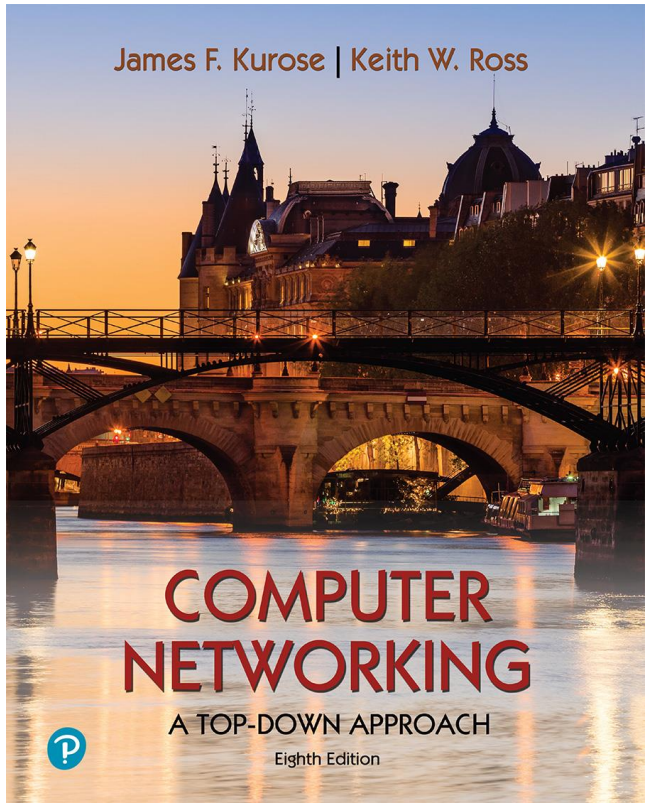
Chapter 3
Transport Layer



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:

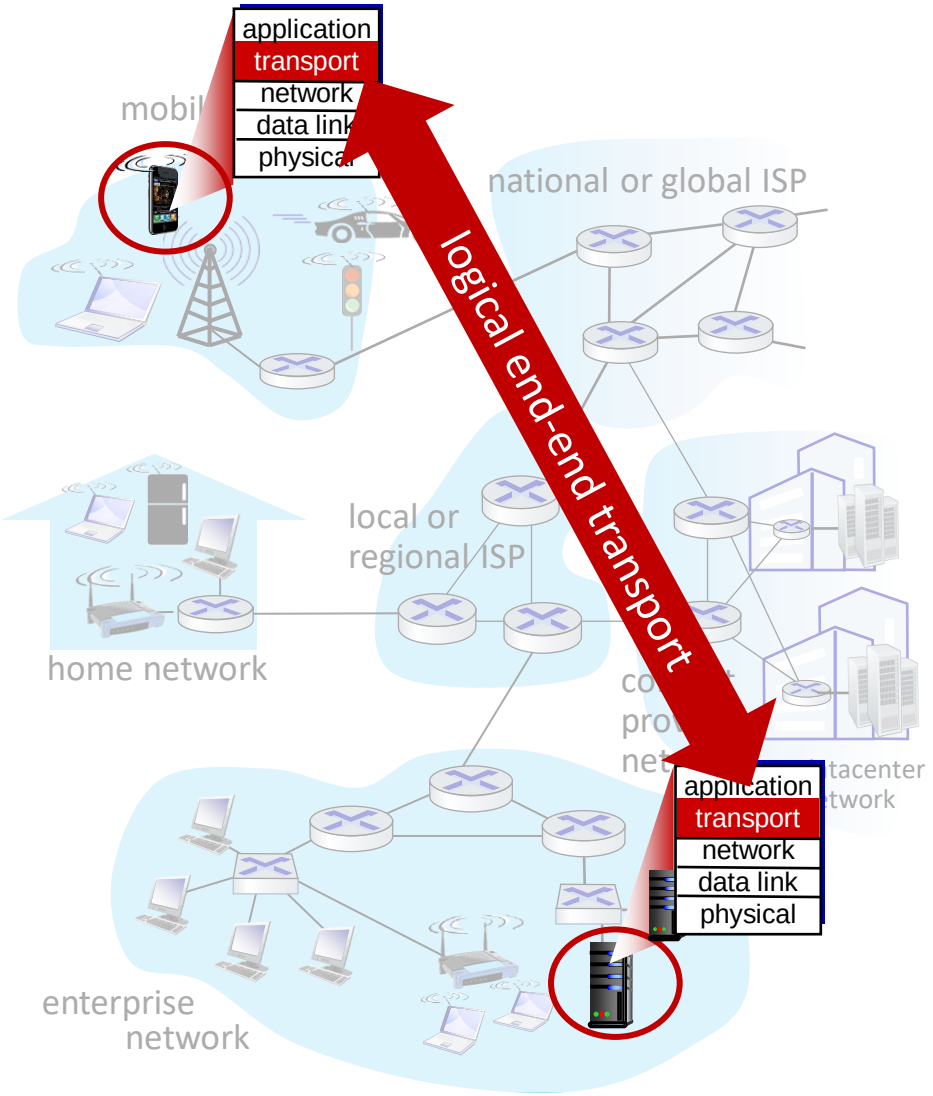
- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

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

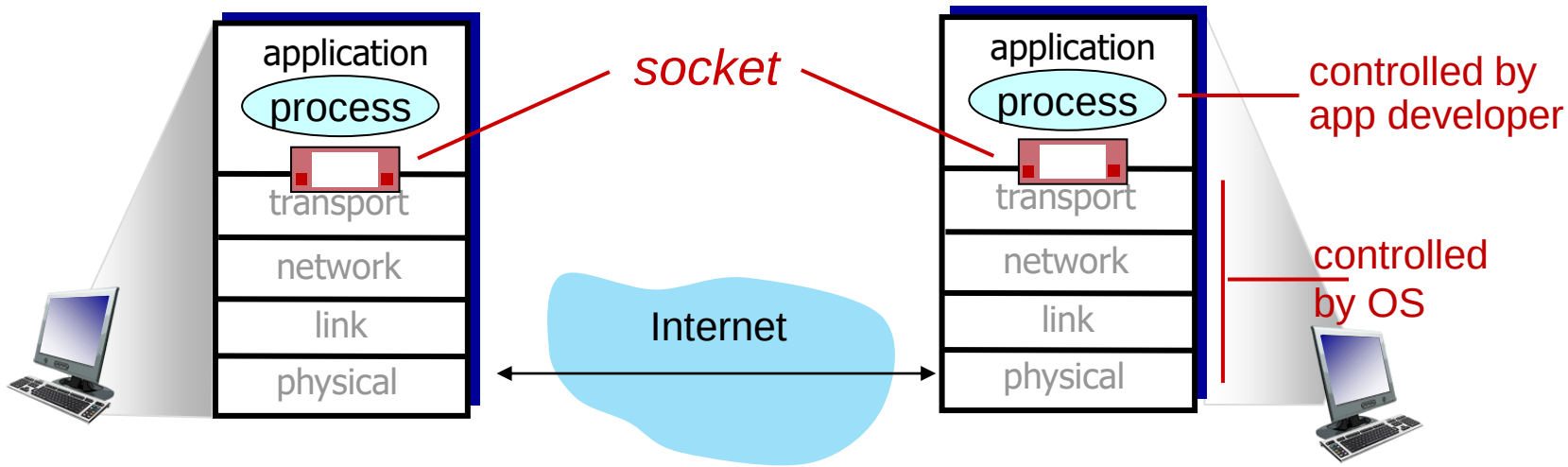
household analogy:

