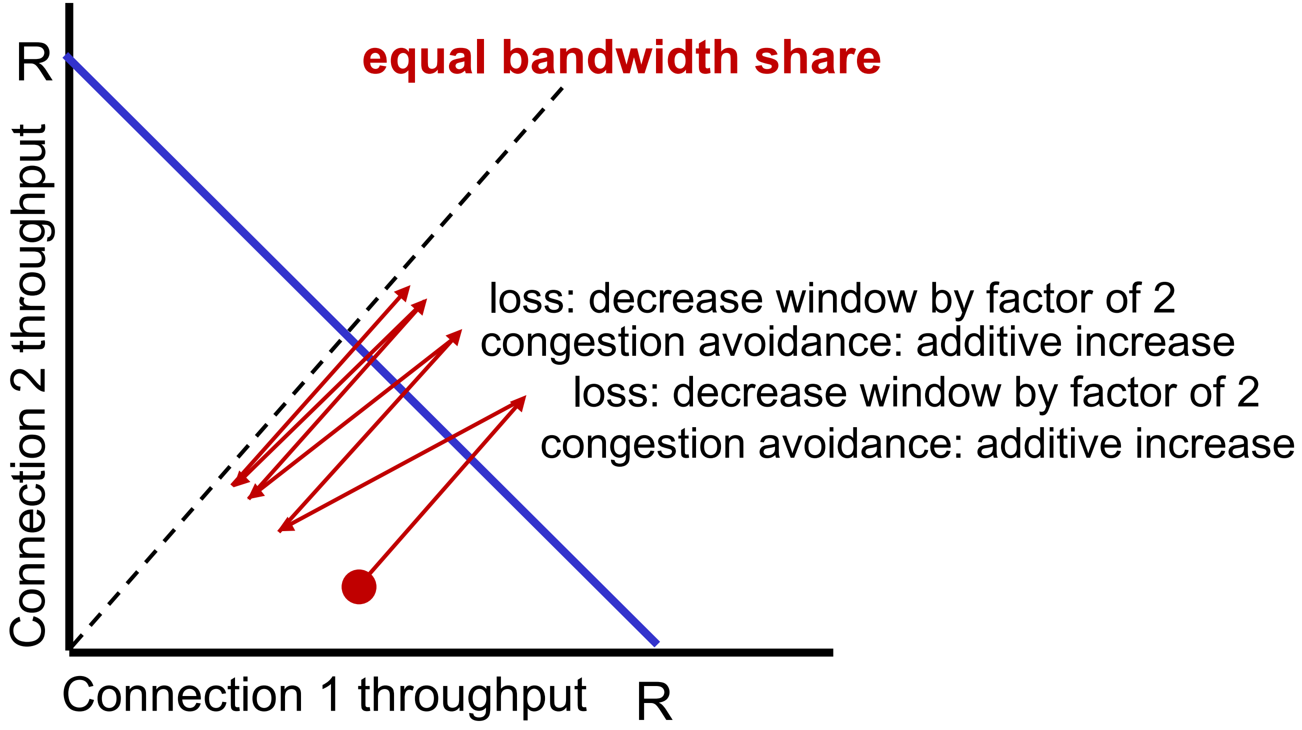
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

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



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be “fair”?

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
- there is **no** “Internet police” policing use of **congestion control**

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

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 (***reading assignment***)

Evolving transport-layer functionality (reading assignment)

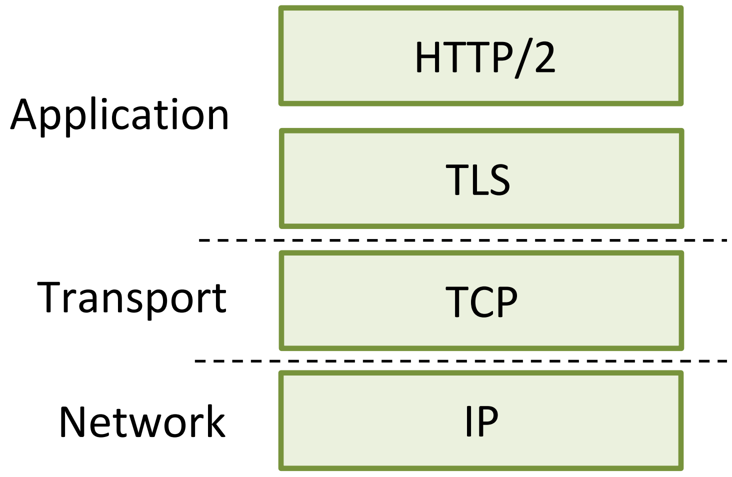
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

