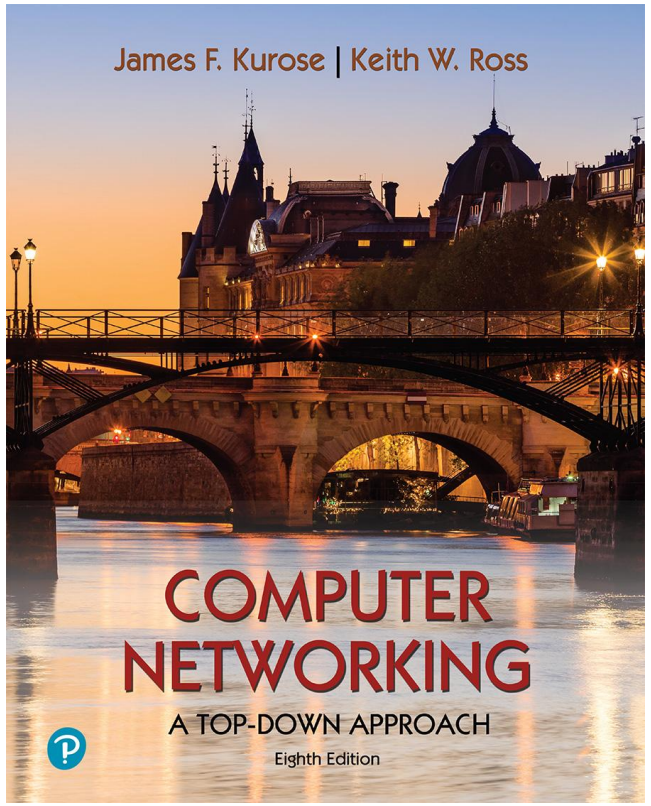
Chapter 3
Transport Layer



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

A note on the use of these PowerPoint slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

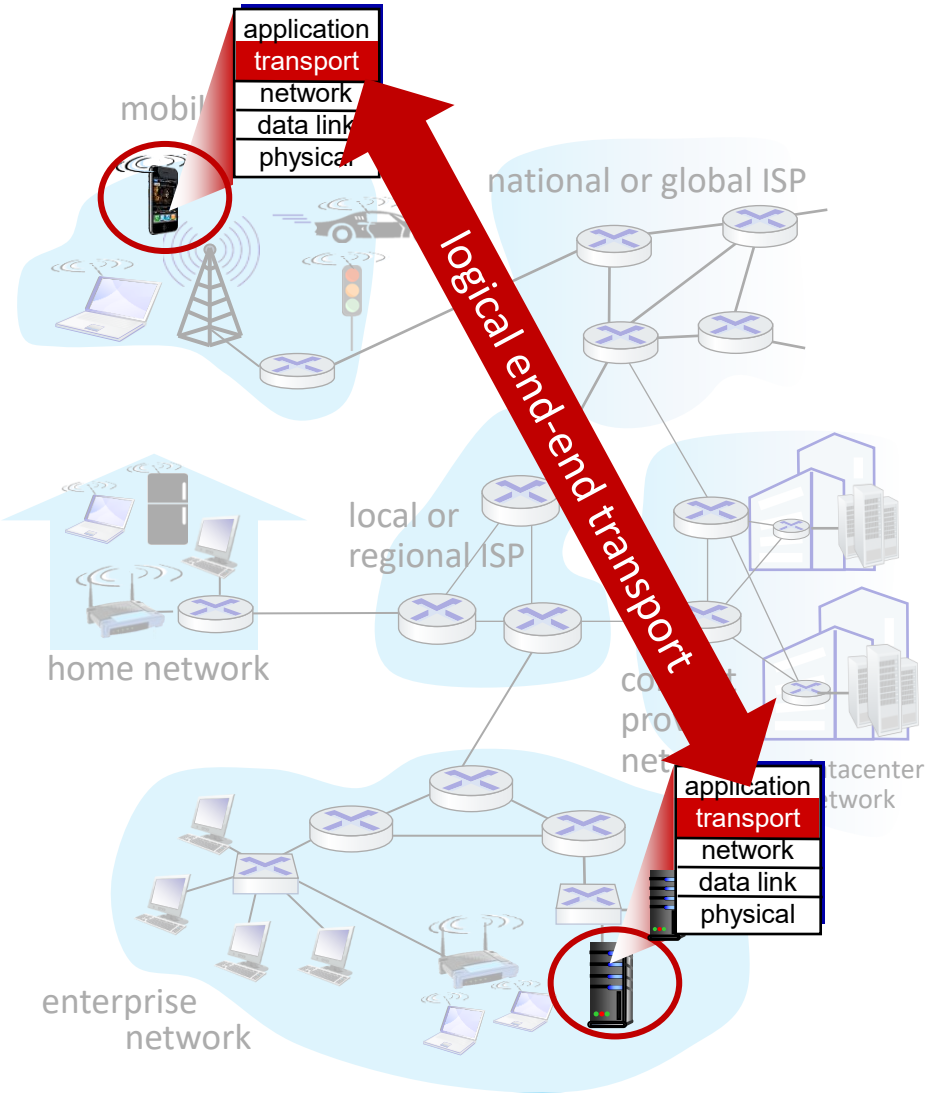
- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR

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 major transport protocols available to Internet applications
  - TCP, UDP



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

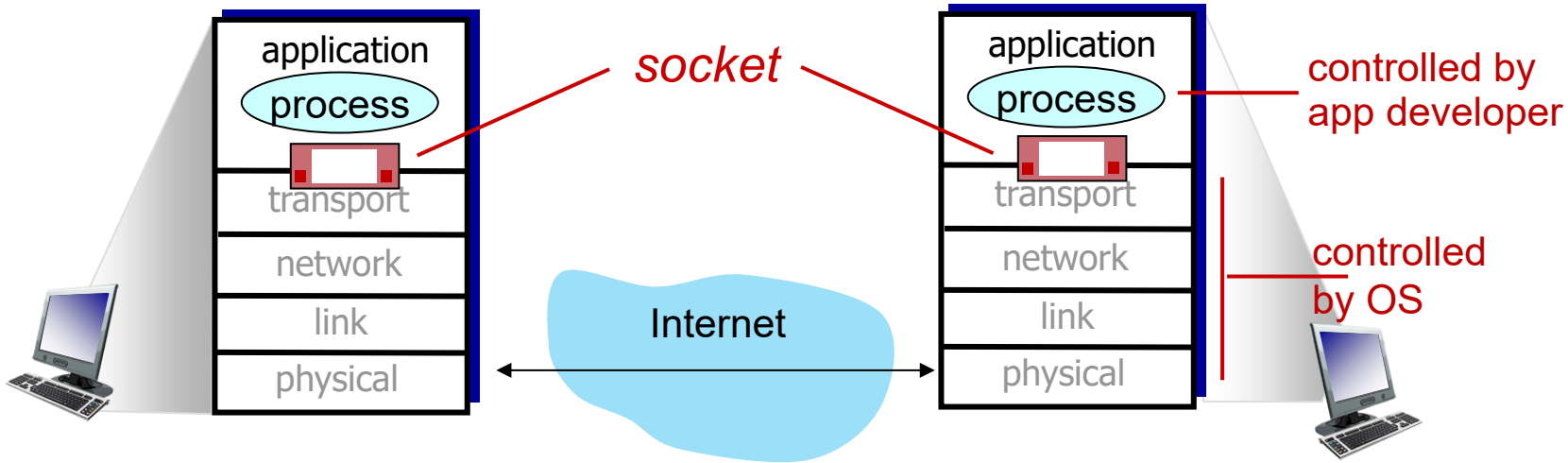
household analogy:

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

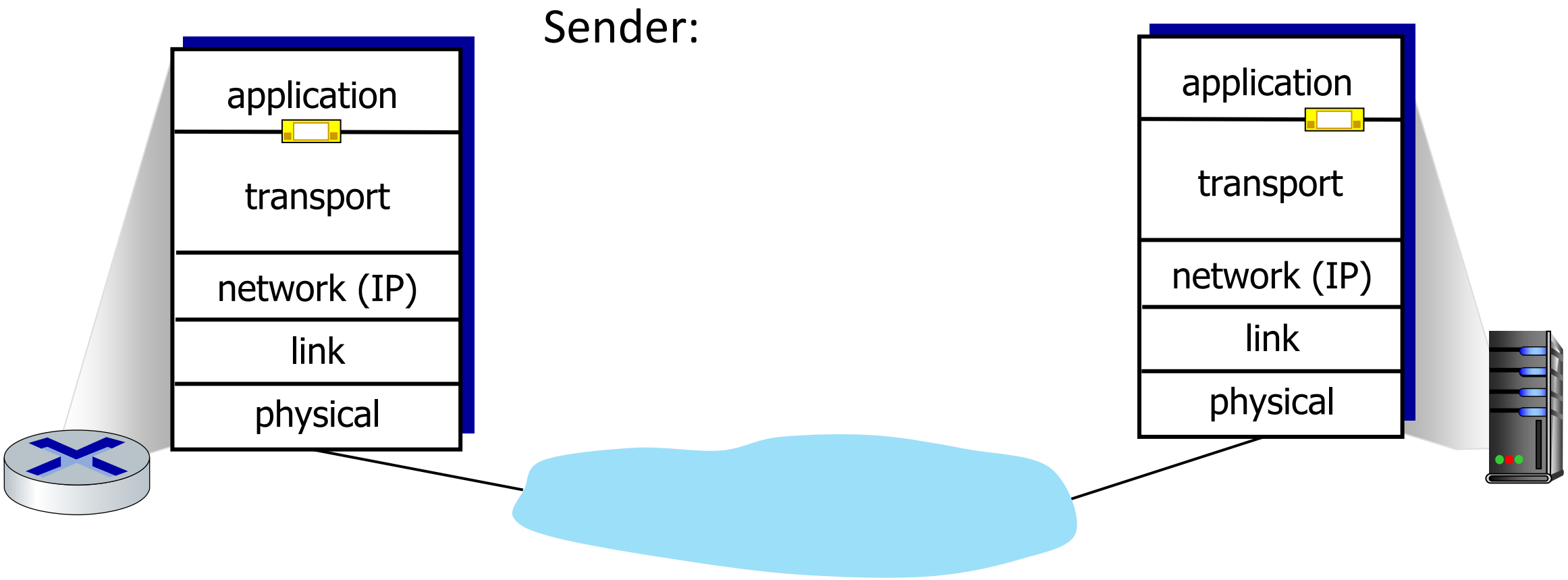
- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill
- network-layer protocol = postal service

A fundamental concept: the socket

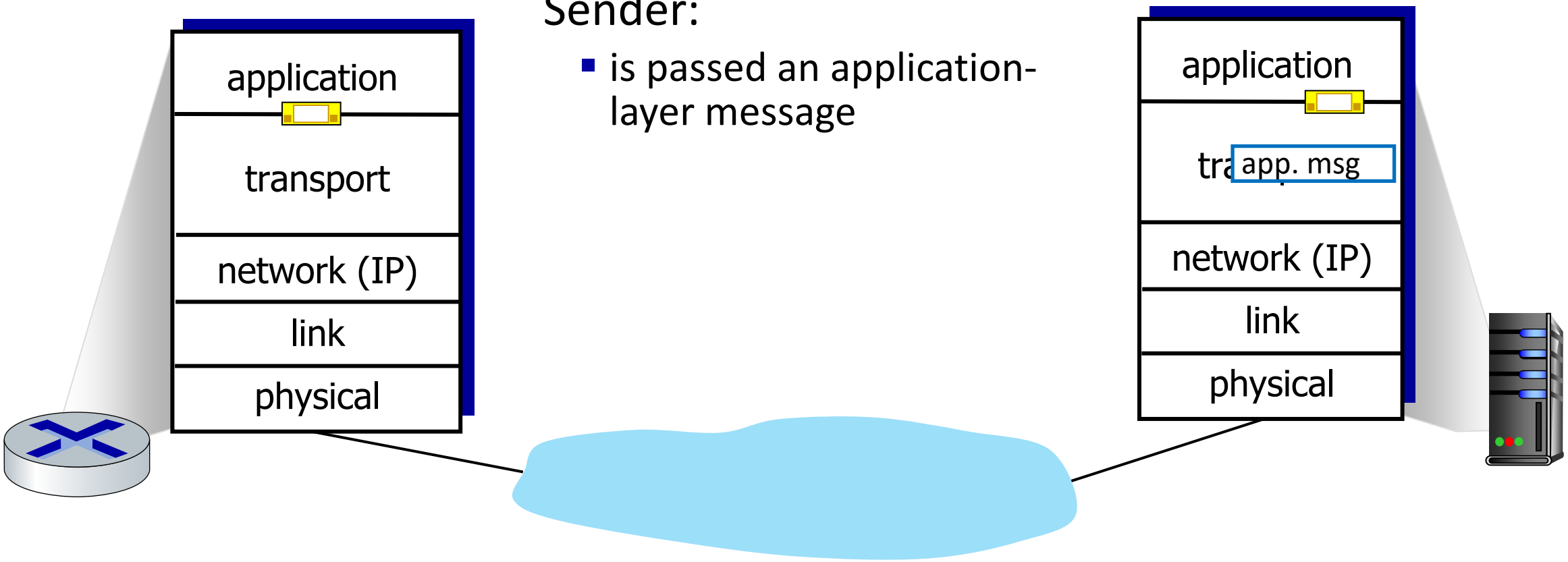
- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message outdoor
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



Transport Layer Actions



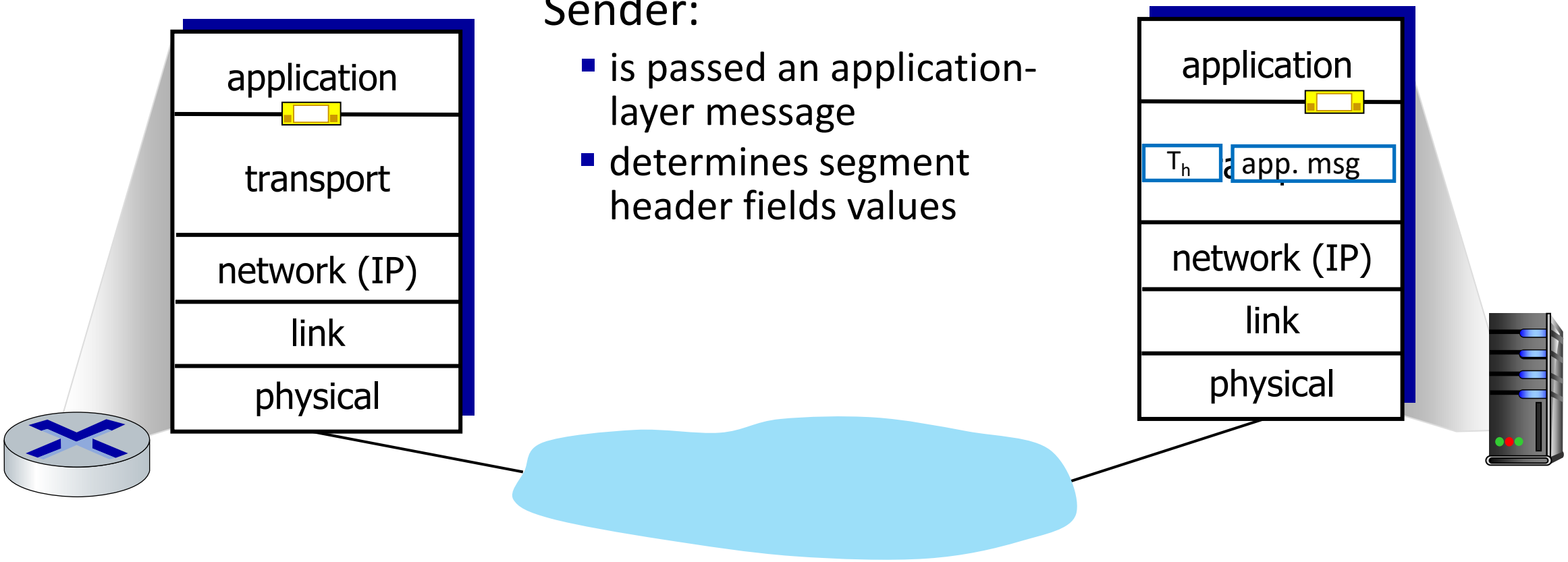
Transport Layer Actions



Sender:

- is passed an application-layer message

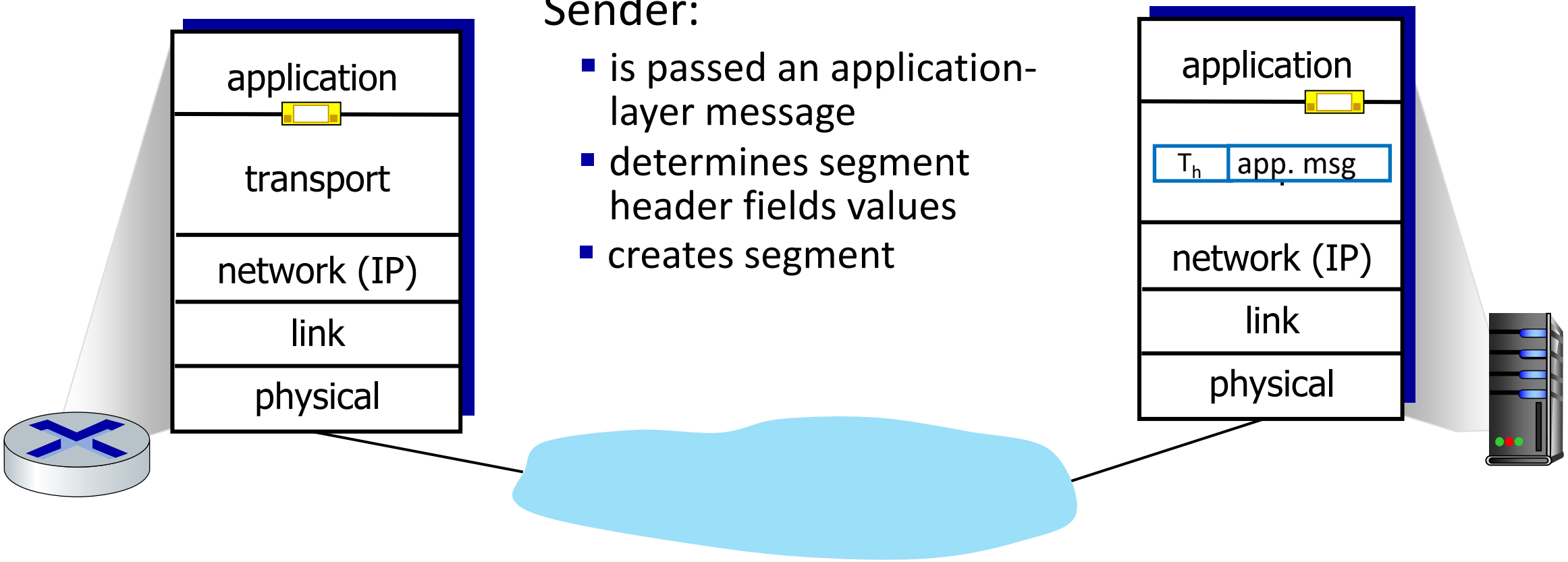
Transport Layer Actions



Sender:

- is passed an application-layer message
- determines segment header fields values

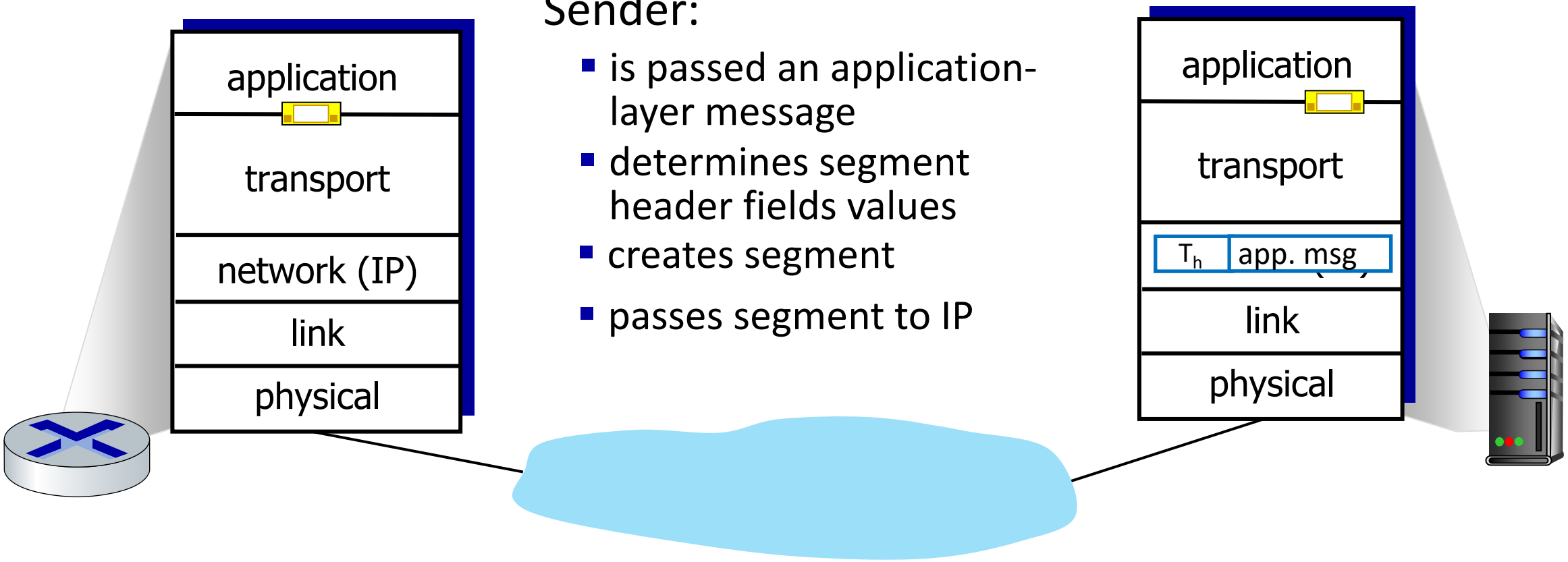
Transport Layer Actions



Sender:

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

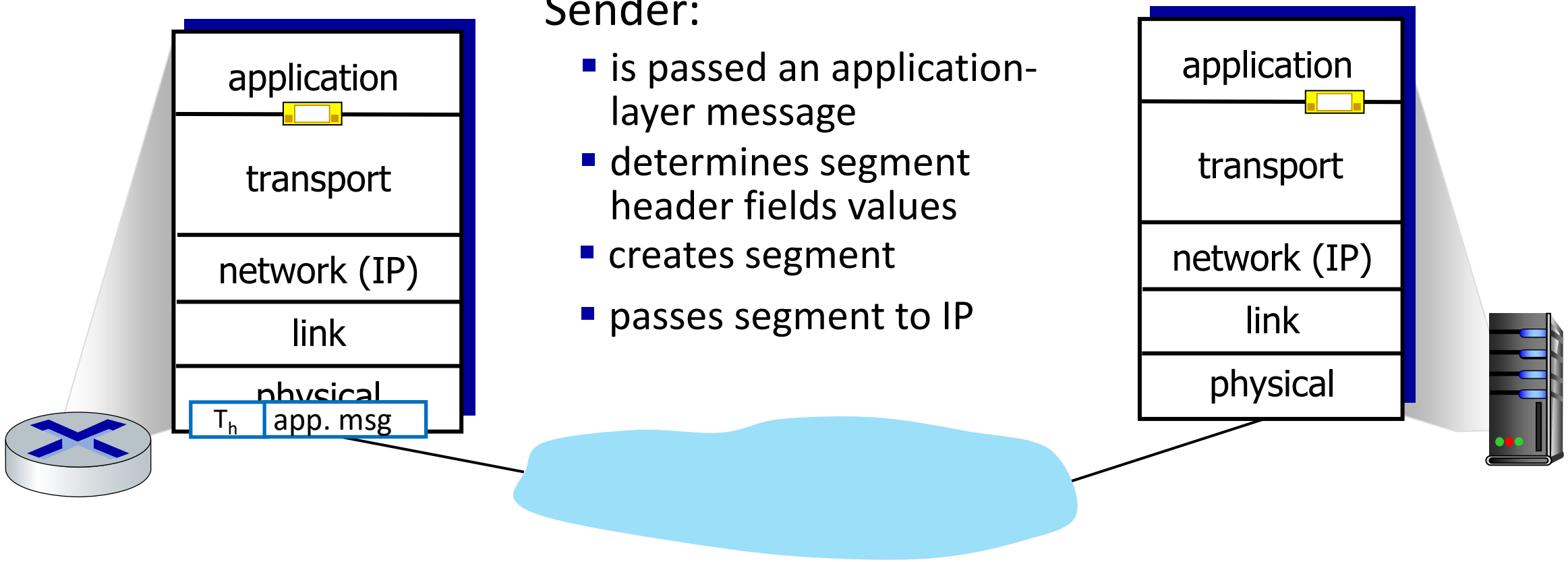
Transport Layer Actions



Sender:

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

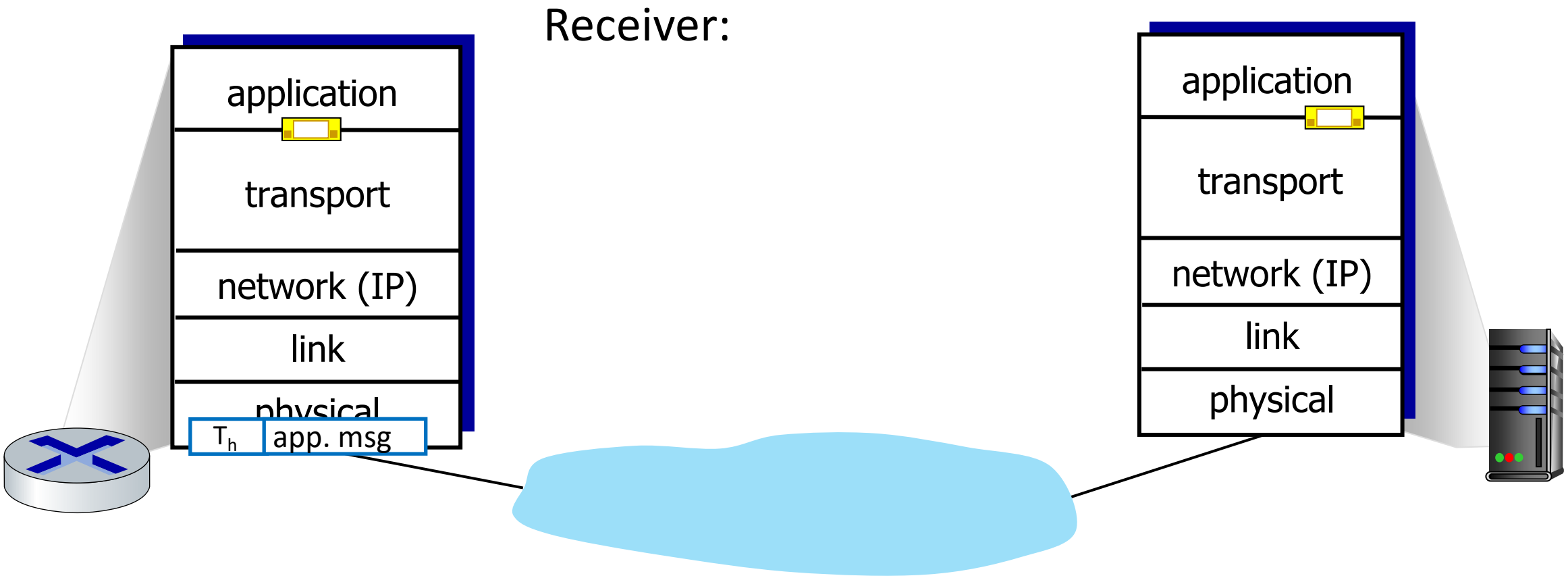
Transport Layer Actions



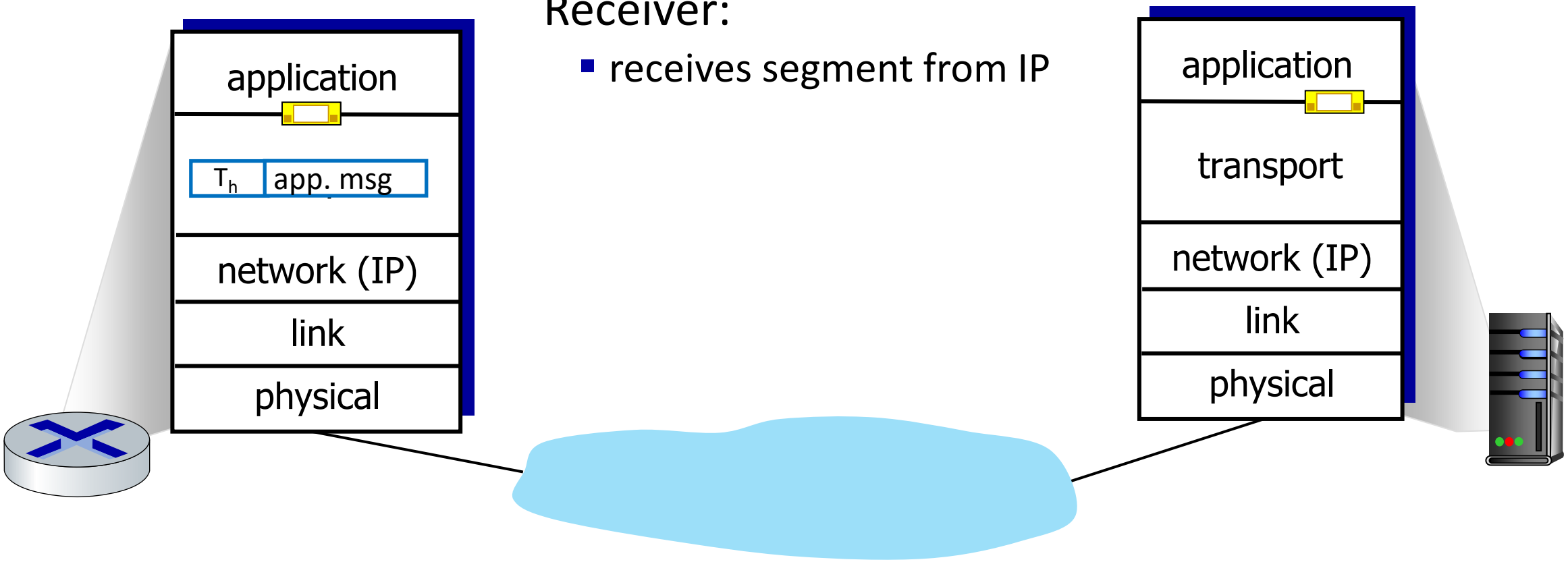
Sender:

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

Transport Layer Actions



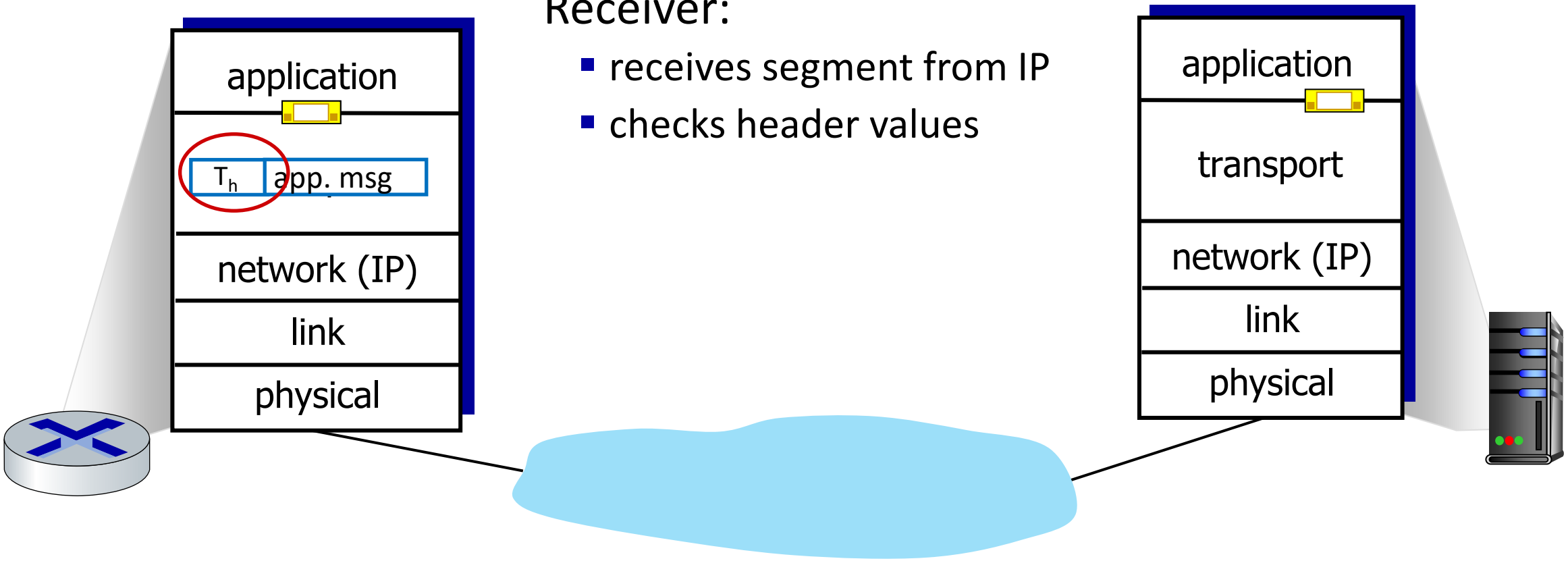
Transport Layer Actions



Receiver:

- receives segment from IP

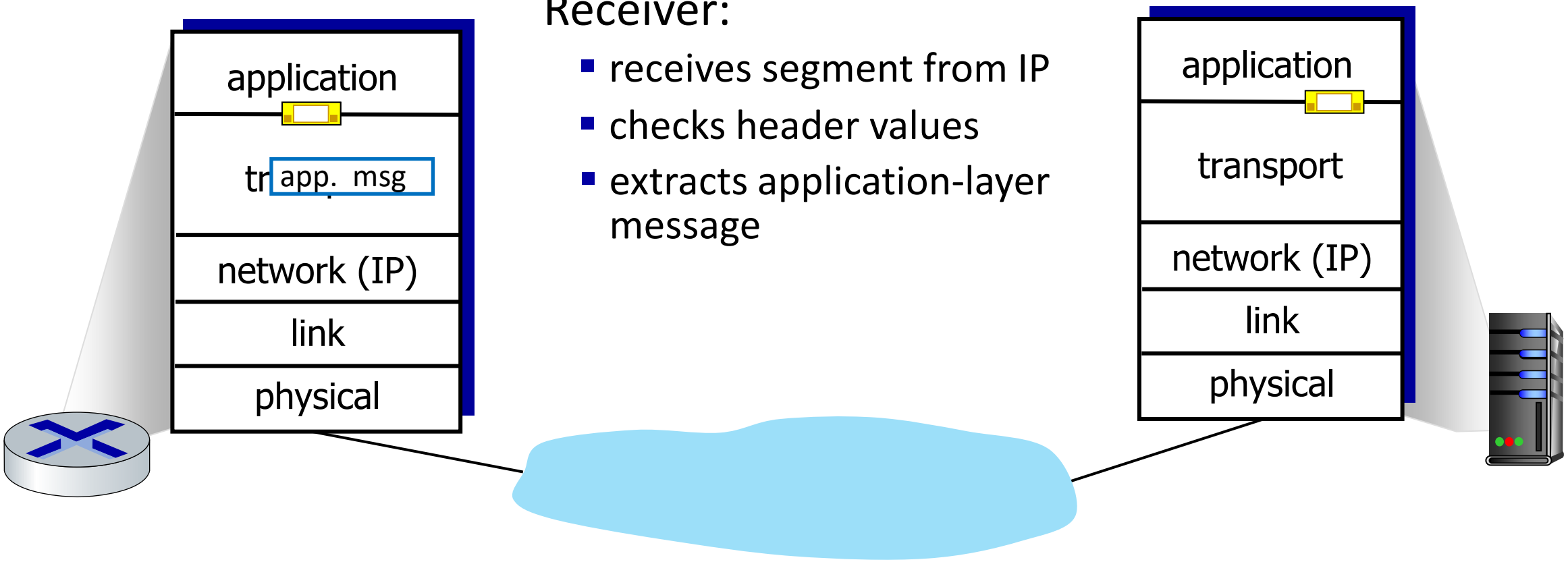
Transport Layer Actions



Receiver:

- receives segment from IP
- checks header values

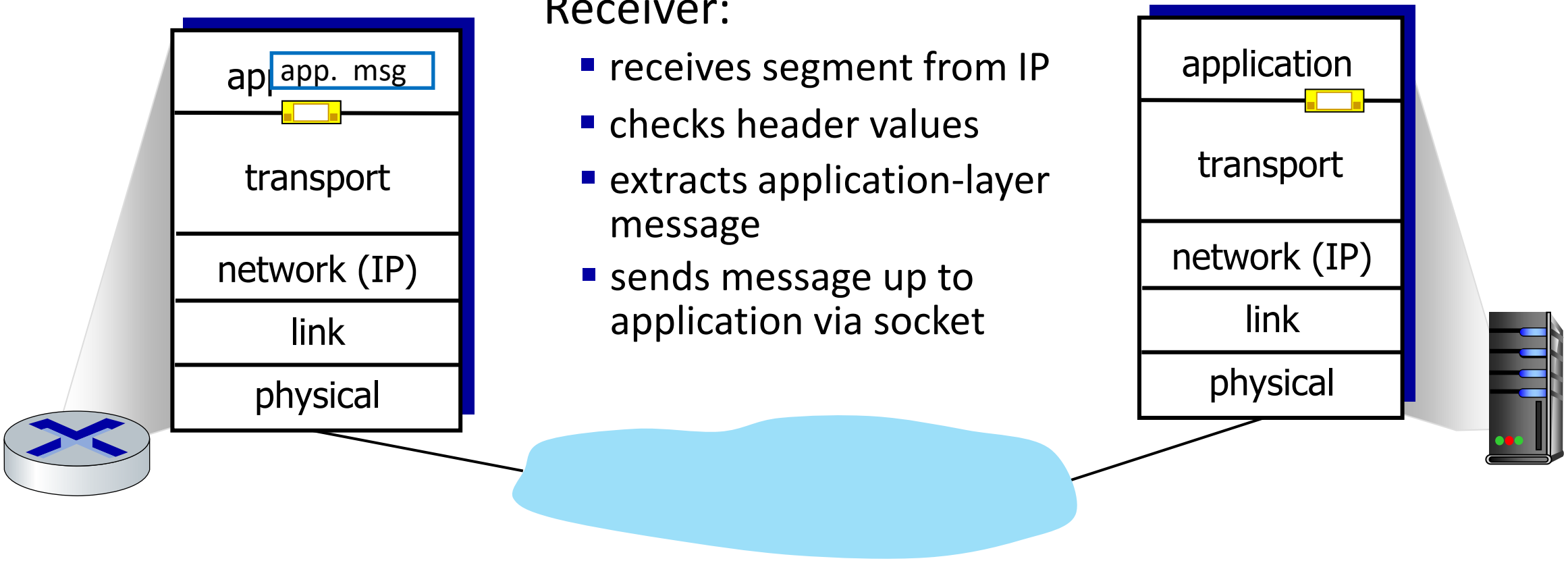
Transport Layer Actions



Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message

Transport Layer Actions

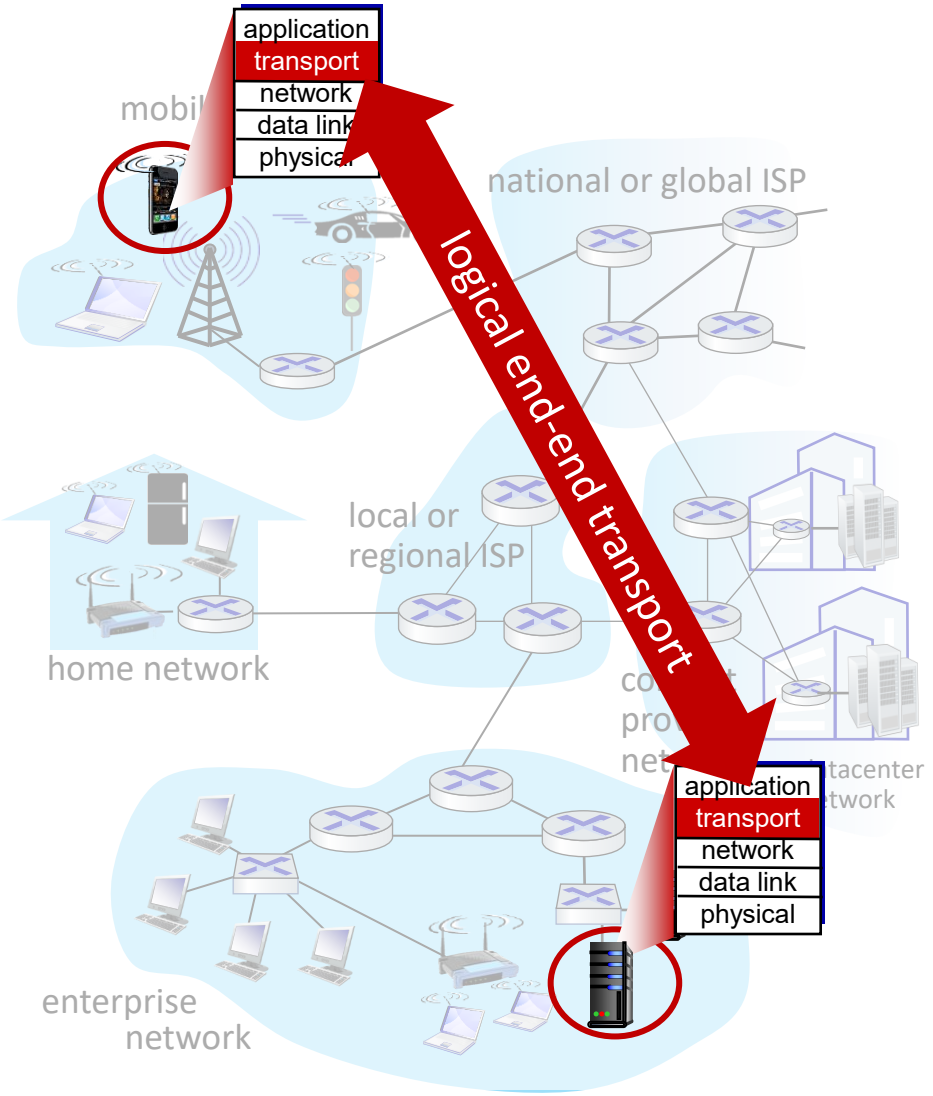


Receiver:

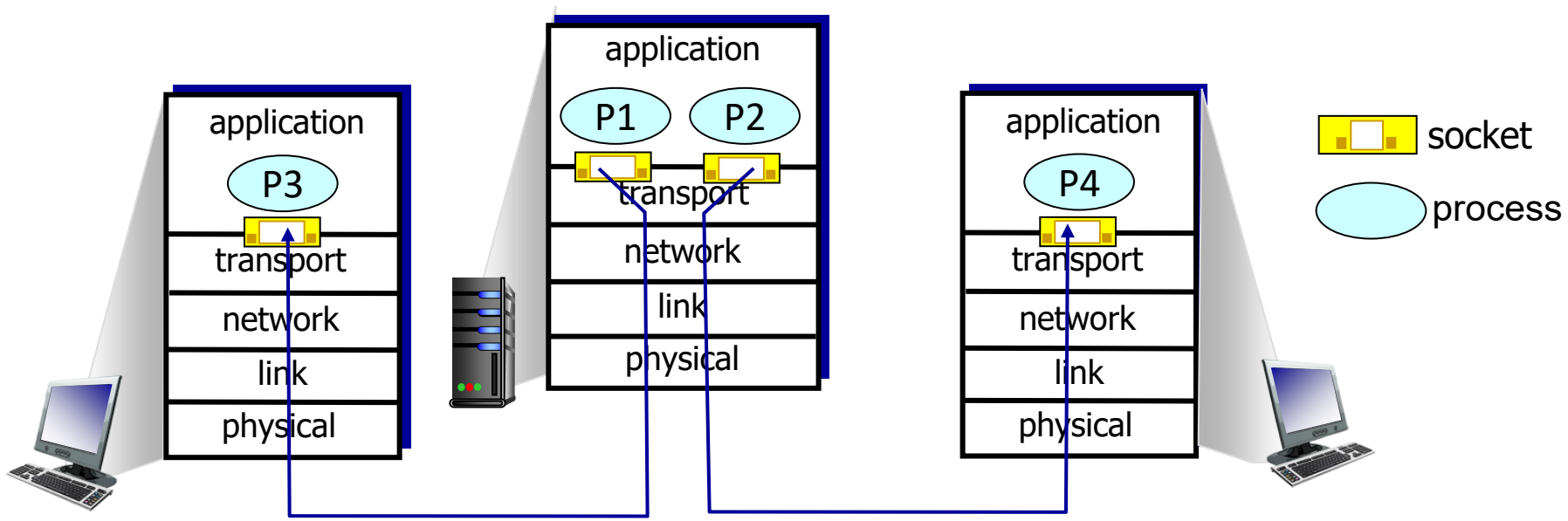
- receives segment from IP
- checks header values
- extracts application-layer message
- sends message up to application via socket

Two principal Internet transport protocols

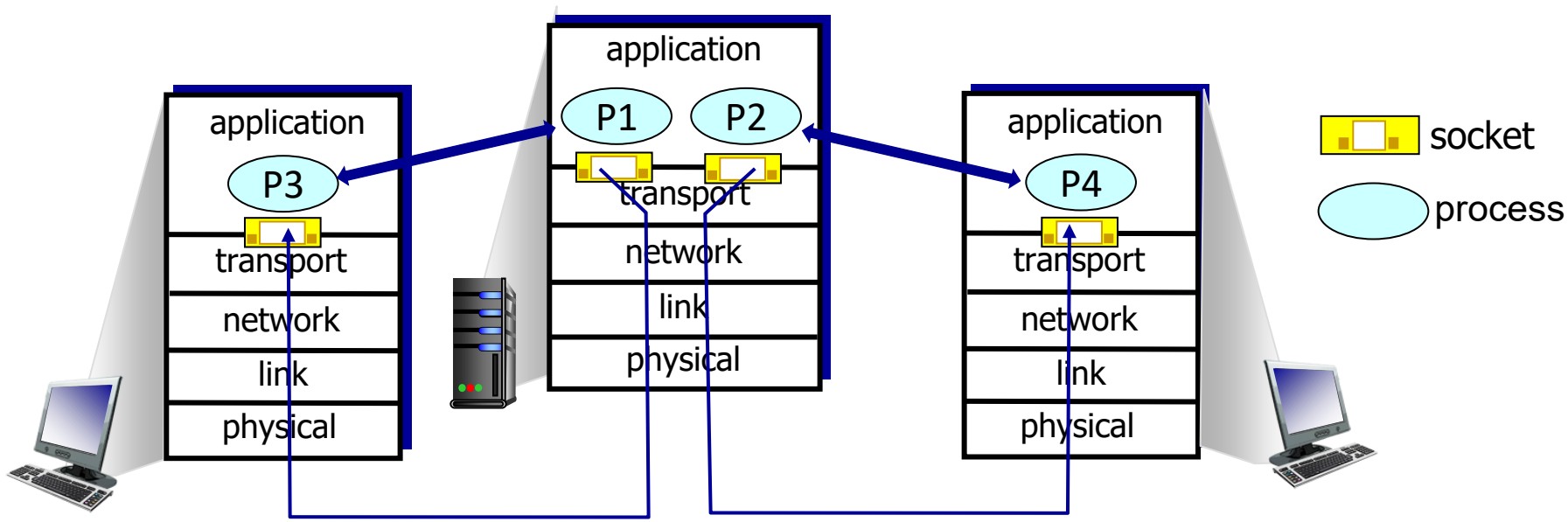
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth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



Multiplexing/demultiplexing



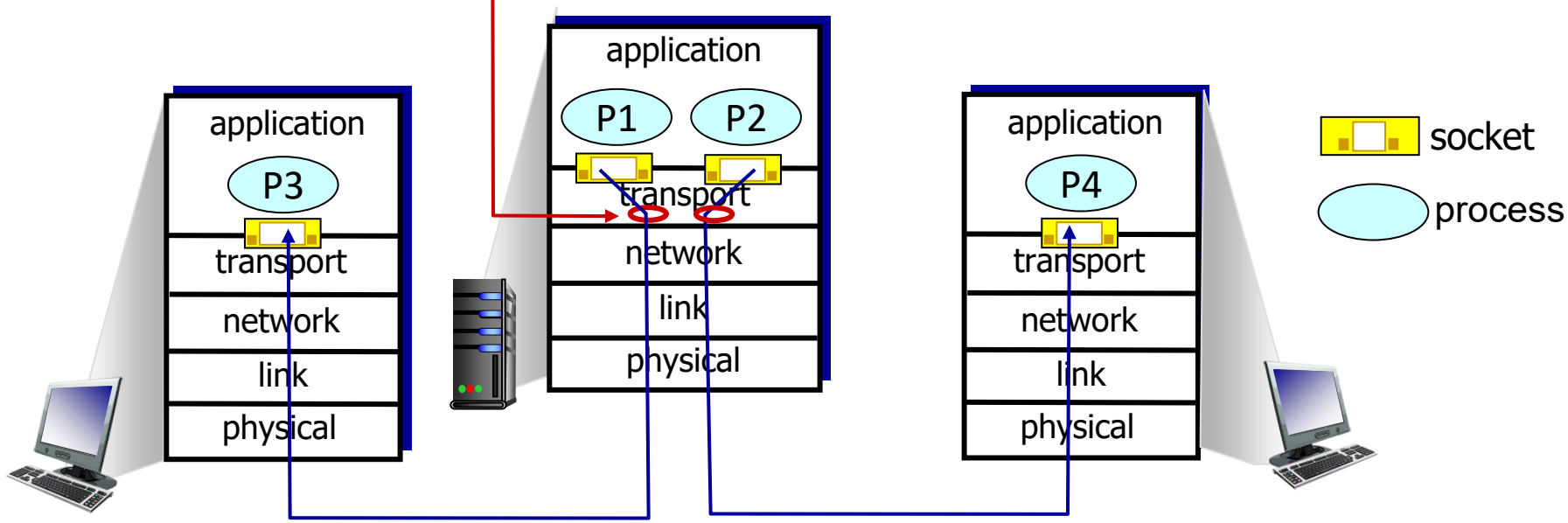
Multiplexing/demultiplexing



Multiplexing/demultiplexing

multiplexing at sender:

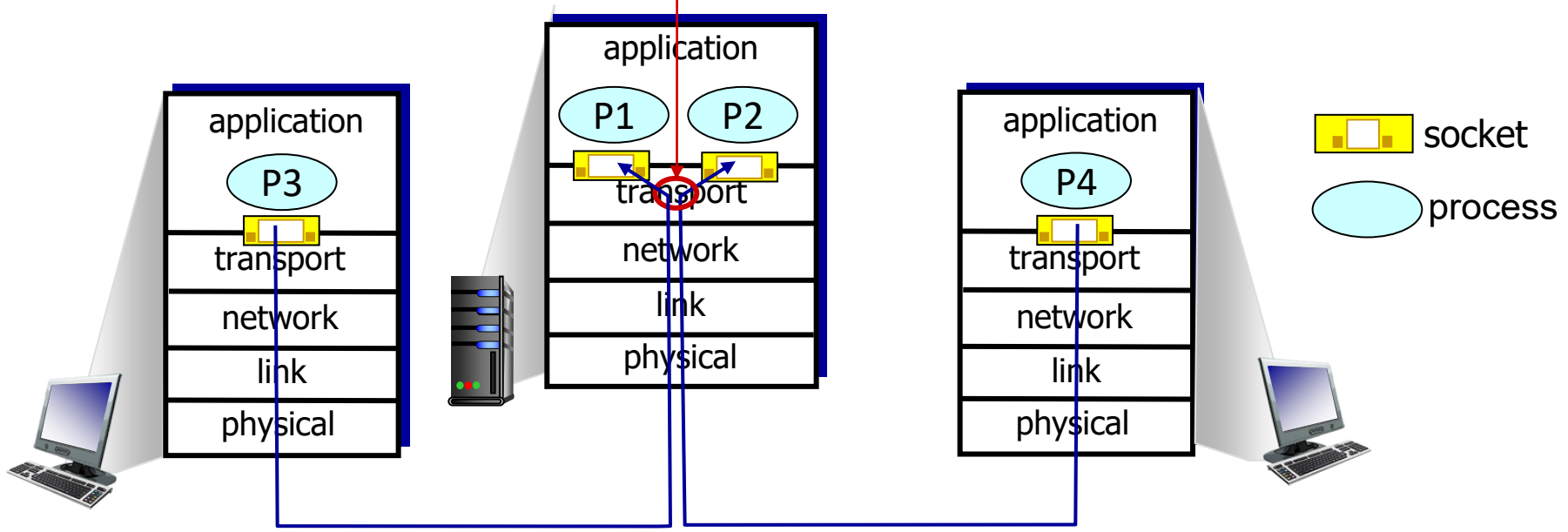
handle data from multiple sockets, add transport header (later used for demultiplexing)



Multiplexing/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
- sender host uses *IP addresses & port numbers* to direct segment to appropriate socket

Connectionless demultiplexing

Recall:

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

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

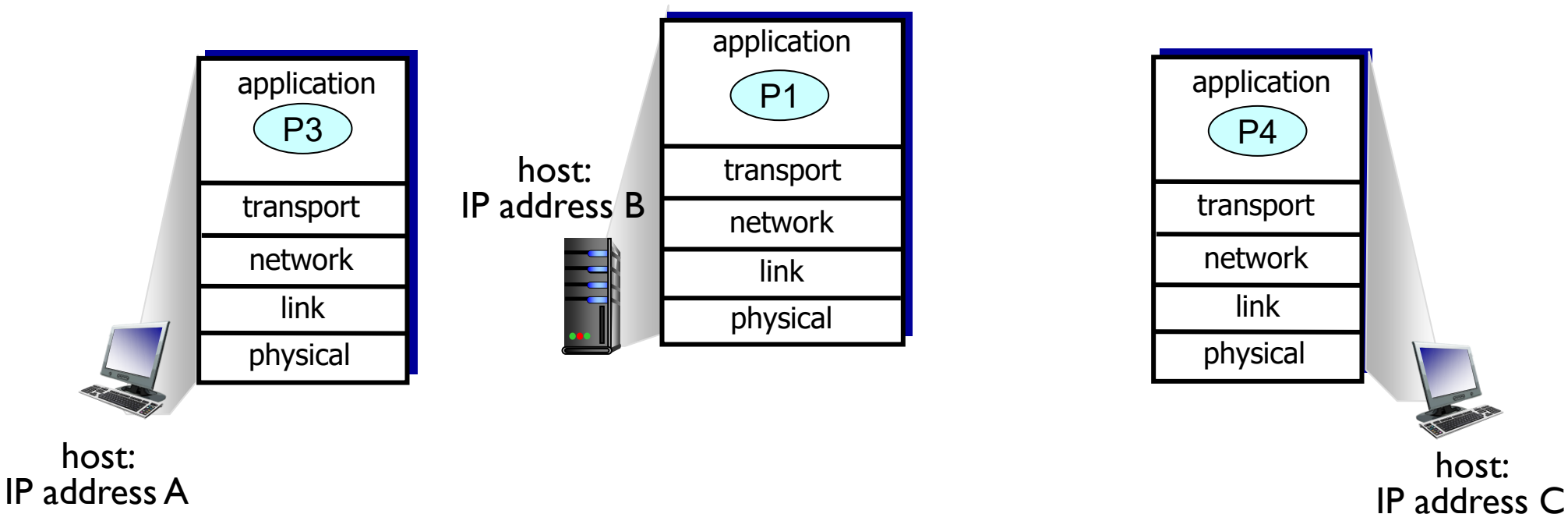
- two values univocally identify the UDP socket when a packet is created and sent to the process:
  1. *destination IP address*
  2. *destination port #*

when receiving host receives *UDP* segment:

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

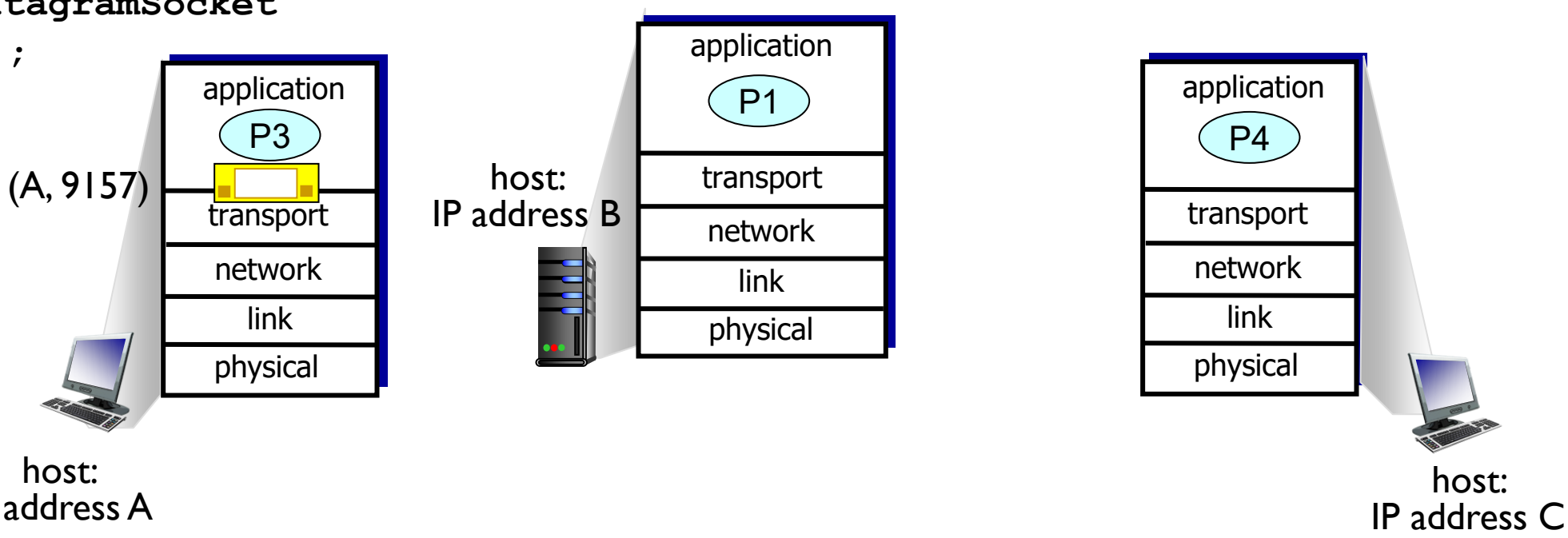
IP/UDP packets with *same dest. port # and dest. IP addresses,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example



Connectionless demultiplexing: an example

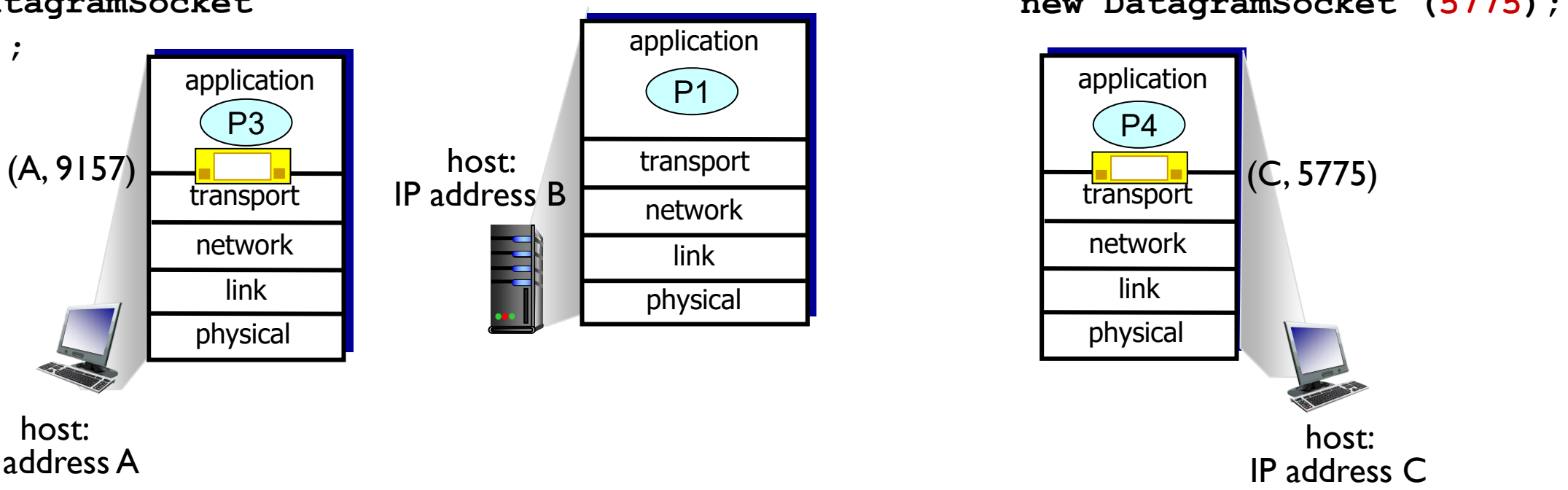
```
DatagramSocket mySocket2 =
 new DatagramSocket
 (9157);
```



Connectionless demultiplexing: an example

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

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

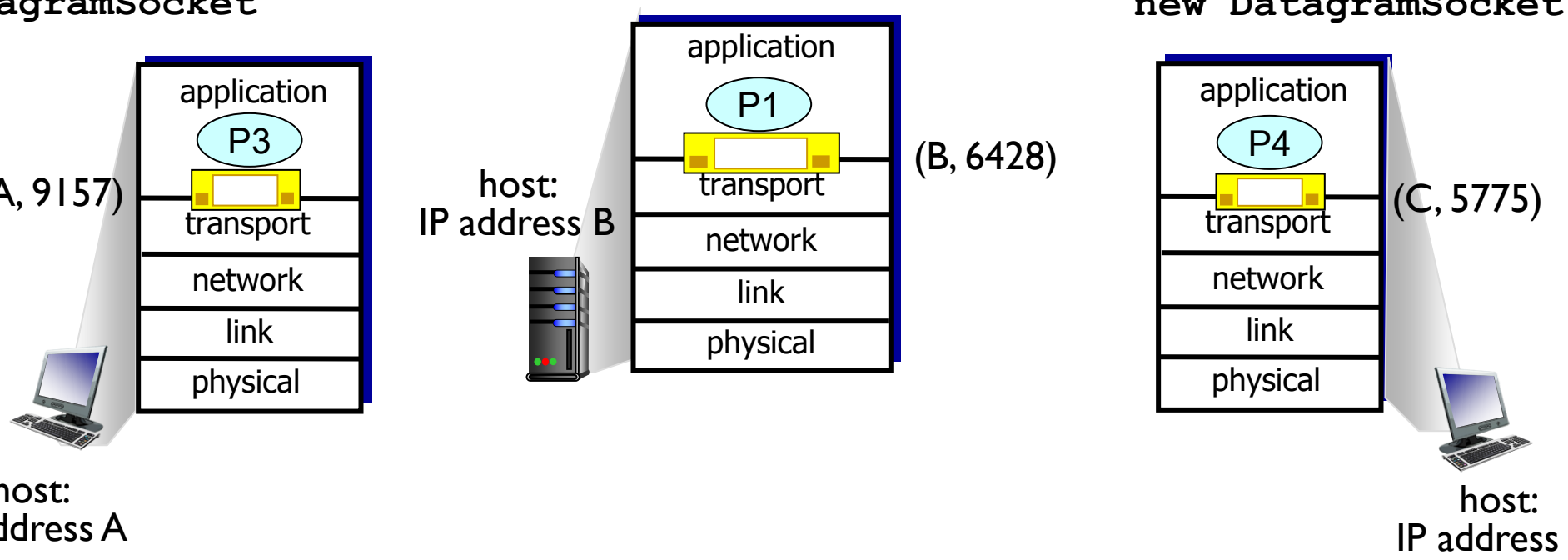


Connectionless demultiplexing: an example

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

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

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

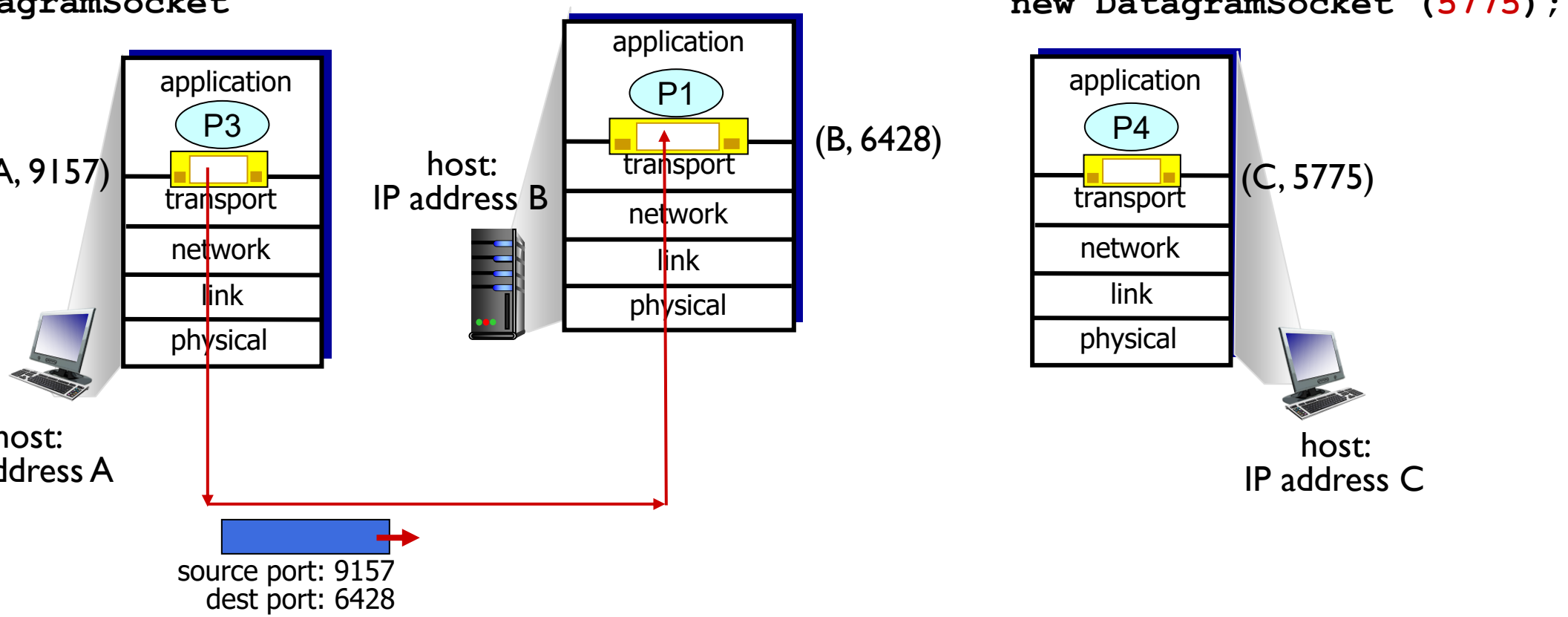


Connectionless demultiplexing: an example

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

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

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

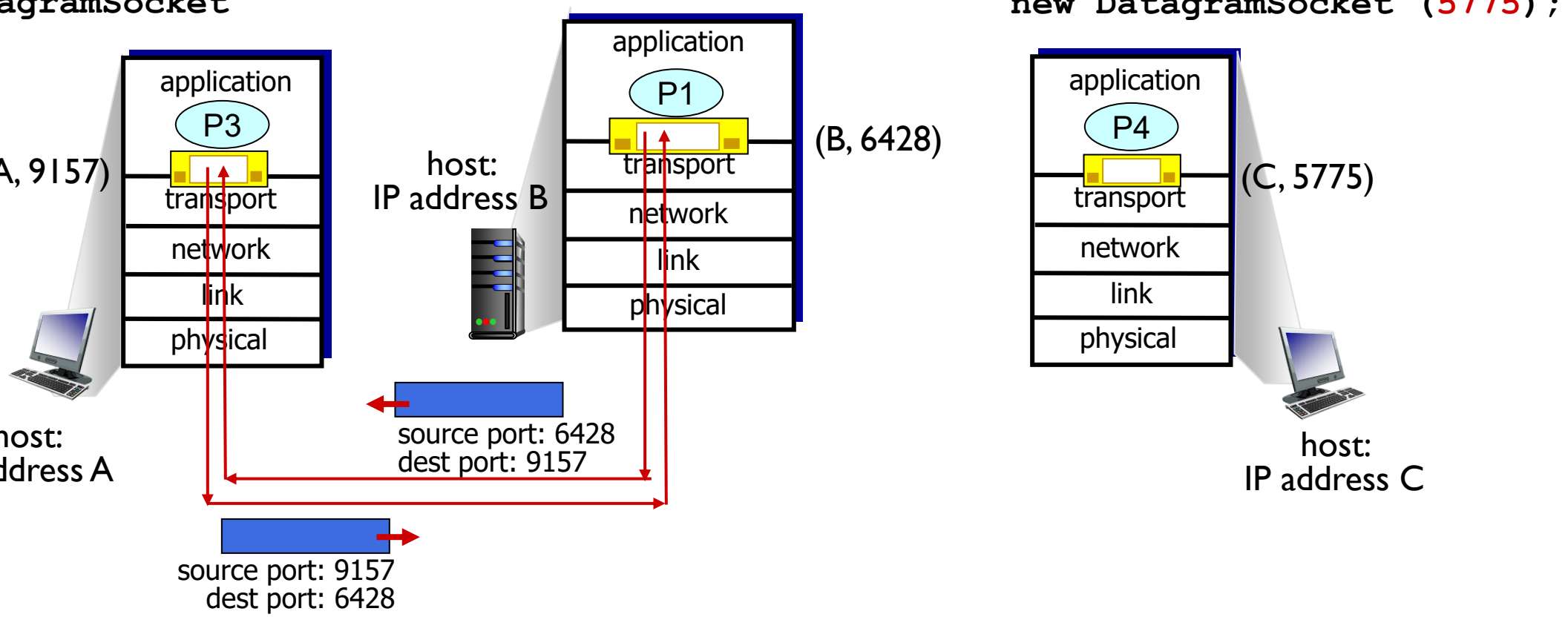


Connectionless demultiplexing: an example

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

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

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

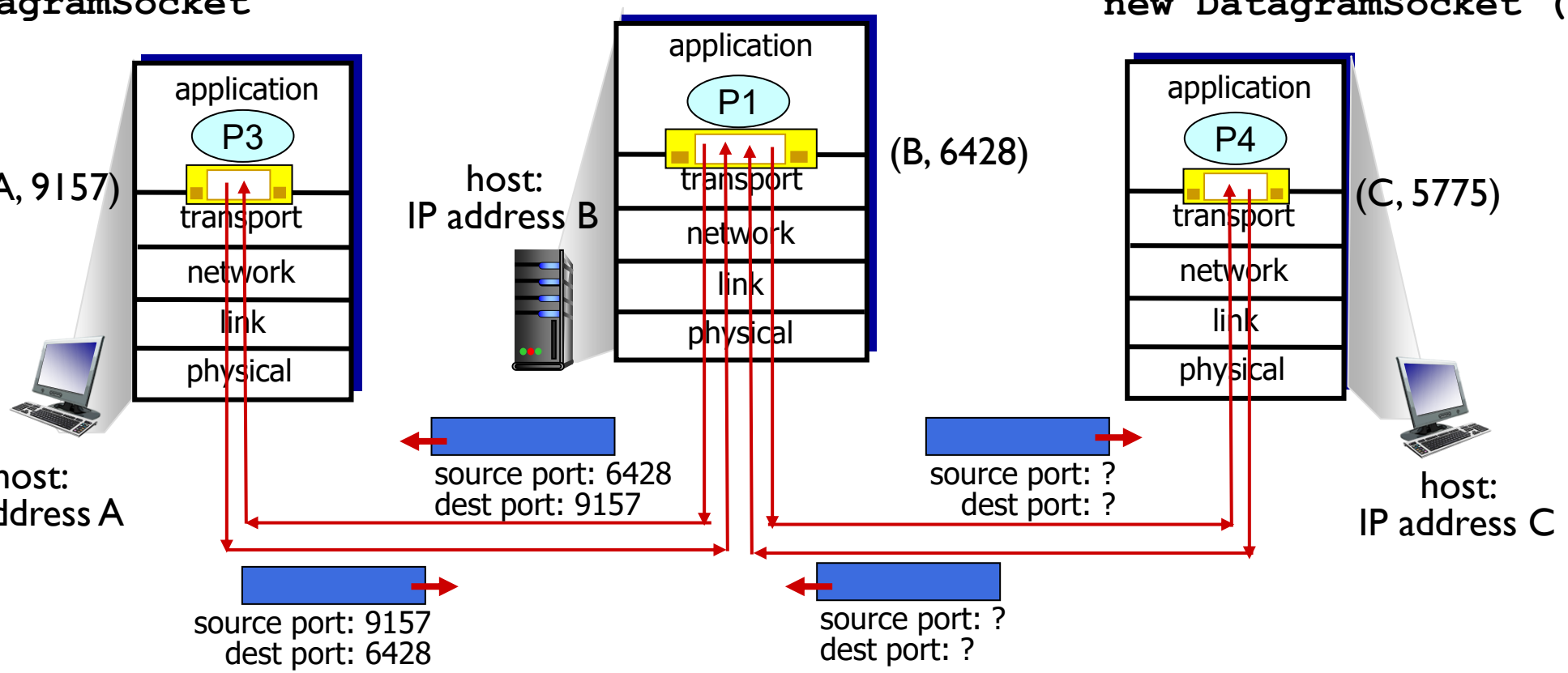


Connectionless demultiplexing: an example

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

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

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

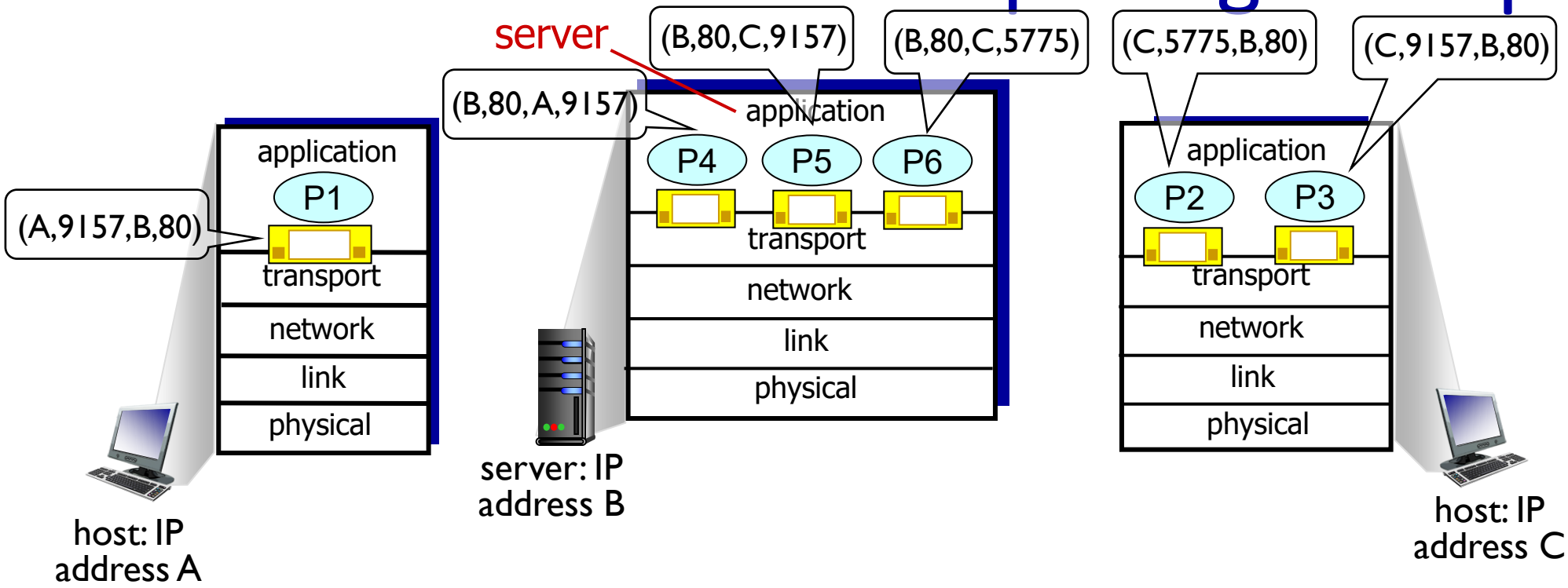


Connection-oriented demultiplexing

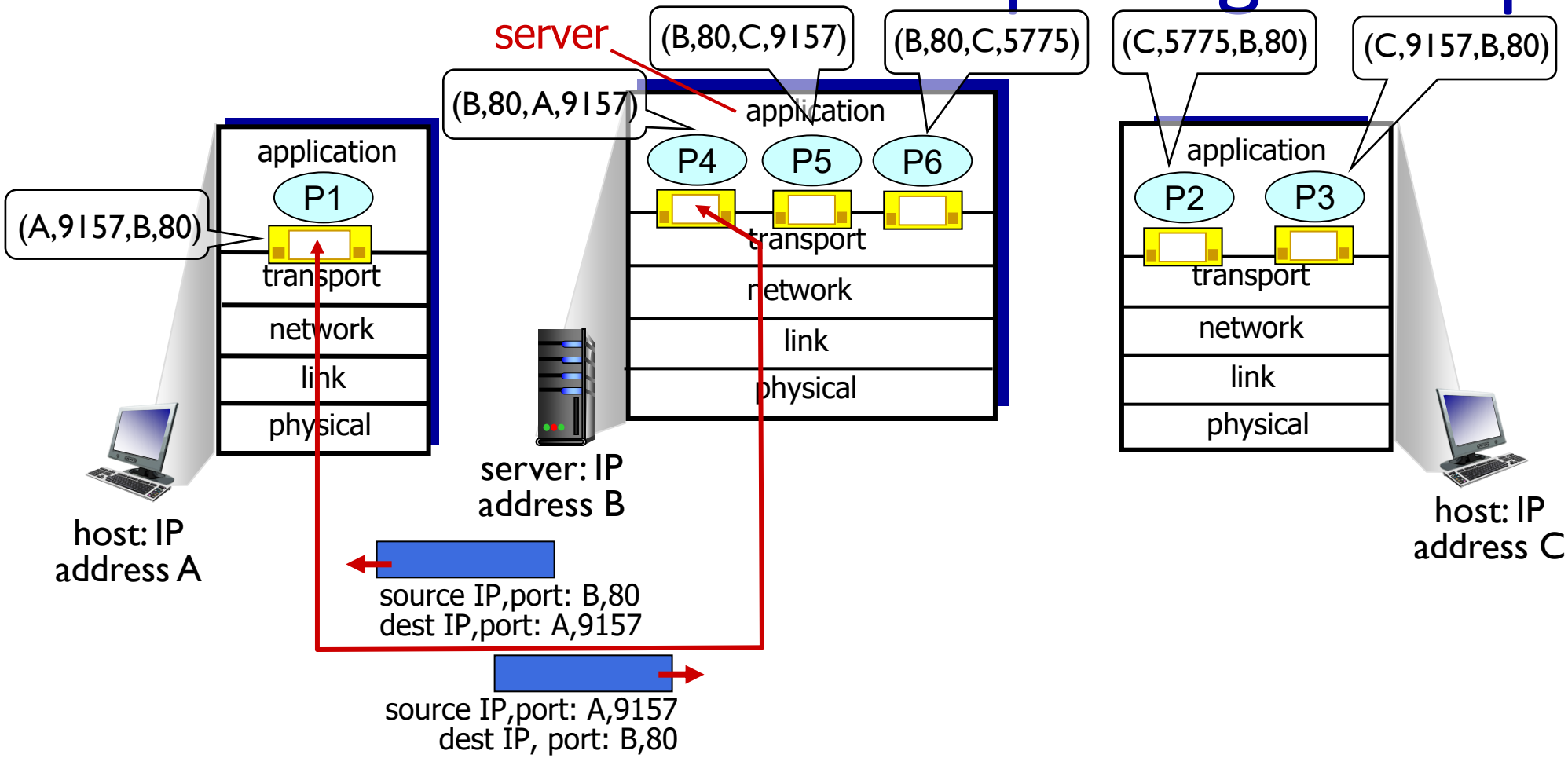
- TCP socket identified by 4-tuple:
  - *source IP address*
  - *source port number*
  - *dest IP address*
  - *dest port number*
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

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

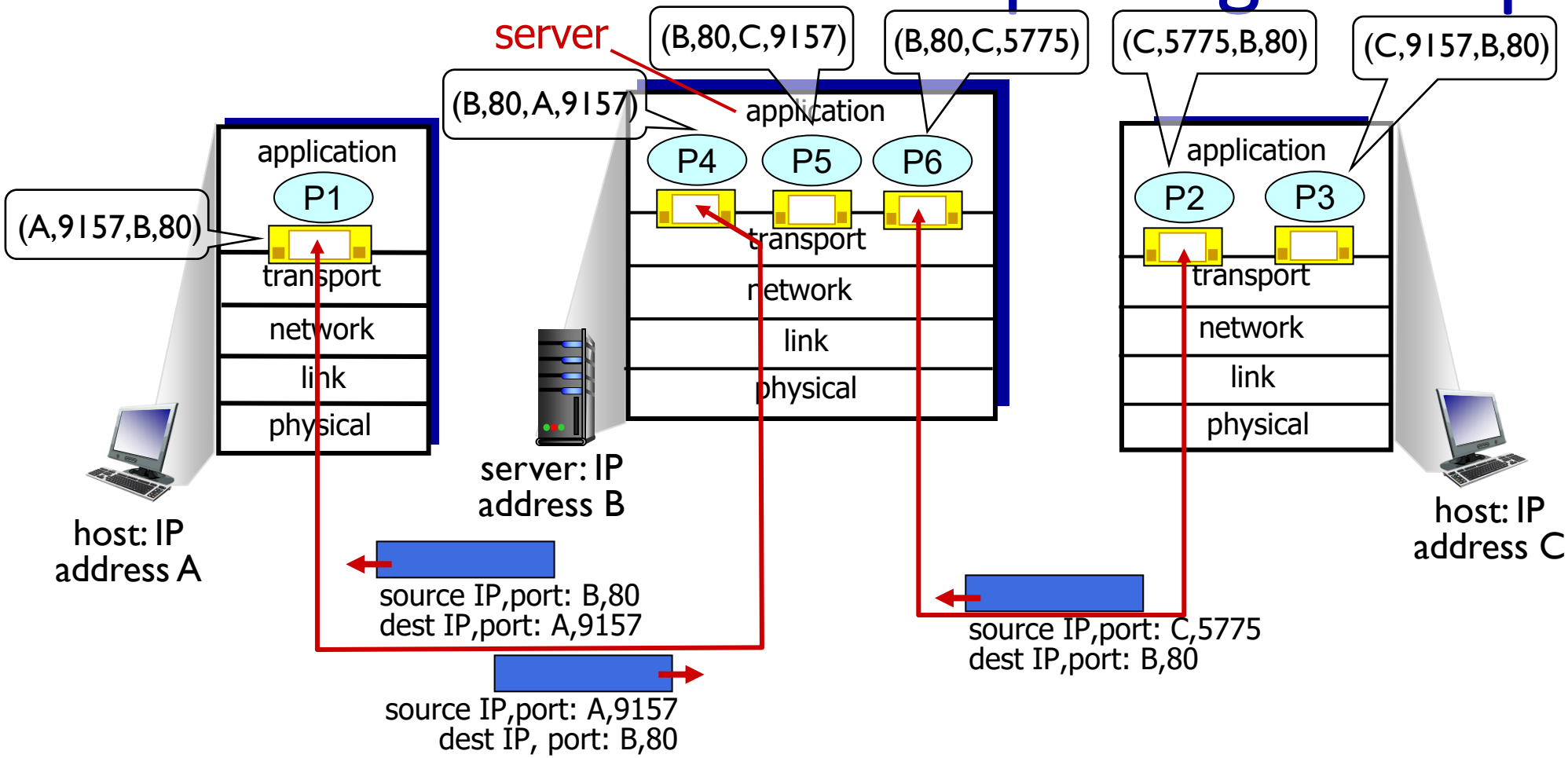
Connection-oriented demultiplexing: example