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

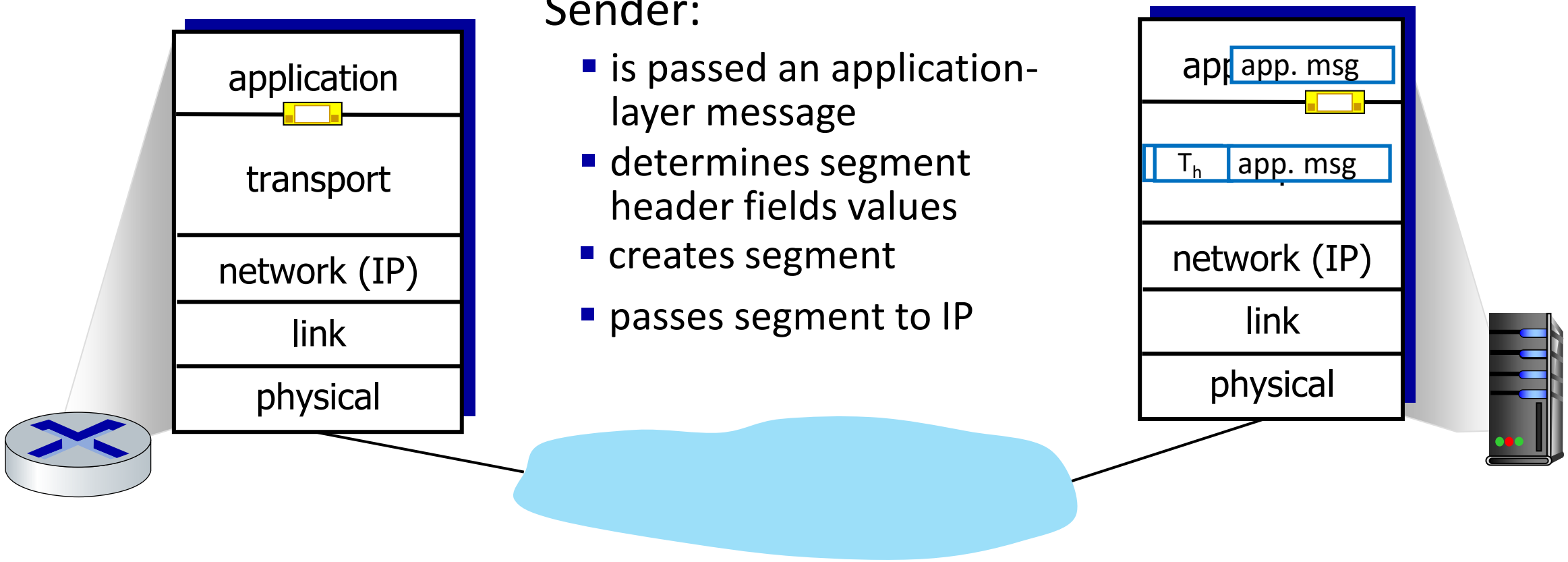
- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill
- network-layer protocol = postal service

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



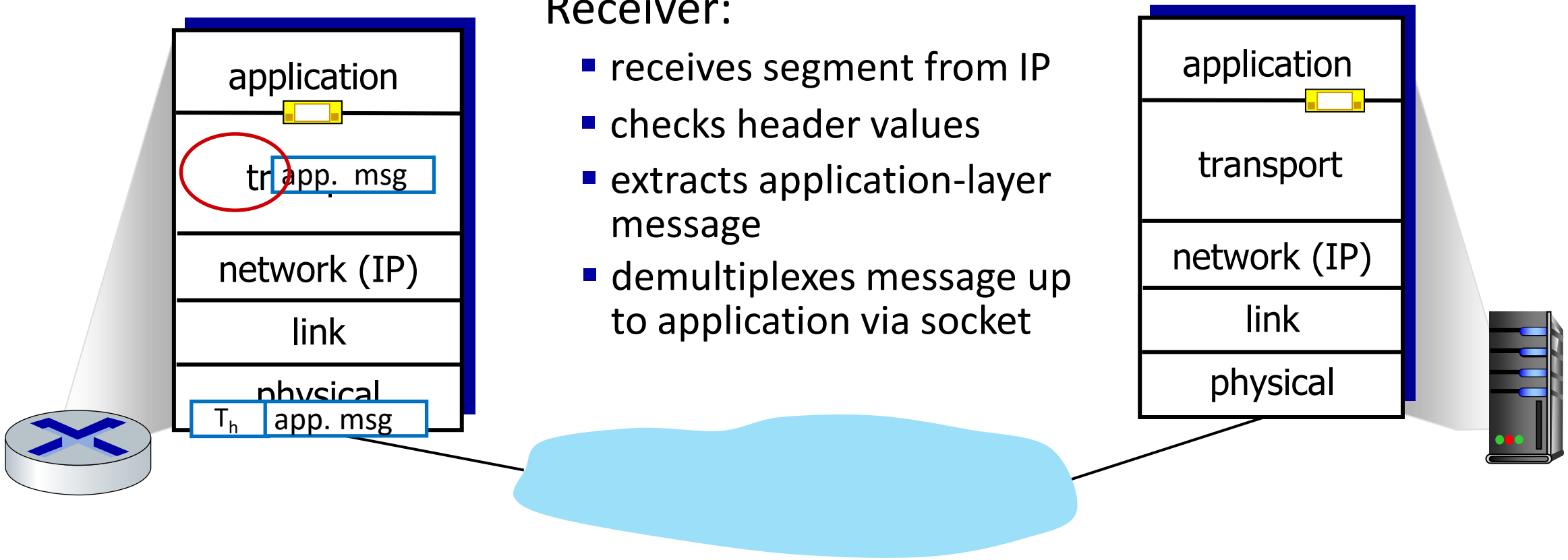
Transport Layer Actions



Sender:

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

Transport Layer Actions

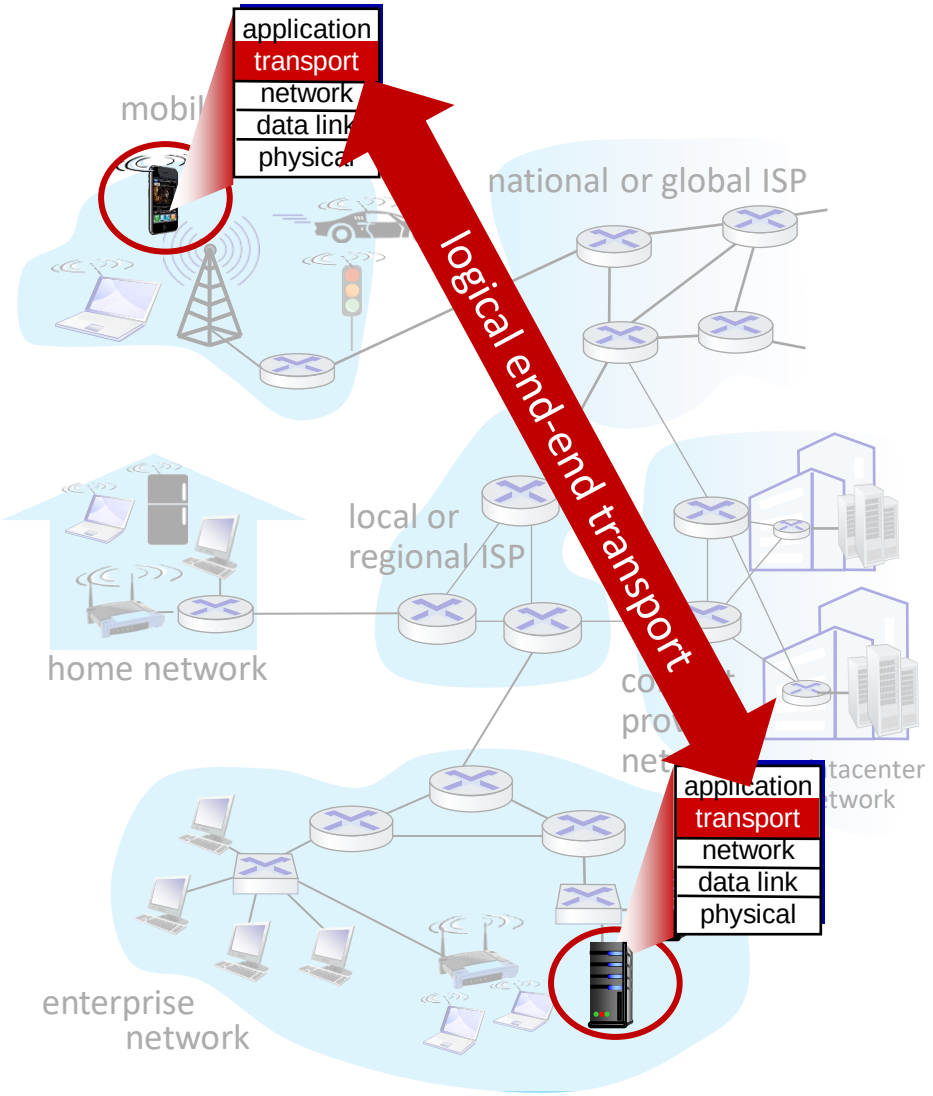


Receiver:

- 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



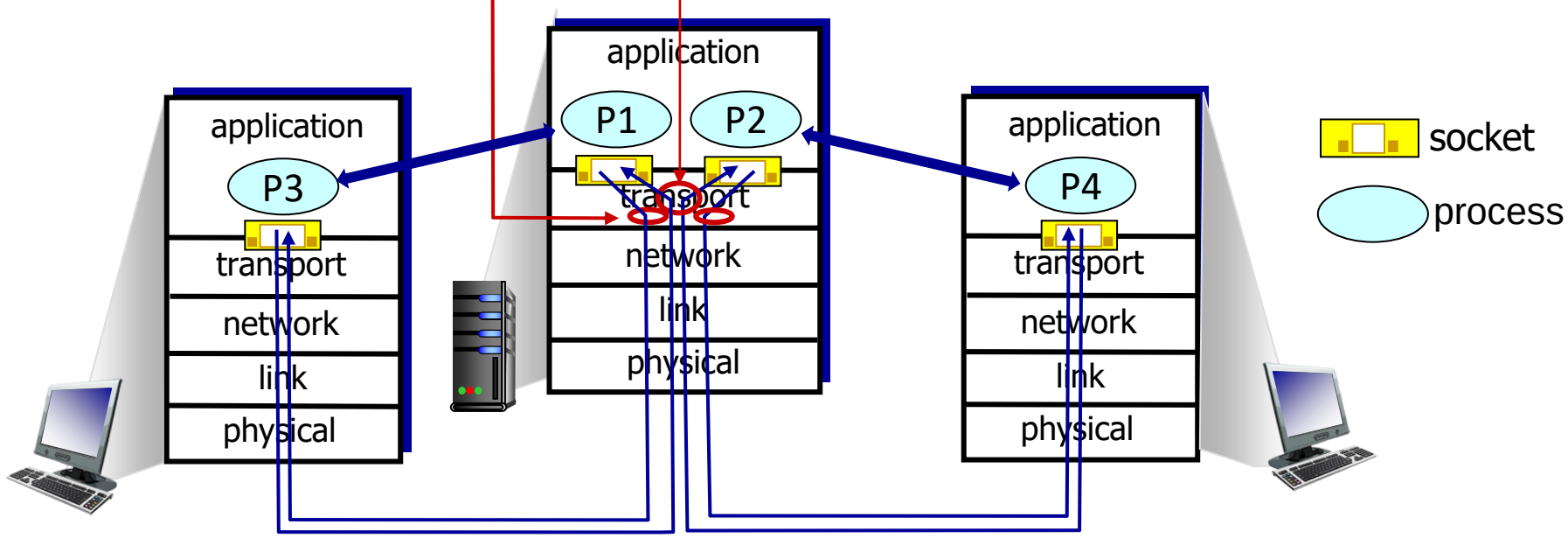
Multiplexing/demultiplexing

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

Connectionless demultiplexing

Recall:

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

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

- when creating datagram to send into UDP socket, must specify
  1. *destination IP address*
  2. *destination port #*
  - these two values univocally identify a UDP socket!

when receiving host receives *UDP* segment:

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

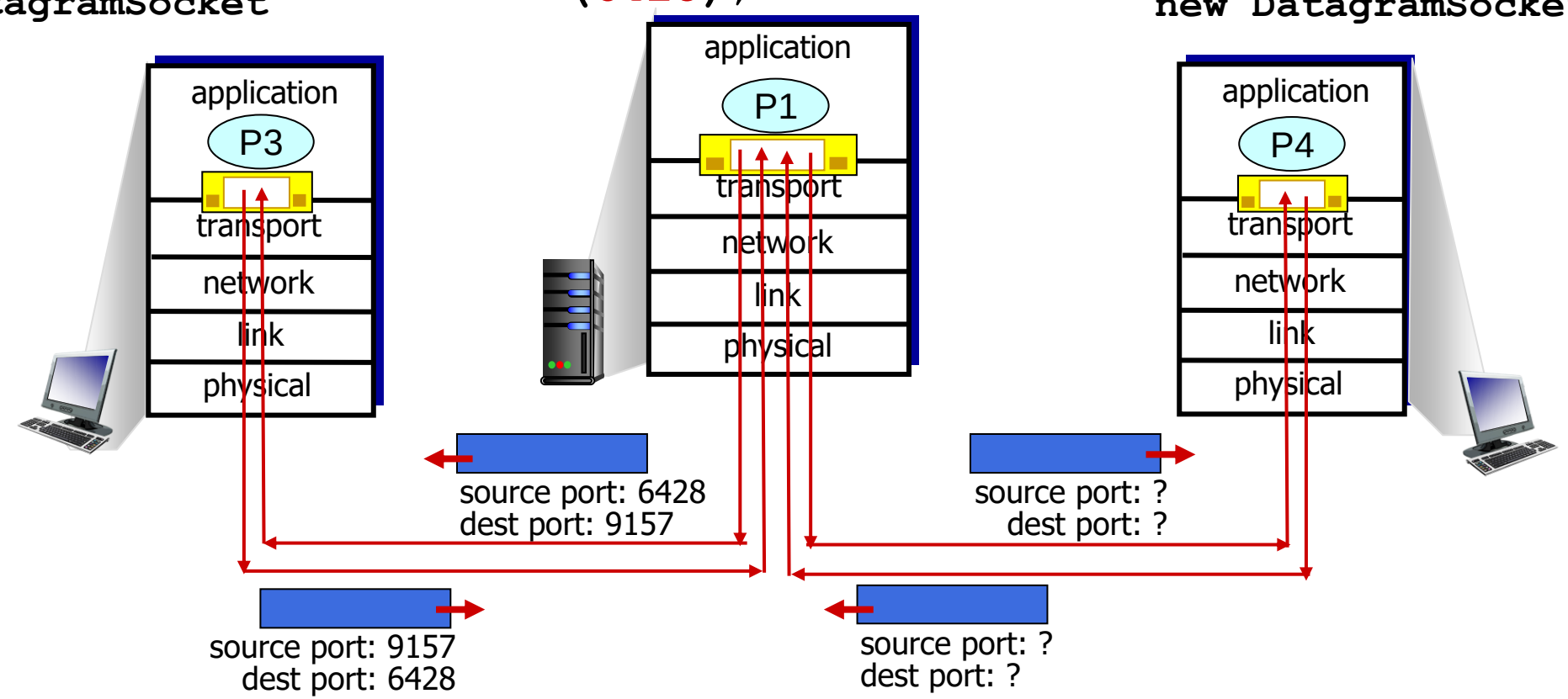
IP/UDP datagrams with *same dest. port # and dest. IP addresses,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