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

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



HTTP/2 over TCP

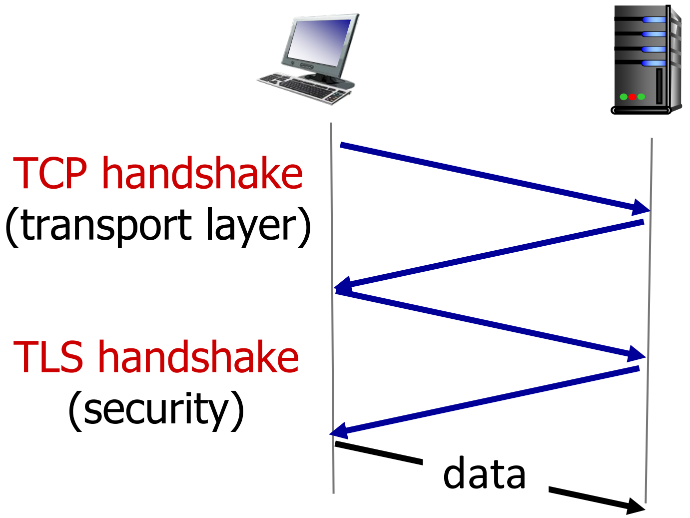
QUIC: Quick UDP Internet Connections

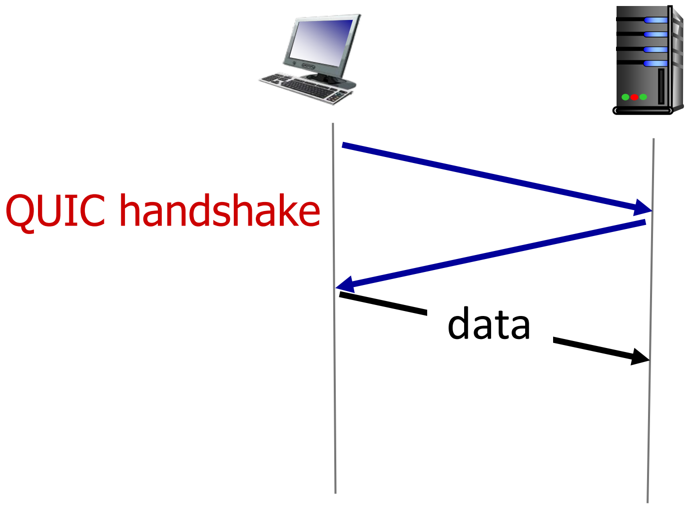
adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment





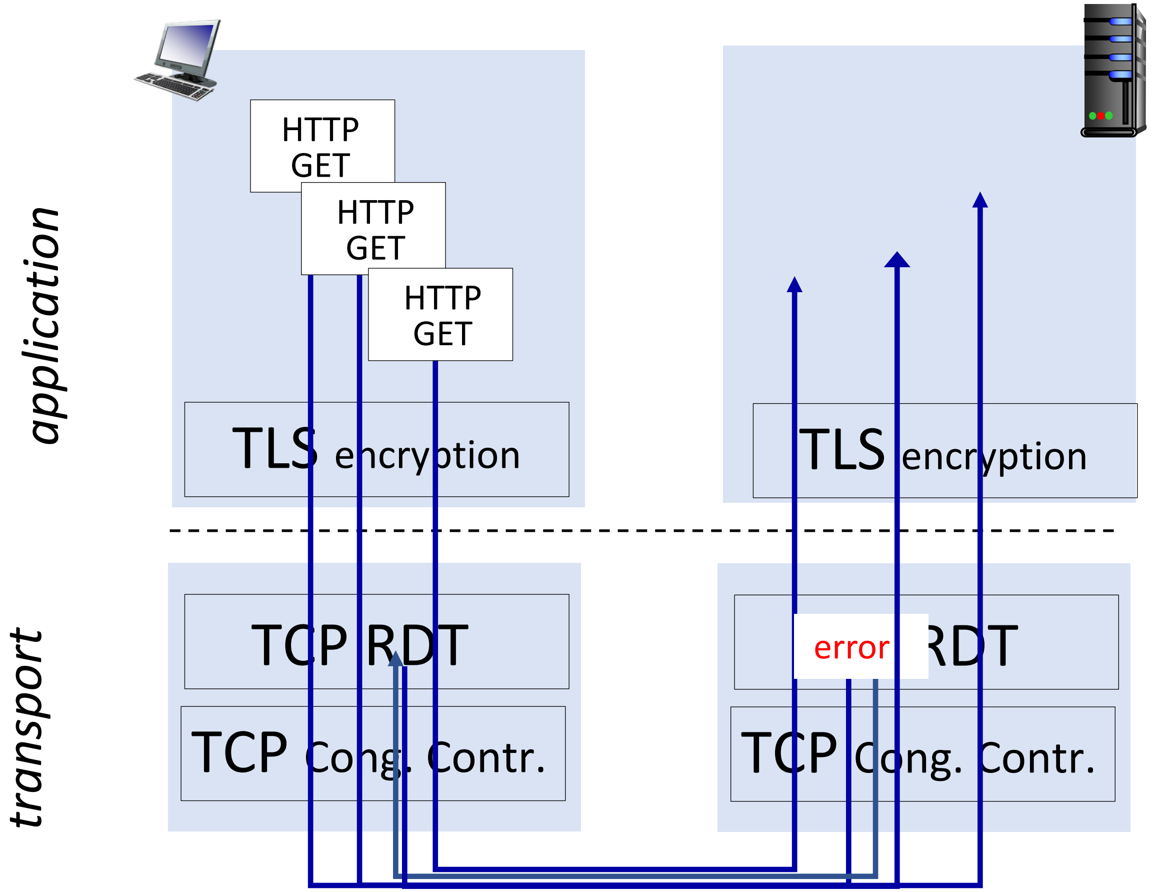
TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