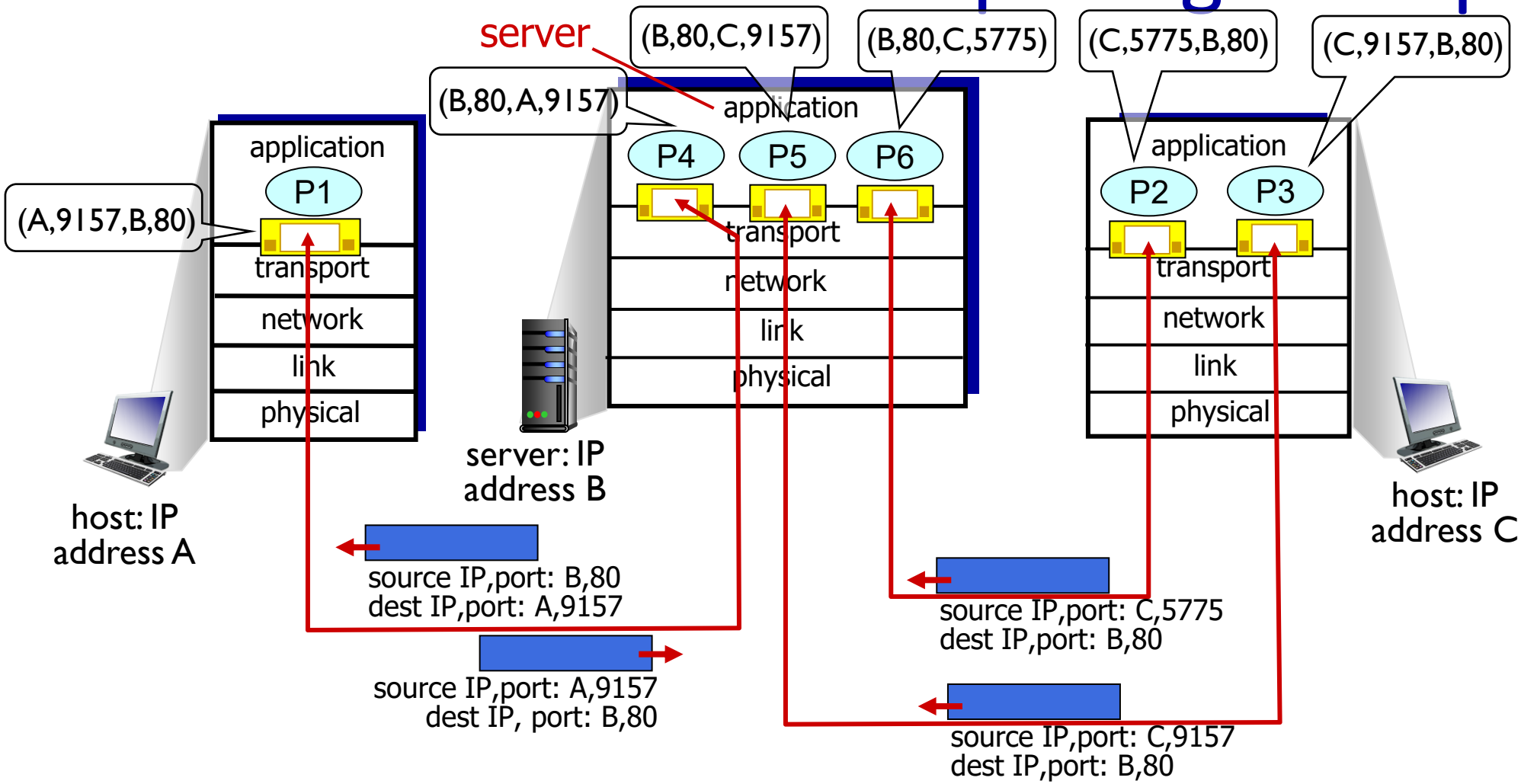
Connection-oriented demultiplexing: example



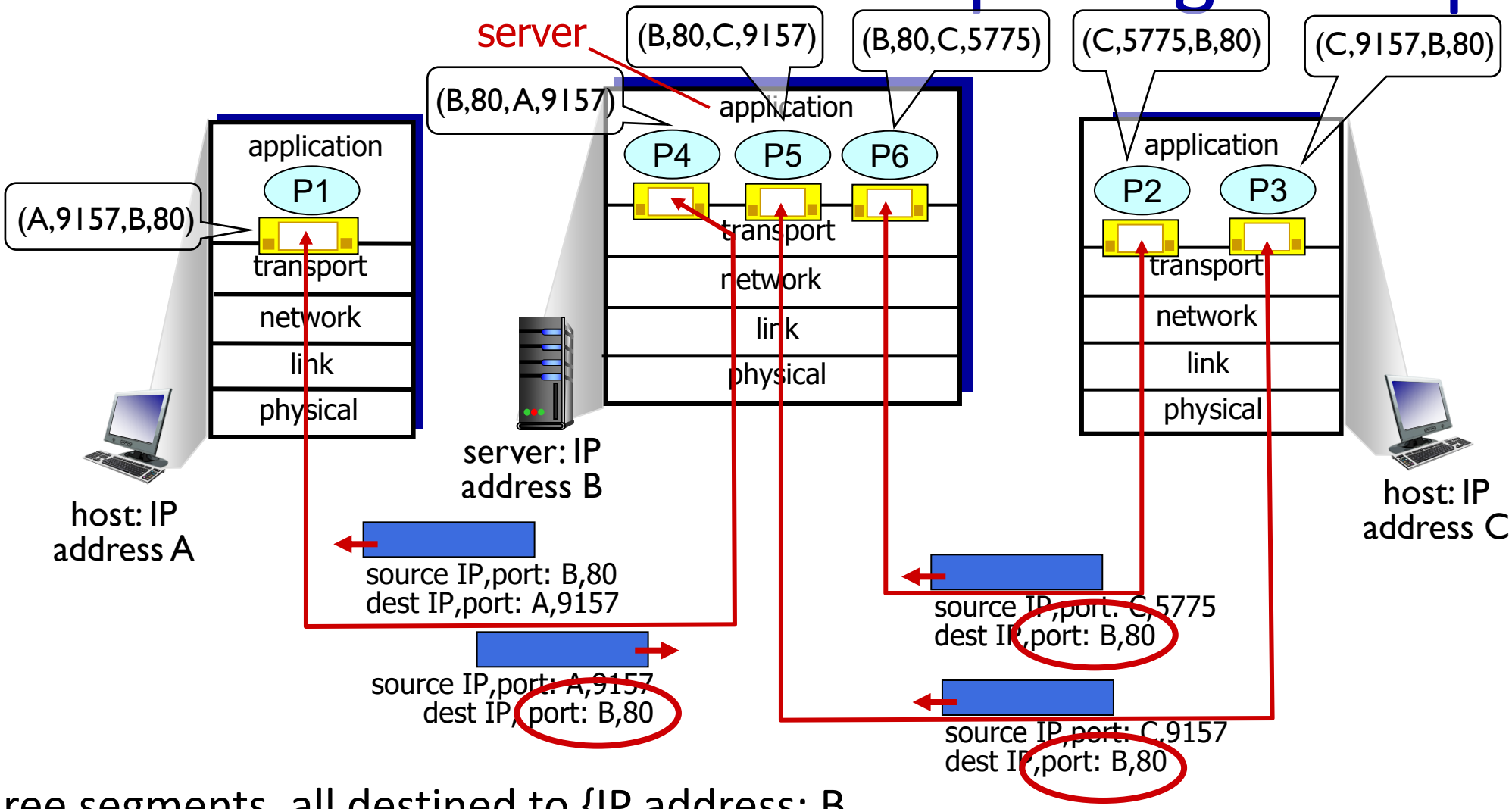
Connection-oriented demultiplexing: example



Connection-oriented demultiplexing: example



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 destination port number and IP address (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers

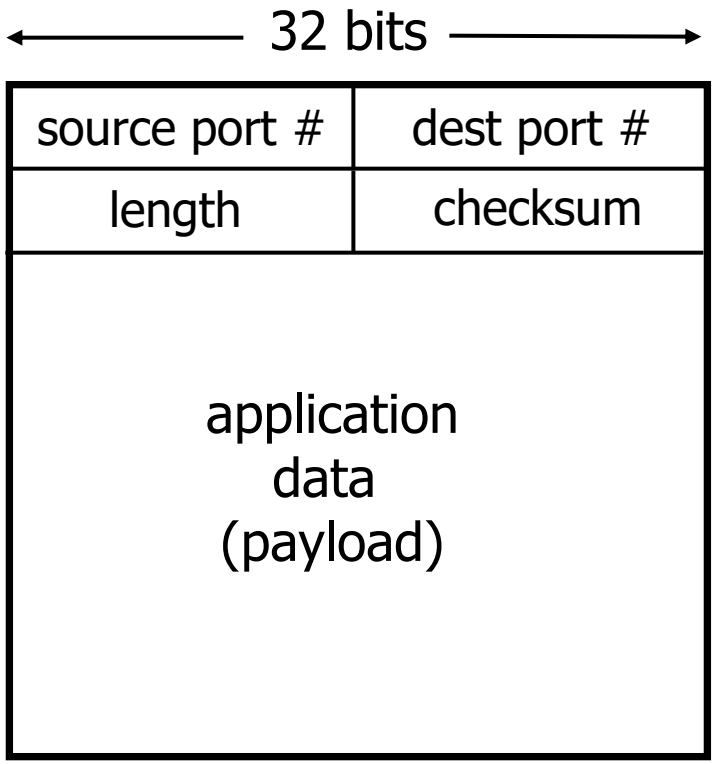
UDP: User Datagram Protocol

- "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?

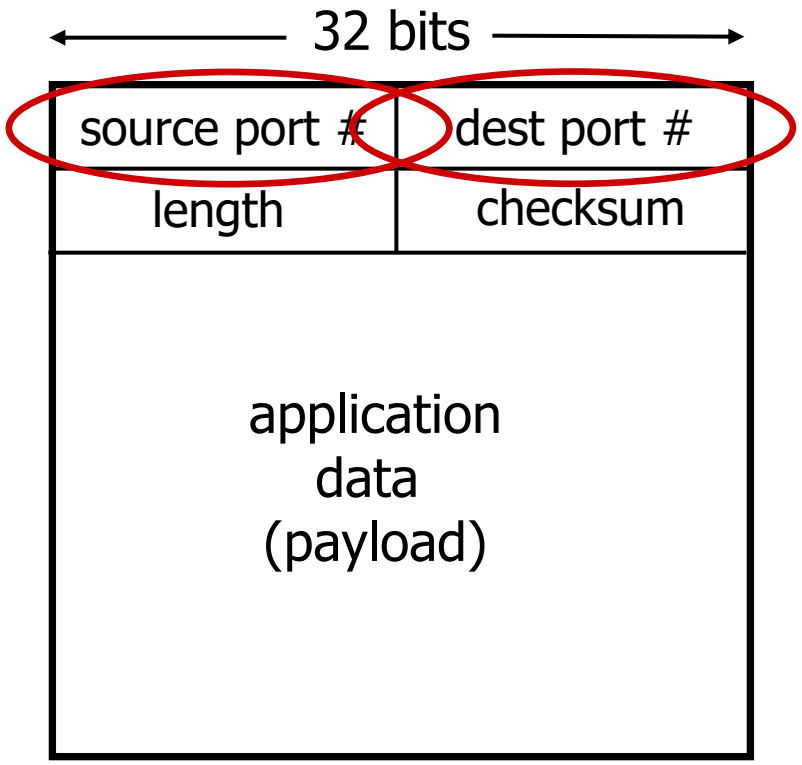
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

UDP segment header



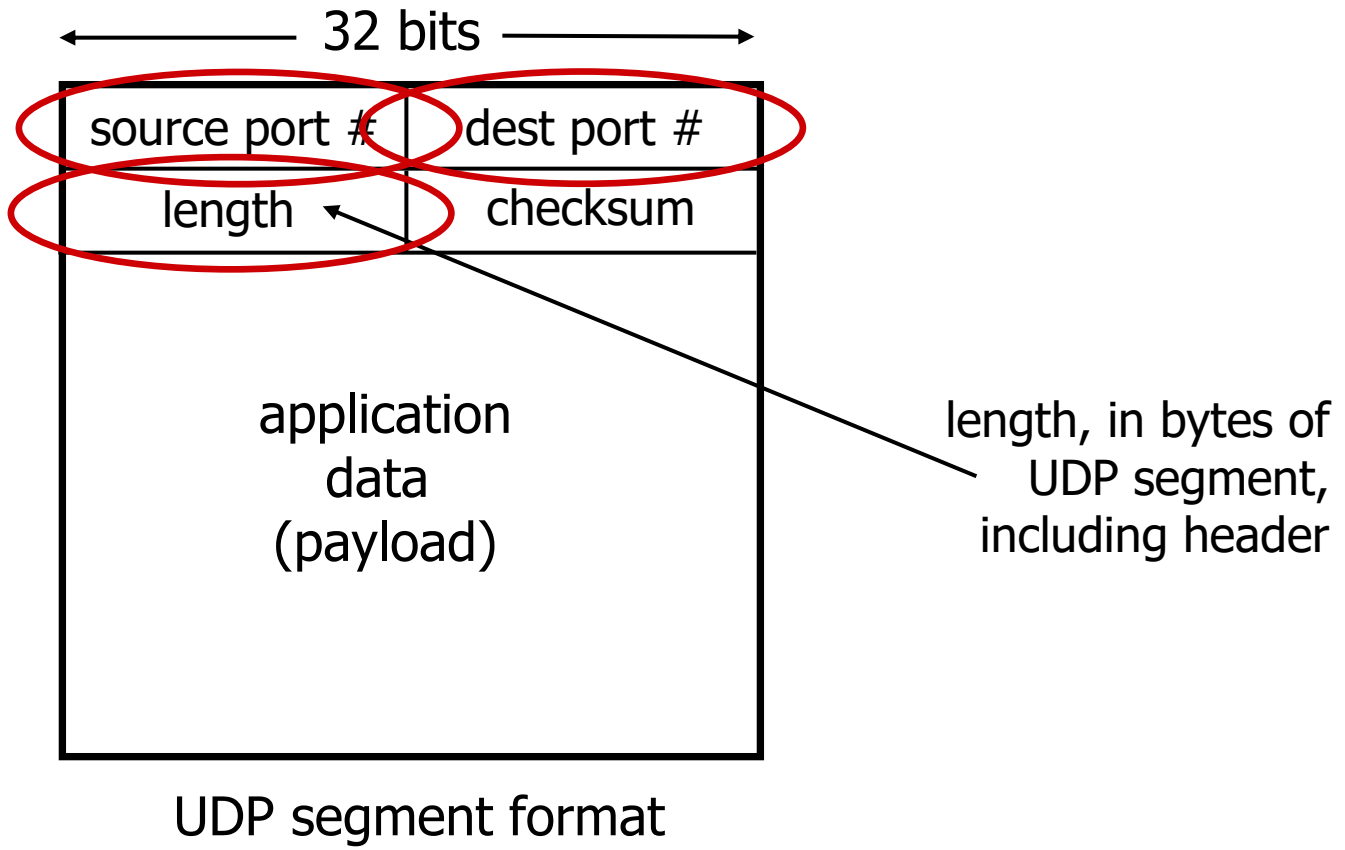
UDP segment format

UDP segment header



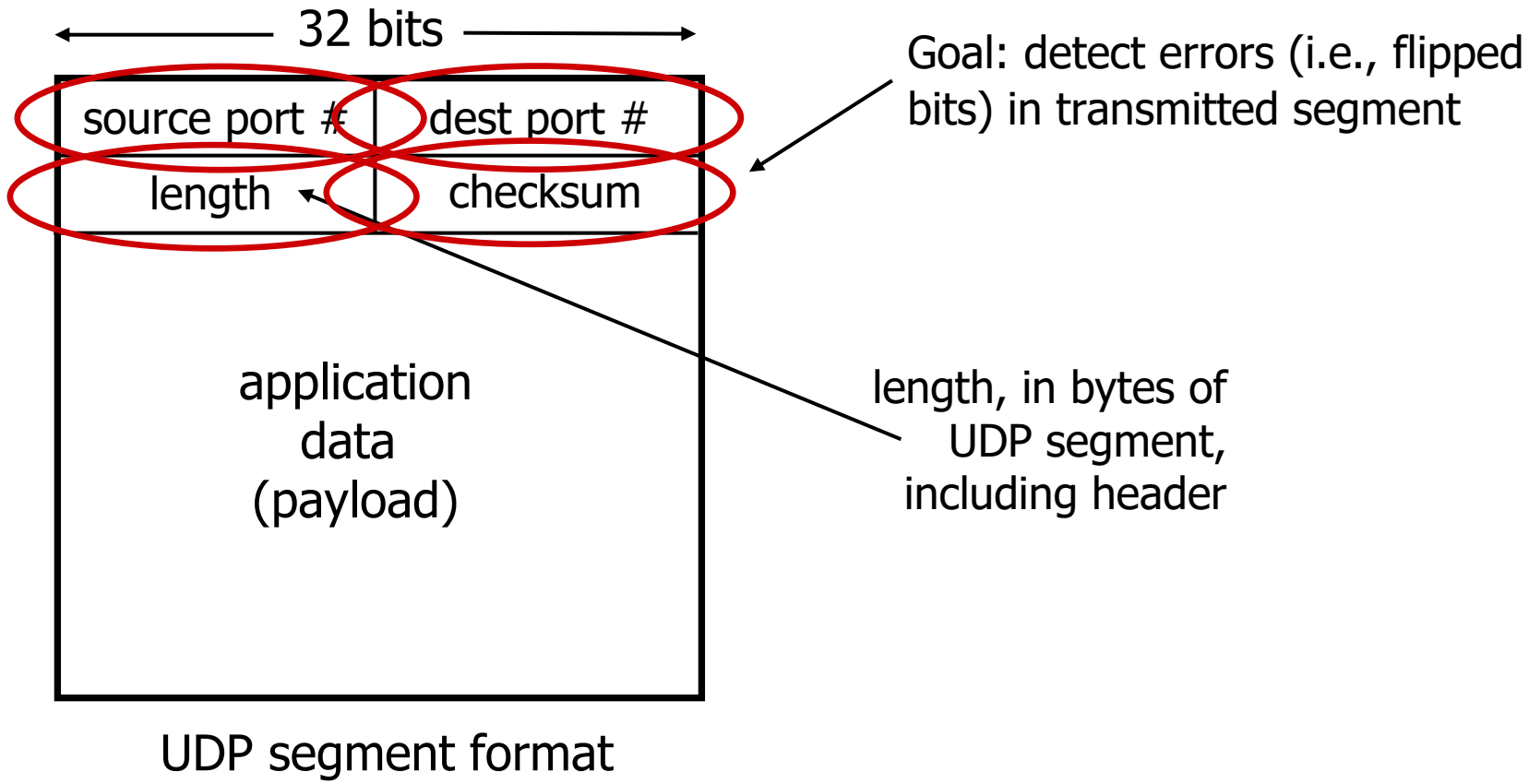
UDP segment format

UDP segment header



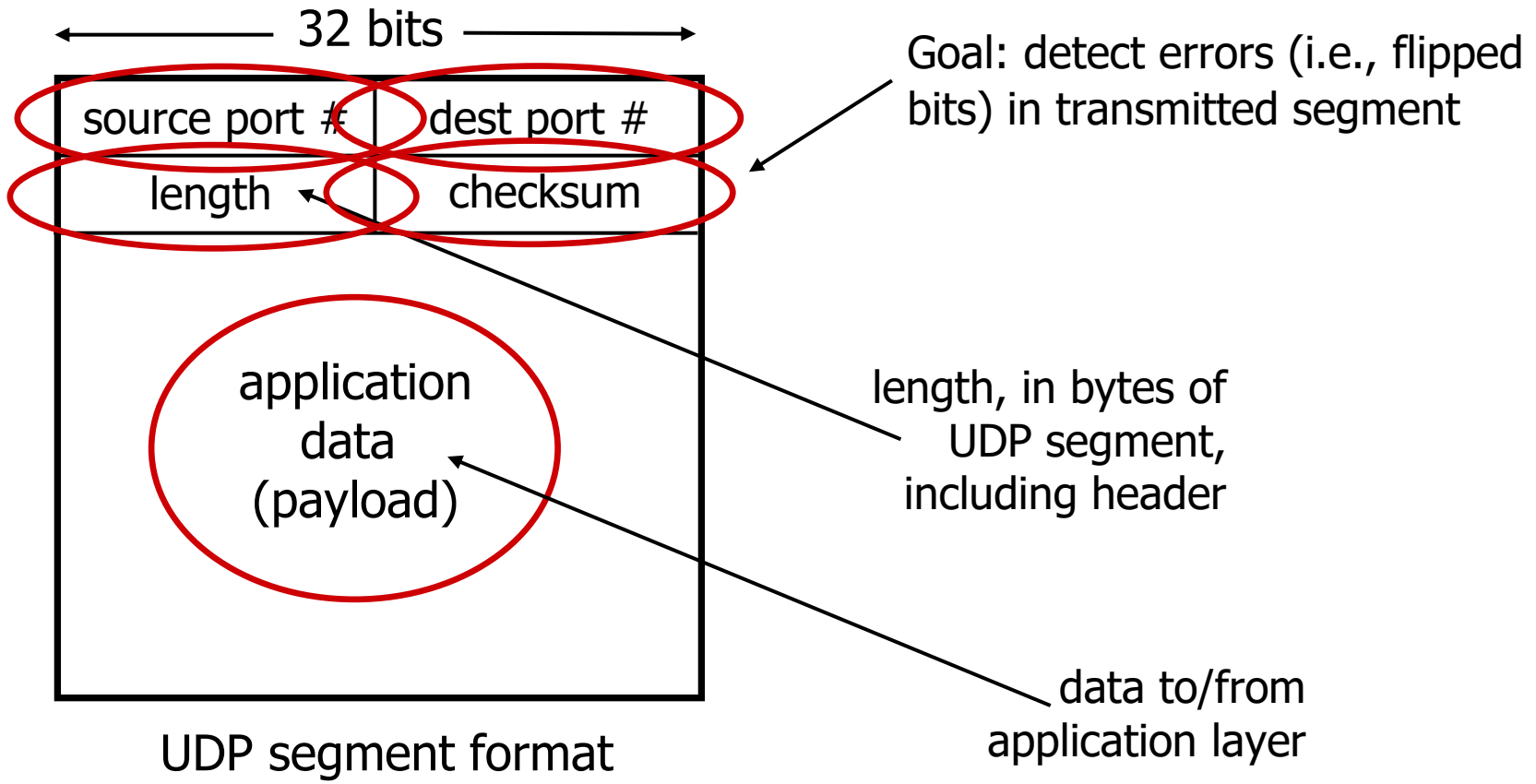
UDP segment format

UDP segment header



UDP segment format

UDP segment header



Summary: UDP

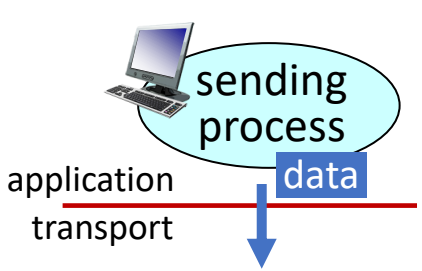
- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no delay incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer

Principles of reliable data transfer

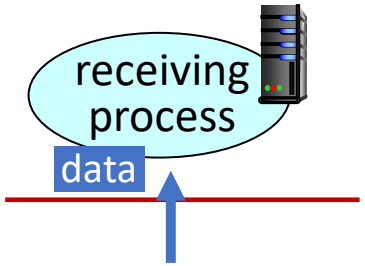
reliable service *abstraction*

Principles of reliable data transfer

reliable service *abstraction*



sender-side of reliable data transfer protocol

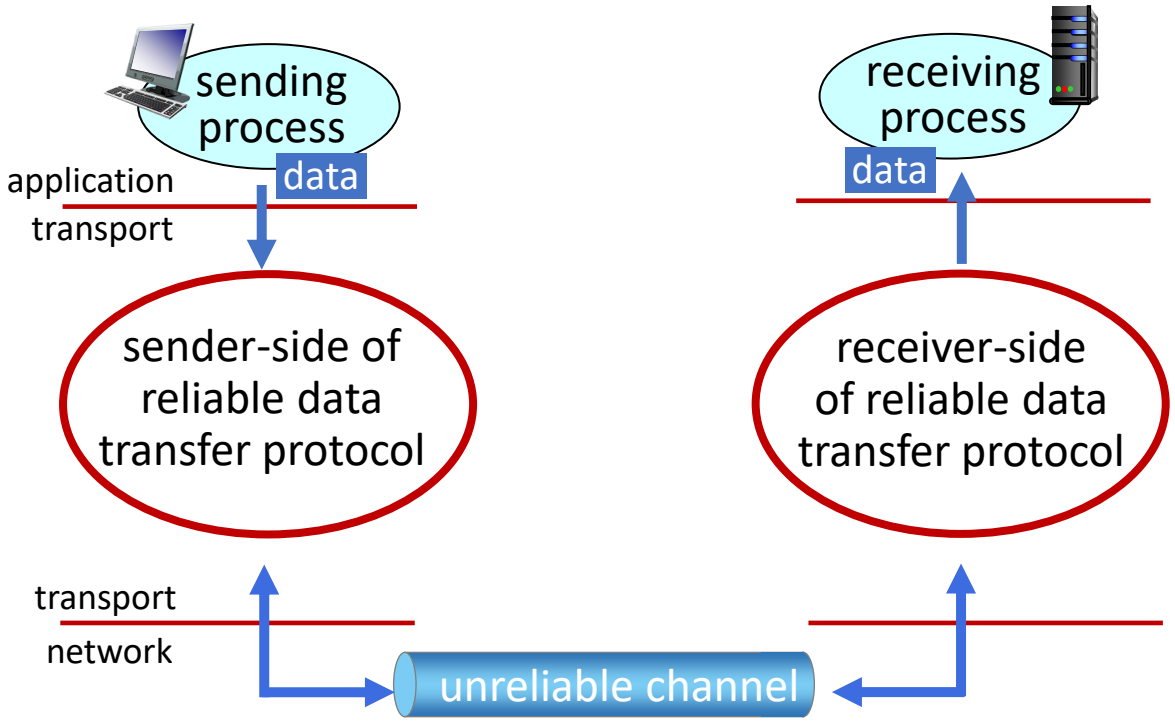


receiver-side of reliable data transfer protocol

reliable service *implementation*

Principles of reliable data transfer

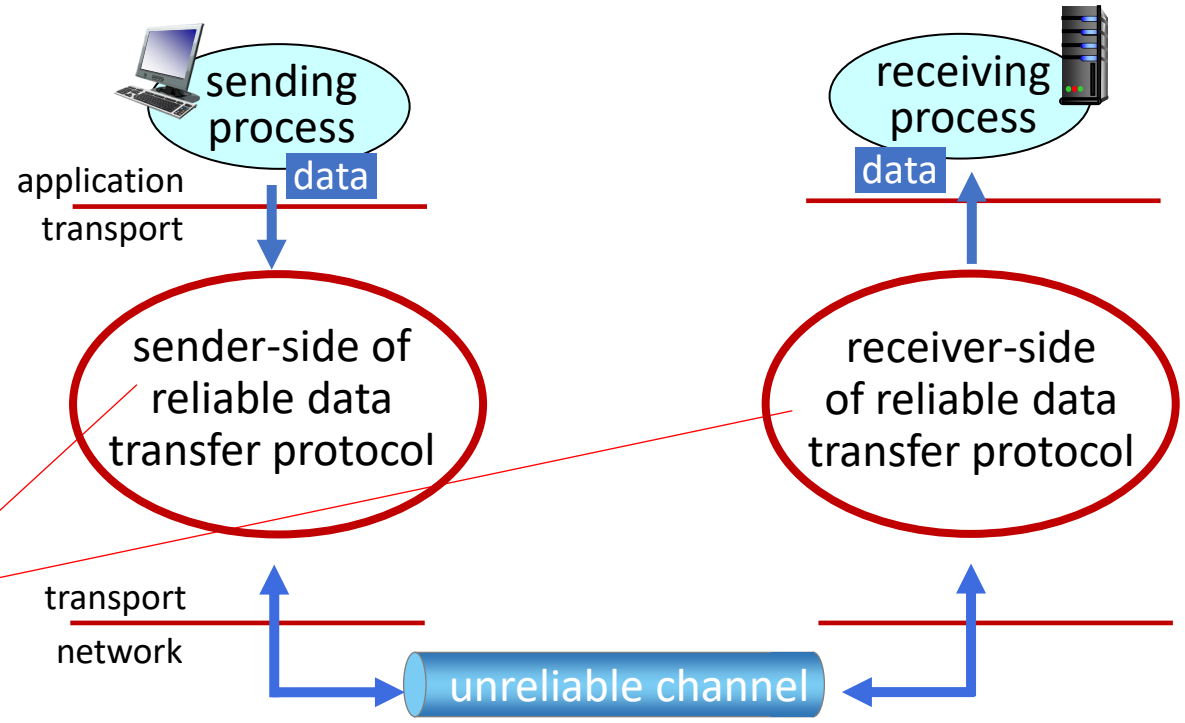
reliable service *abstraction*



reliable service *implementation*

Principles of reliable data transfer

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

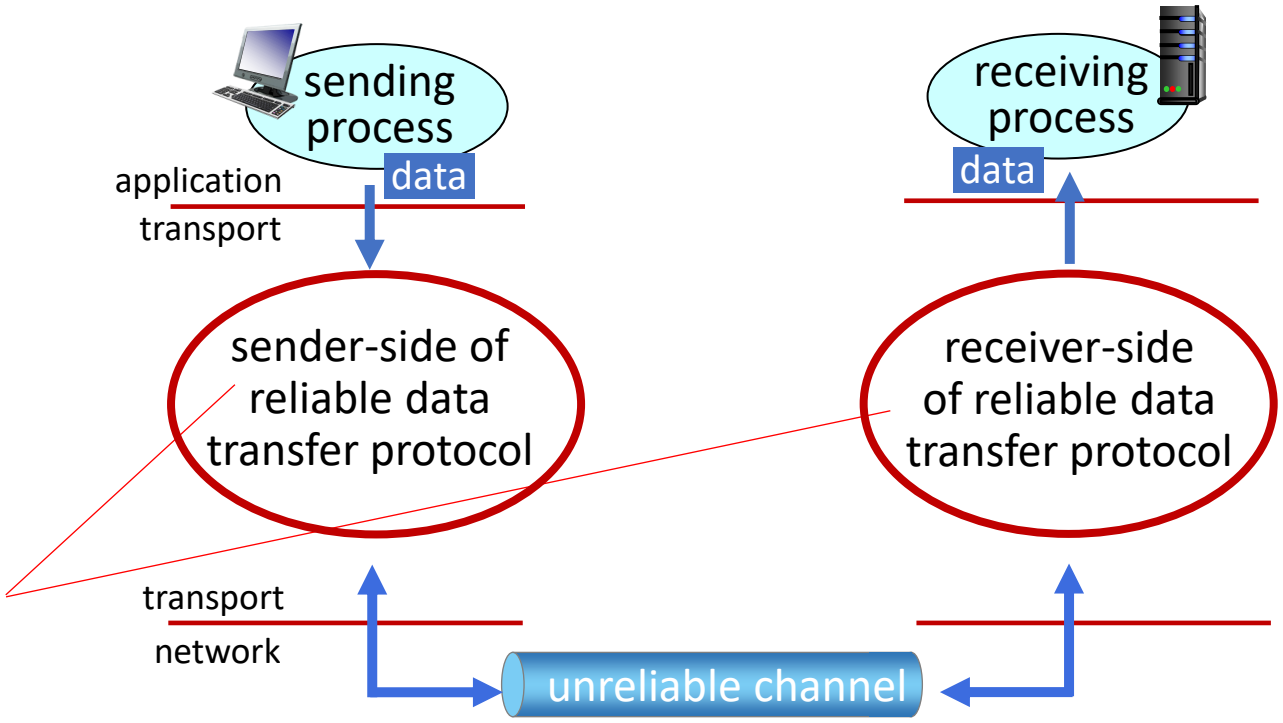


reliable service *implementation*

Principles of reliable data transfer

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

- **unless communicated via a message**

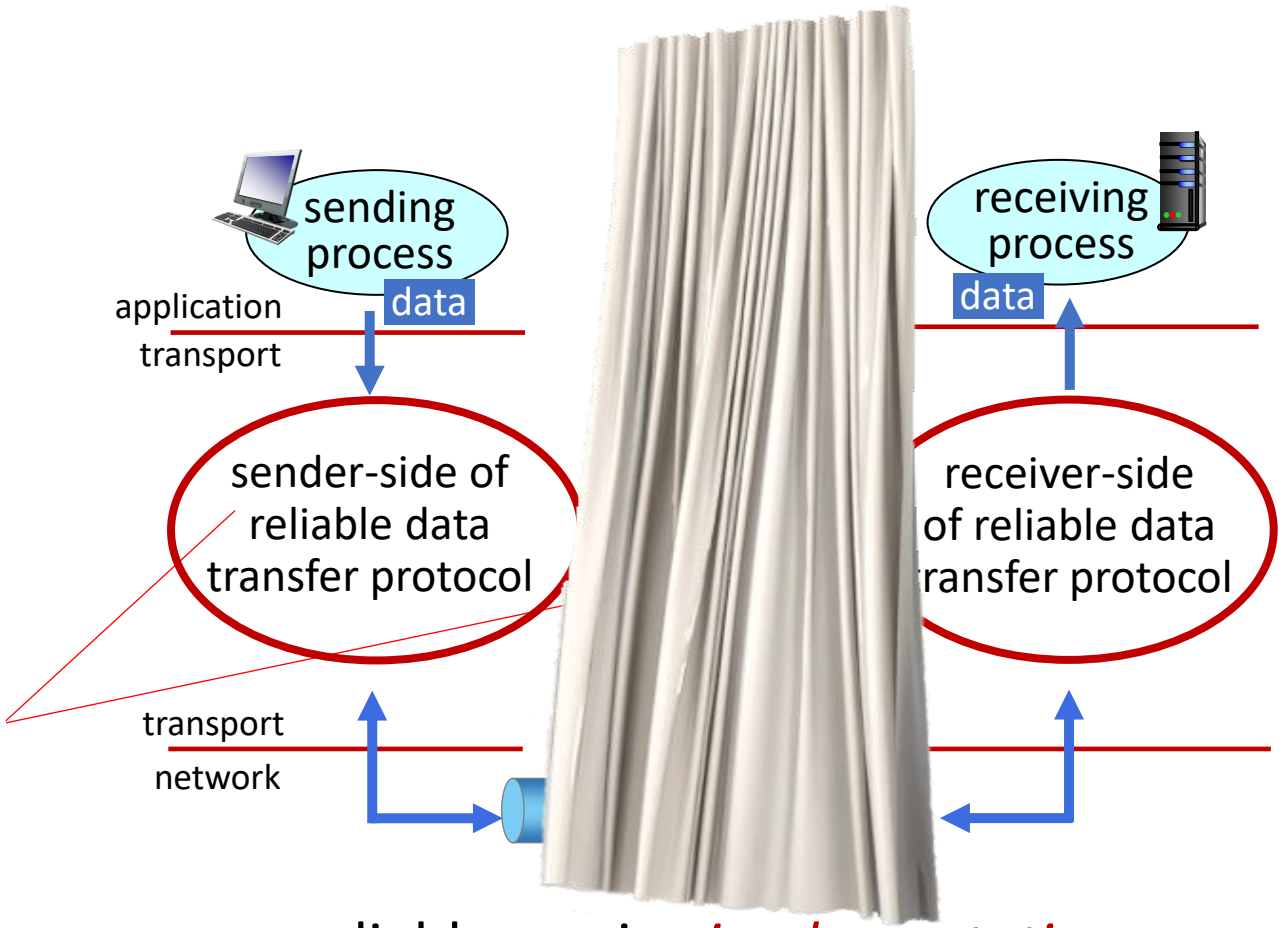


reliable service *implementation*

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**

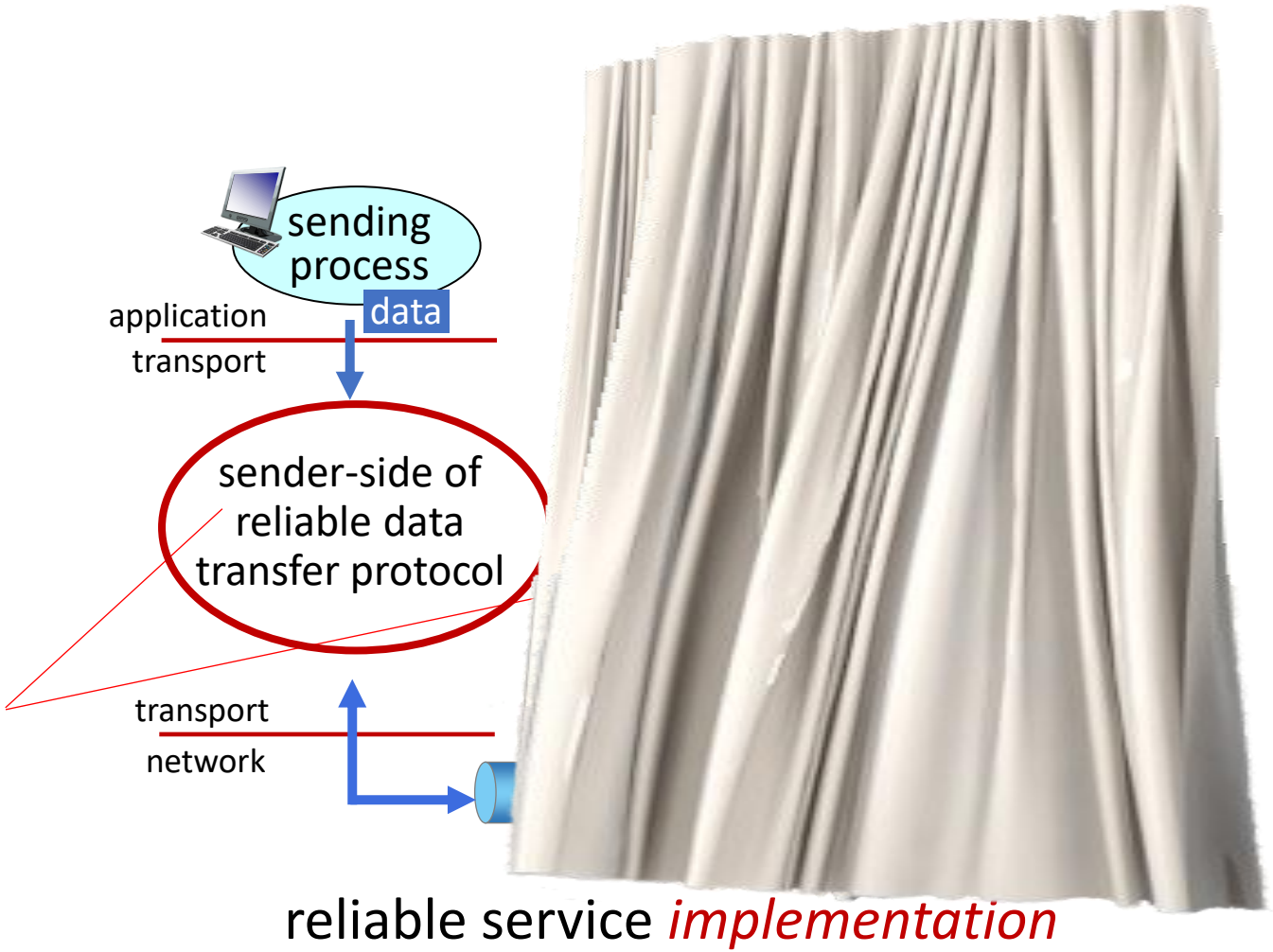


reliable service *implementation*

Principles of reliable data transfer

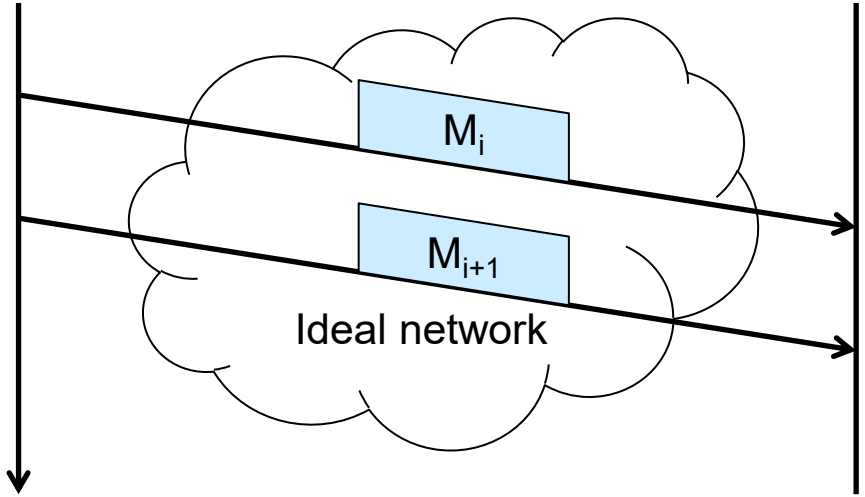
Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- **unless communicated via a message**



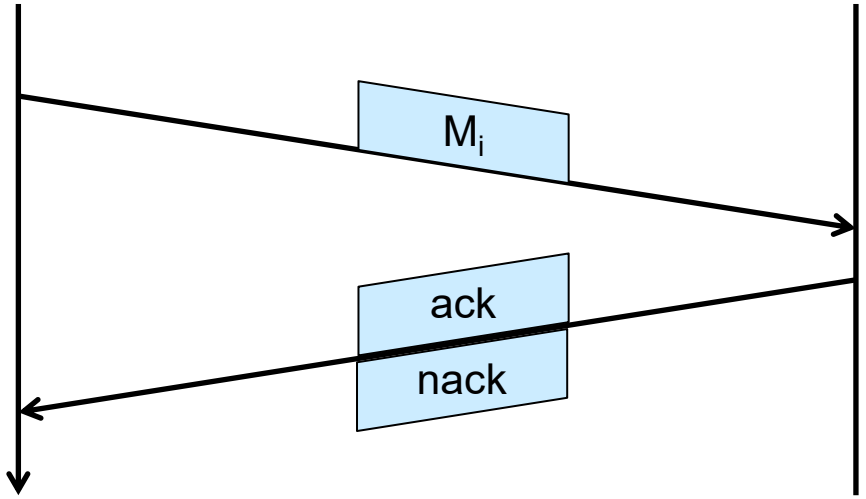
reliable service *implementation*

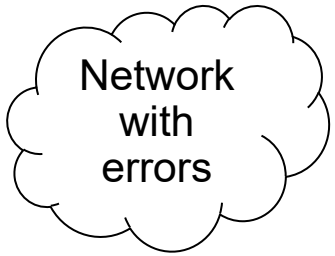
Reliable data transfer



- Let's consider an «**ideal network**», meaning that it does not introduce
  - Bit errors
  - Discards of segments
  - Out-of-sequence segments
- The trasport layer does not need to correct anything and the protocol is trivial
  - The sender sends segments in sequence (one after the other) and the receiver receives them without any need of further checks

Reliable data transfer



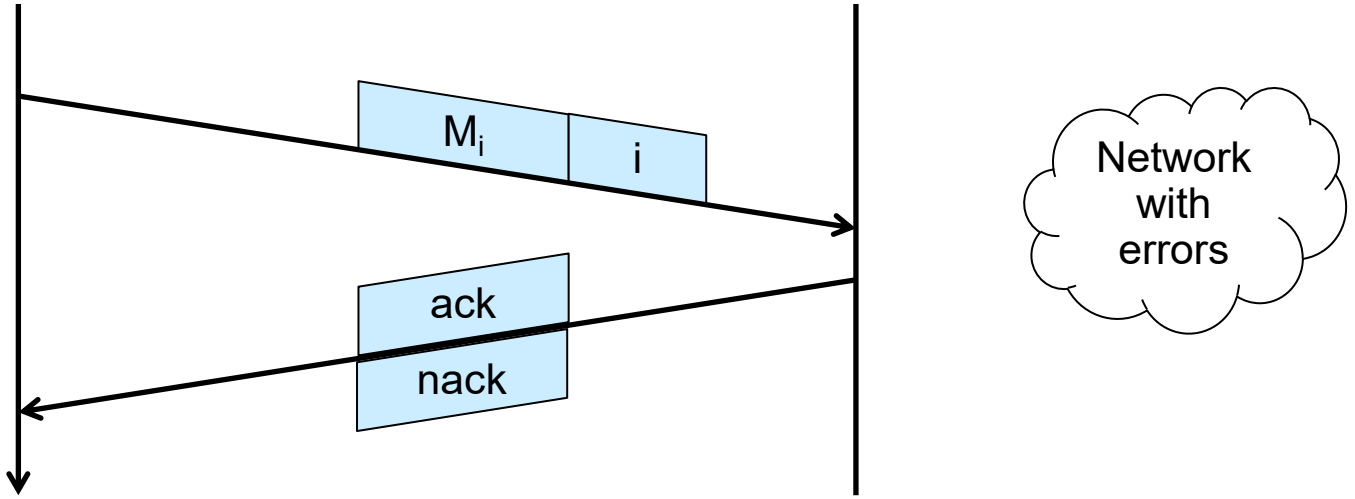


- Unfortunately, ideal networks do not exist…
- In a network with errors it is possible to introduce **positive acknowledgments** (ack) or **negative acknowledgments** (nack)
- Simple possible sender algorithm:

```
IF ack
THEN M_{i+1}
ELSE IF nack
       THEN M_i
       ELSE ?
```

- However, **it does not work**! Ack/nack can also be subject to errors!
  - Possible mitigation strategy: if the ack/nack is corrupted, just retransmit the message → It does not work either! Impossible to understand at the receiver if the segment is a duplicate or not!