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

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

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

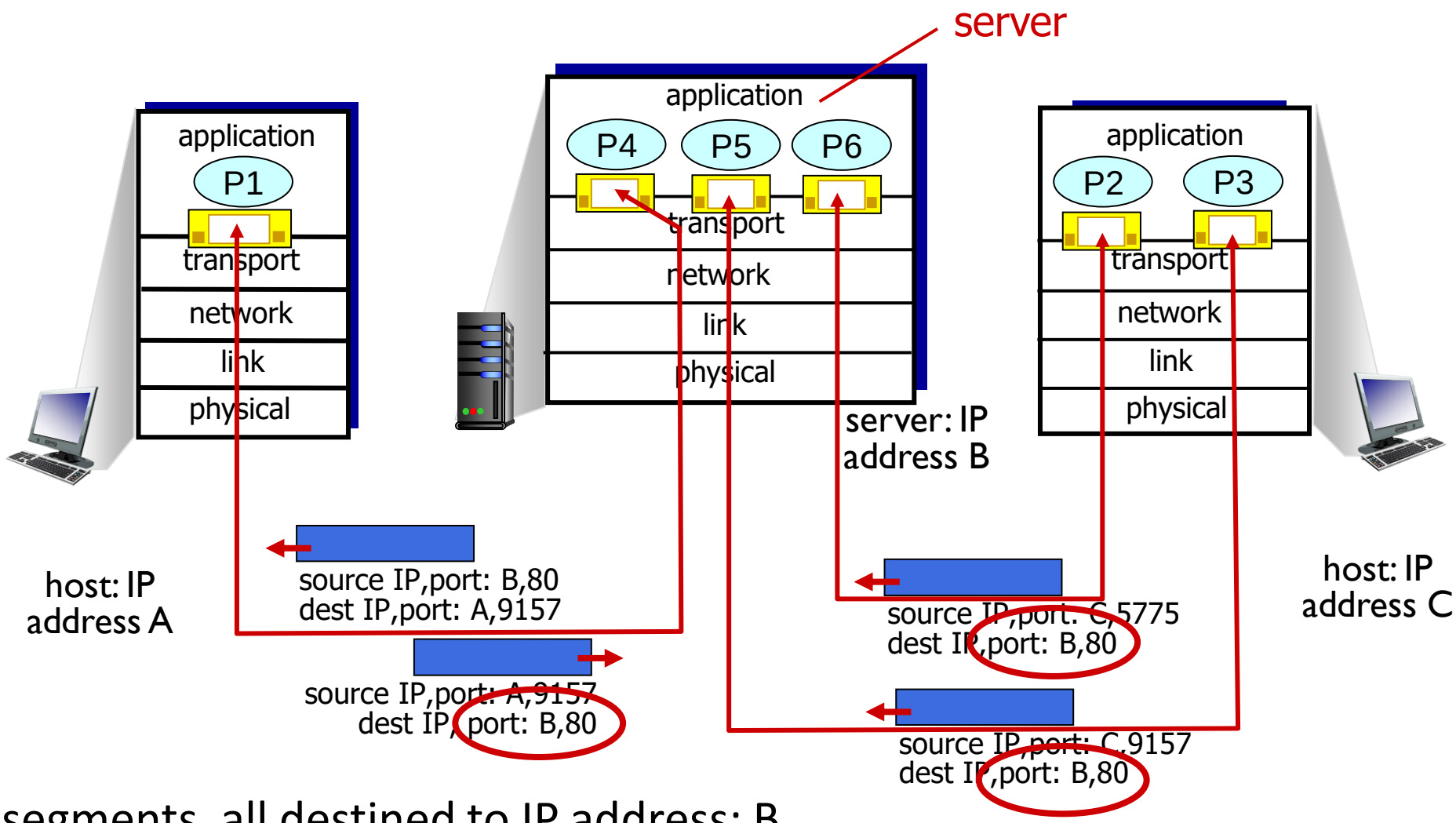


Connection-oriented demultiplexing

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

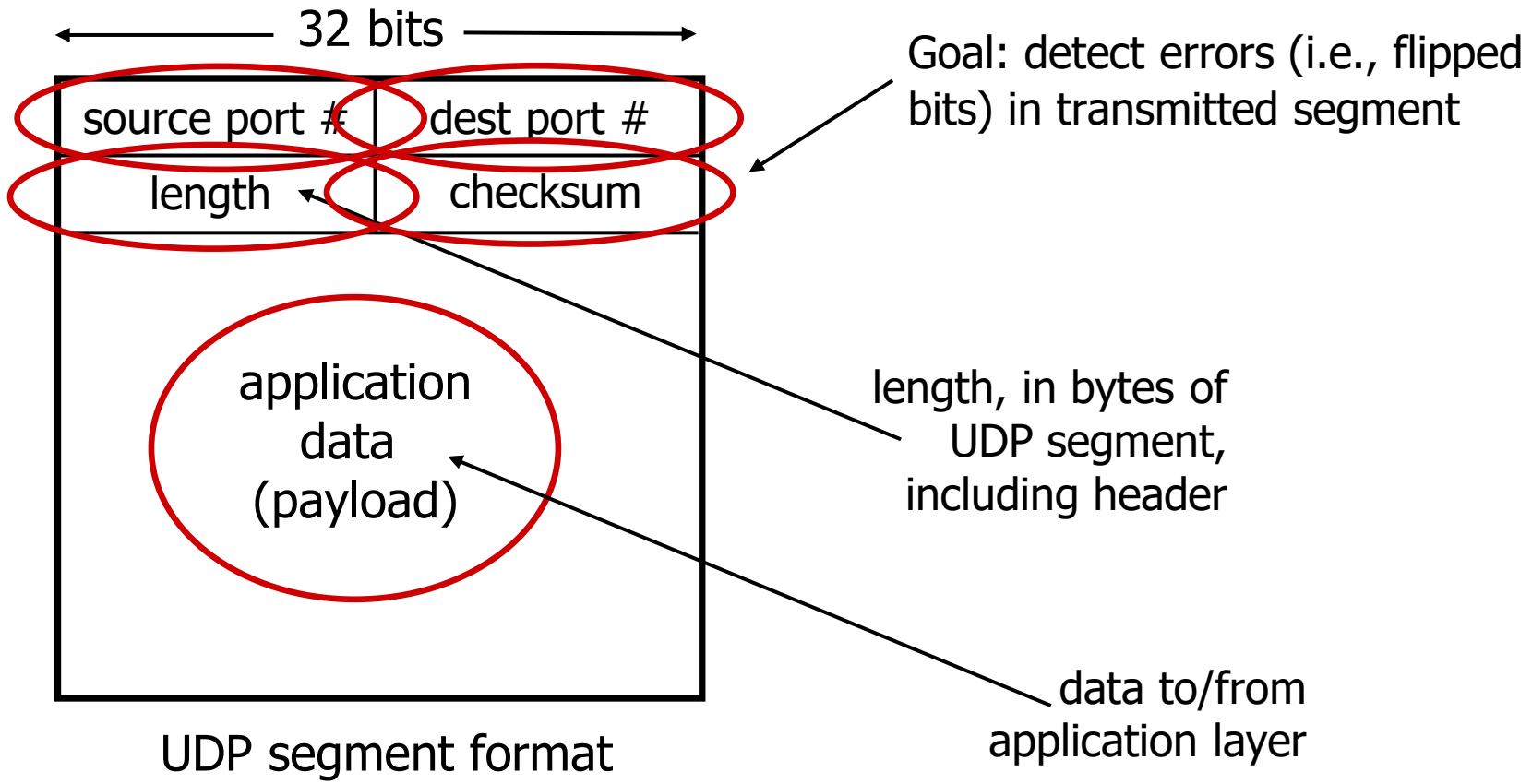
UDP: User Datagram Protocol

- "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