Chapter 3: summary

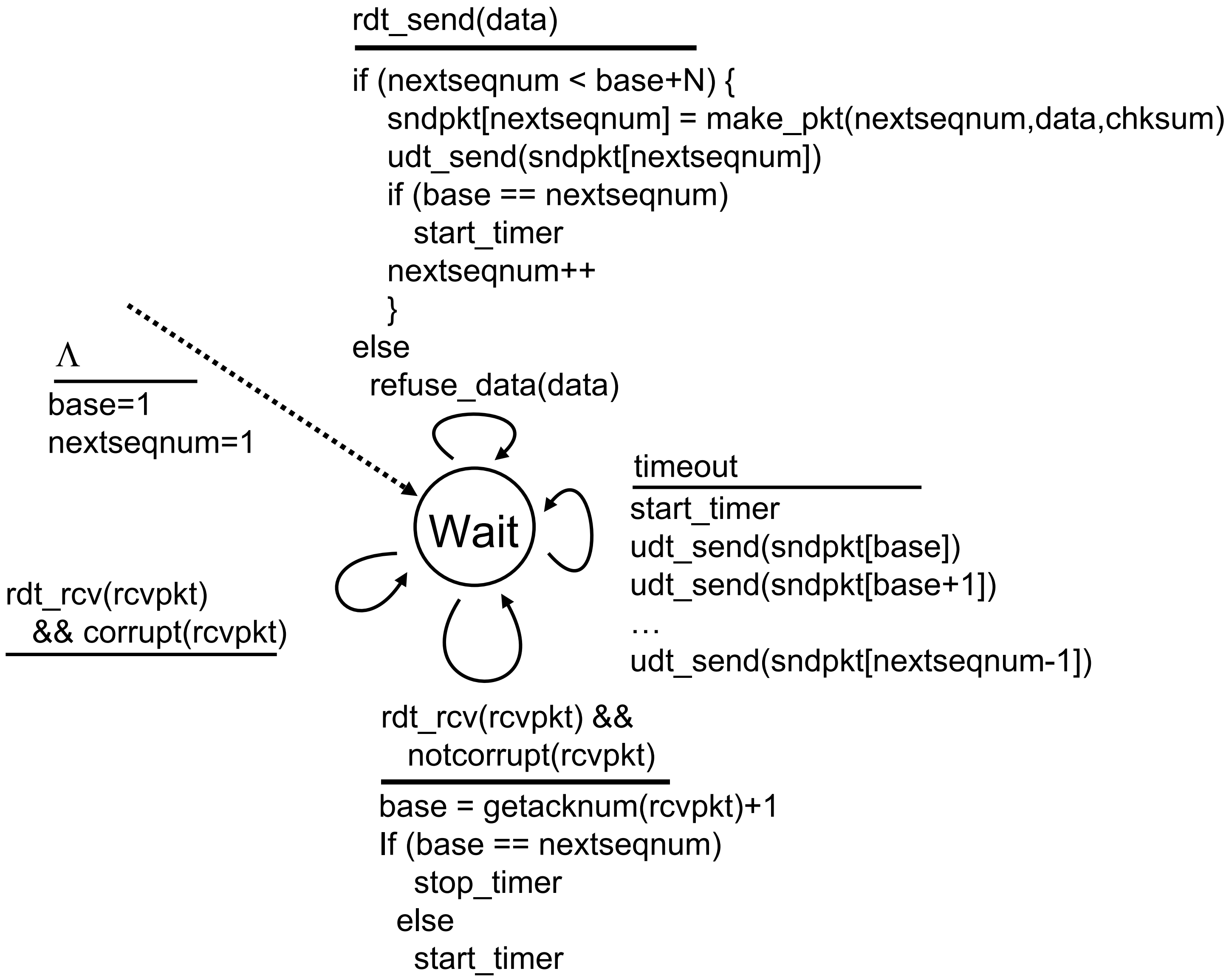
- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

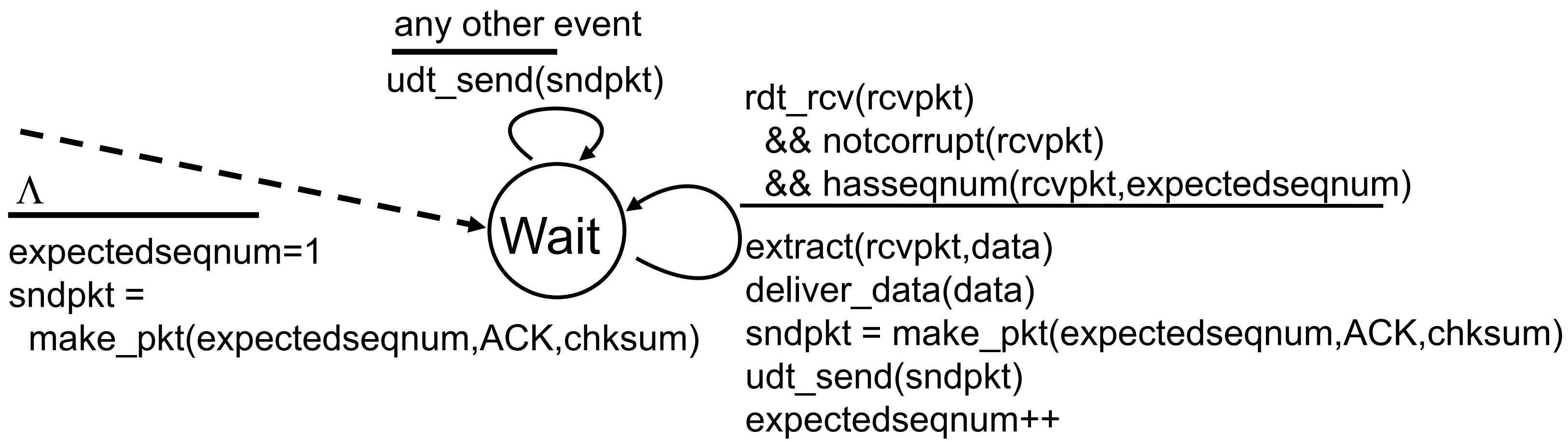
- leaving the network **"edge"** (application, transport layers)
- into the network **"core"**
- two network-layer chapters:
  - data plane
  - control plane