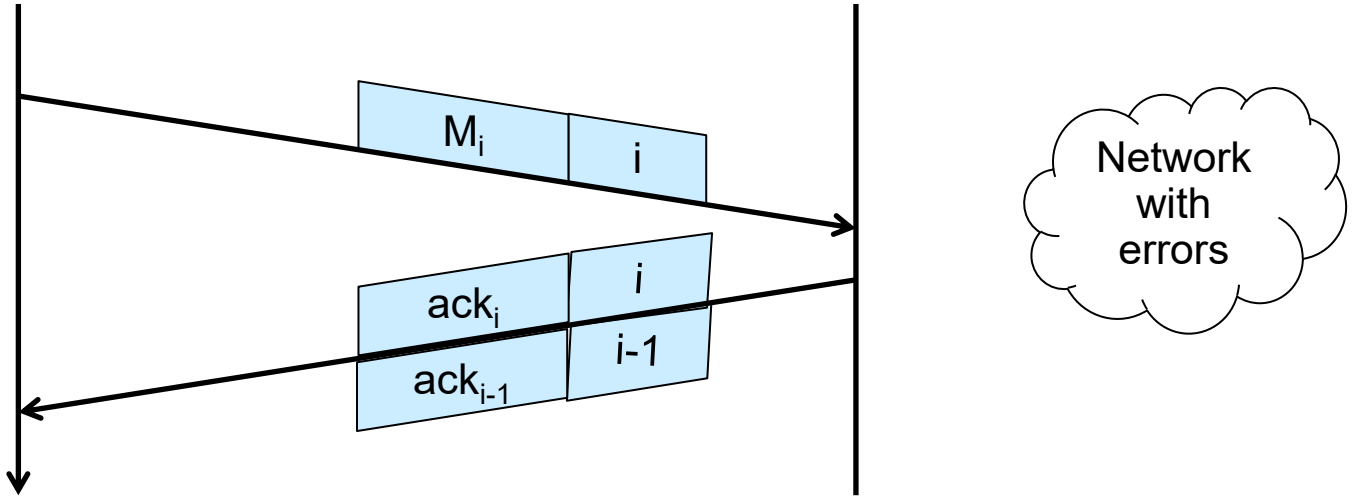
Reliable data transfer



- If the **segments are numbered**, there is no risk also in the case of duplicates

```
IF ack
THEN M_{i+1}
ELSE M_i        ← A nack is received
```

Reliable data transfer

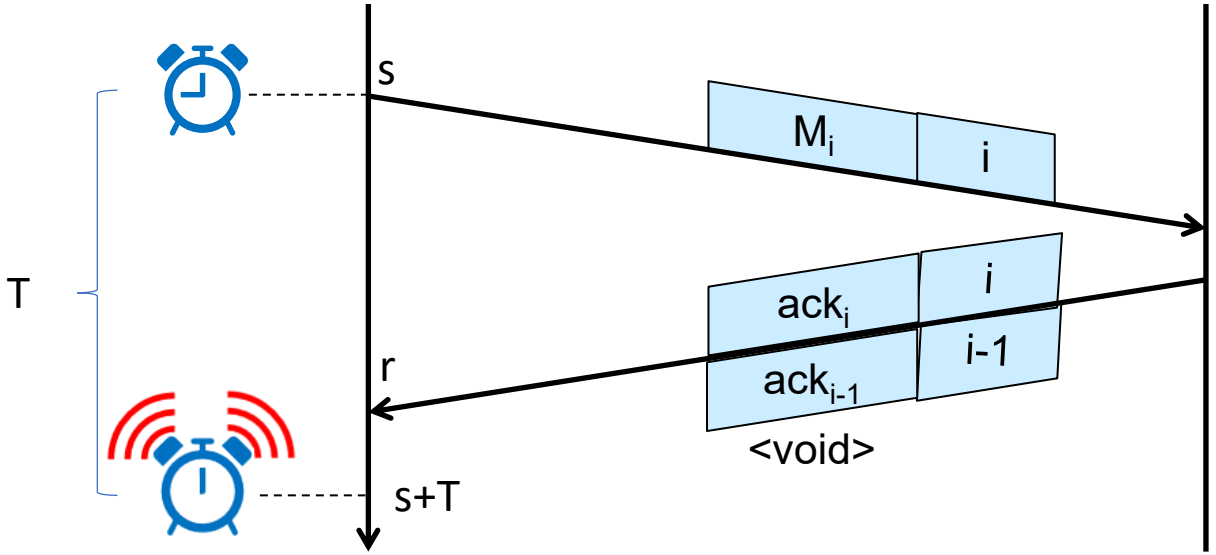


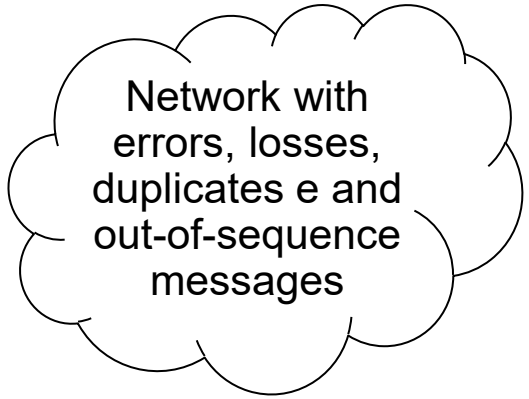
- By also **numbering the acks**, it is possible to avoid nacks thanks to the following rule: *a second ack_{i-1} is equal to a $nack_i$*

```
IF ack_i
THEN M_i+1
ELSE M_i        ← An ack_i-1 is received
```

- Unfortunately, network with errors but without losses do not exist

Reliable data transfer



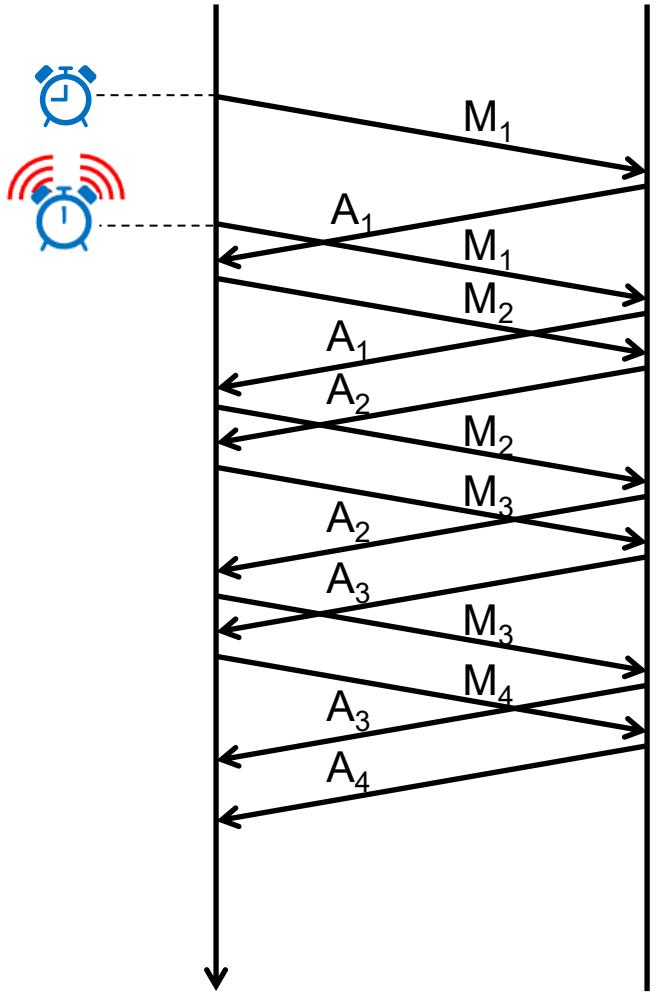


- By adding a **timer** we can handle losses (of segments or of acks)

```
IF s+T is reached
THEN Mi, set (s+T)+T
ELSE IF acki      ← s+T has not been reached yet (r)
    THEN Mi+1, set r+T
    ELSE Mi, set r+T
```

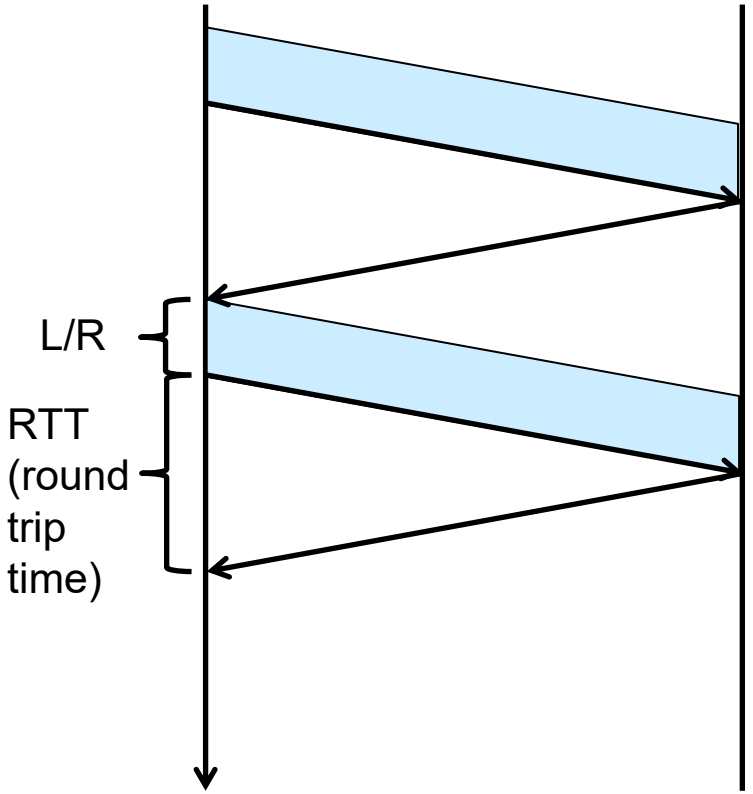
- This is the most sophisticated implementation of a **stop-and-wait** protocol

Reliable data transfer



- Setting the timer's deadline is a complex problem
- A **too long timer** stops the transmission for long periods in the case of a loss
- A **too short timer** may cause useless retransmissions
  - In the case seen so far (stop-and-wait) there may be even very long sequences of useless retransmissions

Reliable data transfer

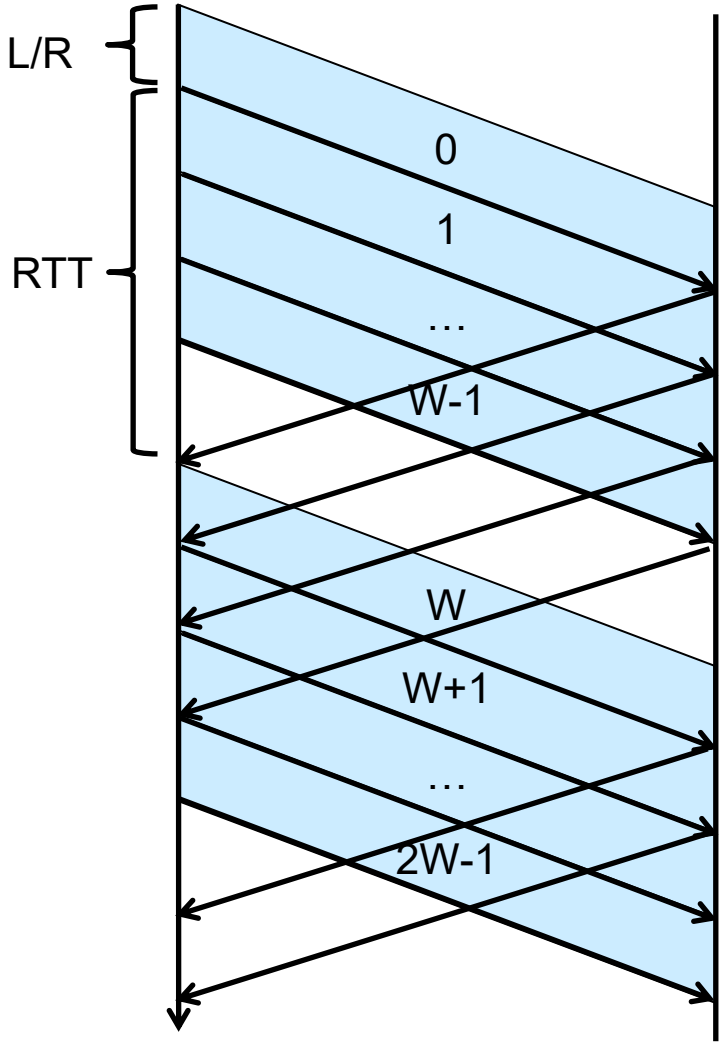


- Transmitting a segment at a time and waiting for its ack before a further transmission (stop-and-wait) significantly limits performance

$$U = \frac{L/R}{RTT + L/R}$$

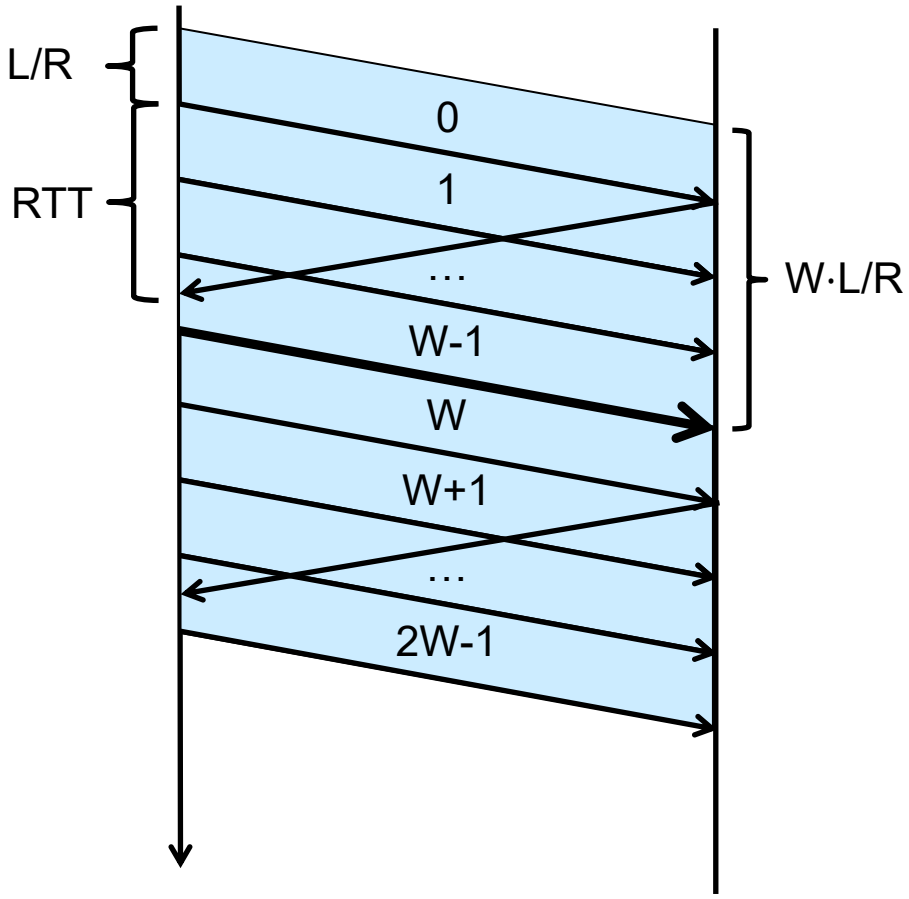
- Example: $RTT = 100\ ms$, $L = 1\ kbyte$, $R = 100\ Mbit/s \rightarrow U = 0{,}0008$

Reliable data transfer



- **Sliding window protocols** can transit up to W segments while waiting the ack of the first one
  - W is the **transmission window** size
  - The transmission window slides by one segment after any ack reception

Reliable data transfer



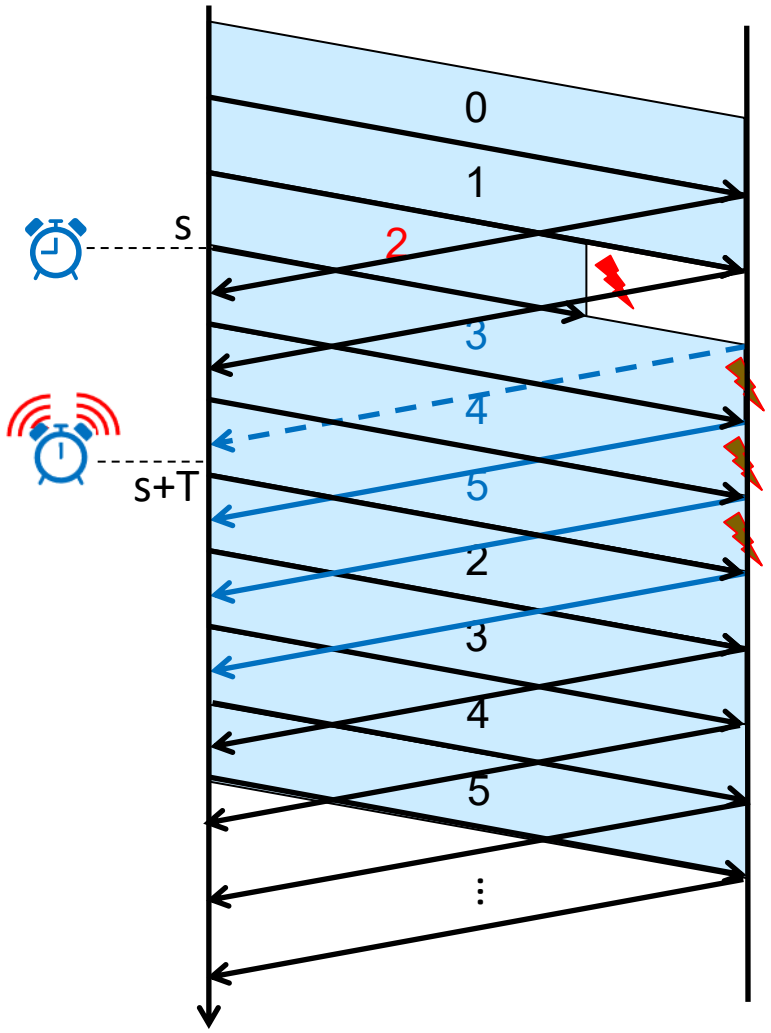
- The condition for a continuous transmission ($U = 1$) is that the transmission window will not close before the arrival of the first ack

$$W \cdot {}^{L}/_{R} \geq RTT + {}^{L}/_{R}$$

$$W \geq \frac{RTT \cdot R}{L} + 1$$

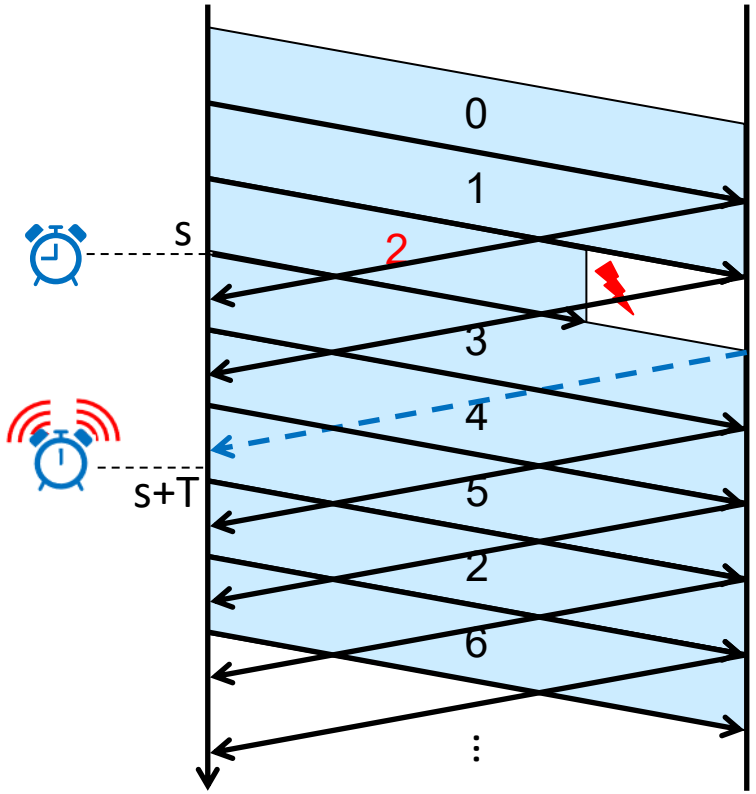
- In the example of two slides ago: $W \geq 1251$

Reliable data transfer



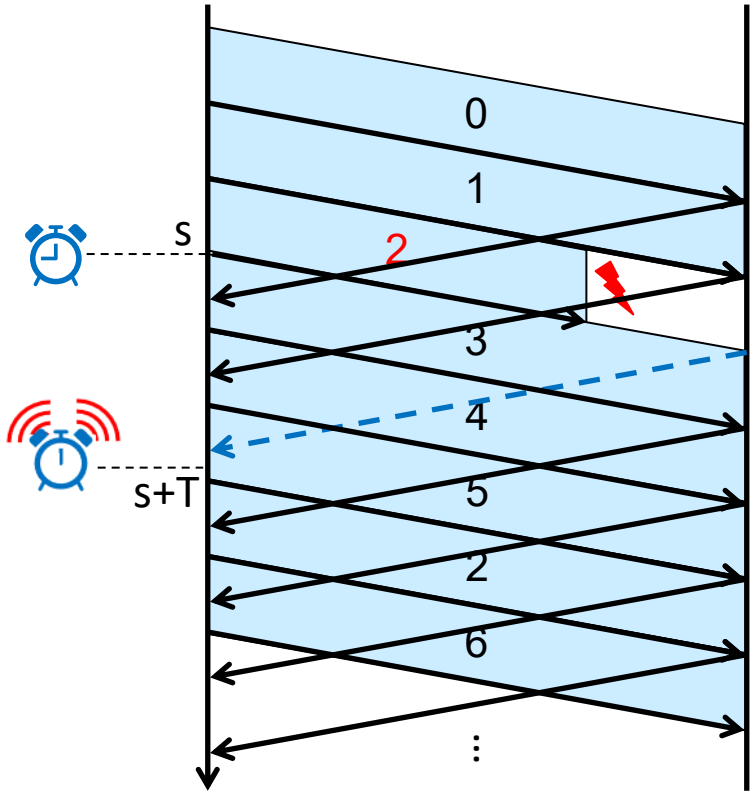
- What is retransmitted when a segment is lost?
- First possibility: **Go-Back-N** protocol
  - If s+T is reached for a segment, retransmit all the segments starting from it
- There is no need of buffering at the receiver
  - Out-of-order segments received after a loss are simply discarded
  - No further duplications once the segment in order is retransmitted
- Acks are cumulative
  - It is possible to recover from multiple ack losses
- One timer is sufficient
  - It is restarted once an ack is received
  - It always refers to the oldest segment without ack

Reliable data transfer



- Second possibility: **Selective Repeat** protocol
  - If s+T is reached only the lost segment is retransmitted
- A buffer is needed at the receiver
  - Segments are reordered and sent to the application in the right order
- Acks are individual
- Timers are associated to single segments

Reliable data transfer



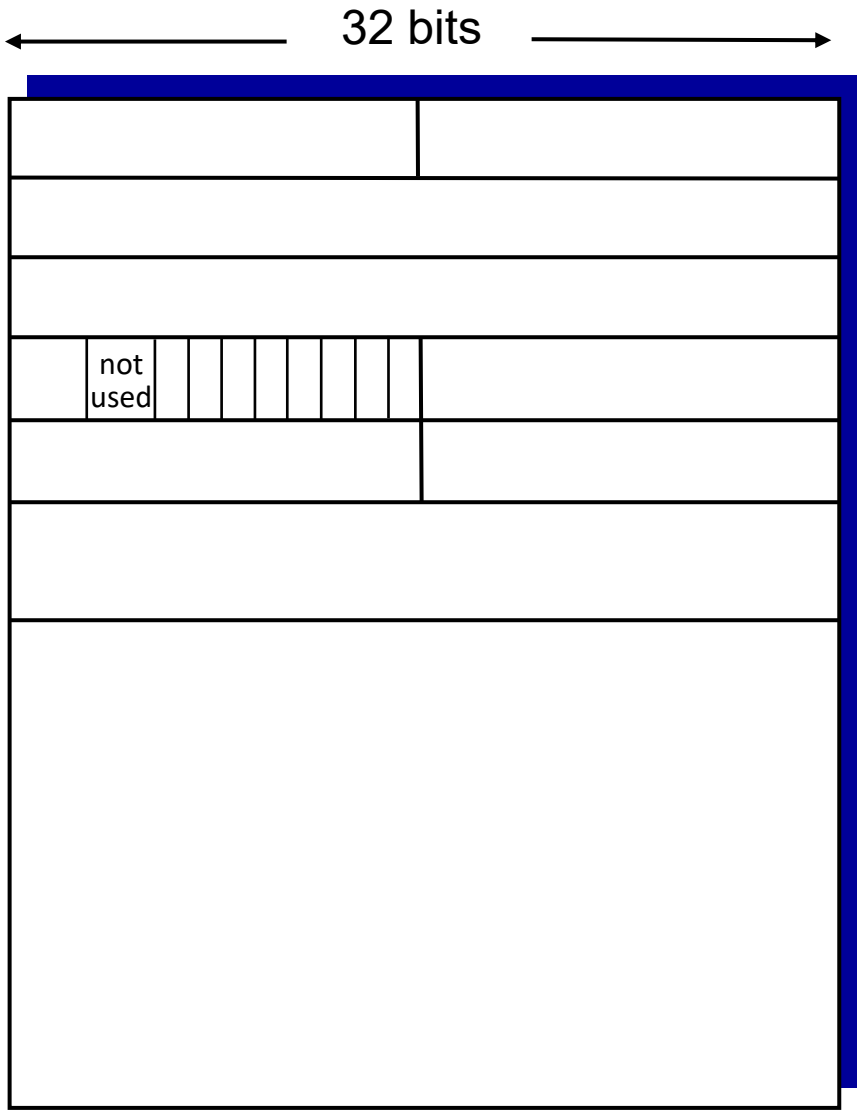
- Second possibility: **Selective Repeat** protocol
  - If s+T is reached only the lost segment is retransmitted
- A buffer is needed at the receiver
  - Segments are reordered and sent to the application in the right order
- Acks are individual
- Timers are associated to single segments

- Observation:
  - There are not only two possibilities (i.e., pure Go-Back-N and pure Selective Repeat): real protocols as TCP rely on hybrid solutions

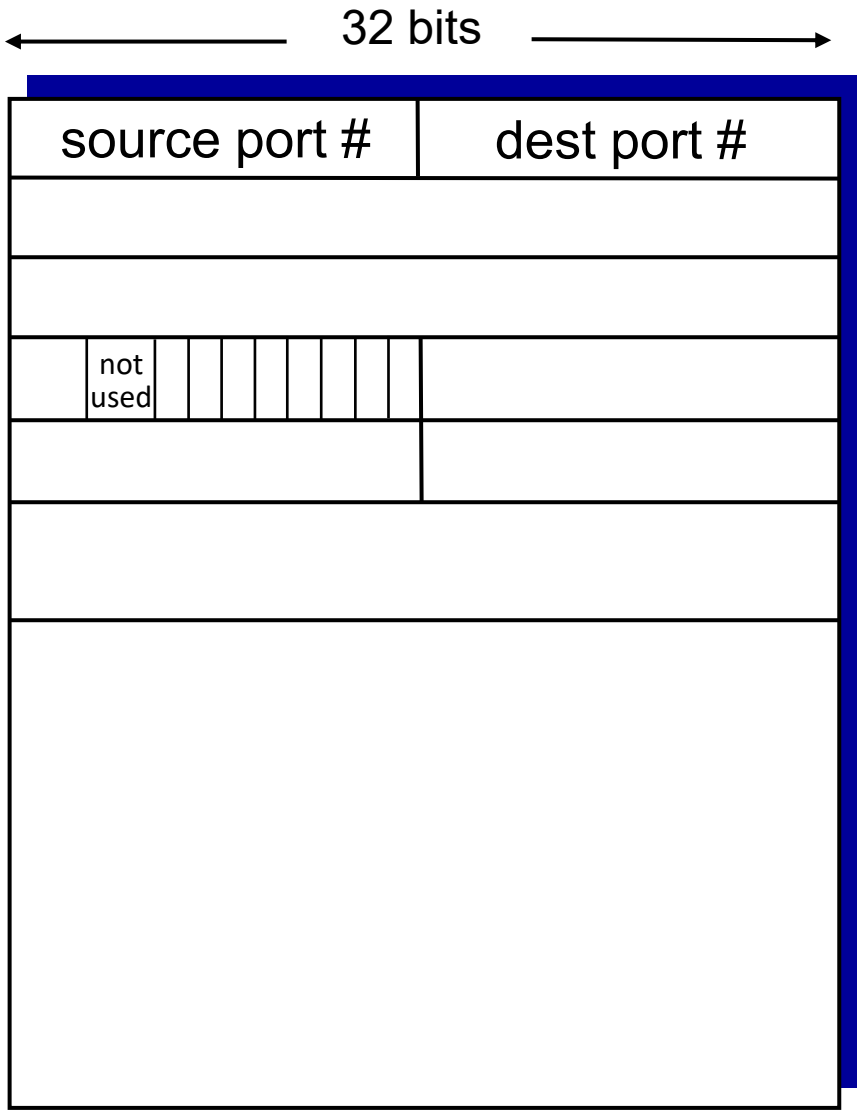
TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - counting bytes of data into byte stream (not segments!)
  - MSS: maximum segment size
- full-duplex data:
  - bi-directional data flow in same connection
- cumulative ACKs
- sliding window:
  - TCP congestion and flow control set transmission window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange

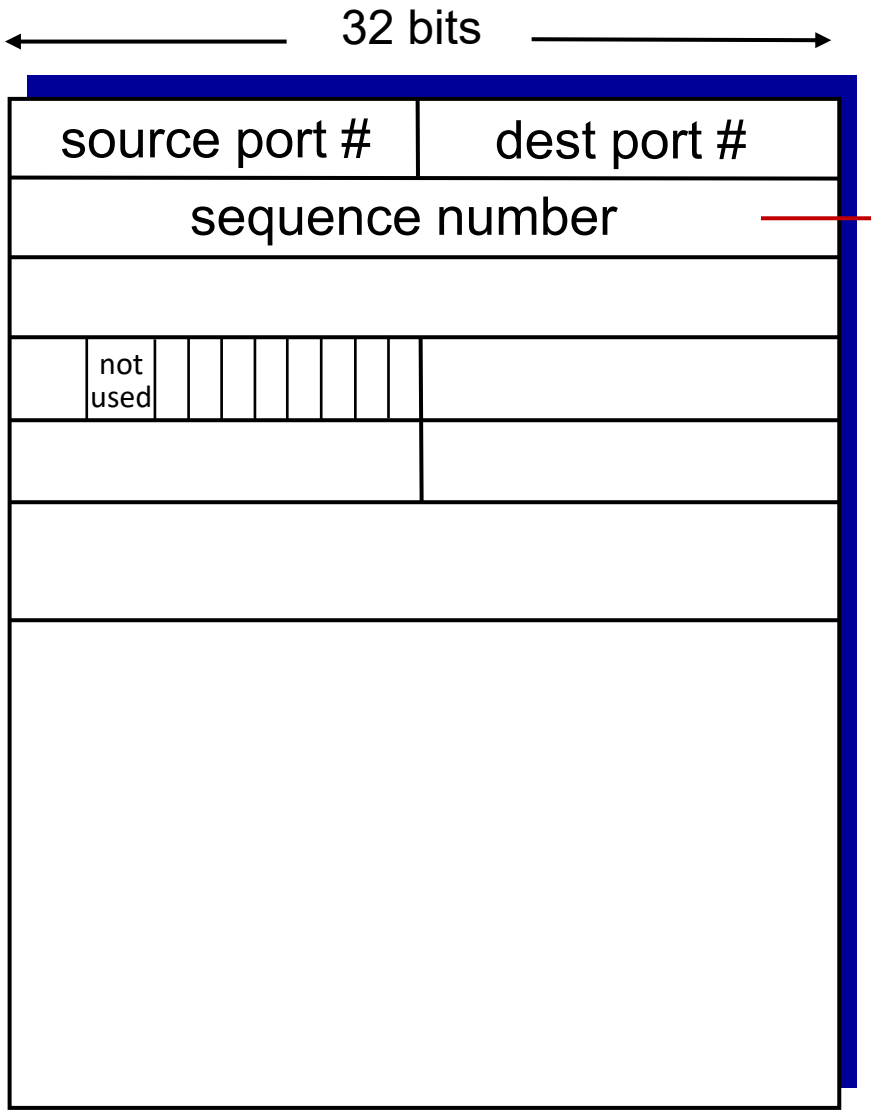
TCP segment structure



TCP segment structure

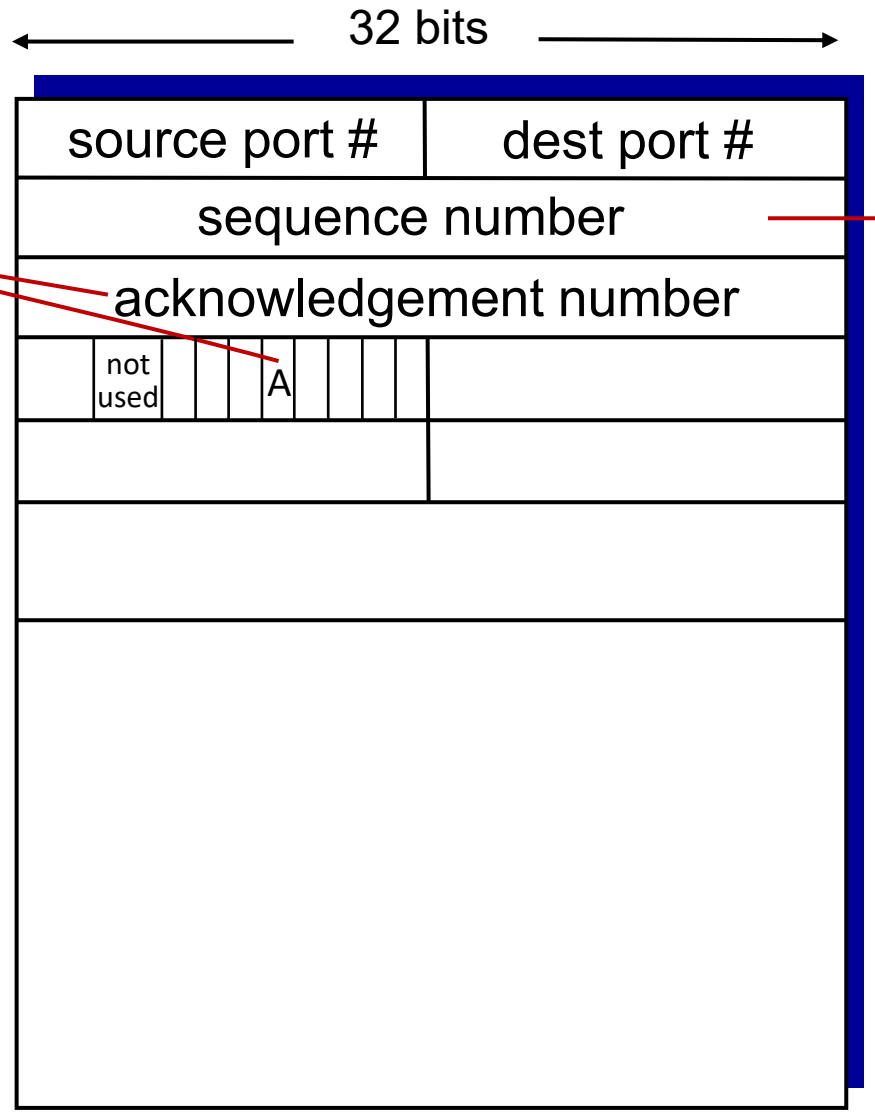


TCP segment structure



segment seq #: byte stream number of first byte in segment's data

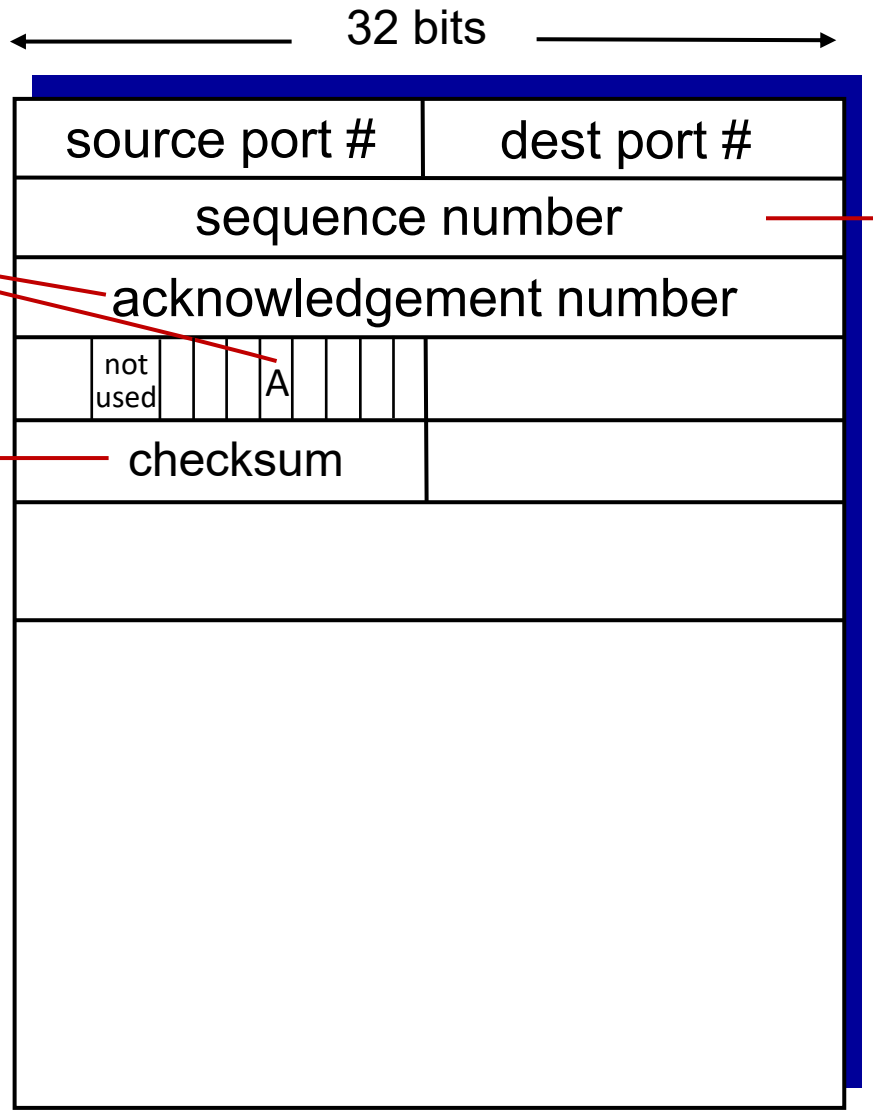
TCP segment structure



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

segment seq #: byte stream number of first byte in segment's data

TCP segment structure

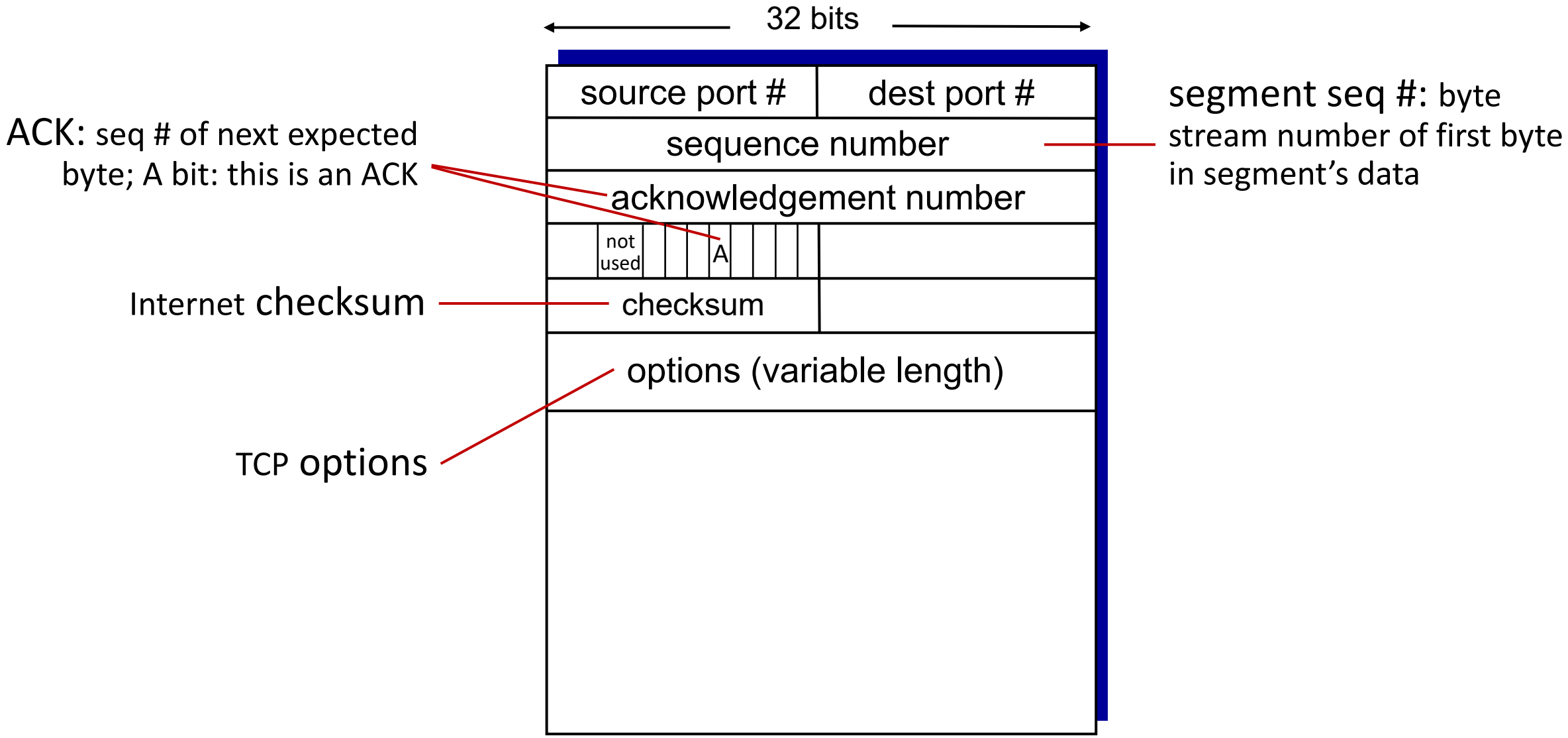


ACK: seq # of next expected byte; A bit: this is an ACK

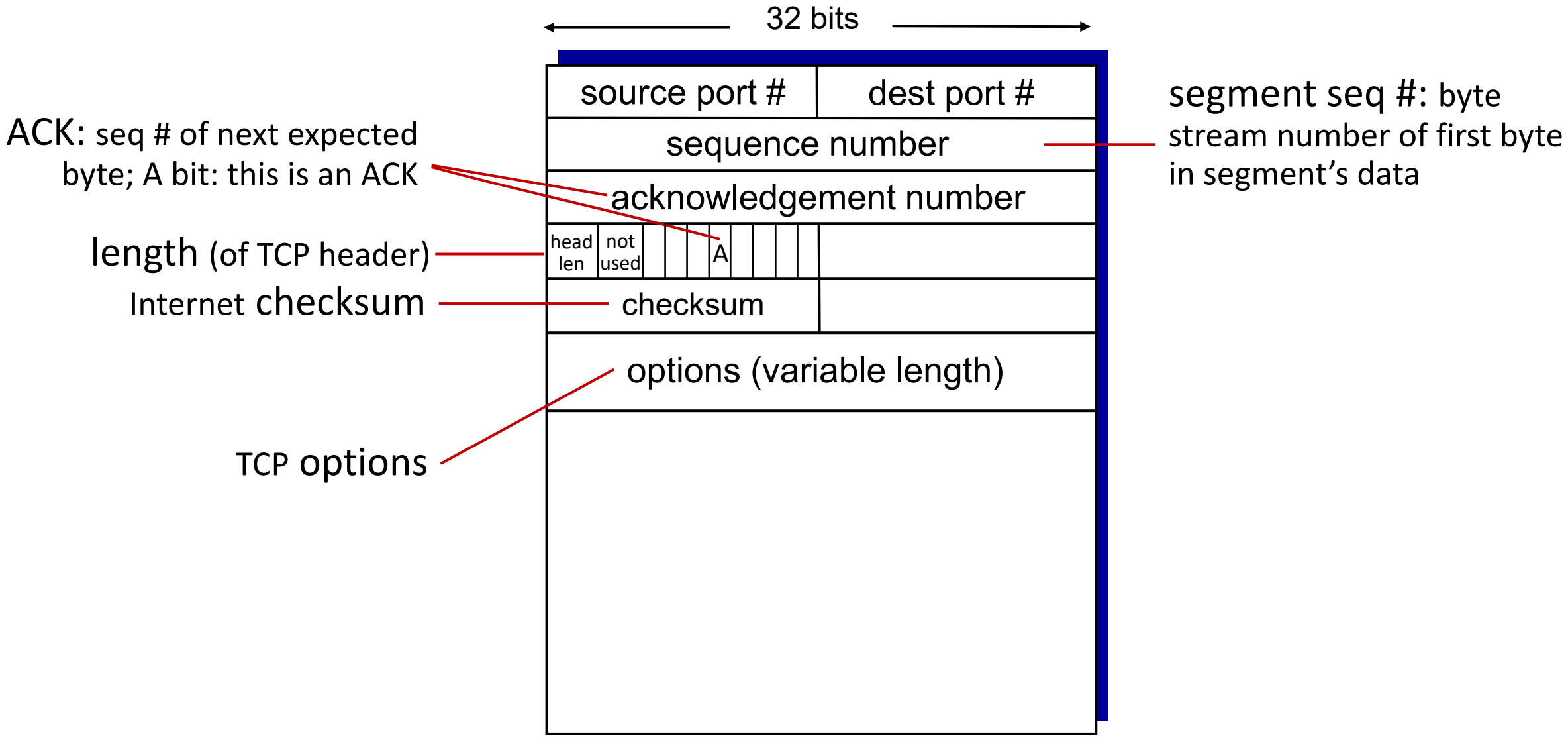
segment seq #: byte stream number of first byte in segment's data

Internet checksum

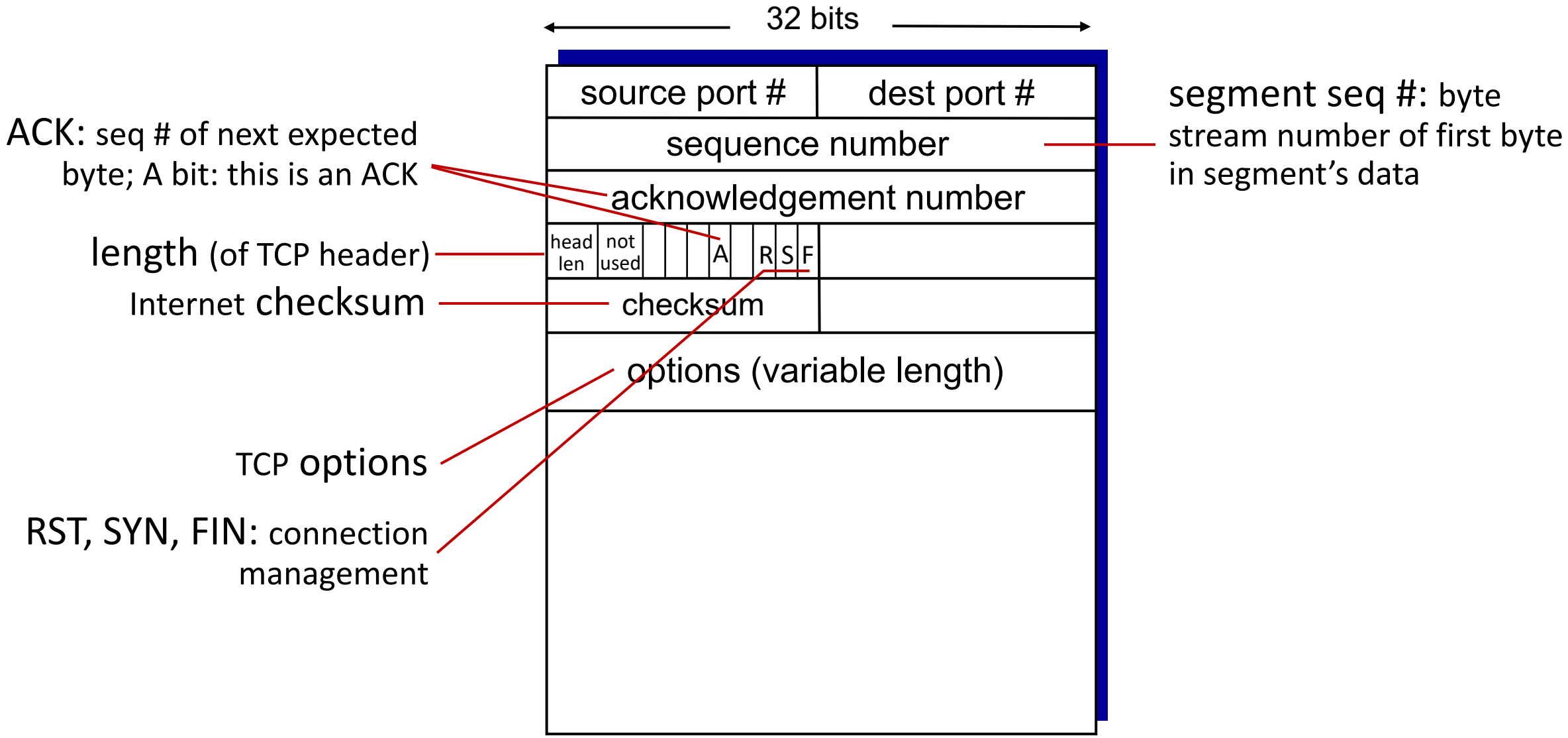
TCP segment structure



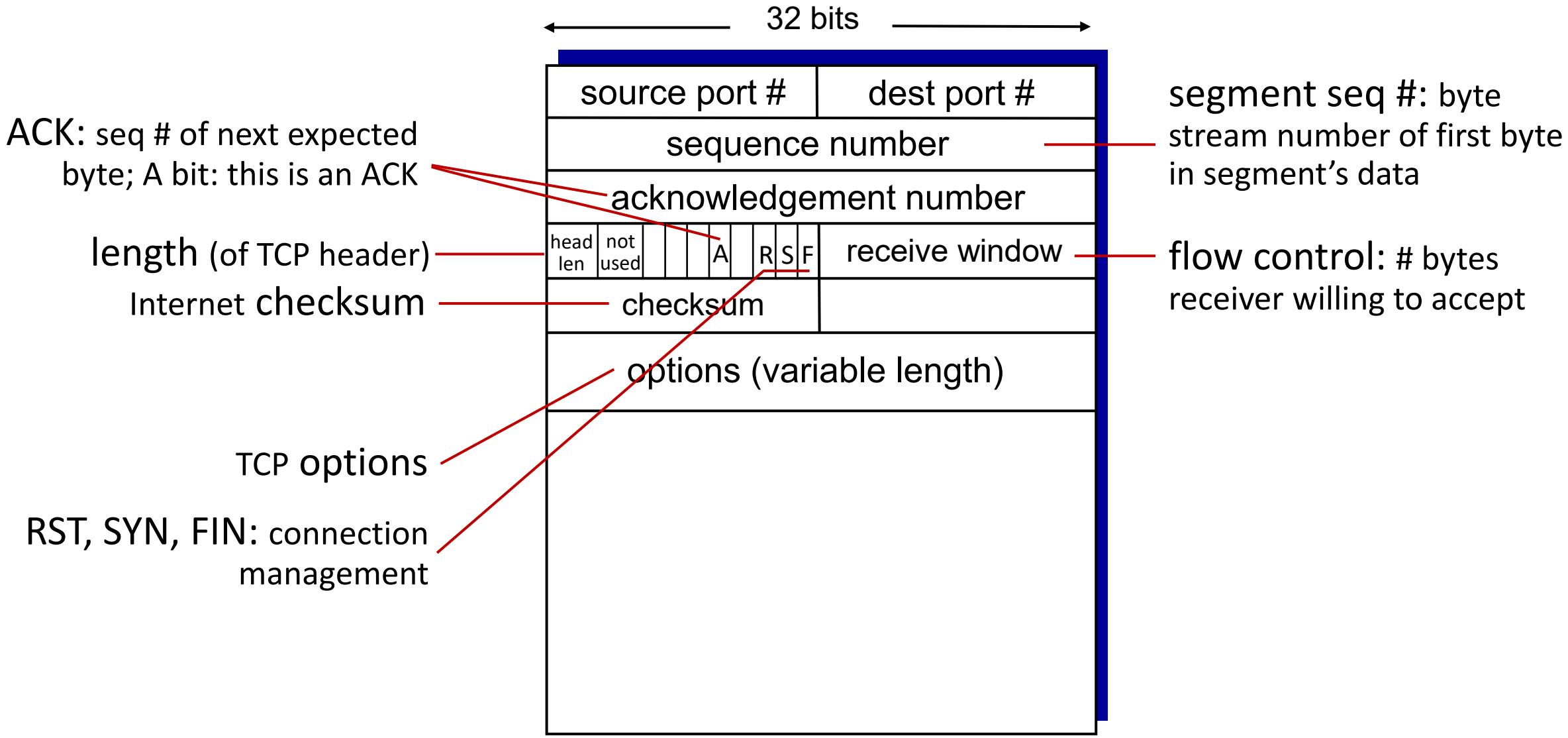
TCP segment structure



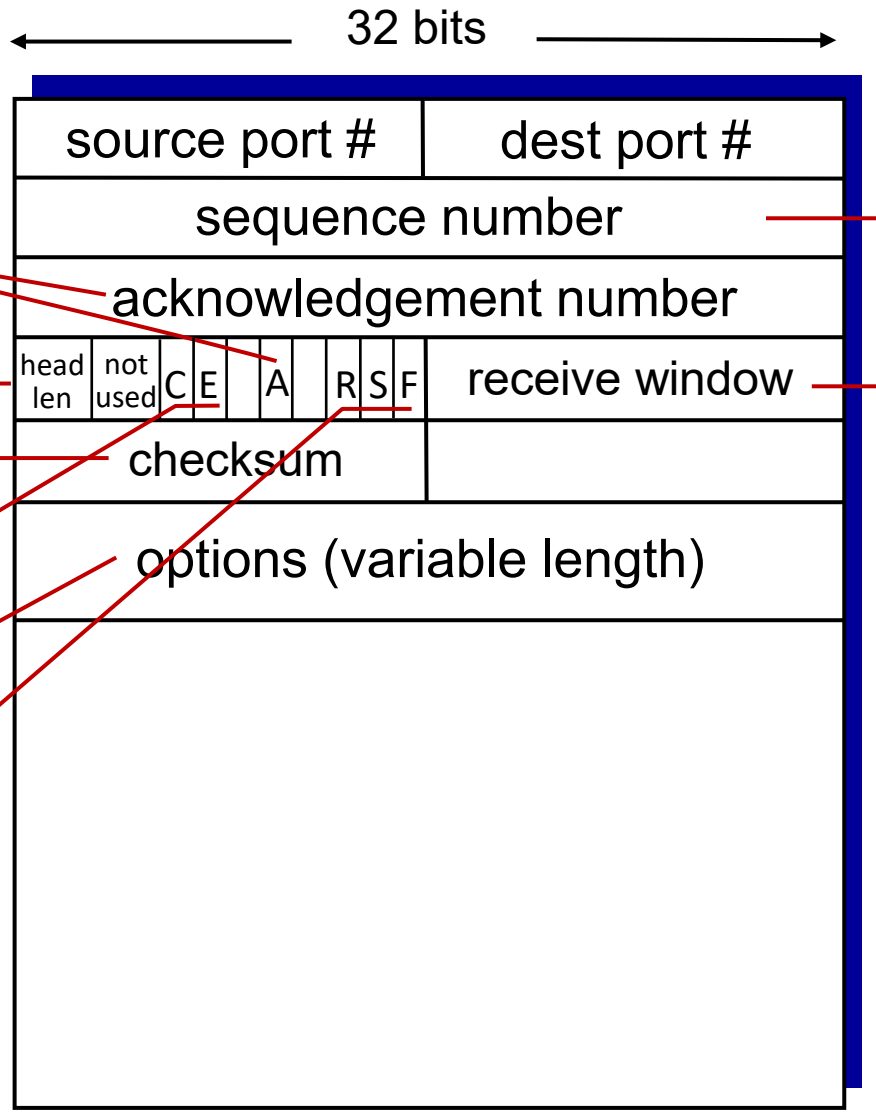
TCP segment structure



TCP segment structure



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

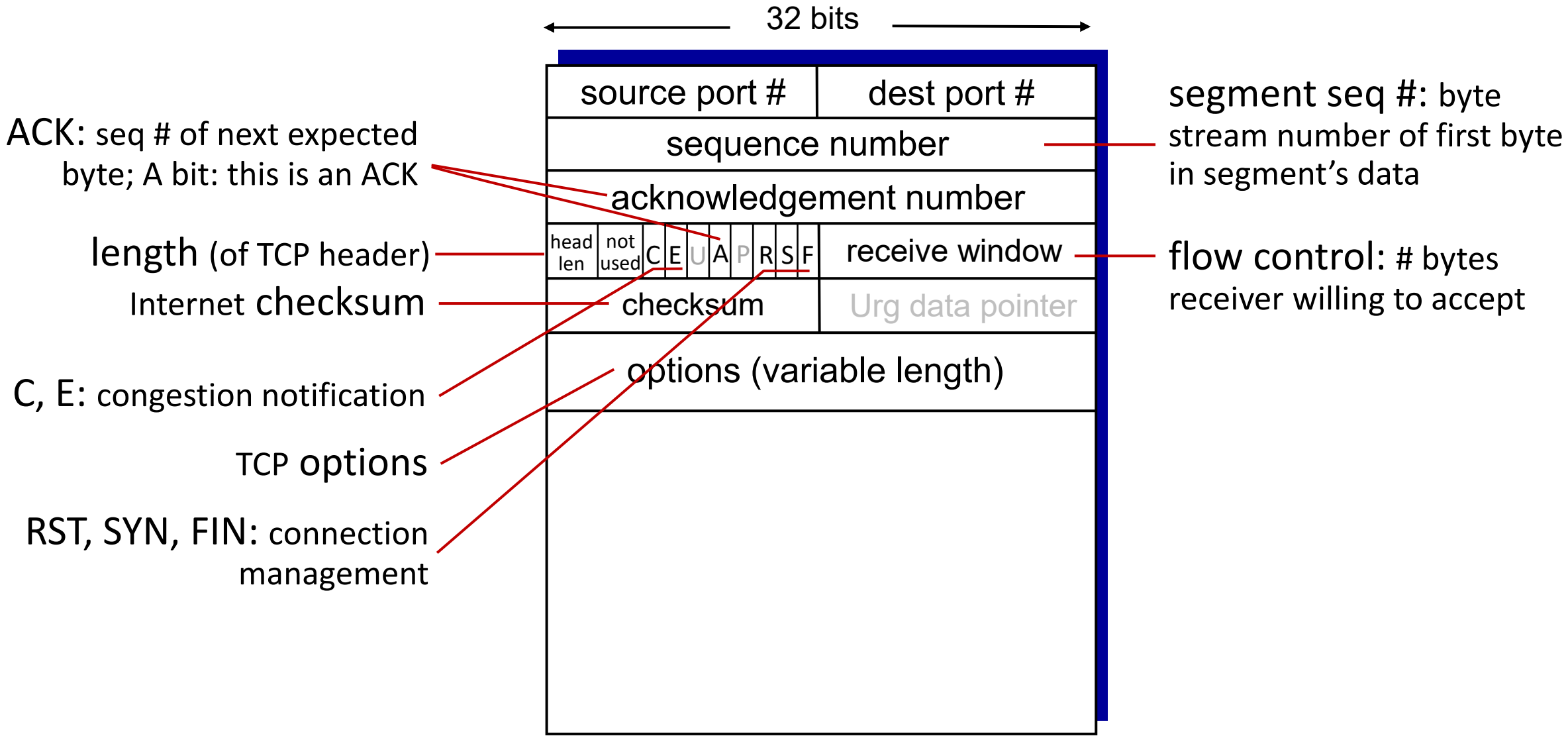
TCP options

RST, SYN, FIN: connection management

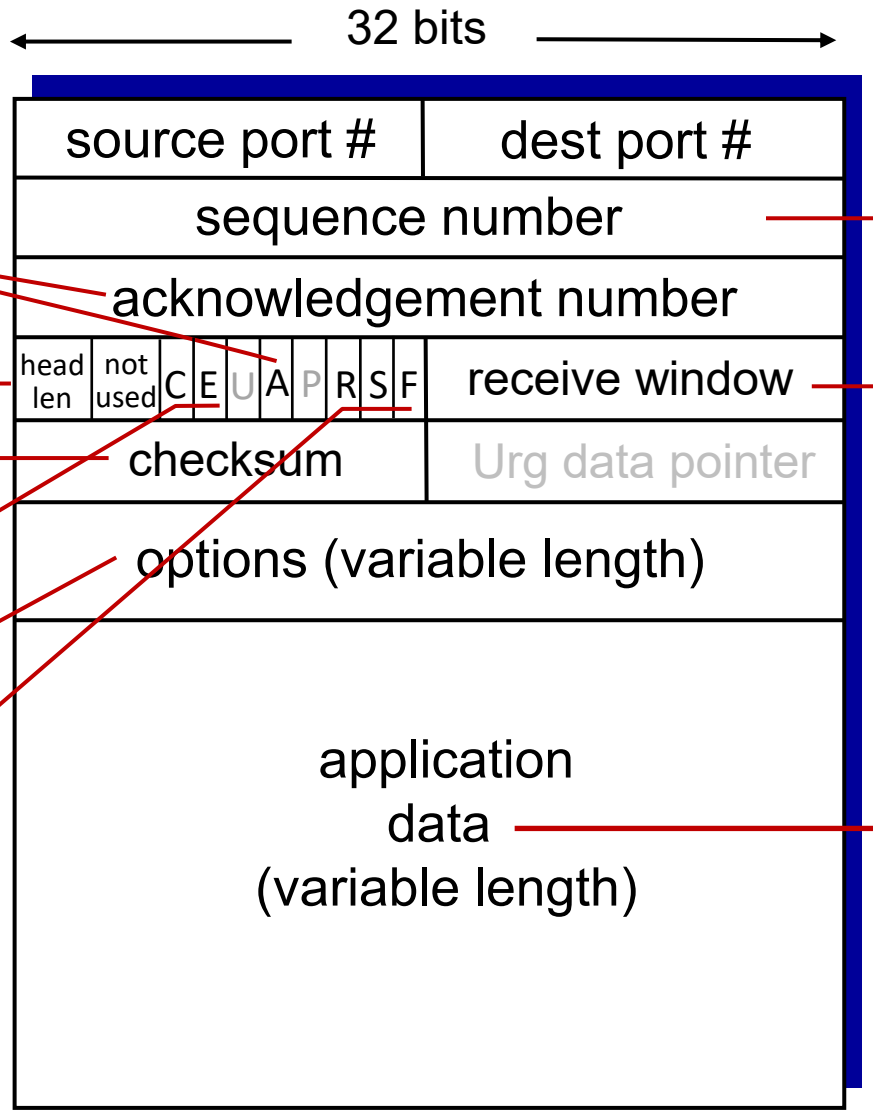
segment seq #: byte stream number of first byte in segment's data

flow control: # bytes receiver willing to accept

TCP segment structure



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 #: byte stream number of first byte in segment's data

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

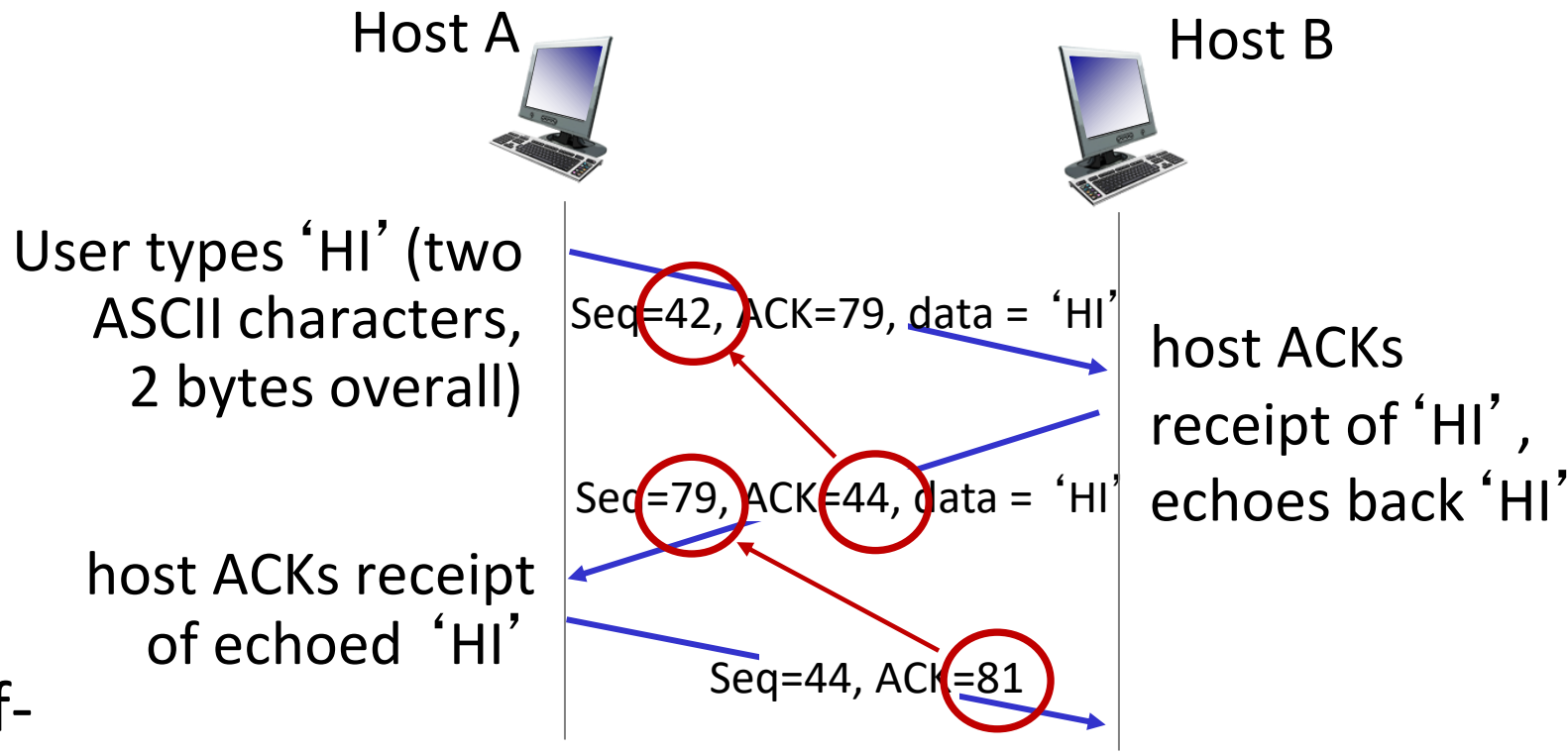
- byte stream "number" of first byte in segment's data

Acknowledgements:

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

Q: how receiver handles out-of-order segments

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



TCP round trip time, timeout

Q: how to set TCP timeout interval?

- longer than RTT, but RTT varies!

- remind that:
  - *too short* (shorter than RTT): premature timeout, unnecessary retransmissions
  - *too long* (much longer than RTT): slow reaction to segment loss

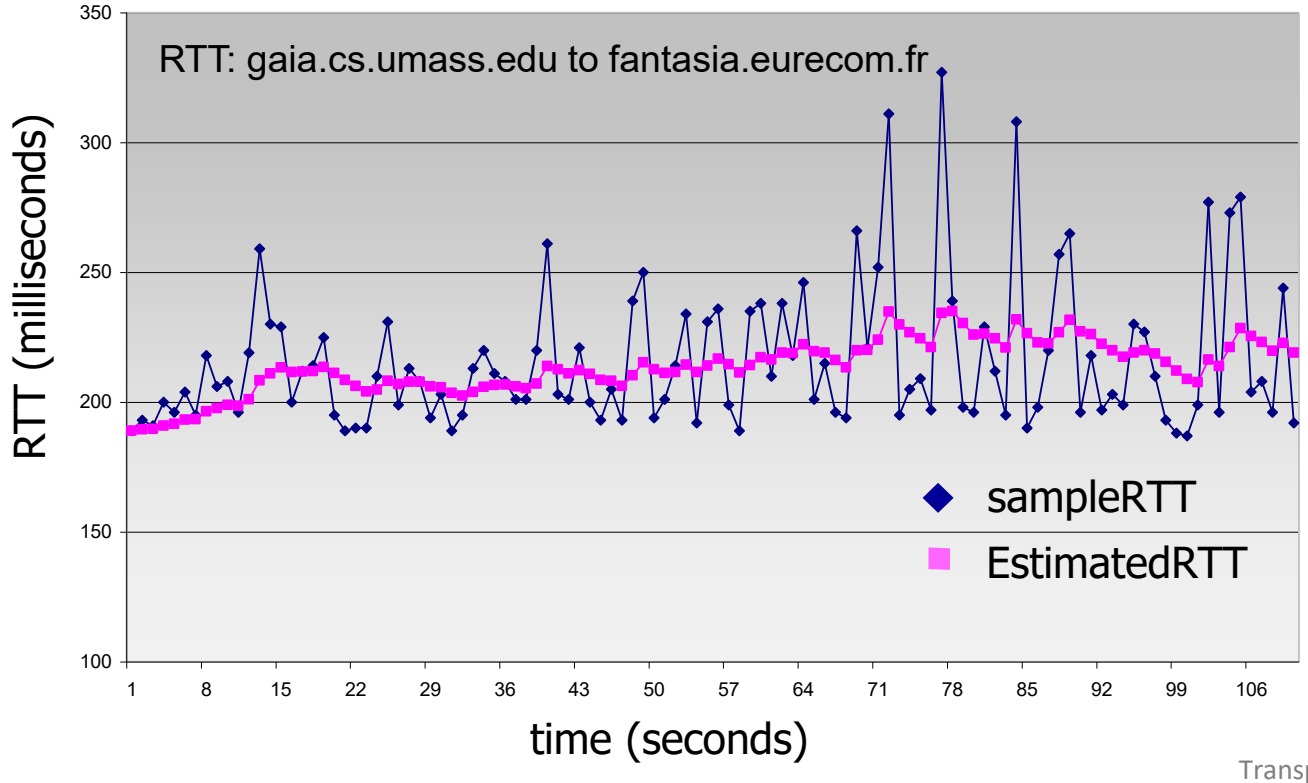
Q: how to estimate RTT?