Summary: UDP

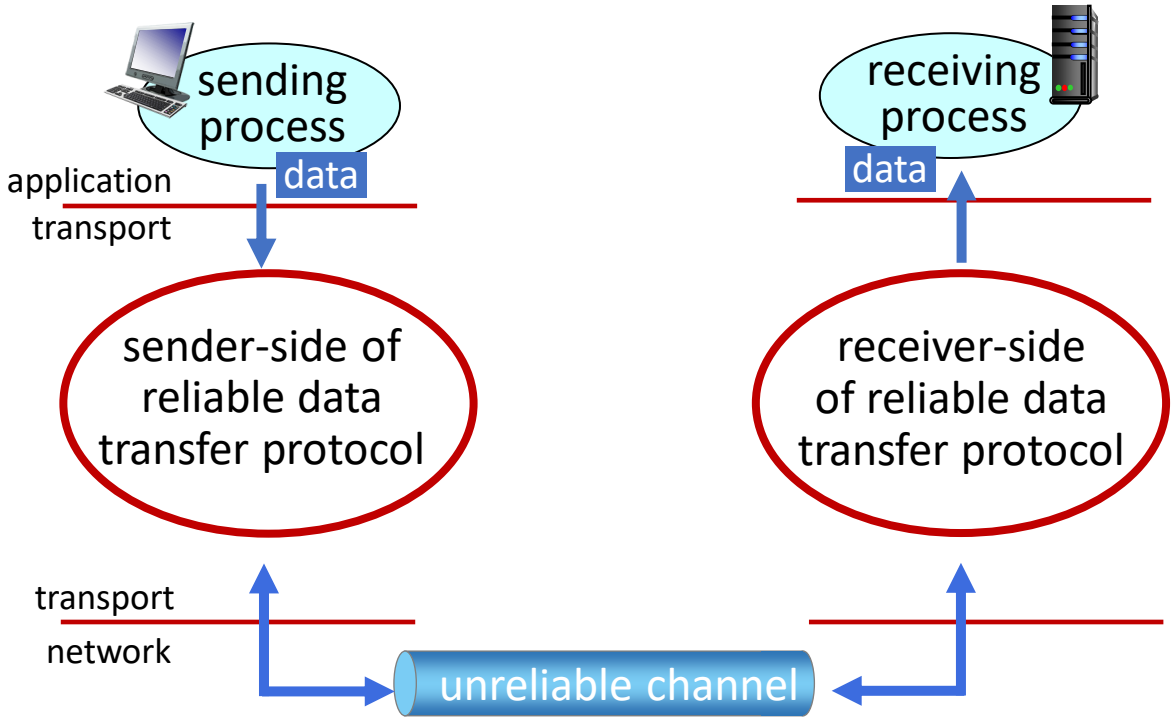
- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no delay incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build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

Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

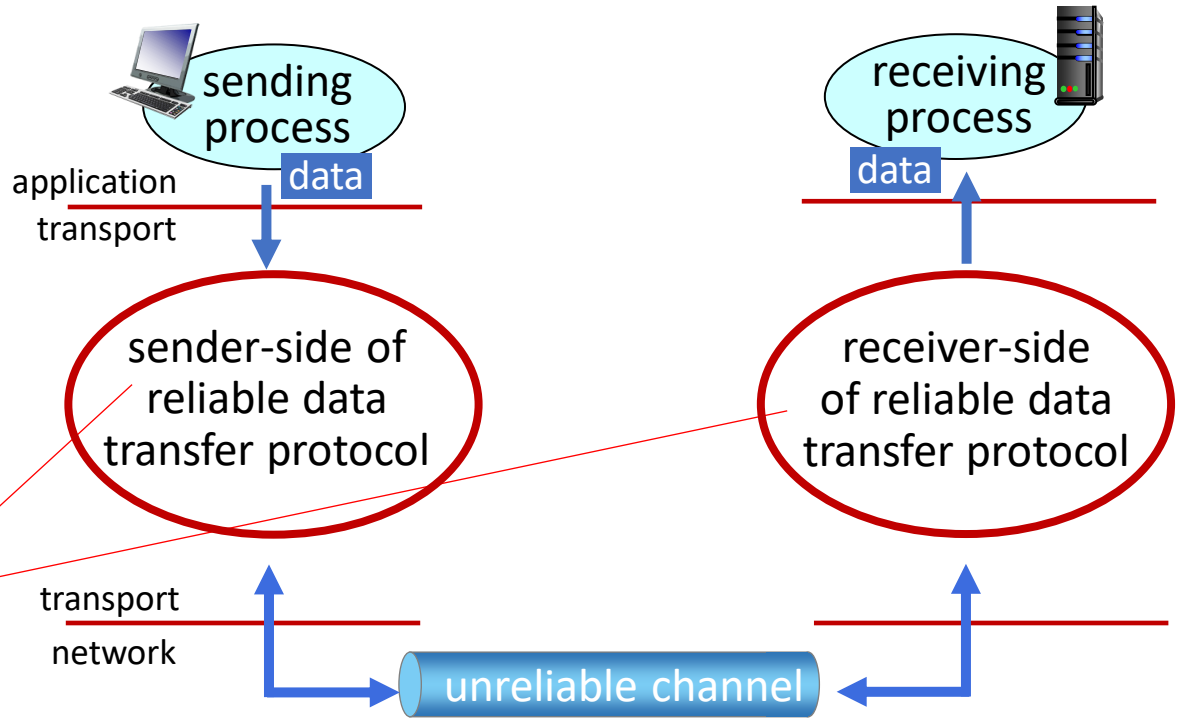
reliable service *abstraction*



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?)