Additional Chapter 3 slides

Go-Back-N: sender extended FSM



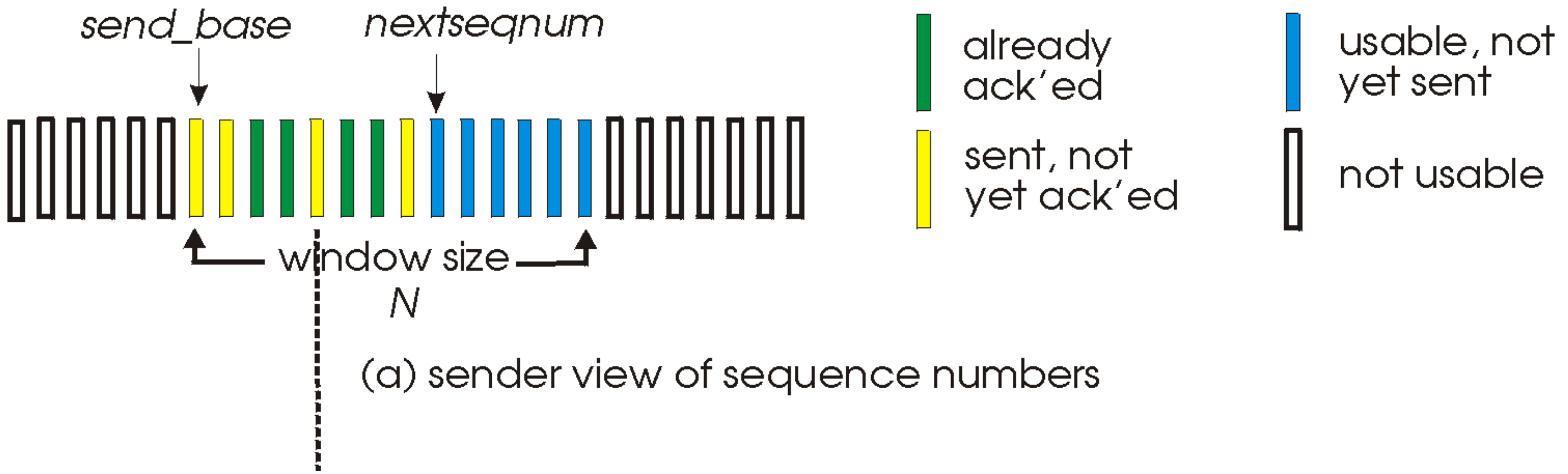
Go-Back-N: receiver extended FSM



ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

- out-of-order packet:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

Selective repeat: sender, receiver windows

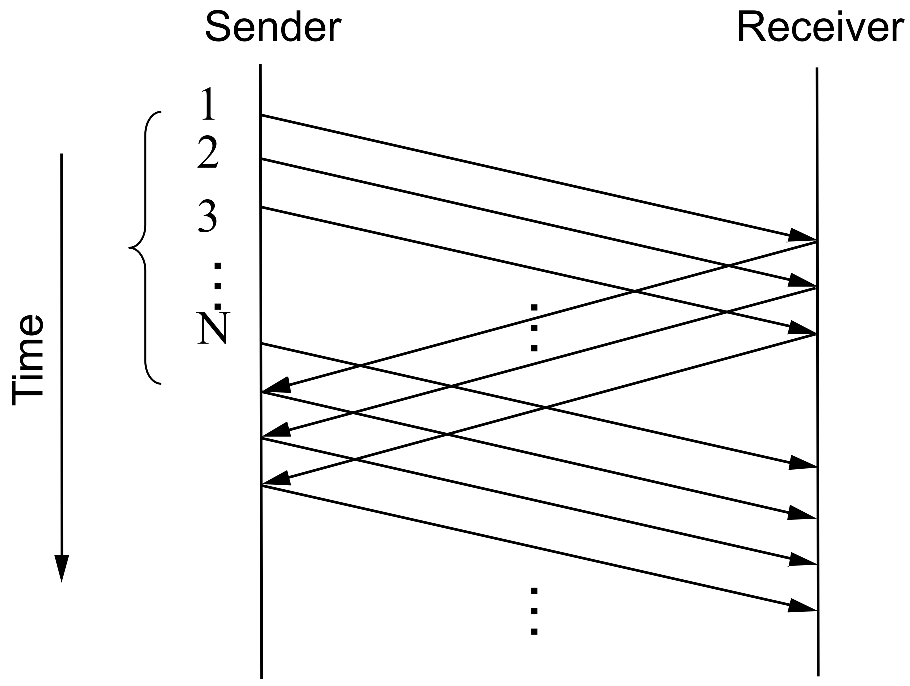


(a) sender view of sequence numbers

Sliding Window (Reference)

https://book.systemsapproach.org/direct/reliable.html?highlight=sliding%20window

- **Basic Idea**:
  - Allow sender to transmit **multiple frames** before receiving an **ACK**, thereby keeping the pipe full. There is an upper limit (called ***window***) on the number of **outstanding** (un-ACKed) frames allowed.

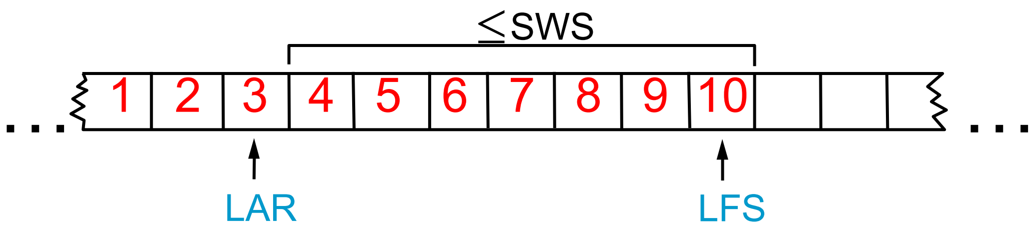


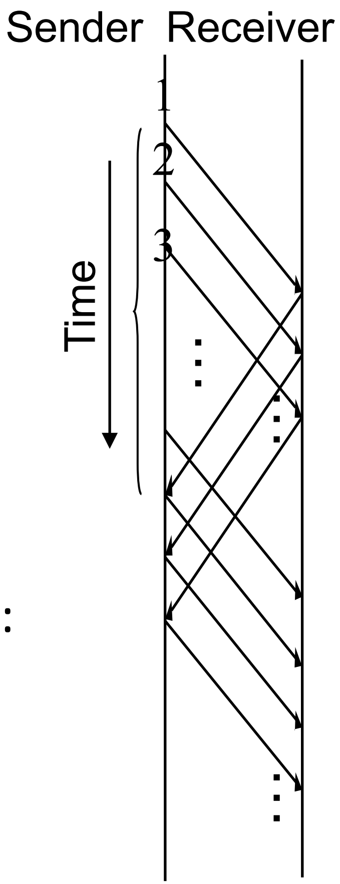
Courtesy of "*Computer Network: A System Approach*" by Larry Peterson and Bruce Davie (Chapter 2.5)

Sliding Window: Sender (Reference)

- Assign sequence number to each frame (**SeqNum**)
- Maintain three state variables:
  - send window size (**SWS**): upper bound on the # of outstanding (un-ACK) frames
  - sequence # of last acknowledgment received (**LAR**)
  - sequence # of last frame sent (**LFS**)