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

TCP round trip time, timeout

EstimatedRTT[t] = (1-α)*EstimatedRTT[t-1] + α*SampleRTT[t]

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



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation of **`EstimatedRTT`** with respect to **`SampleRTT`**: want a larger safety margin

```
TimeoutInterval[t] = EstimatedRTT[t] + 4*DevRTT[t]
```

estimated RTT

"safety margin"

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

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

(typically, β = 0.25)

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

TCP Sender (simplified)

data received from application

- create segment with seq #
  - seq # is byte-stream number of first data byte in segment
- start timer
  - a single timer is adopted
    - think of timer as for oldest unACKed segment
  - expiration interval: **`TimeoutInterval`**

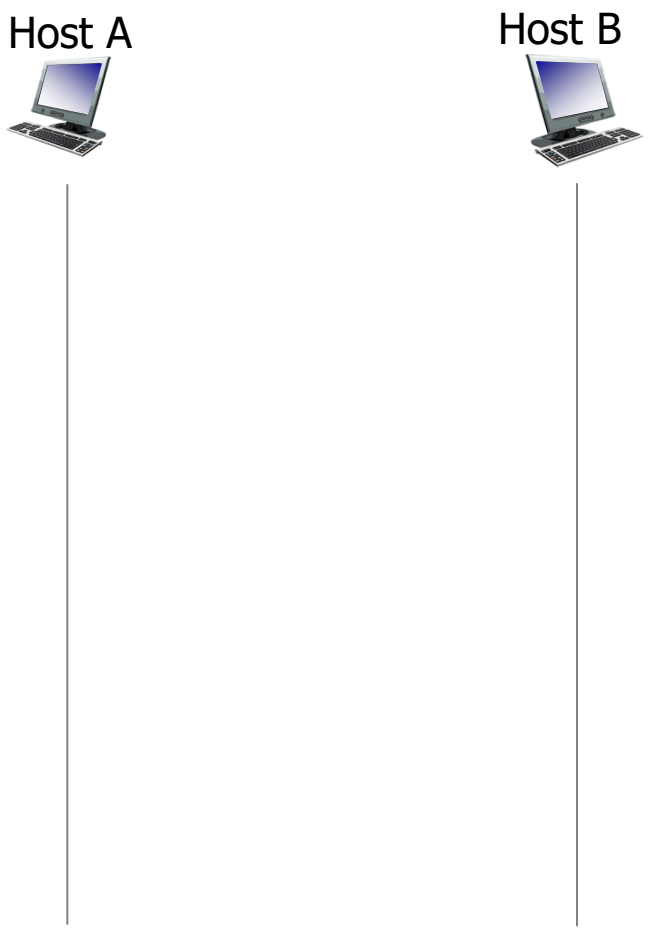
timeout

- retransmit segment that caused timeout
- restart timer

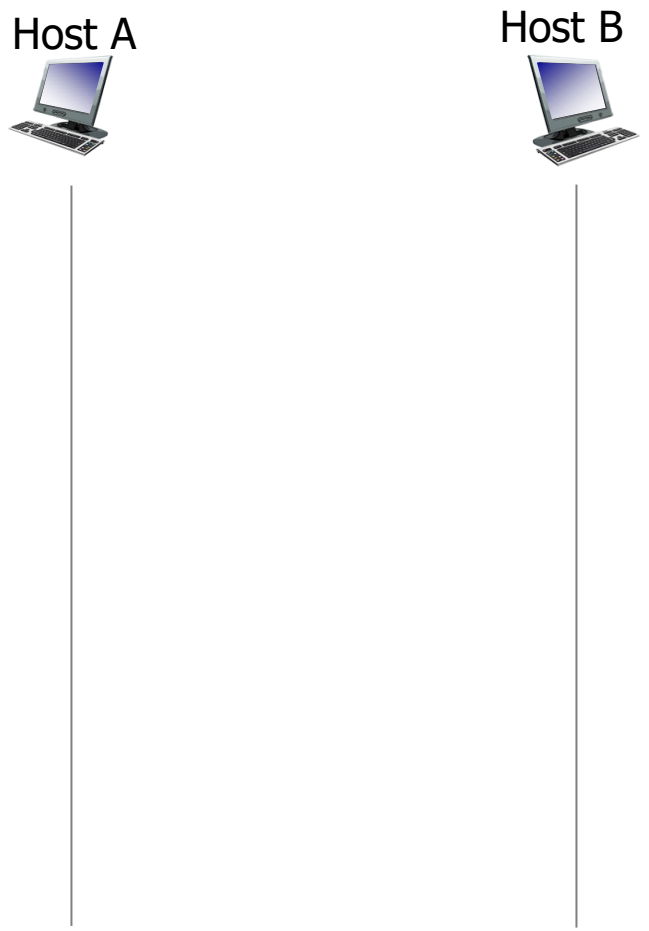
ACK received

- the ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

TCP: retransmission scenarios

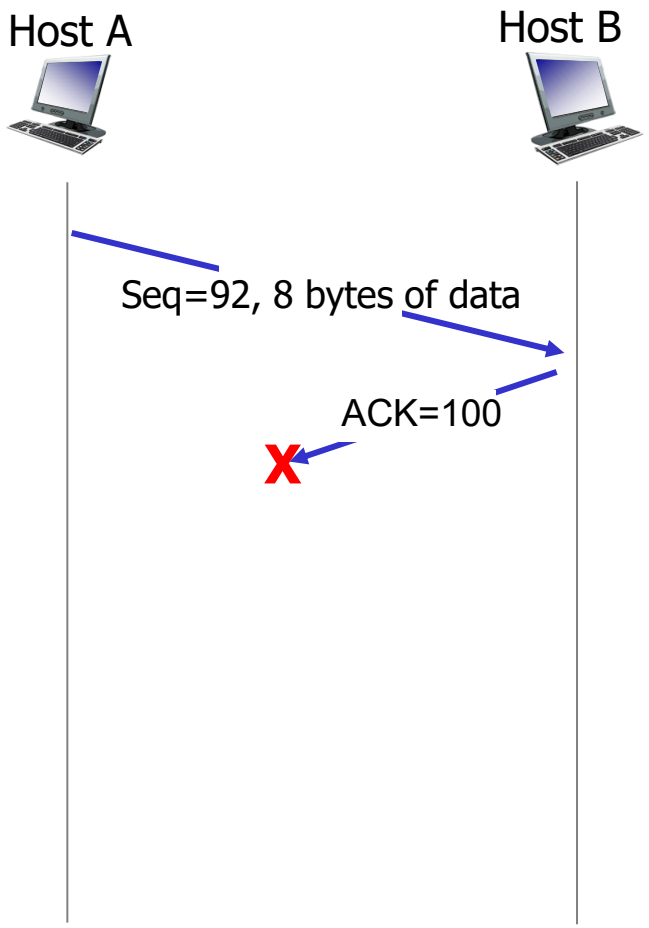


lost ACK scenario

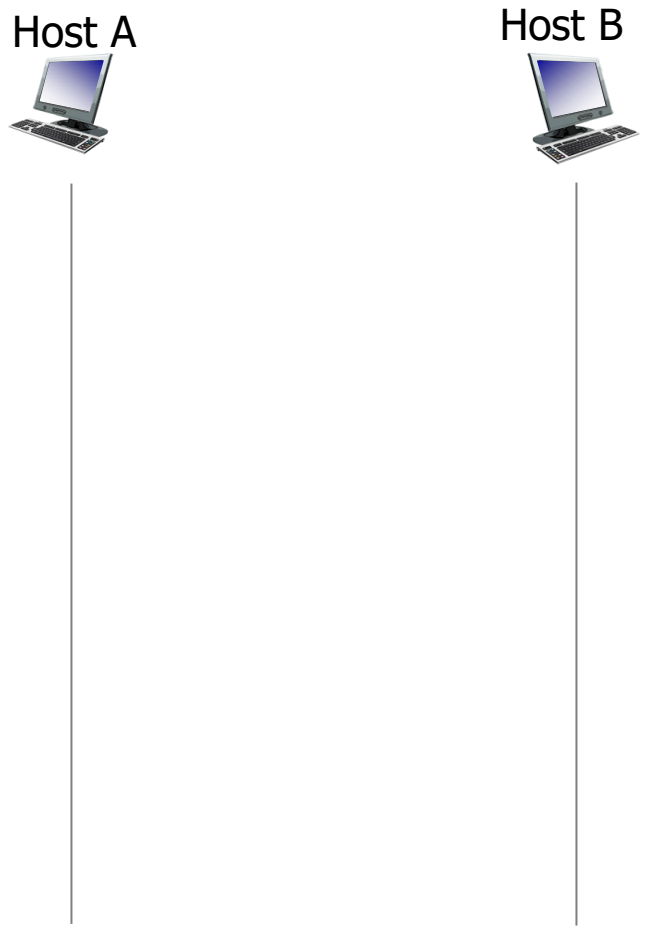


cumulative ACK covers
for earlier lost ACK

TCP: retransmission scenarios

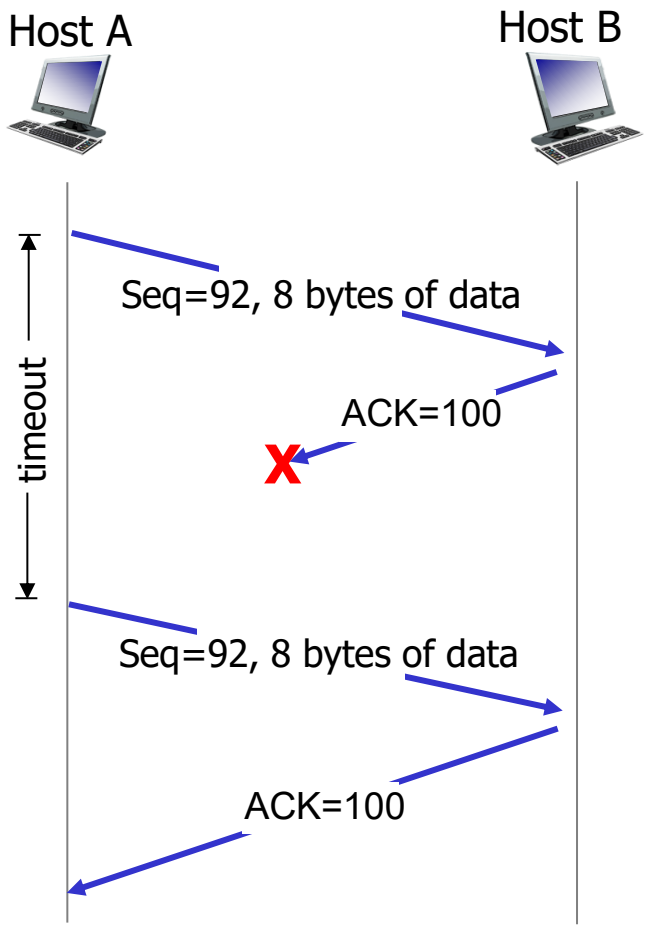


lost ACK scenario

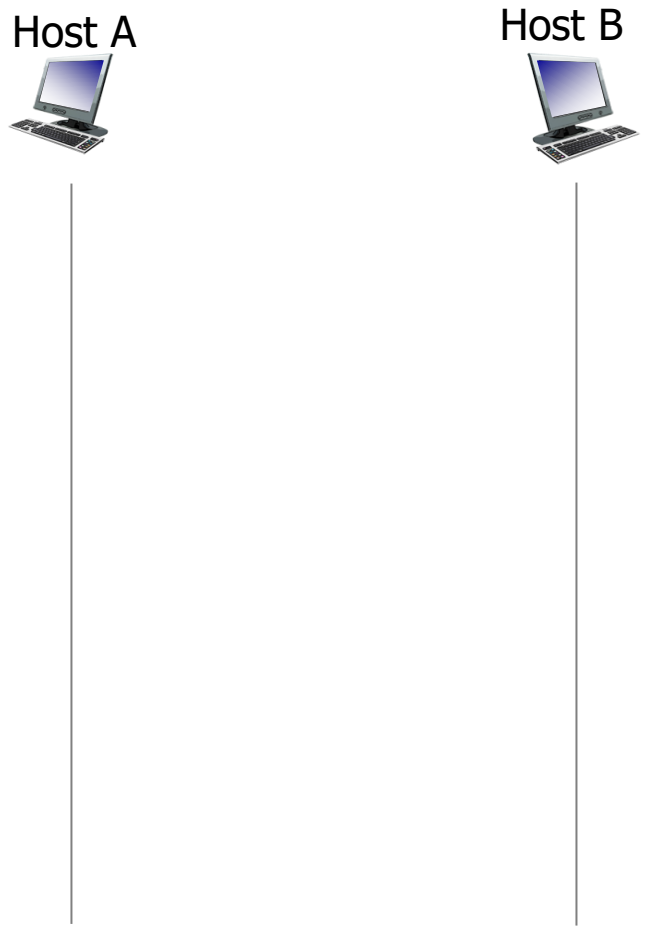


cumulative ACK covers
for earlier lost ACK

TCP: retransmission scenarios

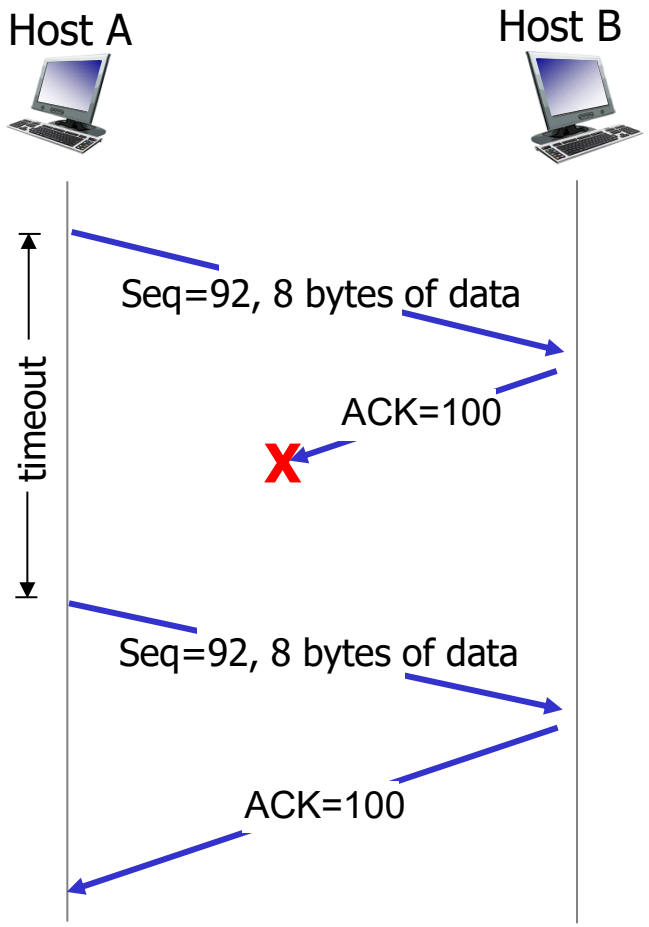


lost ACK scenario

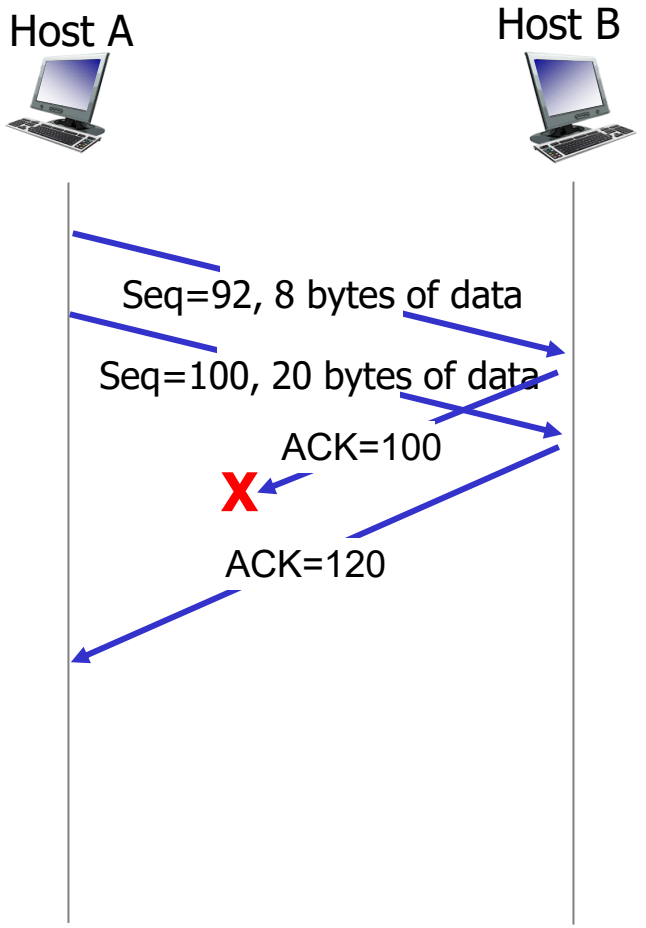


cumulative ACK covers
for earlier lost ACK

TCP: retransmission scenarios

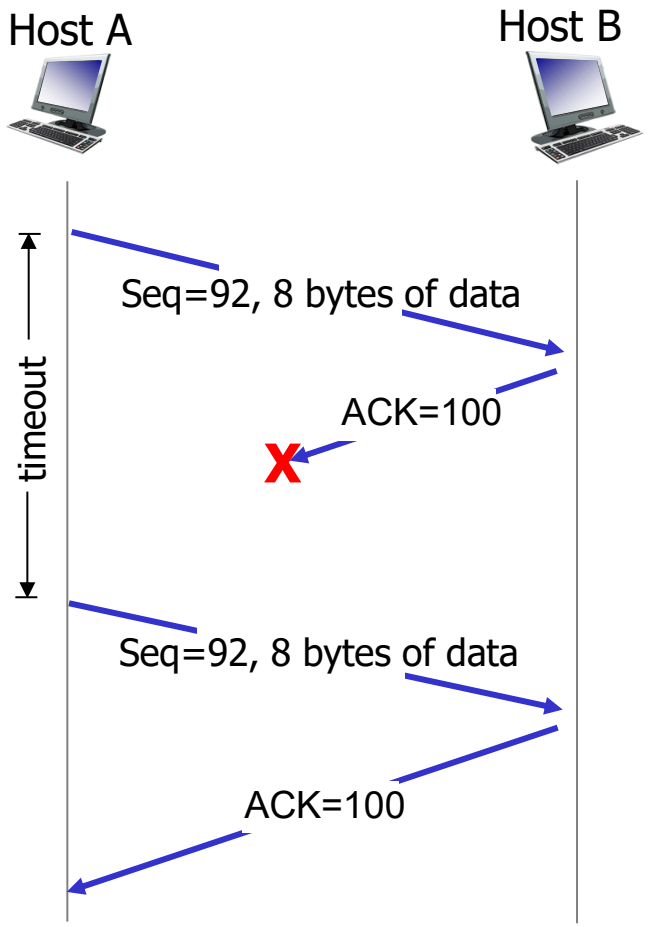


lost ACK scenario

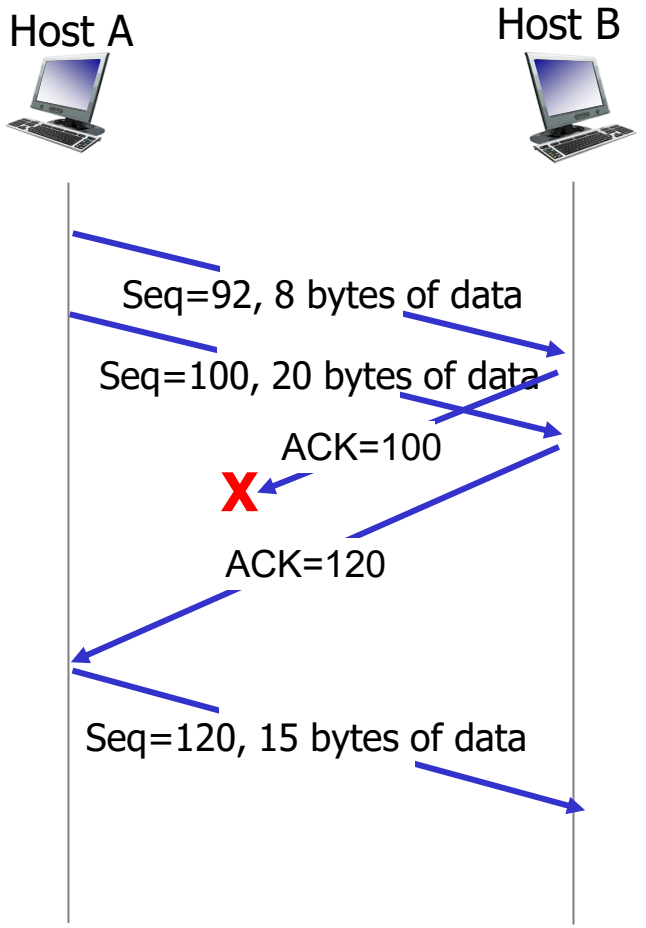


cumulative ACK covers
for earlier lost ACK

TCP: retransmission scenarios

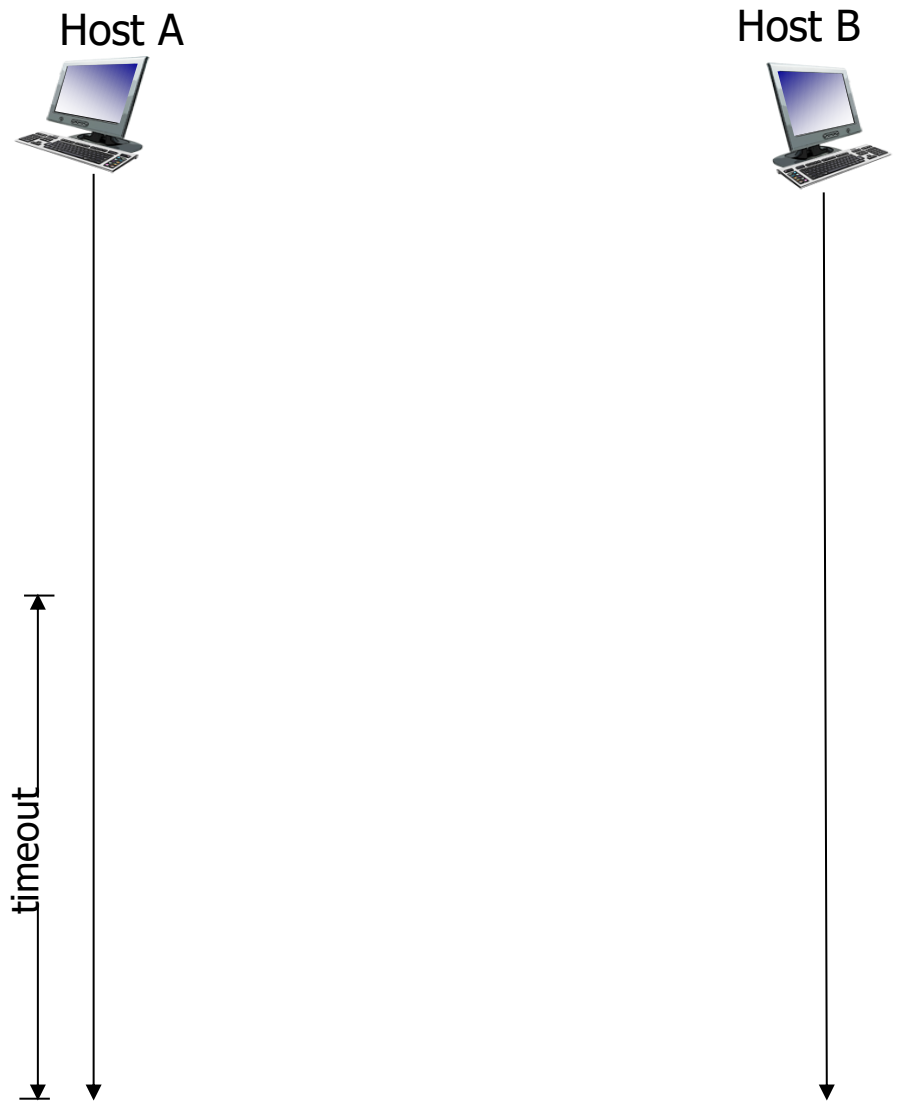


lost ACK scenario

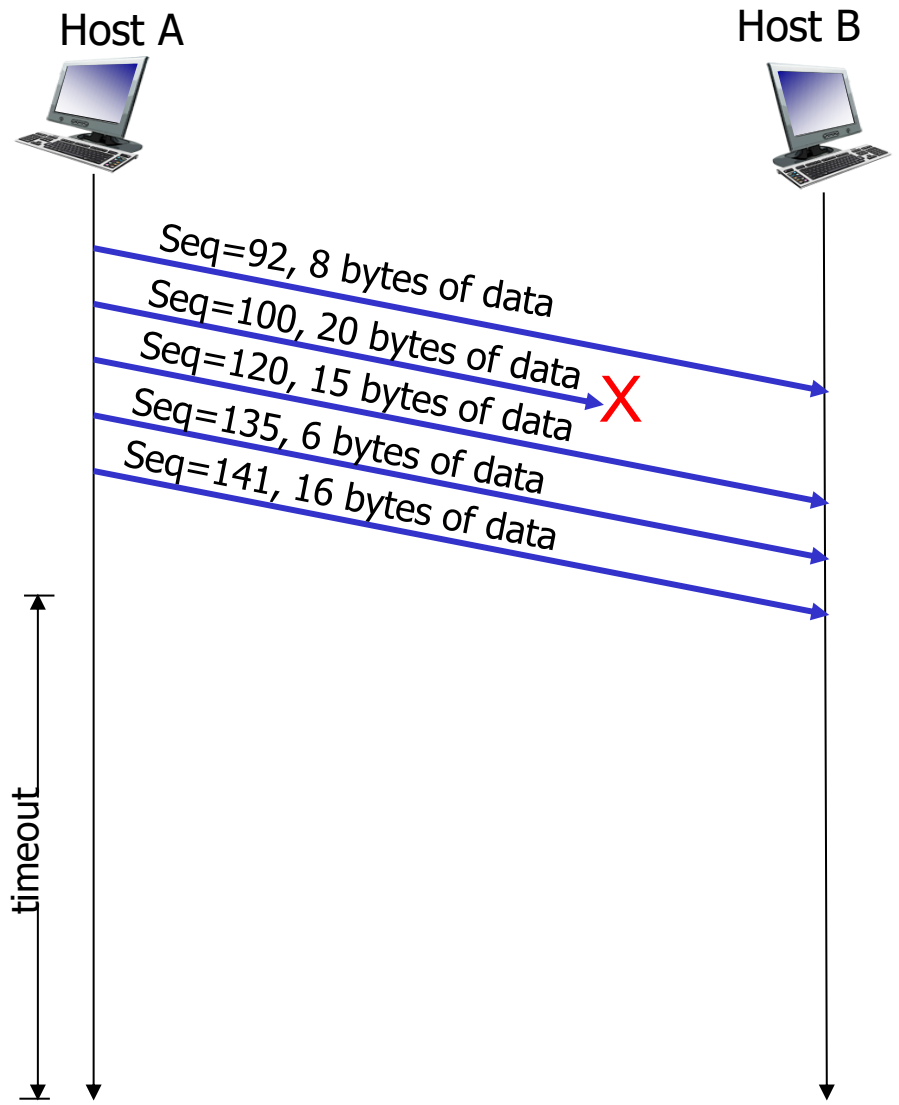


cumulative ACK covers
for earlier lost ACK

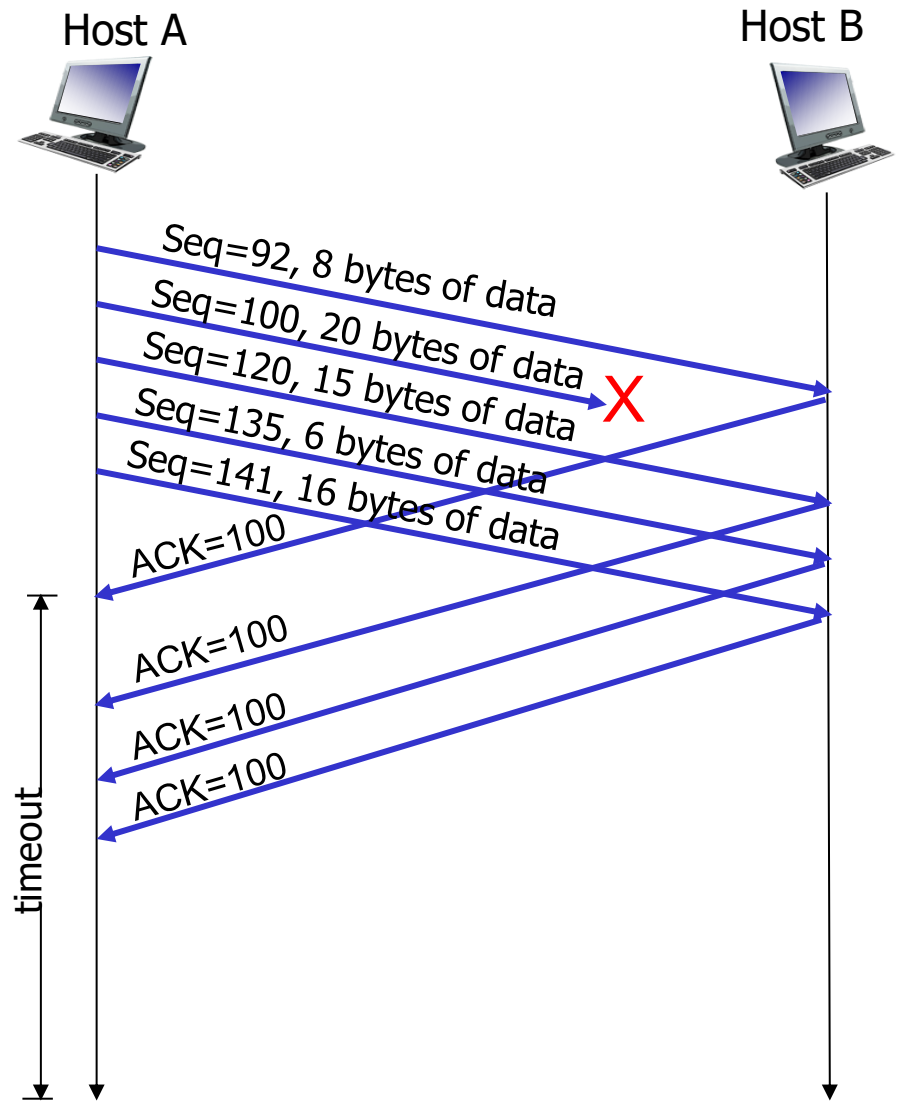
TCP fast retransmit



TCP fast retransmit

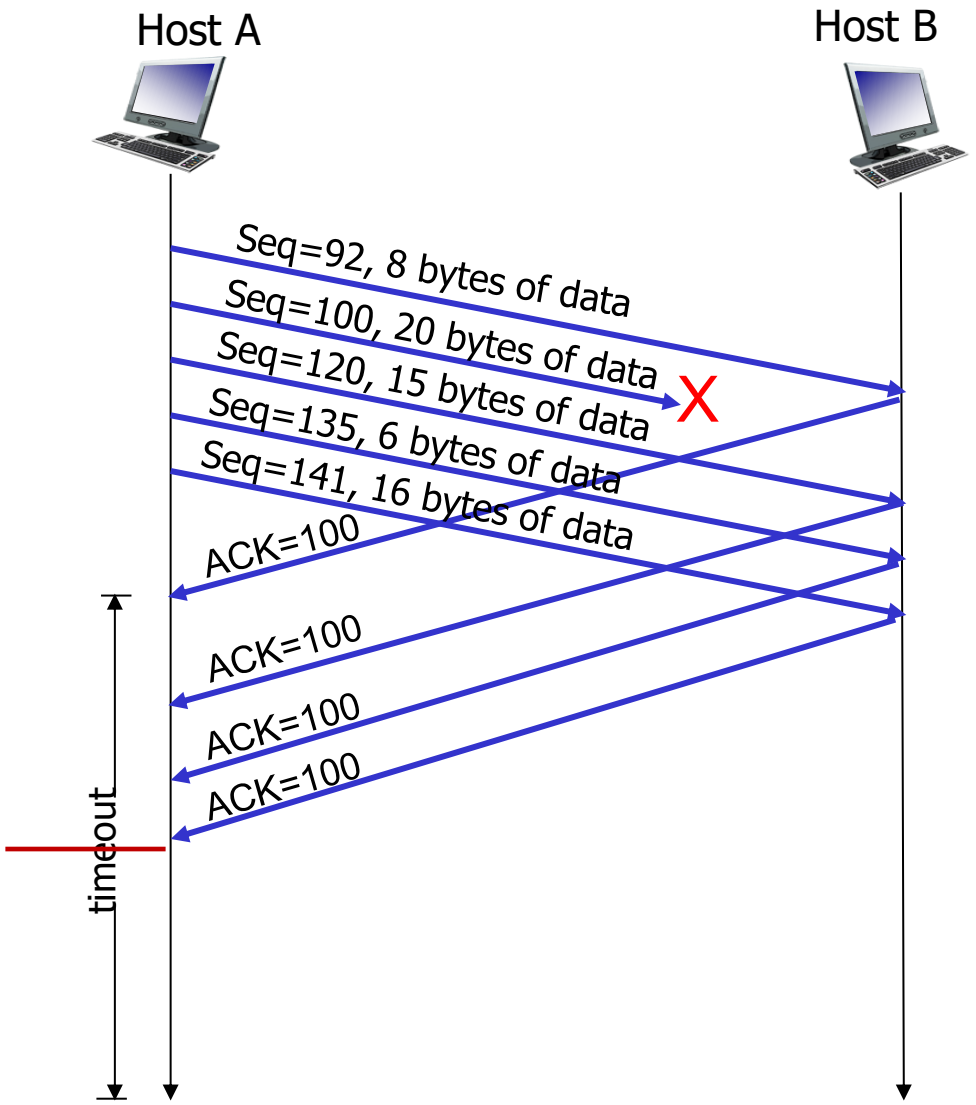


TCP fast retransmit



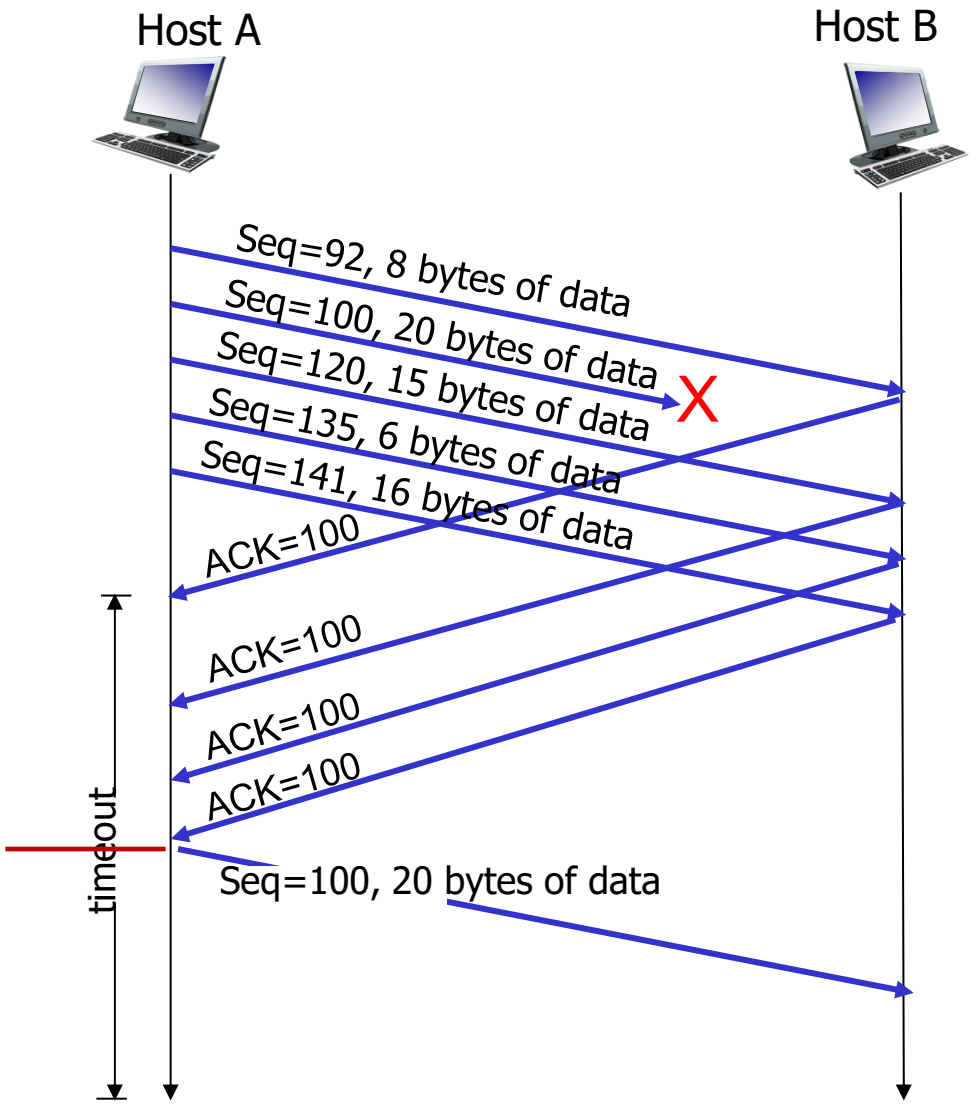
TCP fast retransmit

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



TCP fast retransmit

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



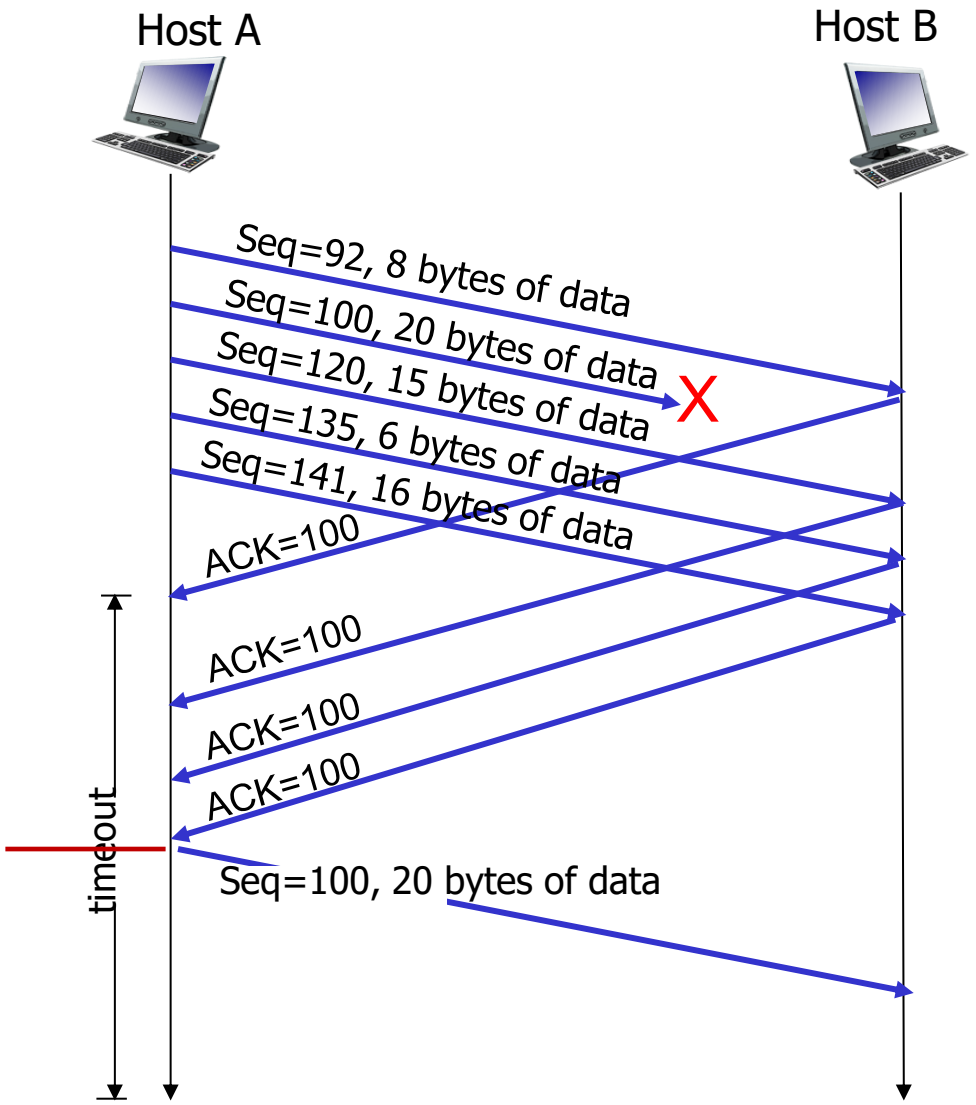
TCP fast retransmit

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

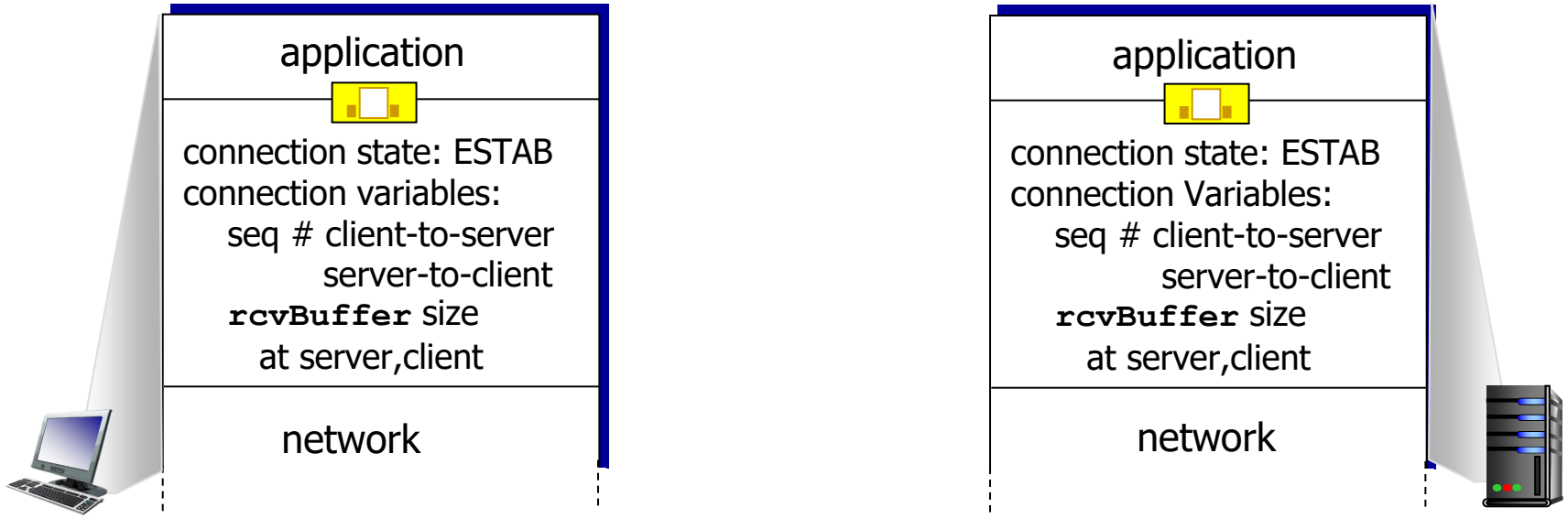
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



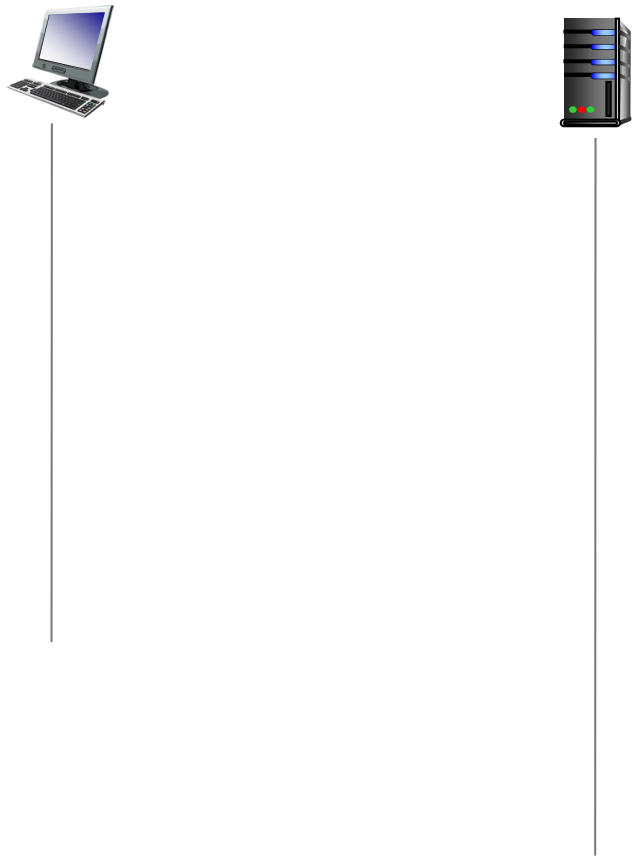
TCP connection management

before exchanging data, sender/receiver “handshake”:

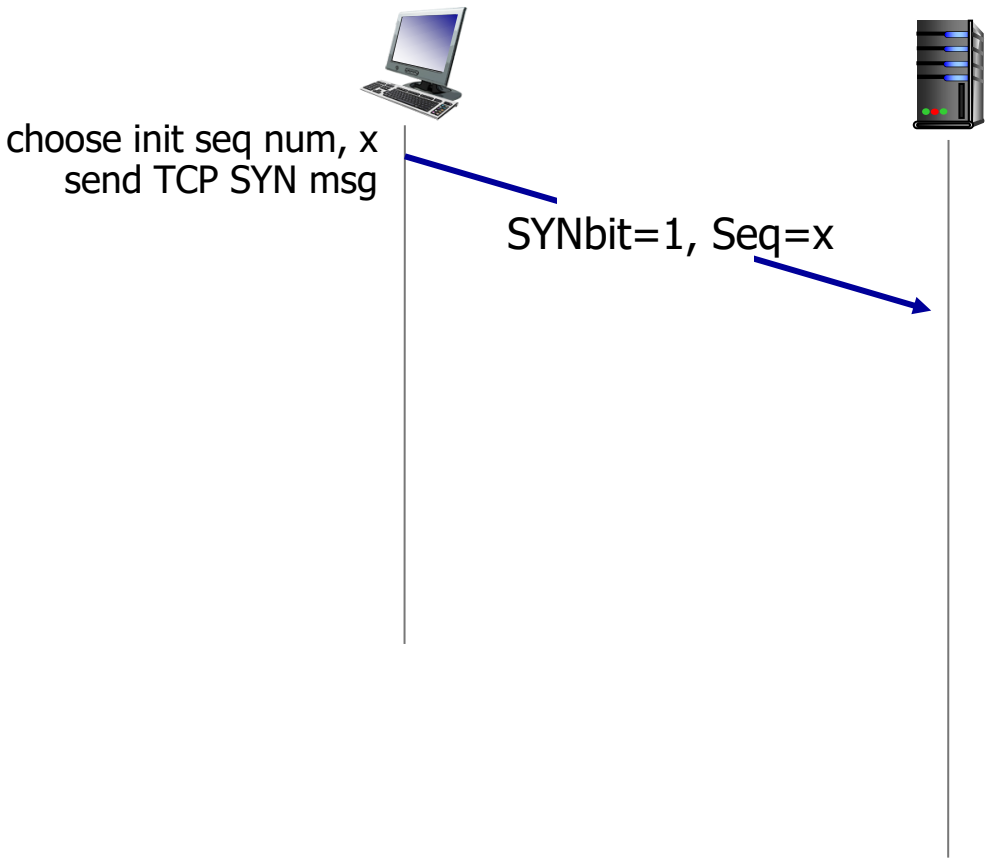
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)