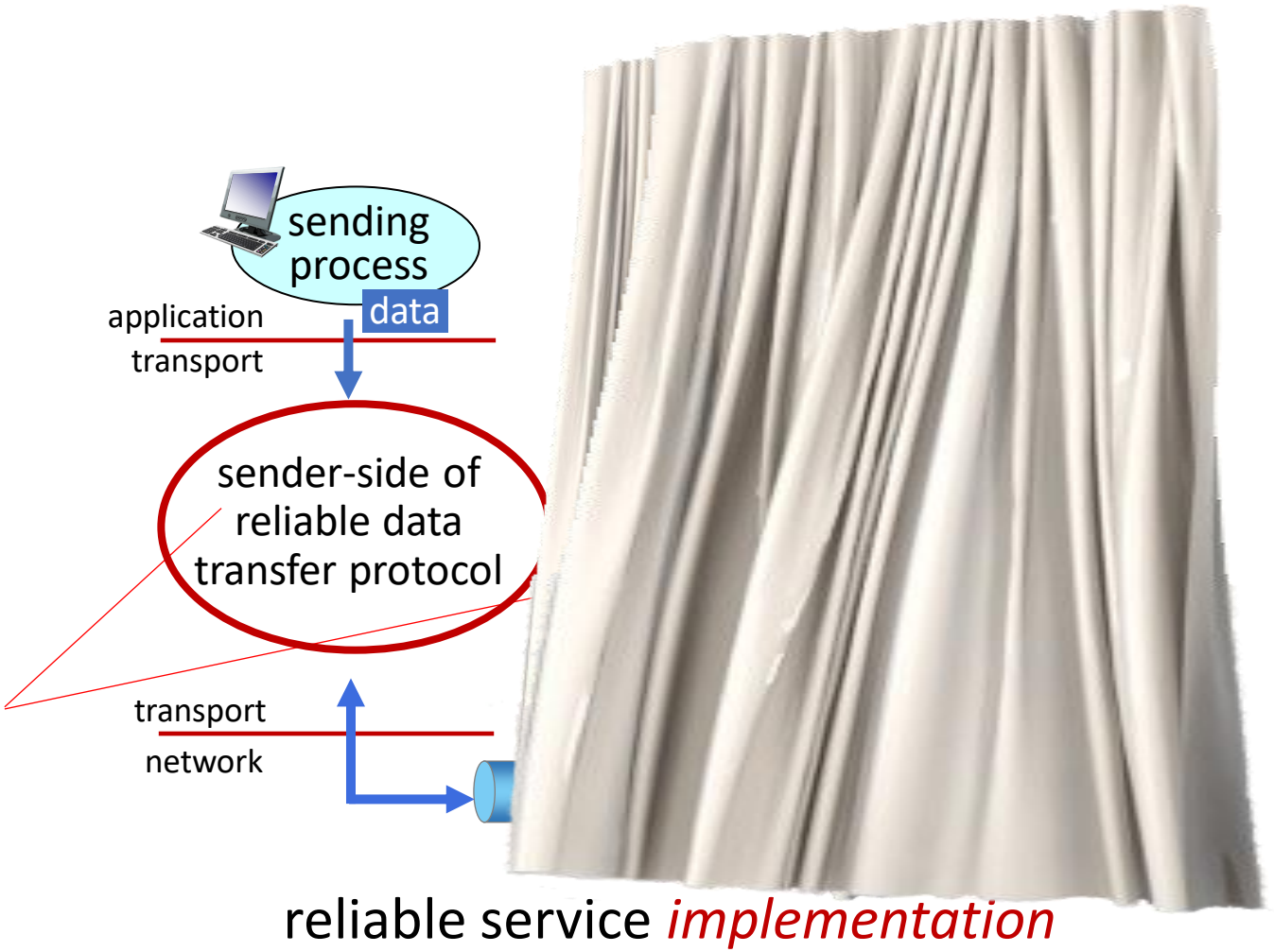


reliable service *implementation*

Principles of reliable data transfer

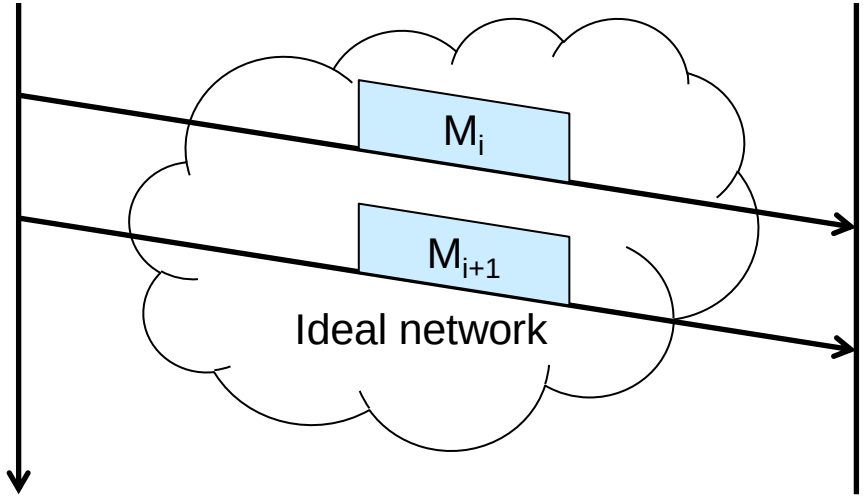
Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- **unless communicated via a message**