- Maintain invariant: **LFS** - **LAR** <= **SWS** at all time



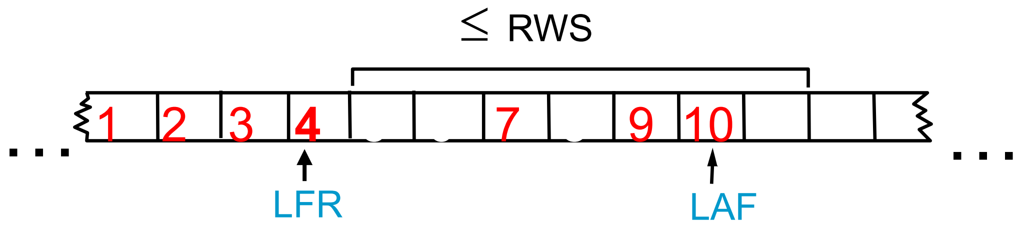


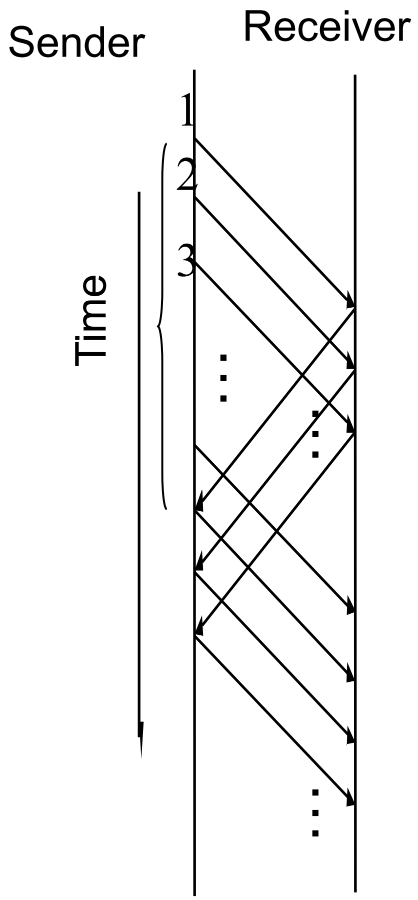
- Advance/update **LAR** when ACK arrives to allow a new frame be sent:
  - *What if 5 received before 4?*

- Buffer up to **SWS** frames for retransmission if needed
  - Worst scenario….
  - Best scenario….
  - Other scenarios

Sliding Window: Receiver (Reference)

- Maintain three state variables
  - **receive window size** (**RWS**): upper bound on the # of out-of-order frames (Why?): size selection?
  - sequence # of **largest acceptable frame** (**LAF**)
  - sequence # of **last frame received** (**LFR**) *in order*
- Maintain invariant: **LAF - LFR <= RWS**





- Frame w/ **SeqNum** arrives:
  - if **LFR < SeqNum < = LAF** ⟶ **accept**
  - if **SeqNum < = LFR** or **SeqNum > LAF** ⟶ **discarded**
- Mechanism of Sending cumulative **ACKs**
  - **LFR = SeqNumtoAck (largest seq # not yet acknowledged)**
  - **LAF = LFR + RWS**
  - **Variations on packet loss notification**
    - **timeout, negative ACK, duplicated ACK, selective ACK**

Sequence Number Space (Reference)

- **SeqNum** field is finite; sequence numbers wrap around (reuse)
- Sequence number space must be **larger** than the number of outstanding frames
- **SWS <= MaxSeqNum-1** is not sufficient
  - suppose 3-bit **SeqNum** field (**0..7**)
  - **SWS=RWS=7**
  - sender transmit frames **0..6**
  - arrive successfully, but ACKs lost
  - sender retransmits old **0..6**
  - receiver expecting **7,0..5**, but receives them as second incarnation of **0..5**
- **SWS < (MaxSeqNum+1)/2** is a correct rule when **SWS = RWS**
  - **If RWS=1, SWS <= MaxSeqNum-1 is sufficient**

TCP sender (simplified)