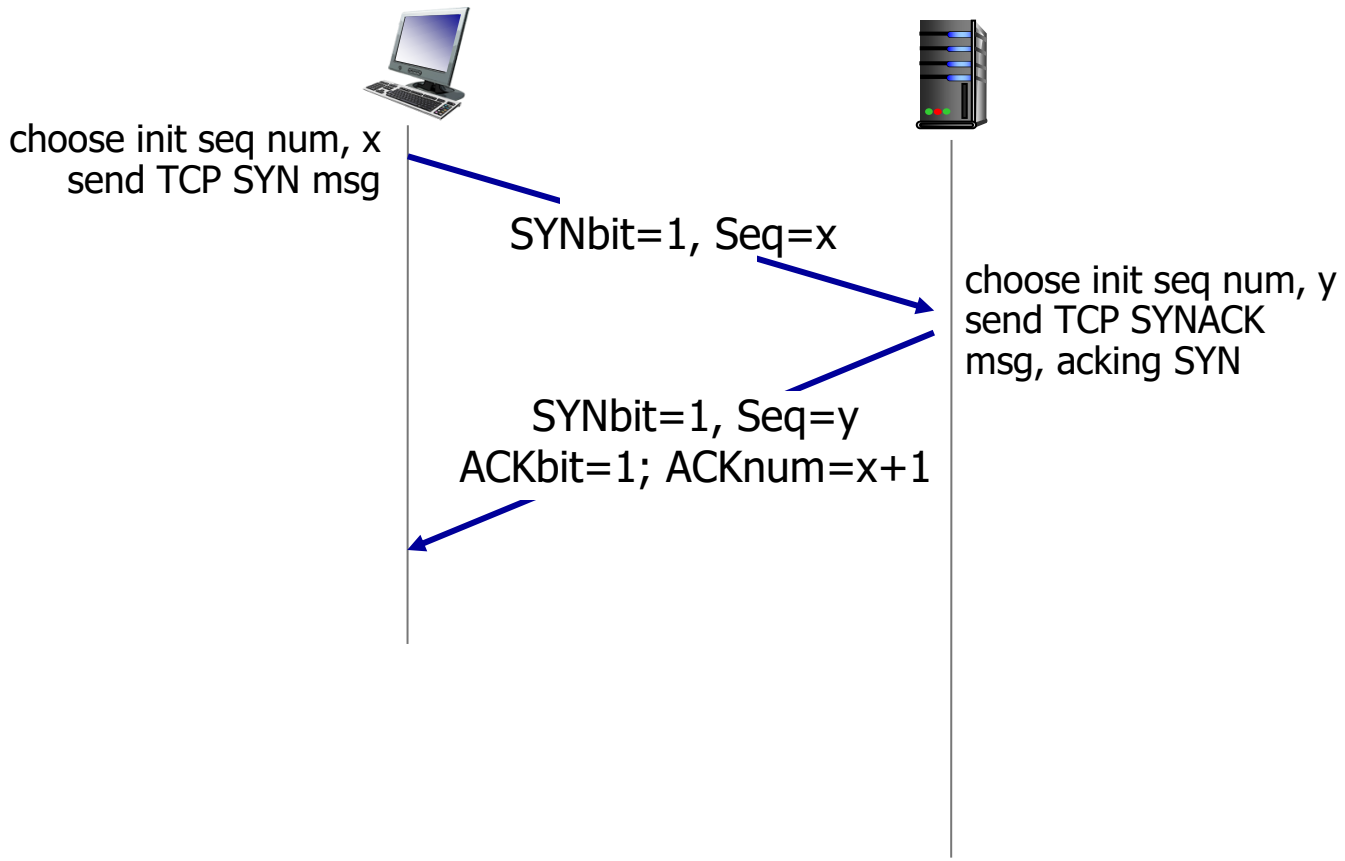
TCP 3-way handshake



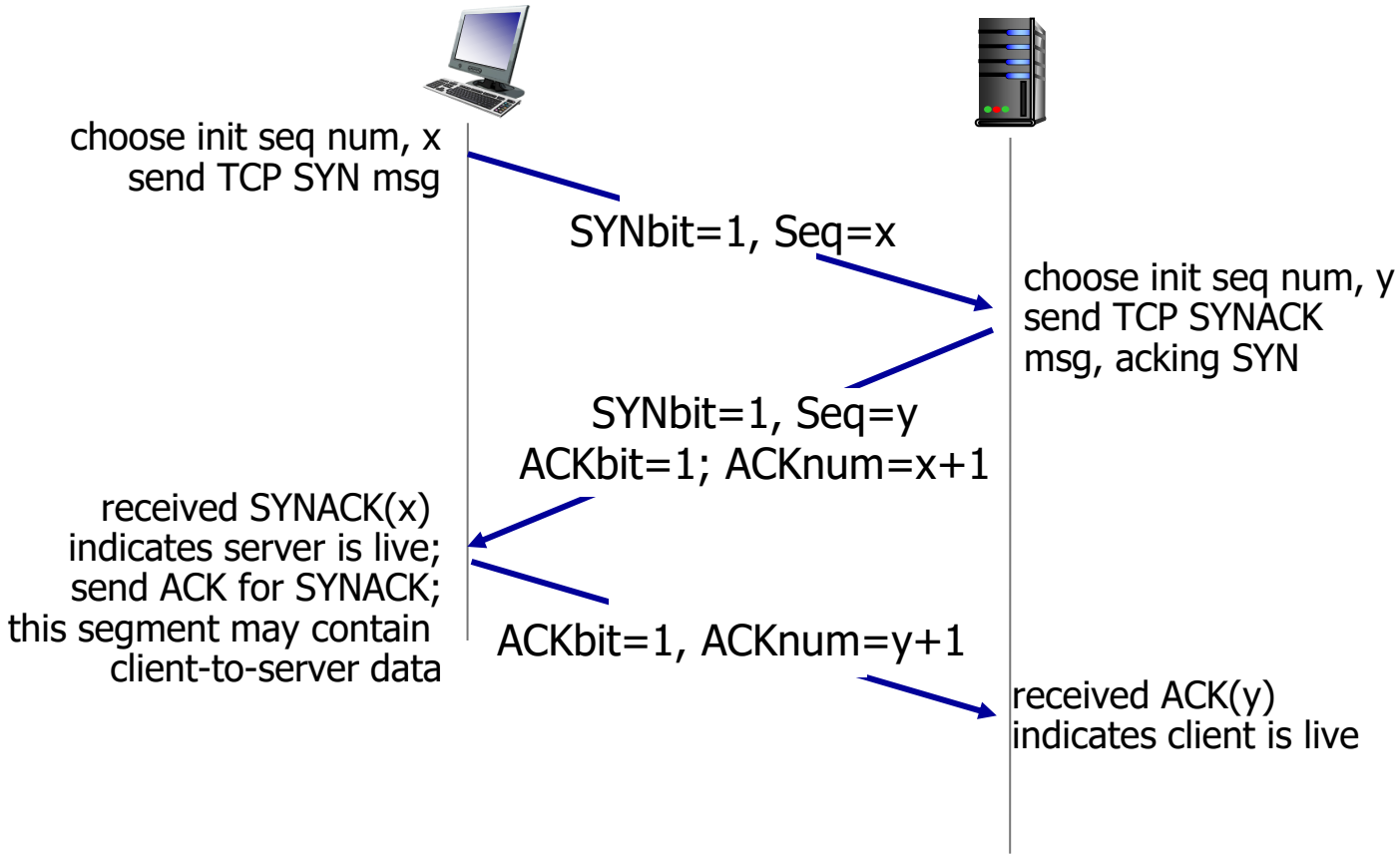
TCP 3-way handshake



TCP 3-way handshake

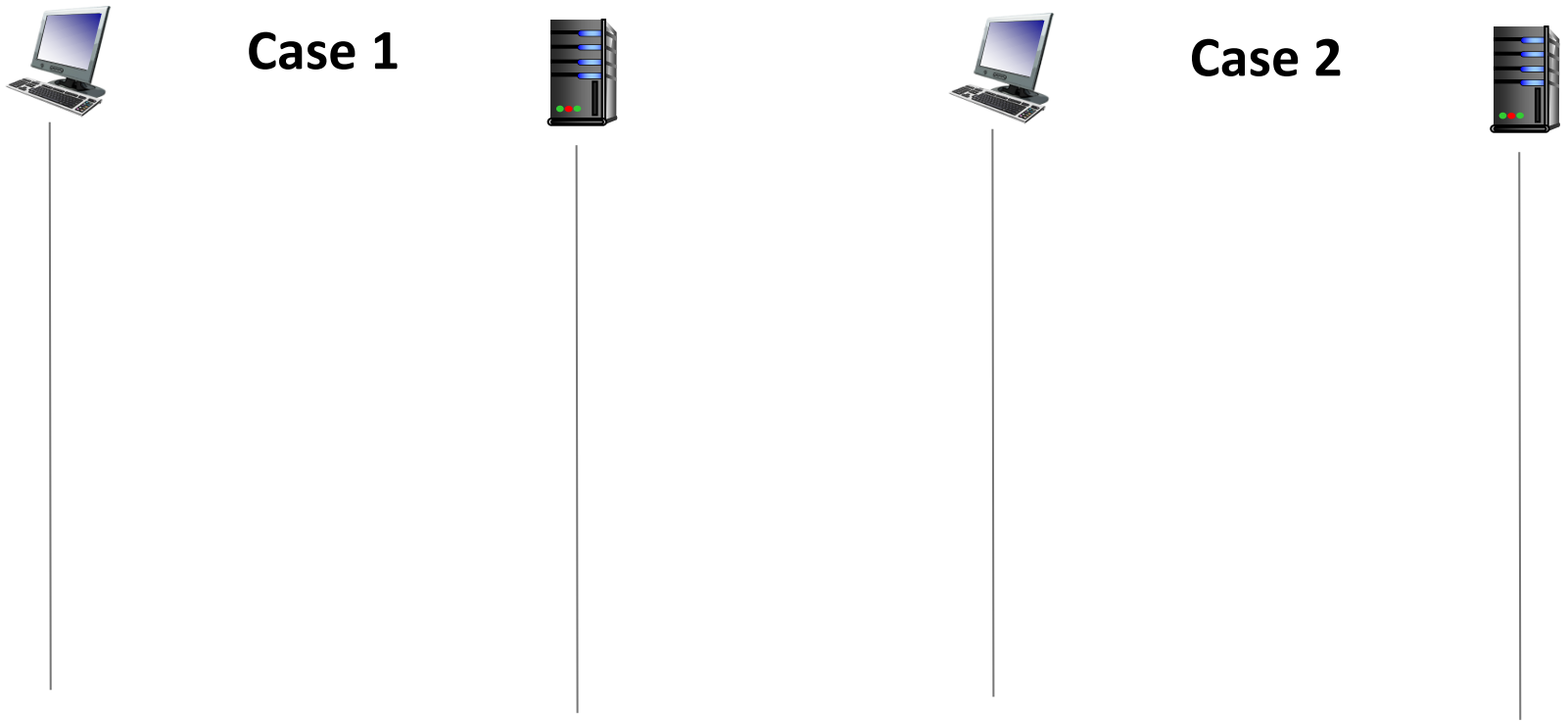


TCP 3-way handshake



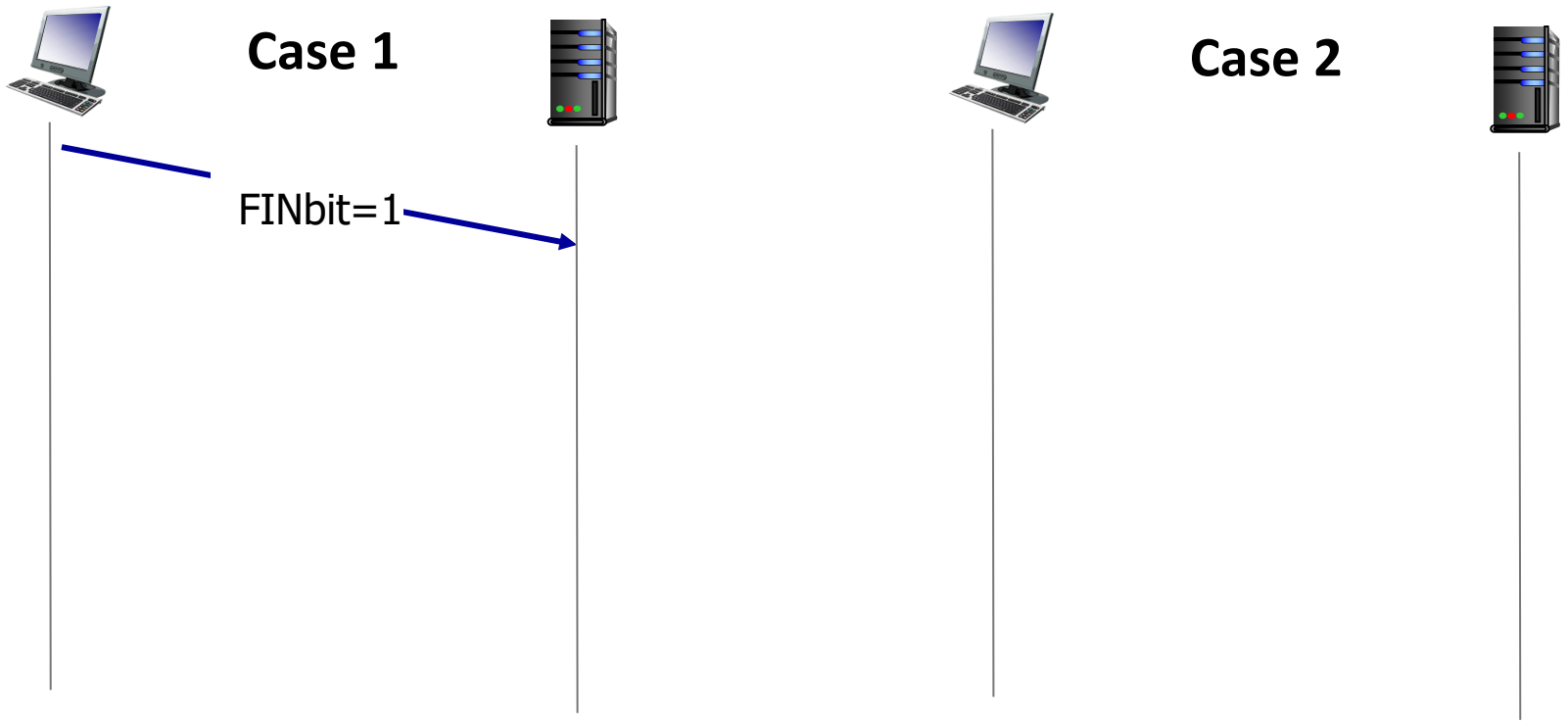
Closing a TCP connection

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



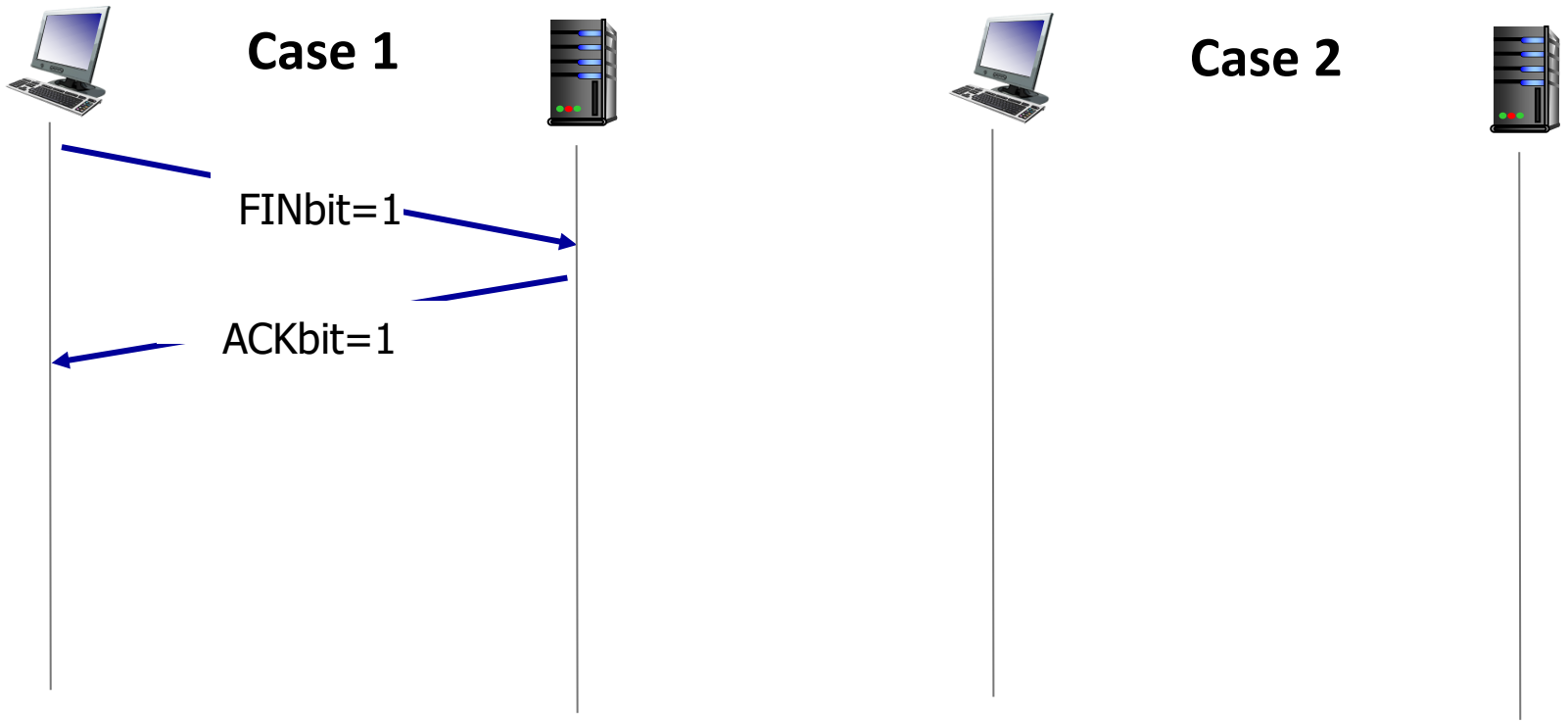
Closing a TCP connection

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



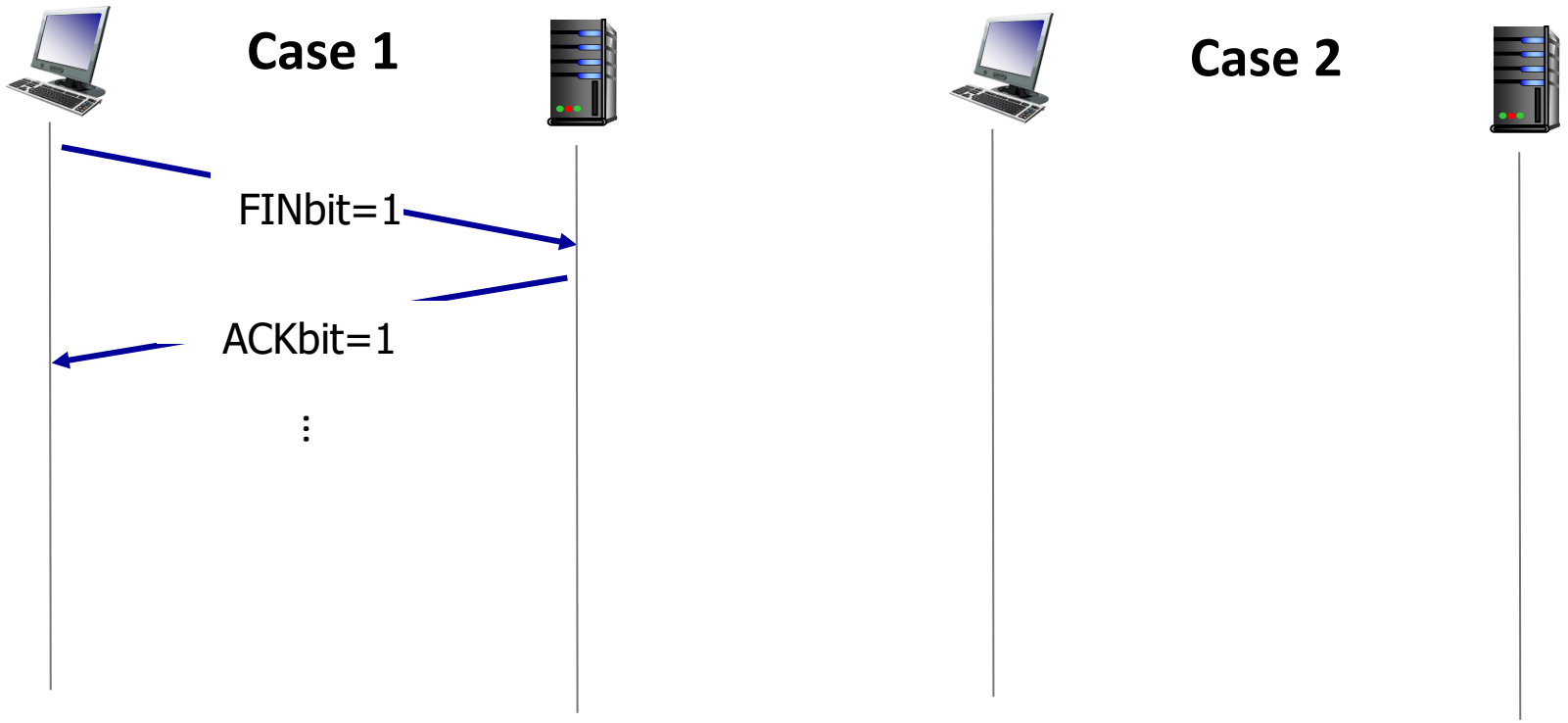
Closing a TCP connection

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



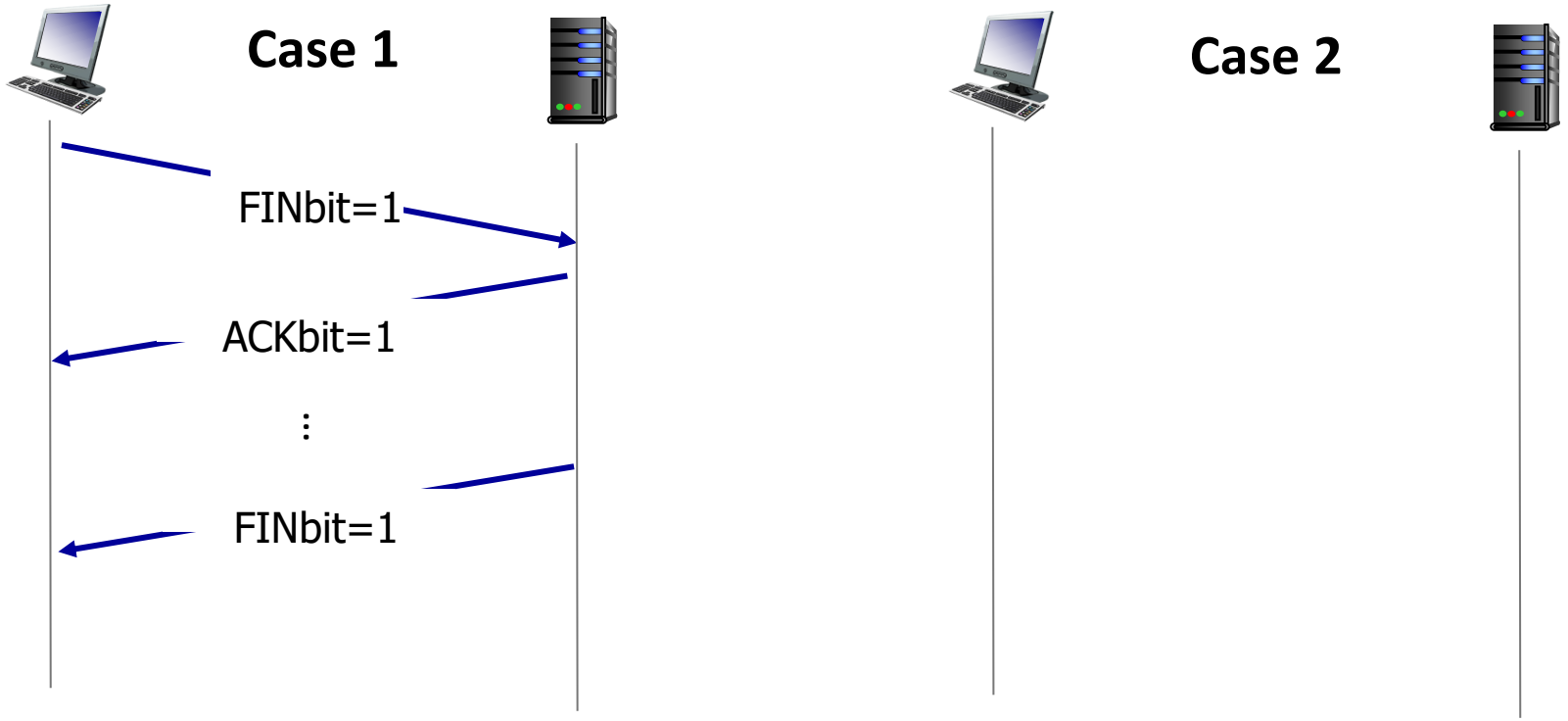
Closing a TCP connection

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



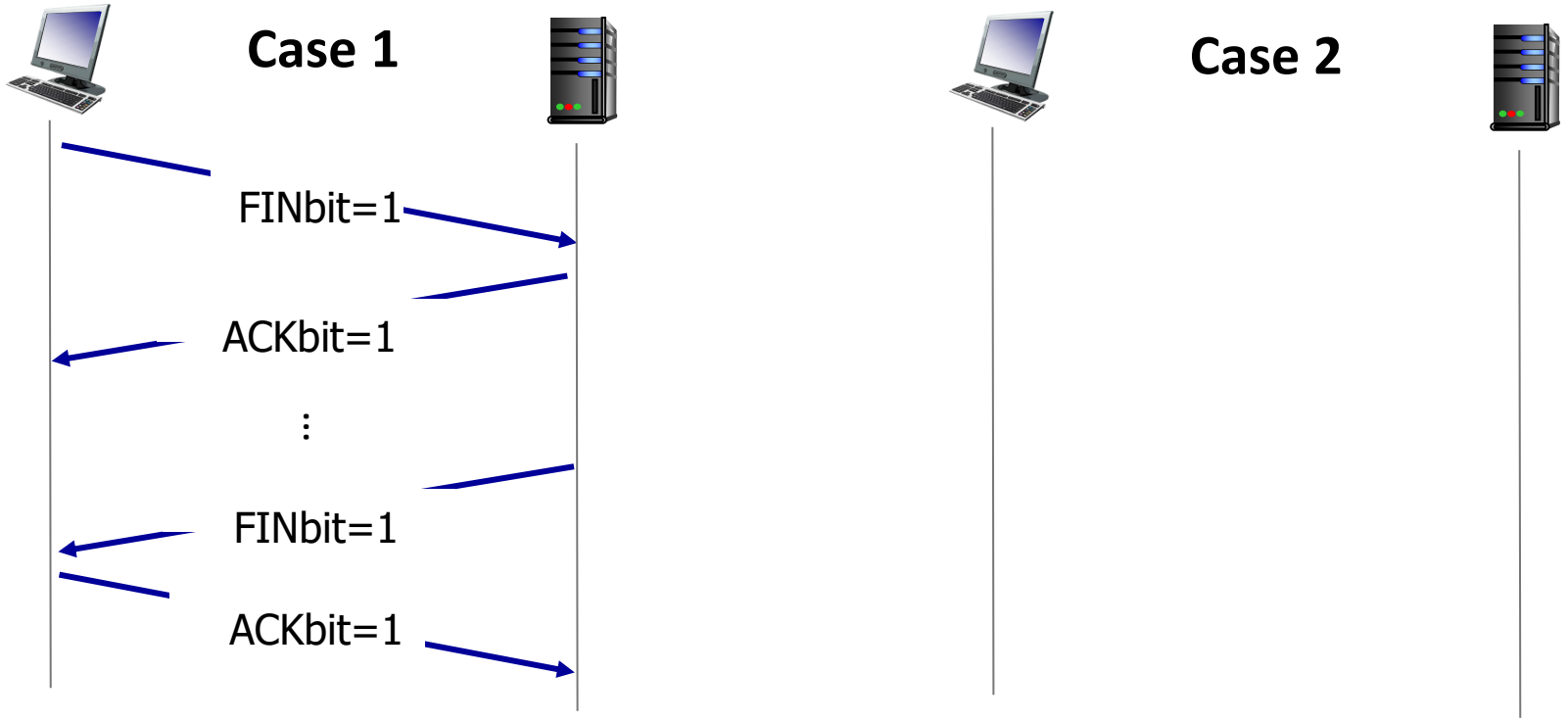
Closing a TCP connection

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



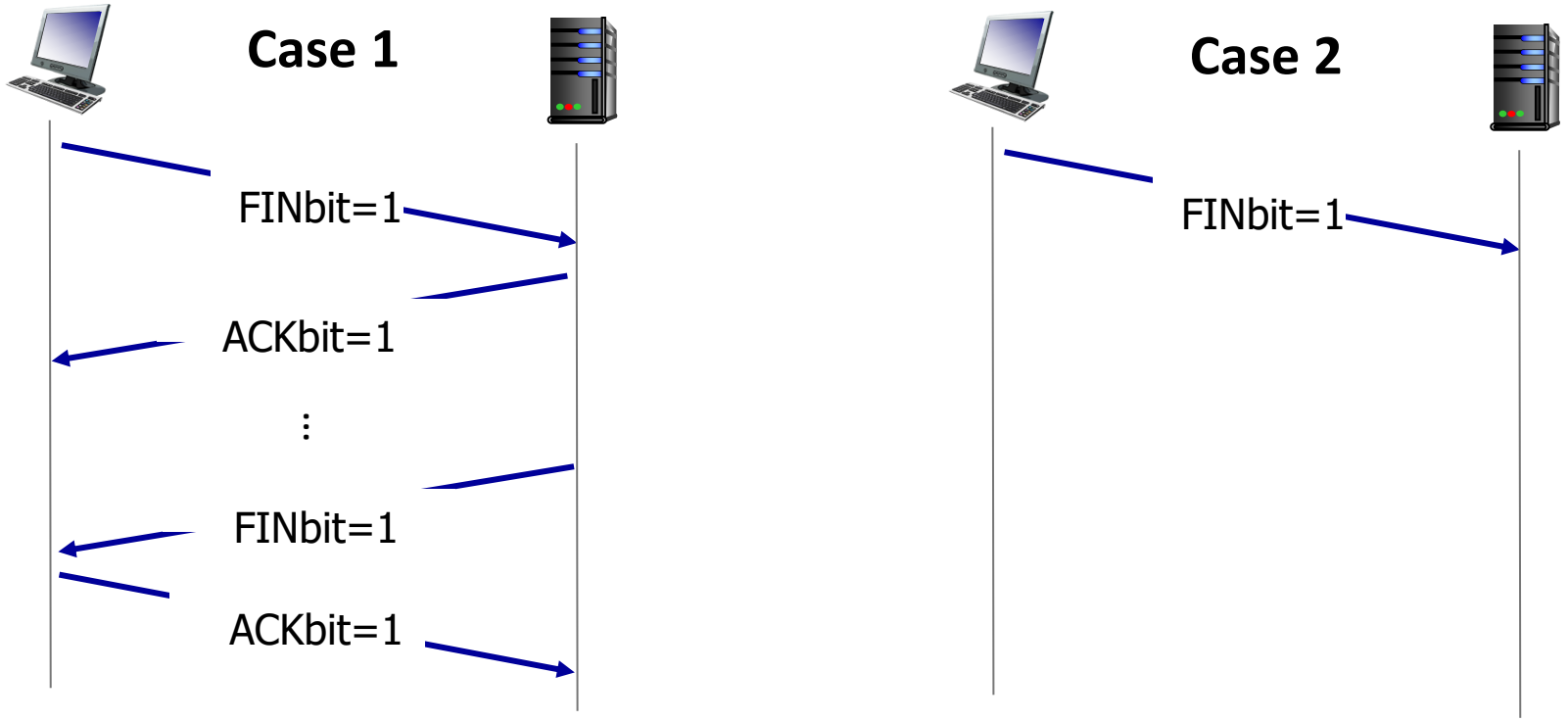
Closing a TCP connection

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



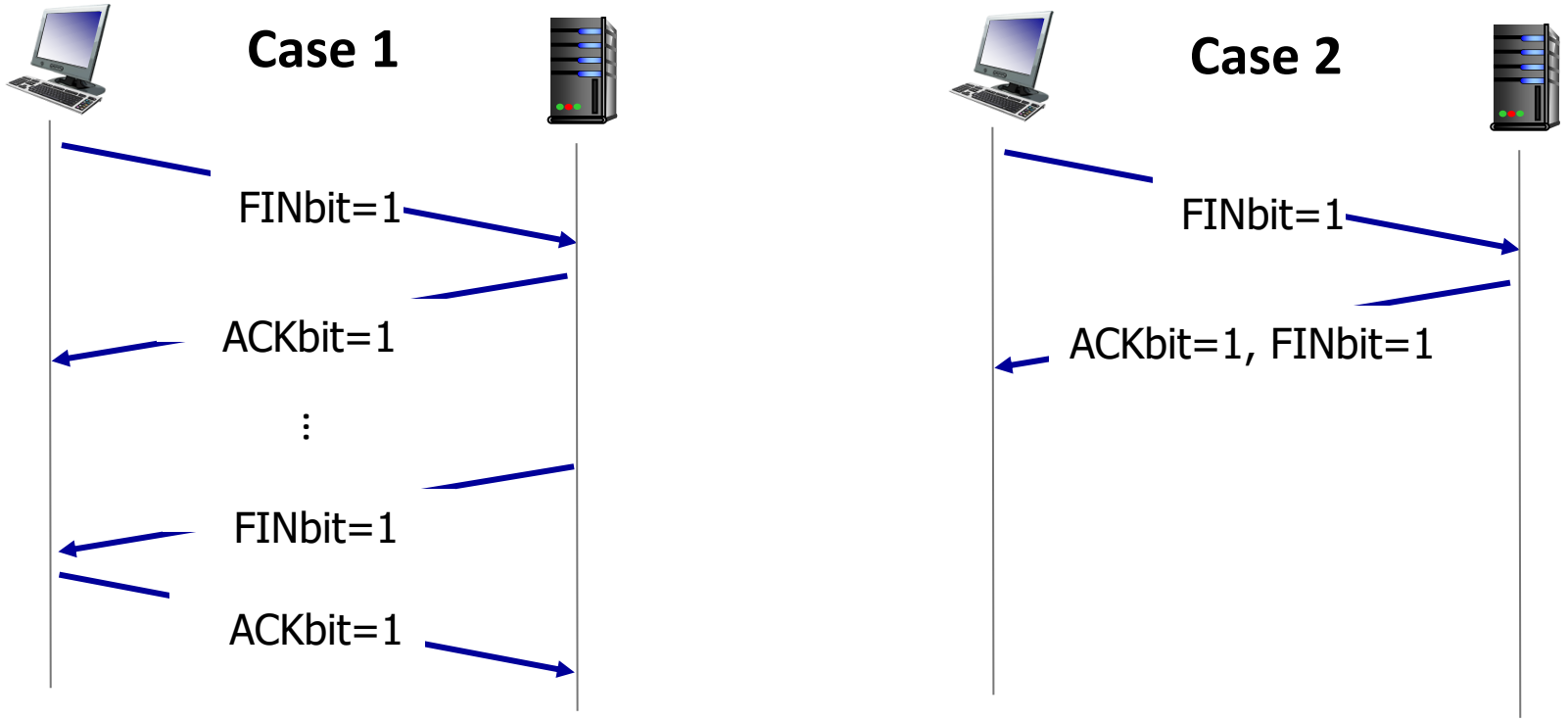
Closing a TCP connection

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



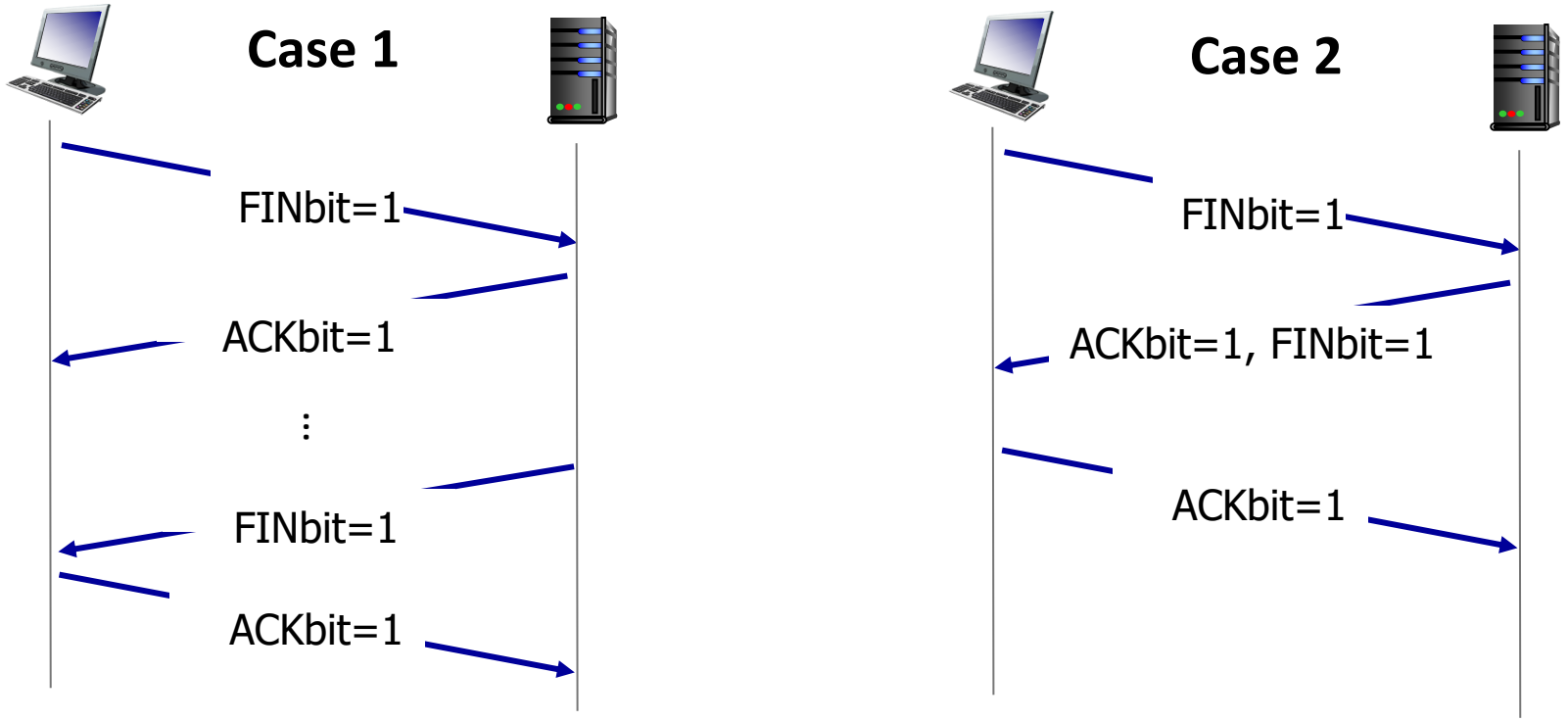
Closing a TCP connection

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

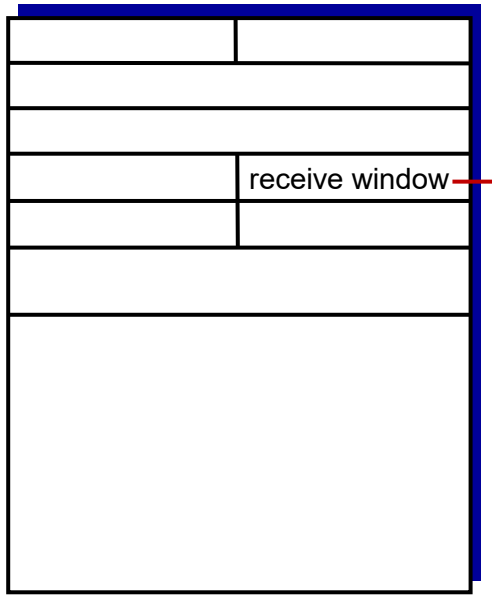


Closing a TCP connection

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

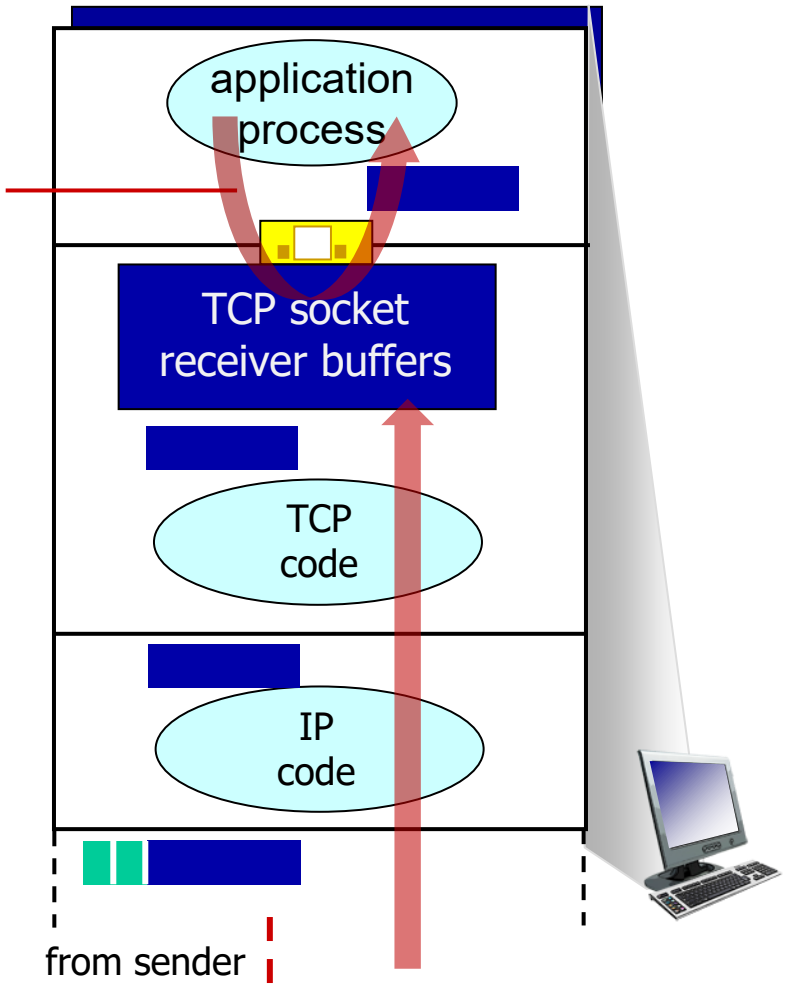


TCP flow control



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



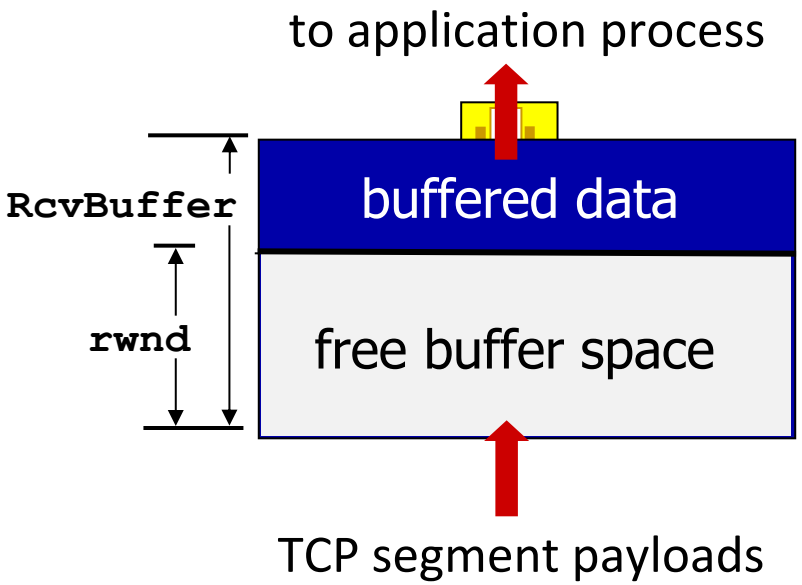
receiver protocol stack

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 receive window (**`rwnd`**) field in TCP header
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