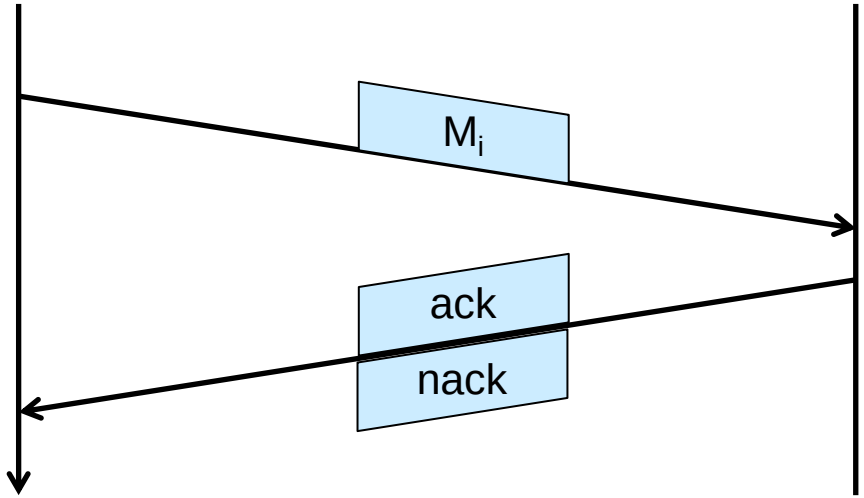
reliable service *implementation*

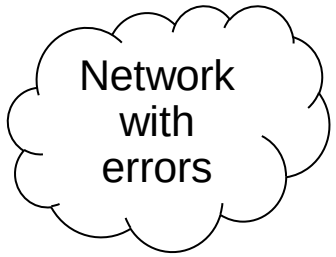
Reliable data transfer



- 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



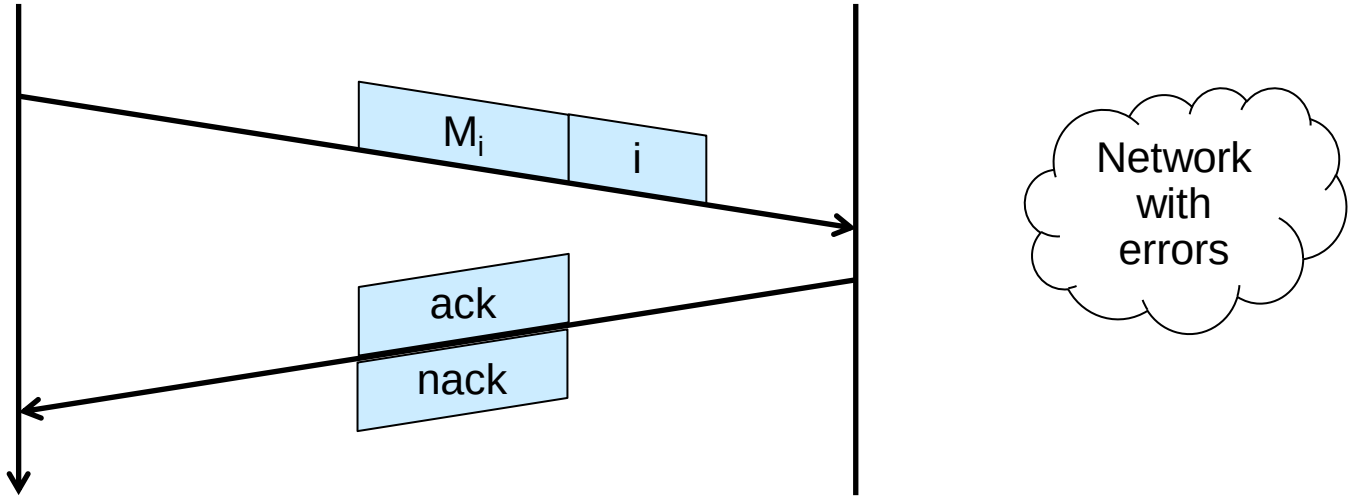


- Unfortunately, ideal networks do not exist…
- In a network with errors it is possible to introduce **positive acknowledgments** (ack) or **negative acknowledgments** (nack)
- Simple possible sender algorithm:

```
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!

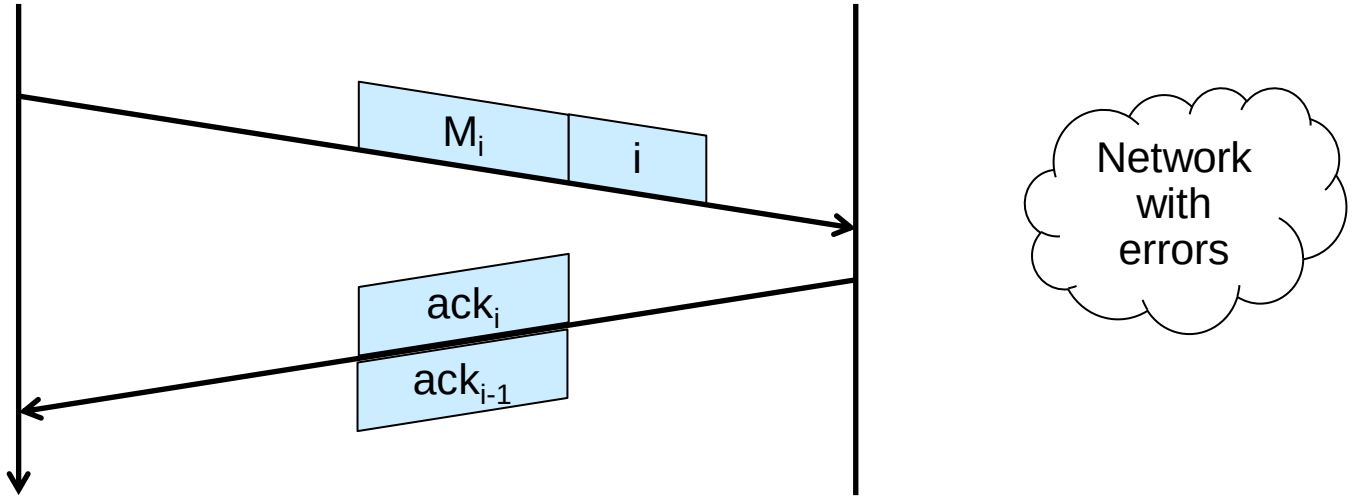
Reliable data transfer



- 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

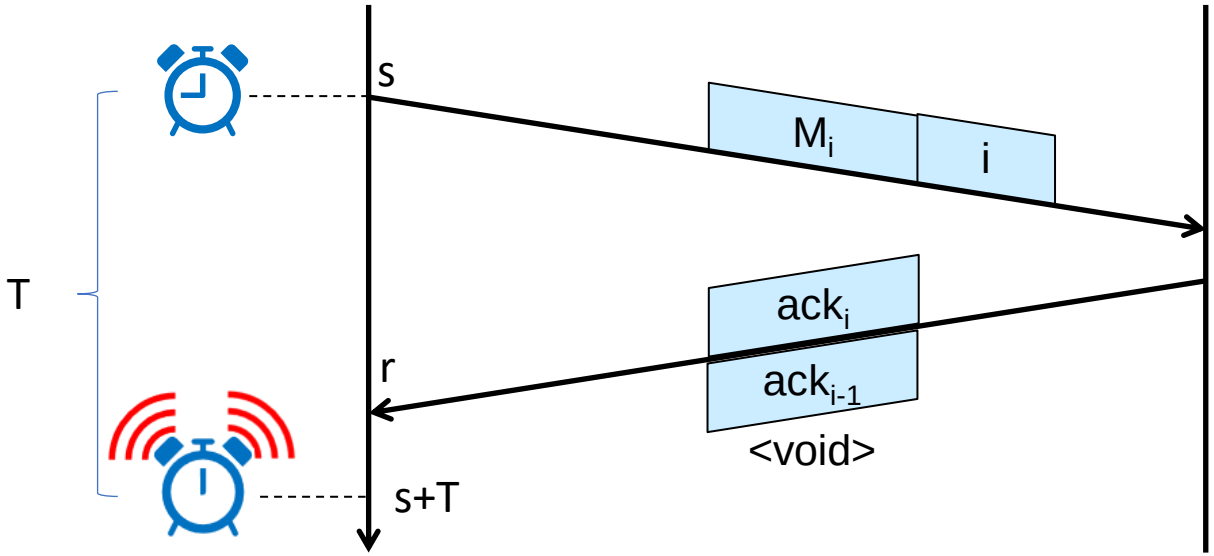


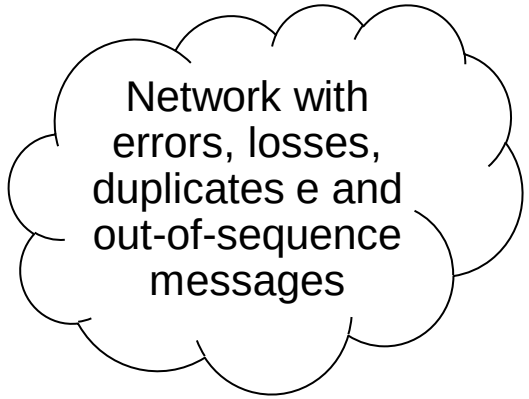
- By also **numbering the acks**, it is possible to avoid nacks thanks to the following rule: *a second ack_{i-1} is equal to a $nack_i$*

```
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



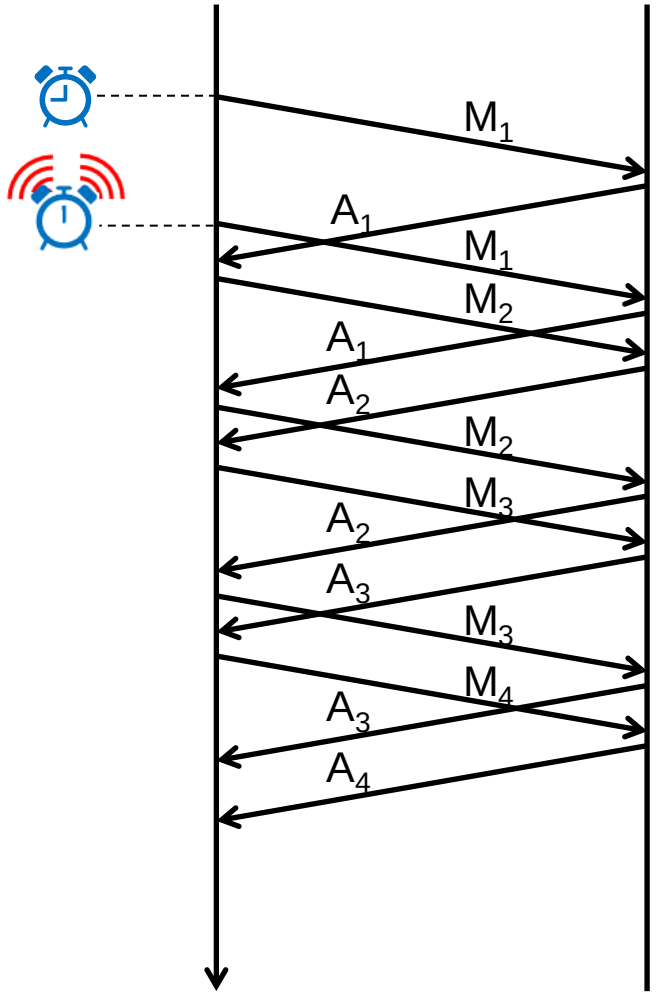


- By adding a **timer** we can handle losses (of segments or of acks)

```
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
```