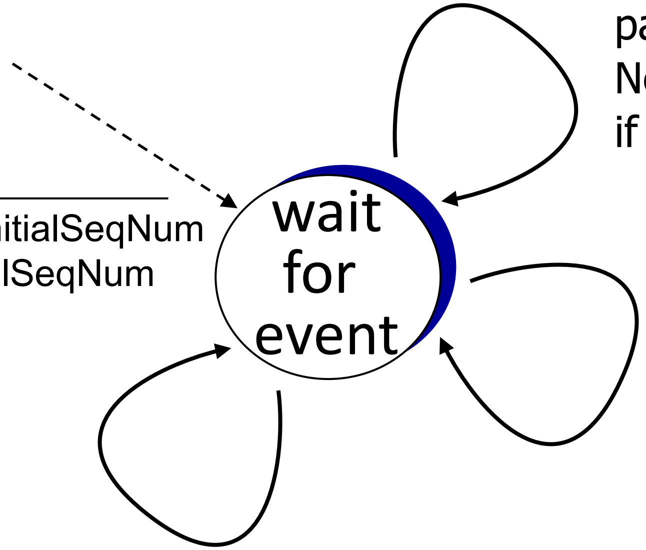
data received from application above

---

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



timeout

---

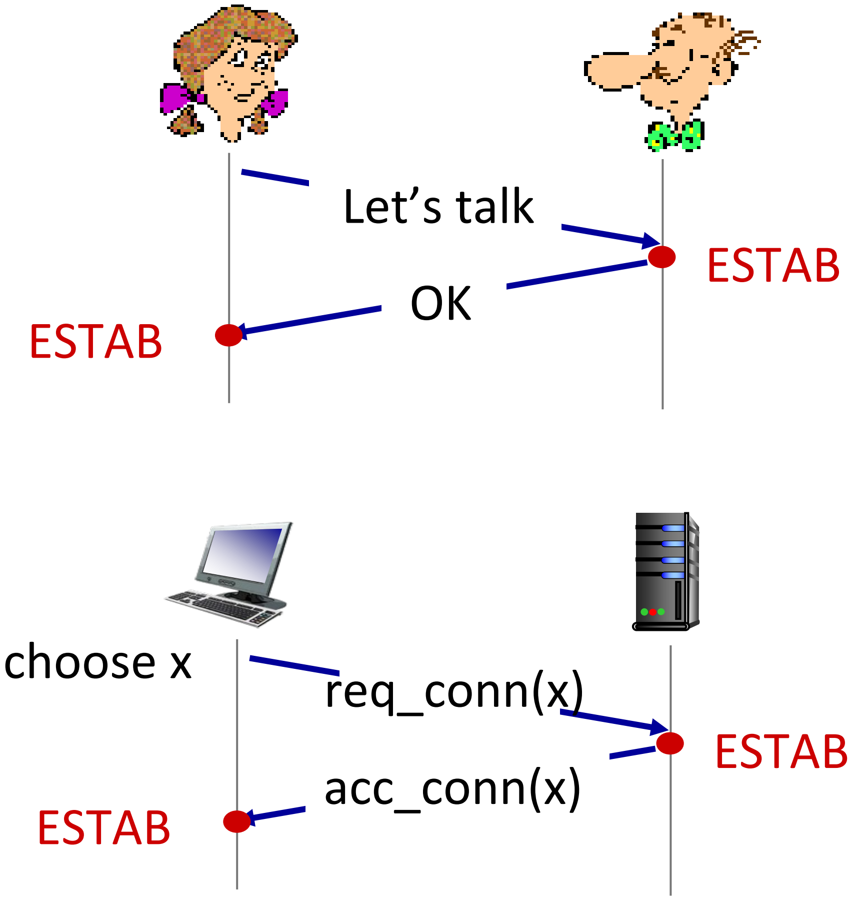
retransmit not-yet-acked segment
    with smallest seq. #
start timer

ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
        start timer
      else stop timer
 }
```

Agreeing to establish a connection

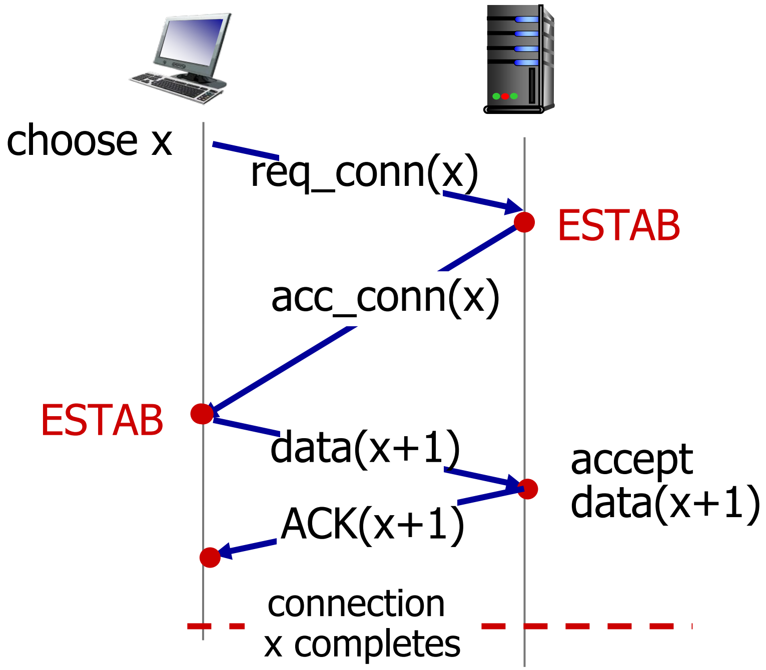
2-way handshake:



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

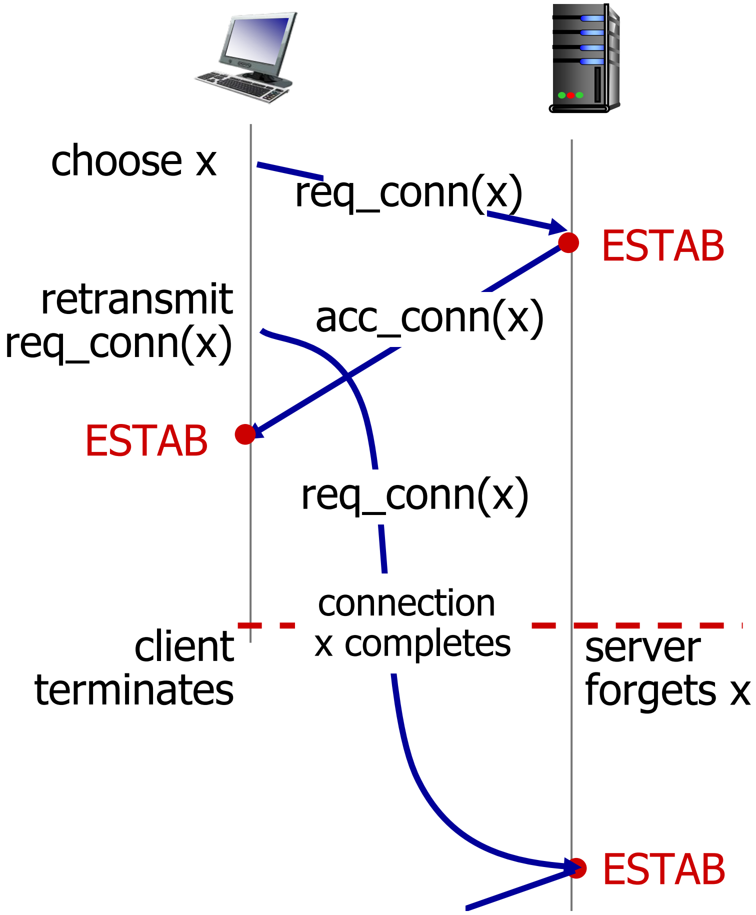
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

2-way handshake scenarios



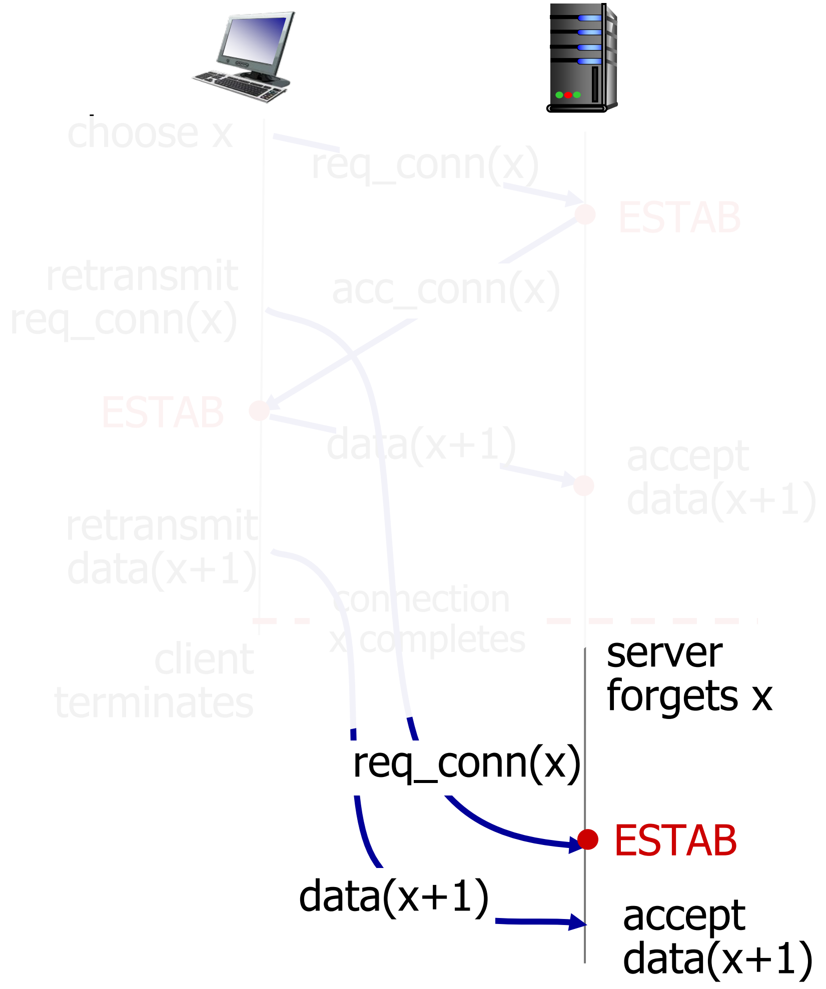
No problem!

2-way handshake scenarios



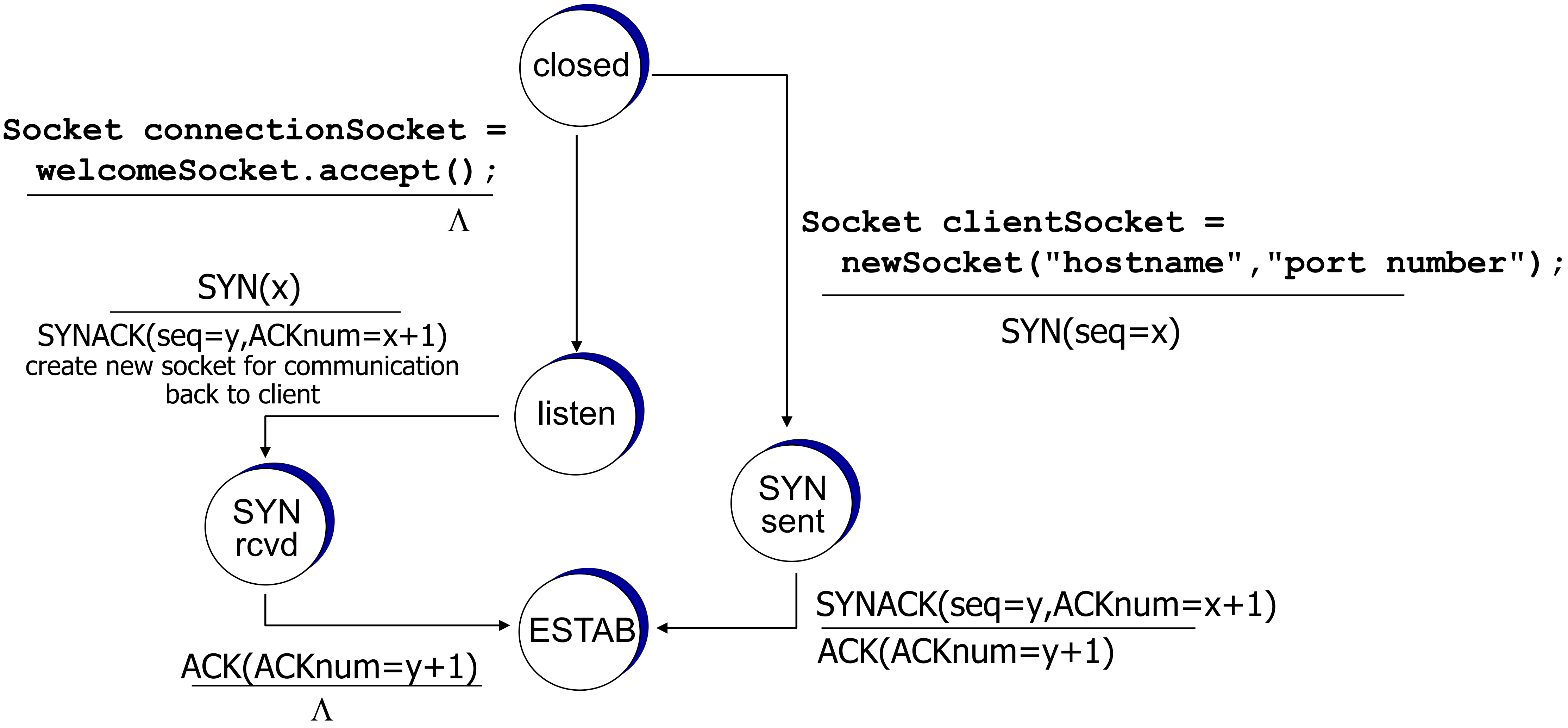