Principles of congestion control

Congestion:

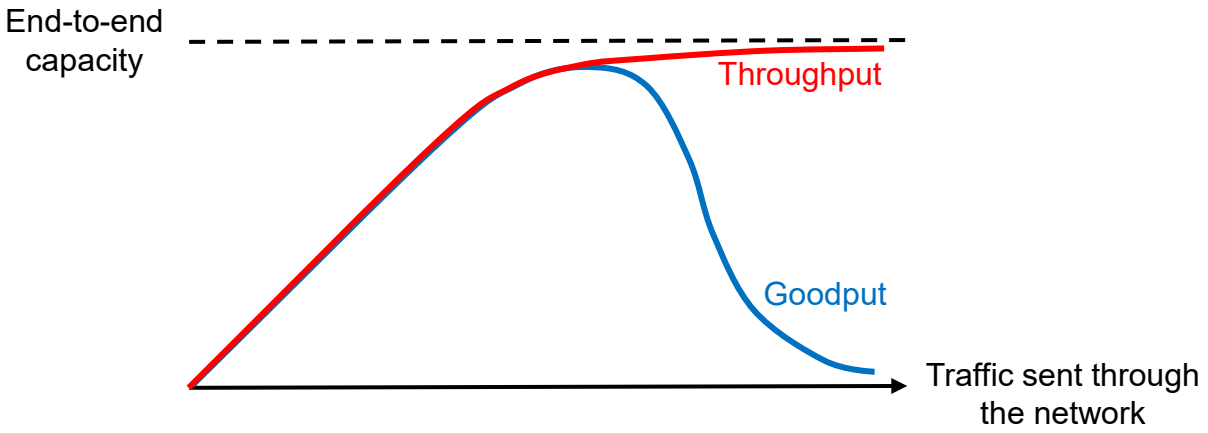
- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!

congestion control:
too many senders,
sending too fast

flow control: one sender
too fast for one receiver

Congestion

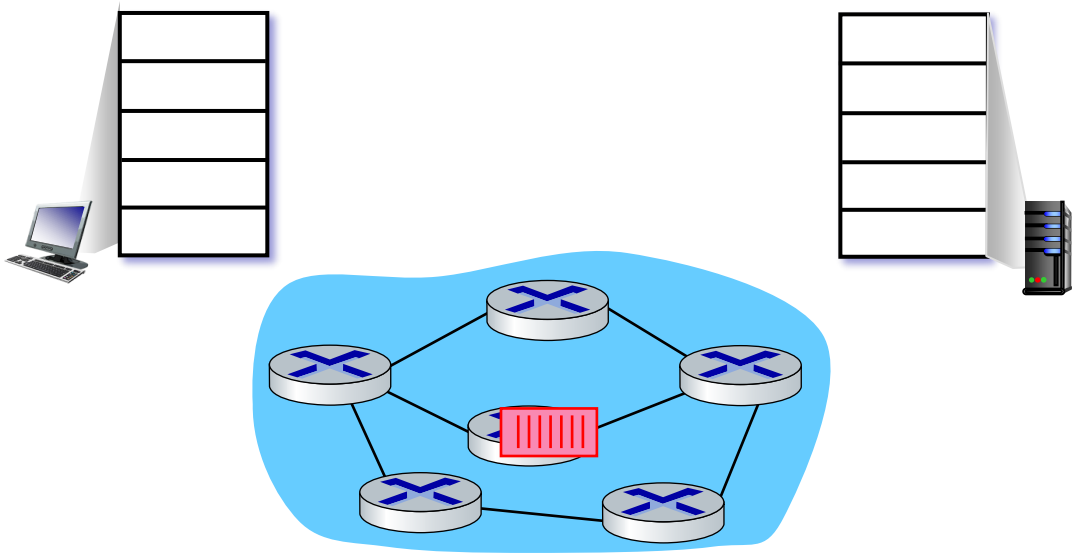
- When network gets closer to saturation of available resources, delay and loss percentage increase
- If the transport layer retransmits messages, the average number of messages retransmissions increases too
- While throughput is close to 100% of capacity, “goodput” experienced by the application decreases!



Approaches towards congestion control

End-to-end congestion control:

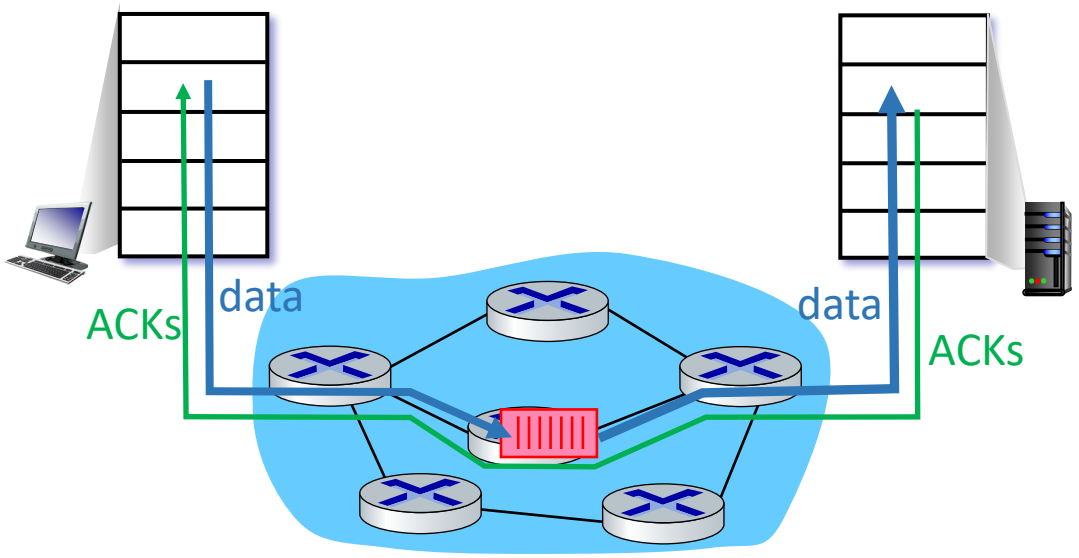
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



Approaches towards congestion control

End-to-end congestion control:

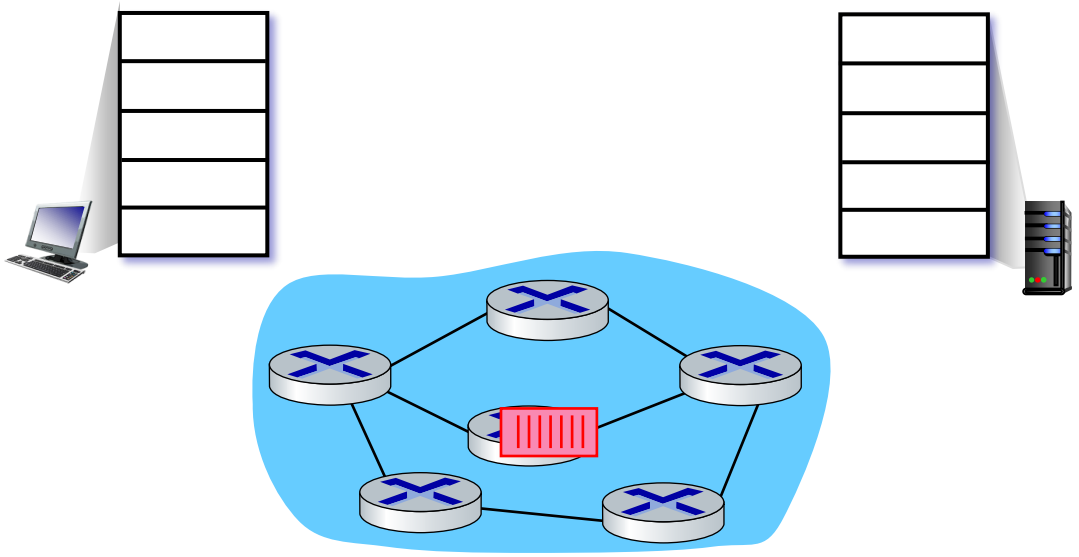
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

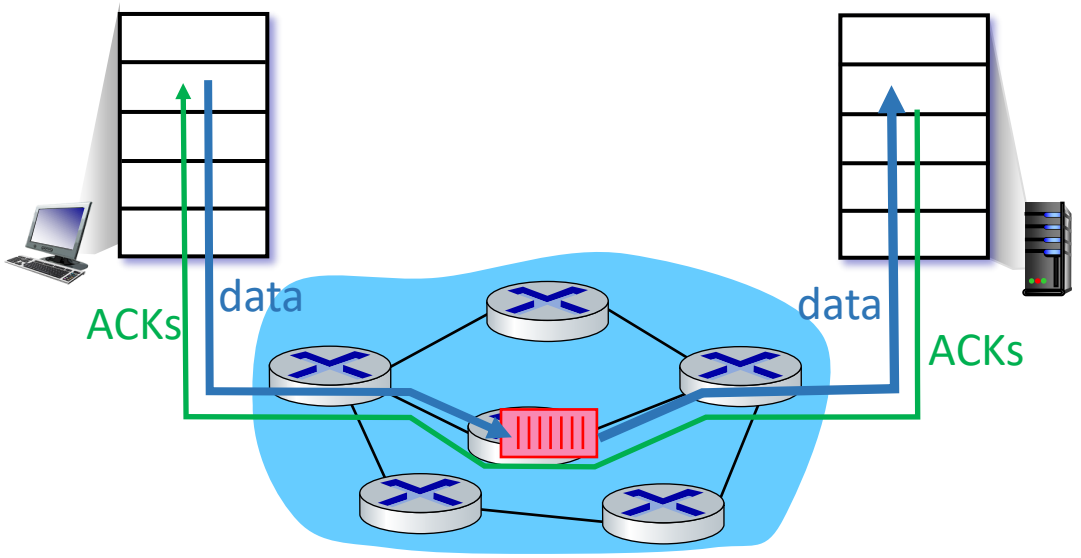
- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate



Approaches towards congestion control

Network-assisted congestion control:

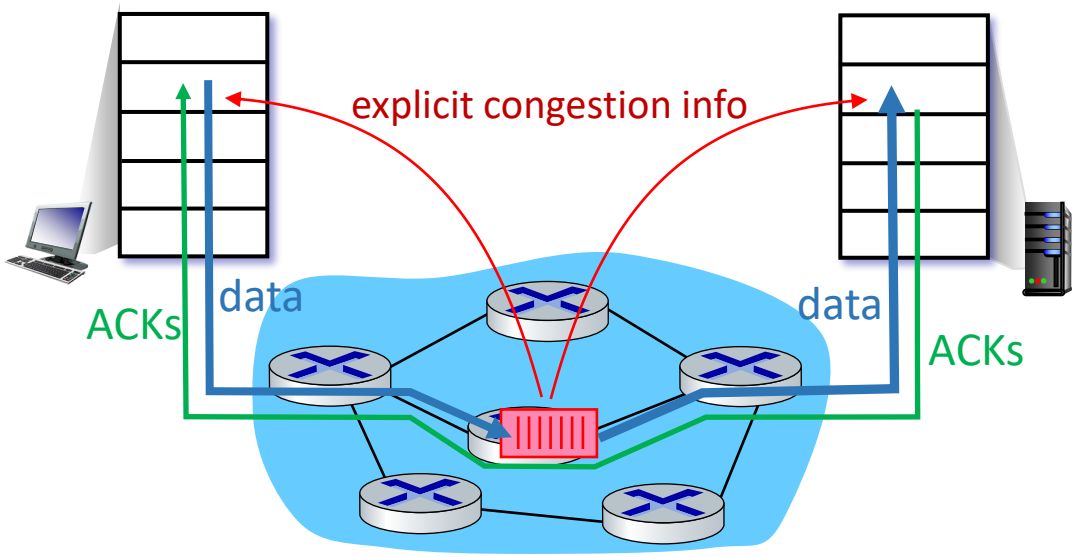
- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate



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

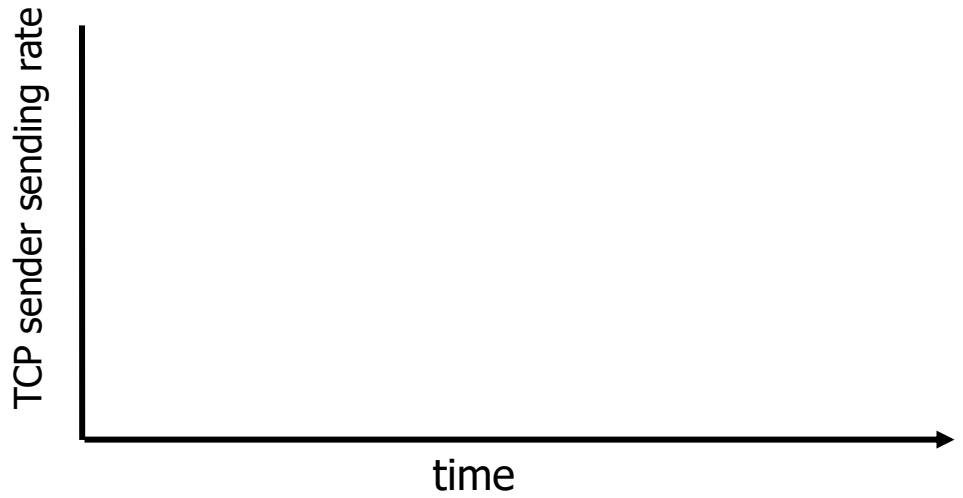


TCP congestion control: Additive Increase Multiplicative Decrease

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

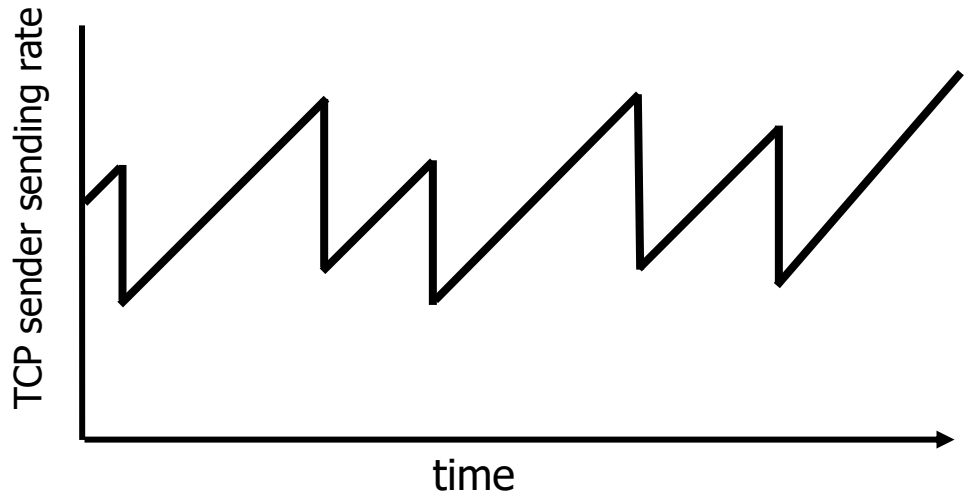
TCP congestion control: Additive Increase Multiplicative Decrease

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



TCP congestion control: Additive Increase Multiplicative Decrease

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

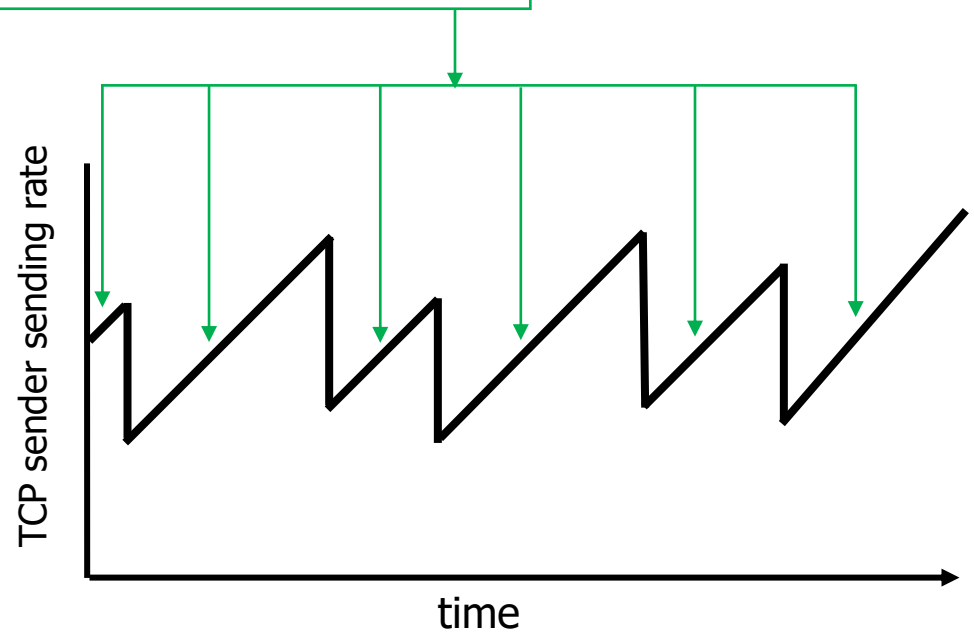


TCP congestion control: Additive Increase Multiplicative Decrease

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

Additive Increase

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



TCP congestion control: Additive Increase Multiplicative Decrease

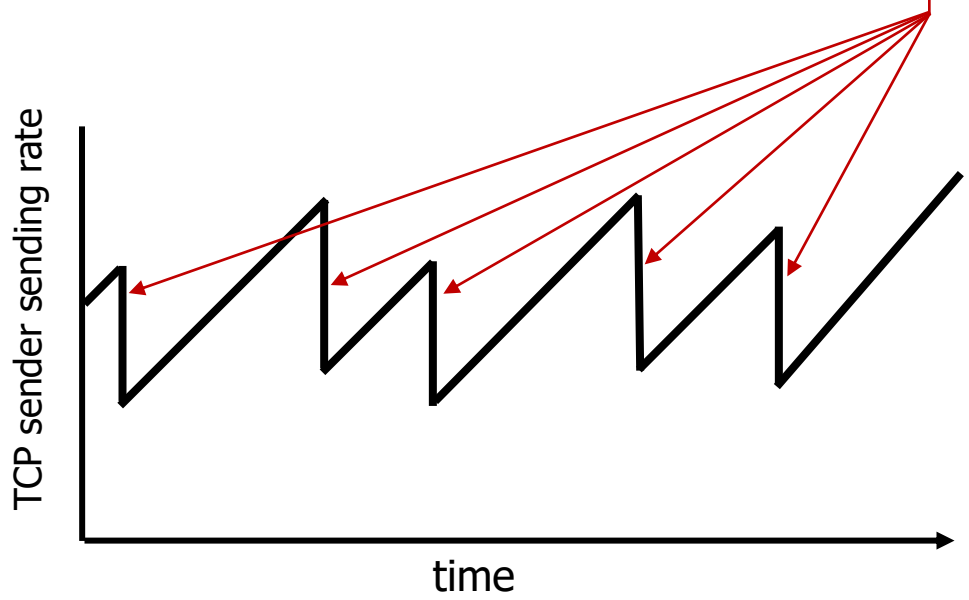
- *approach:* senders can increase sending rate until packet loss (due to congestion) occurs, then decrease sending rate on loss event

Additive Increase

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

Multiplicative Decrease

cut sending rate in half at each loss event



TCP congestion control: Additive Increase Multiplicative Decrease

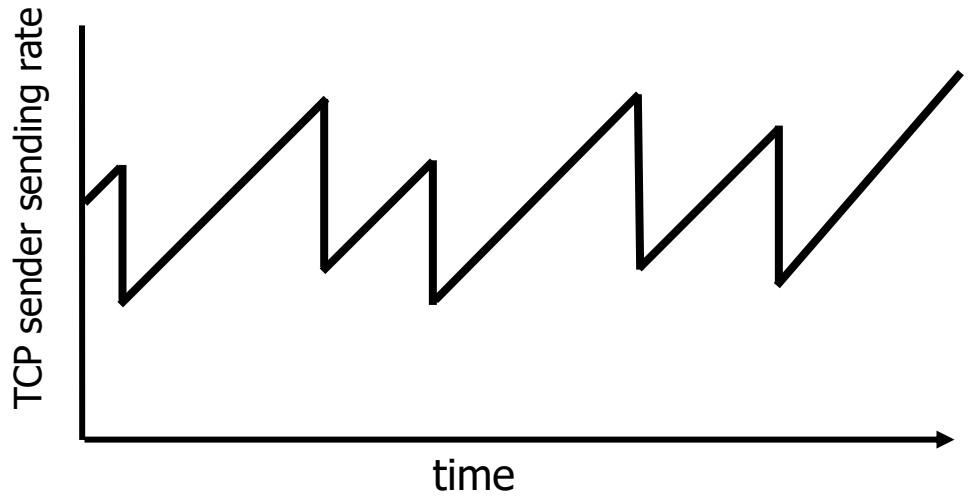
- *approach:* senders can increase sending rate until packet loss (due to congestion) occurs, then decrease sending rate on loss event

Additive Increase

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

Multiplicative Decrease

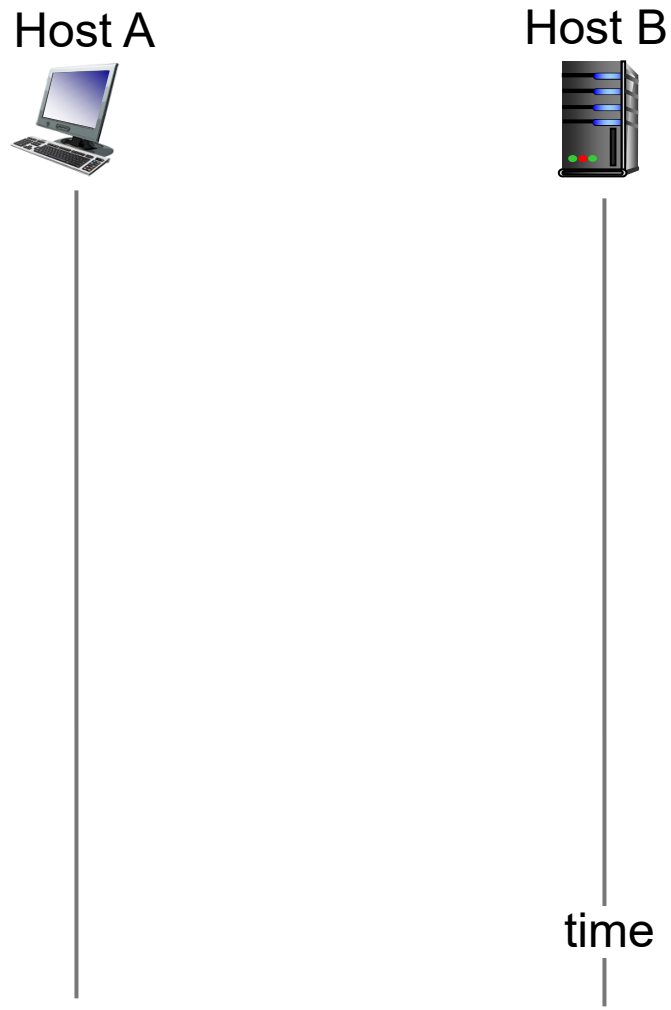
cut sending rate in half at each loss event



AIMD: sawtooth behavior

TCP congestion - Slow Start

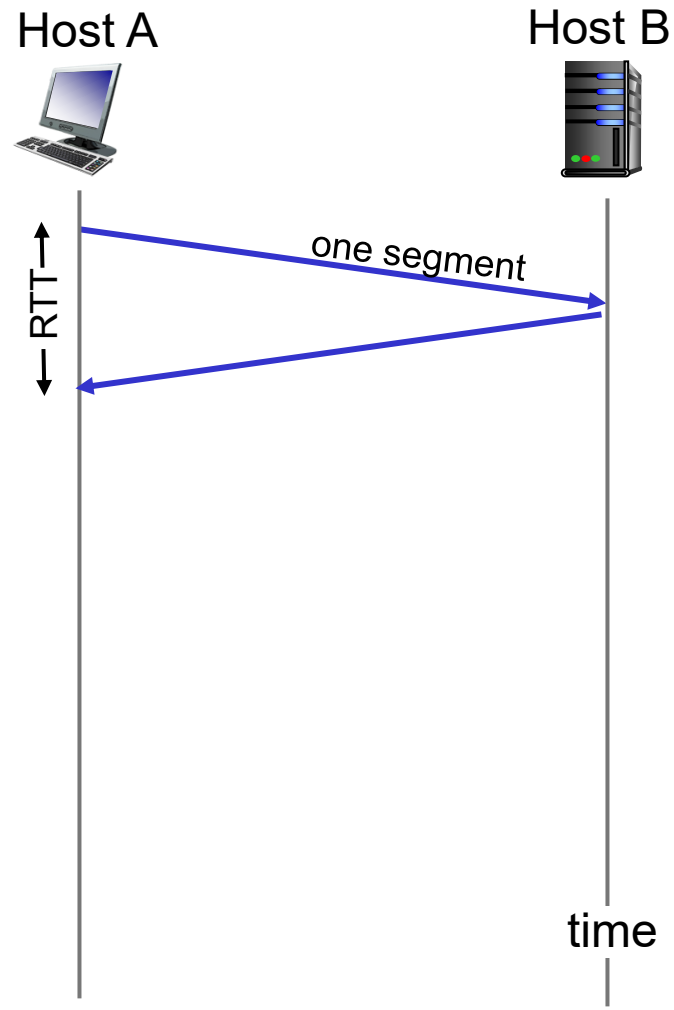
- the number of transmitted segments (MSS) at each RTT is limited by the congestion window (**`cwnd`**)
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT (i.e., transmission round)
  - done by incrementing **`cwnd`** for every ACK received



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

TCP congestion - Slow Start

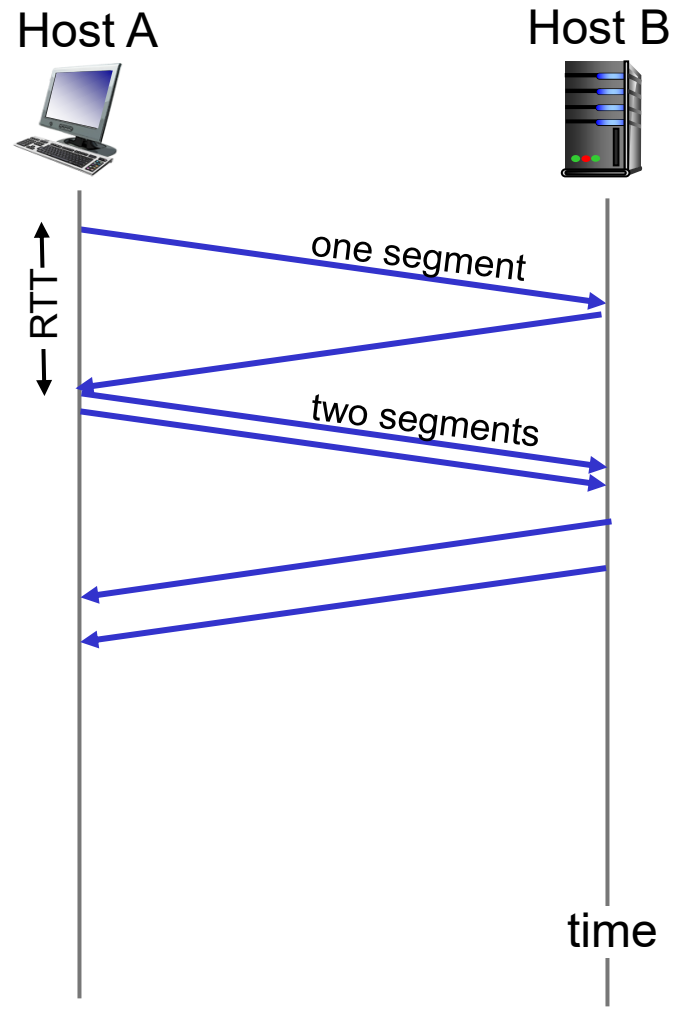
- the number of transmitted segments (MSS) at each RTT is limited by the congestion window (**`cwnd`**)
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT (i.e., transmission round)
  - done by incrementing **`cwnd`** for every ACK received



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

TCP congestion - Slow Start

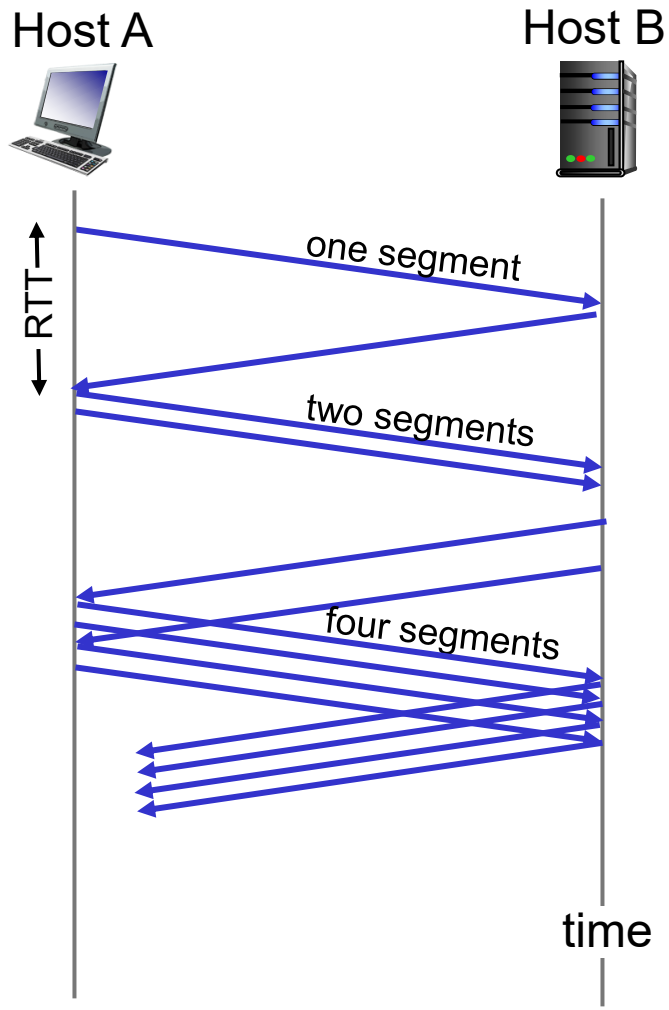
- the number of transmitted segments (MSS) at each RTT is limited by the congestion window (**`cwnd`**)
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT (i.e., transmission round)
  - done by incrementing **`cwnd`** for every ACK received



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

TCP congestion - Slow Start

- the number of transmitted segments (MSS) at each RTT is limited by the congestion window (**`cwnd`**)
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT (i.e., transmission round)
  - done by incrementing **`cwnd`** for every ACK received



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

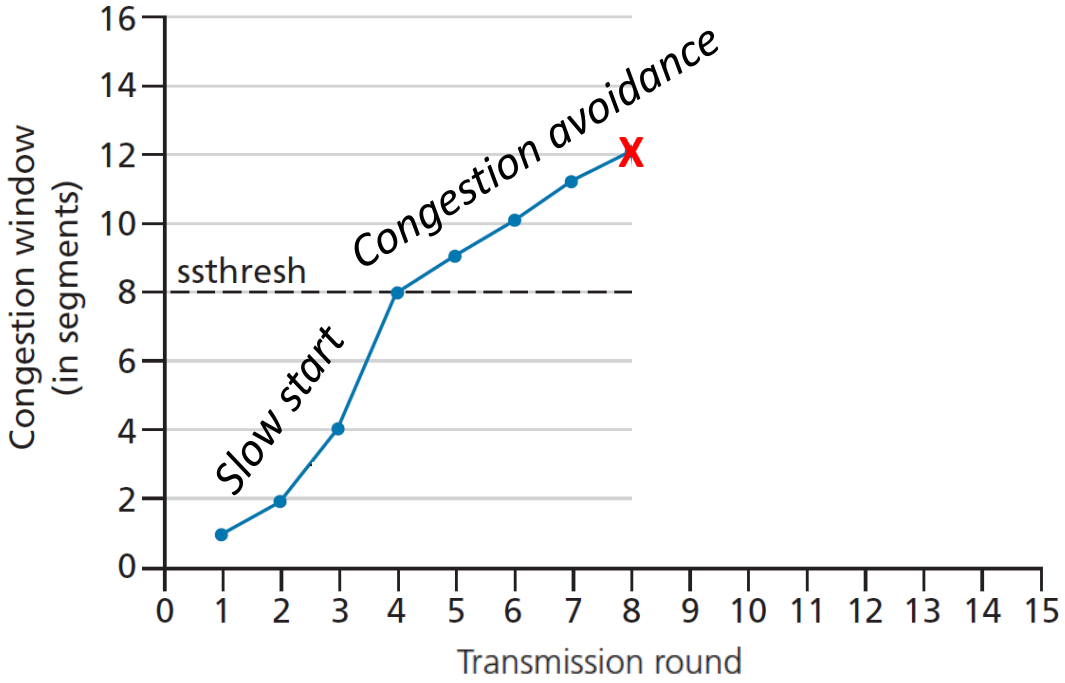
TCP congestion
From Slow Start to Congestion Avoidance

Q: when should the exponential increase switch to linear (Congestion Avoidance)?

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

Implementation:

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



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

TCP congestion
From Slow Start to Congestion Avoidance

Q: when should the exponential increase switch to linear (Congestion Avoidance)?

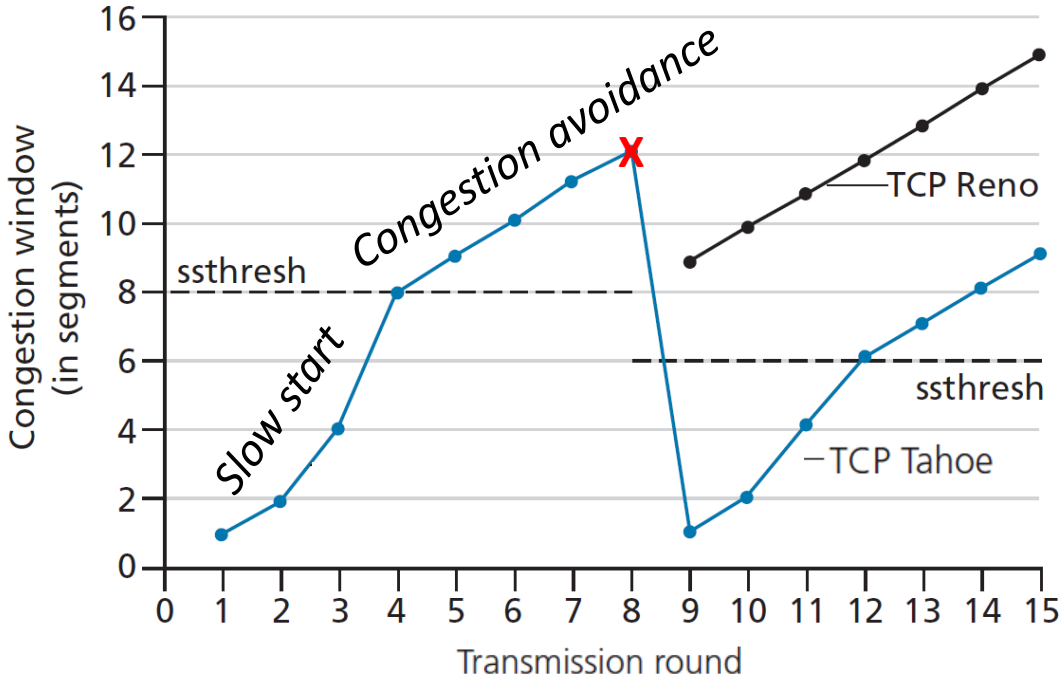
A: when **`cwnd`** gets to 1/2 of its value before last timeout

Implementation:

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

TCP Reno: it adopts the **Fast Recovery** mechanism after three duplicates acks

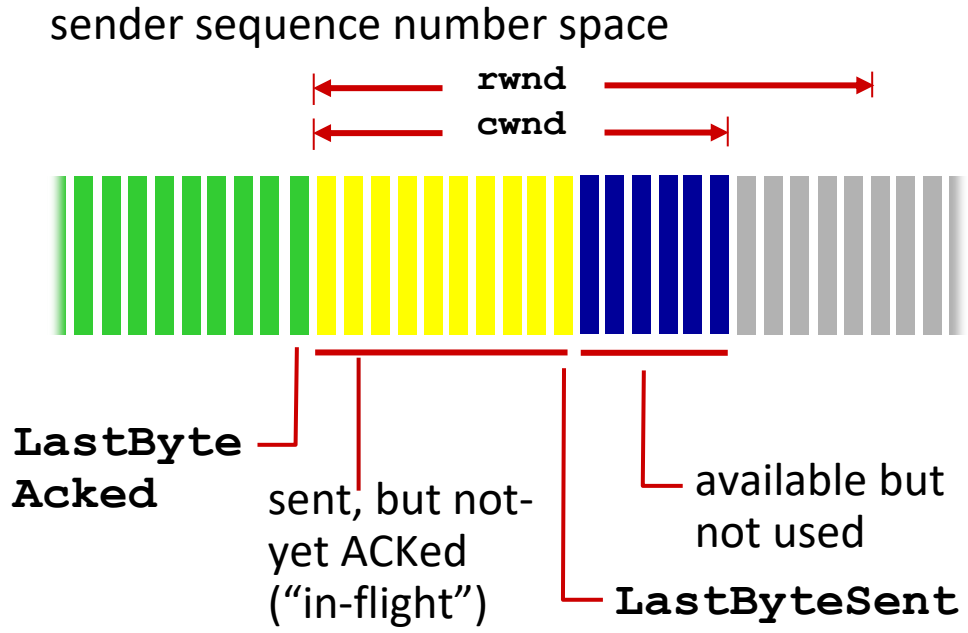
- set `cwnd` to `ssthresh + 3 MSS` (congestion avoidance)
- same as TCP Tahoe when timeout is reached



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

TCP congestion and flow control: details

rwnd: receive window
cwnd: congestion window



TCP sending behavior:

- Send `min{cwnd, rwnd}` bytes (**transmission window**), wait RTT for ACKs, then send more bytes
  - When `cwnd < rwnd`, congestion control is dominant w.r.t. flow control

- TCP sender limits transmission: **`LastByteSent - LastByteAcked ≤ min{cwnd,rwnd}`**
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)
- `rwnd` is instead adjusted according to the received value by the receiver (implementing TCP flow control)

Evolving transport-layer functionality

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

Scenario	Challenges
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**
    - Very interesting, but no time to investigate it!