Problem: half open connection! (no client)

2-way handshake scenarios

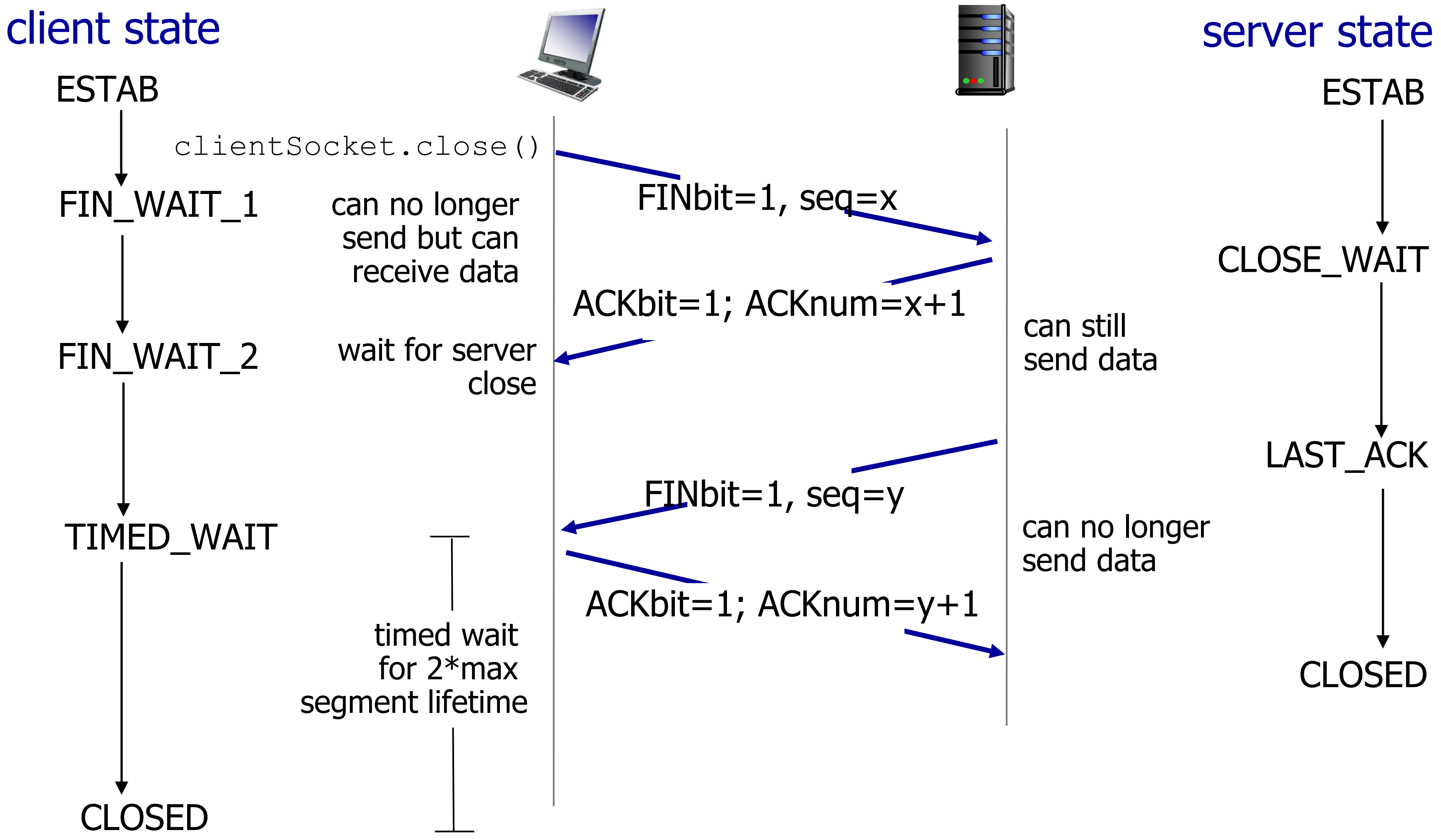


Problem: dup data accepted!

TCP 3-way handshake FSM



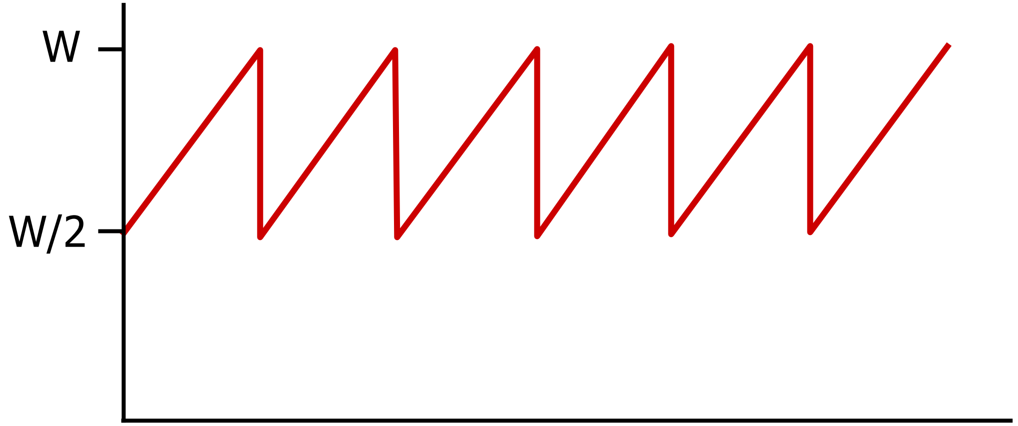
Closing a TCP connection



TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume there is 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 over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ *– a very small loss rate!*

- versions of TCP for long, high-speed scenarios