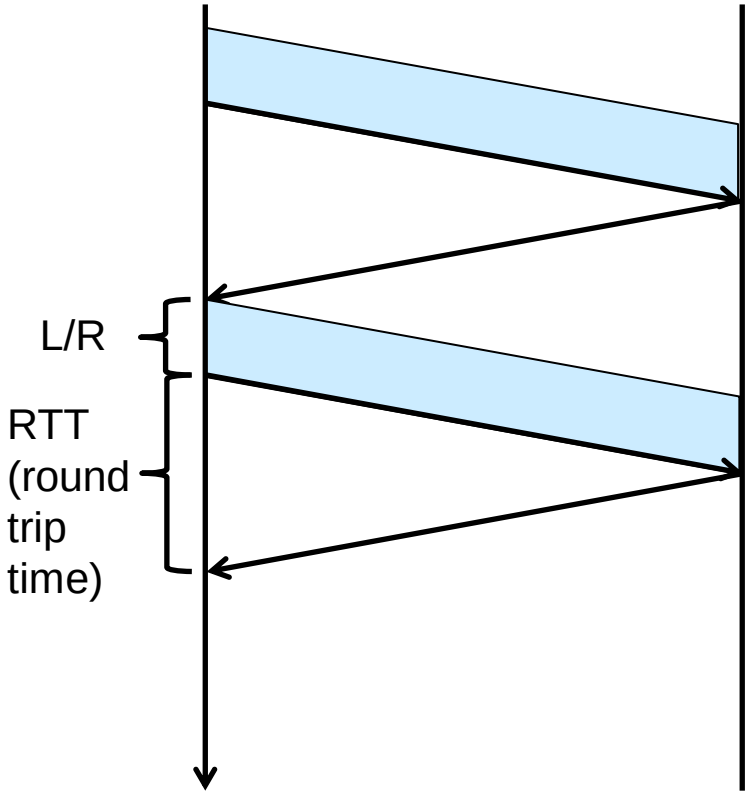
- Note: what is shown is the case when r≤s+T
  - In the case that r≥s+T the ack is ignored!
- This is the most sophisticated implementation of a **stop-and-wait** protocol

Reliable data transfer



- Setting the timer's deadline is a complex problem
- A **too 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
- A too long timer stops the transmission for long periods in the case of a loss

Reliable data transfer

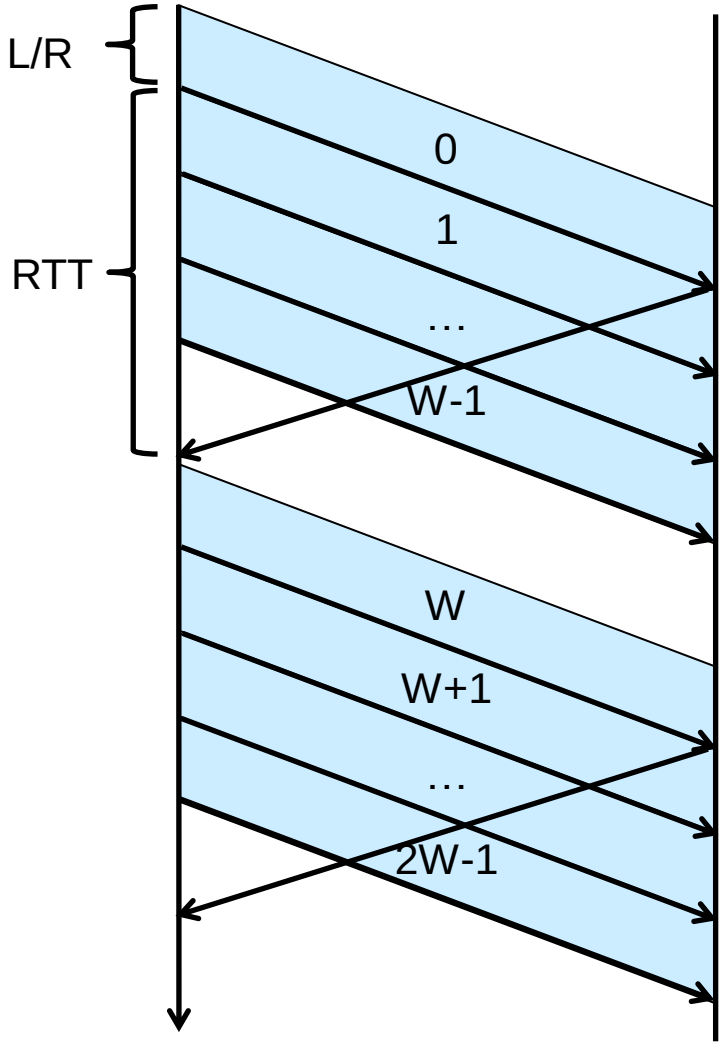


- Transmitting a segment at a time and waiting for its ack before a further transmission (stop-and-wait) significantly limits performance

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

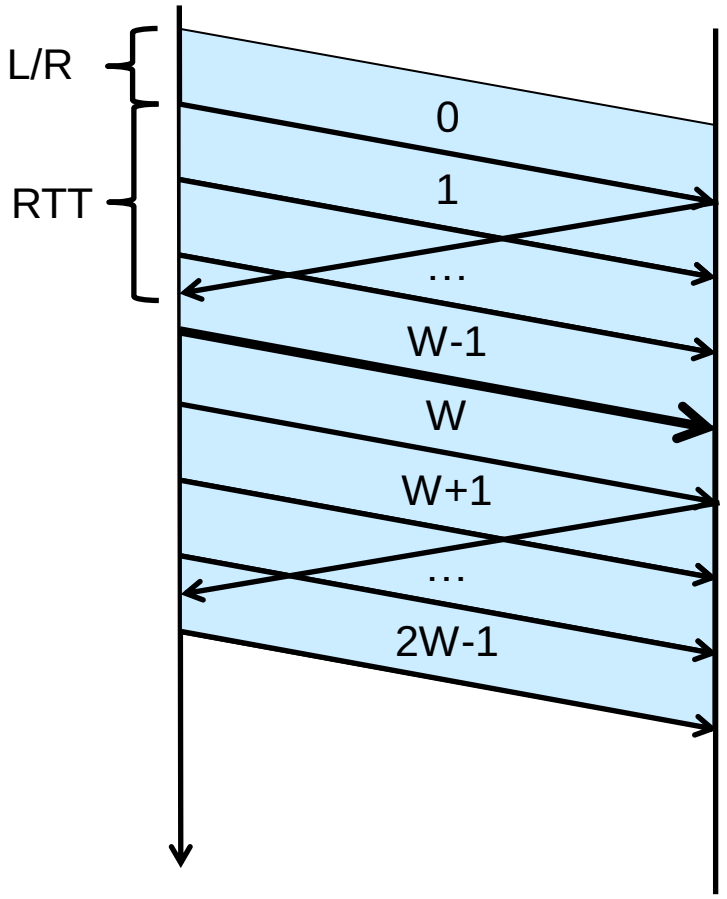
- Example: $RTT = 100\ ms$, $L = 1\ kbyte$, $R = 100\ Mbit/s \rightarrow U = 0{,}008$

Reliable data transfer



- **Sliding window protocols** can transit up to W segments while waiting the ack of the first one

Reliable data transfer



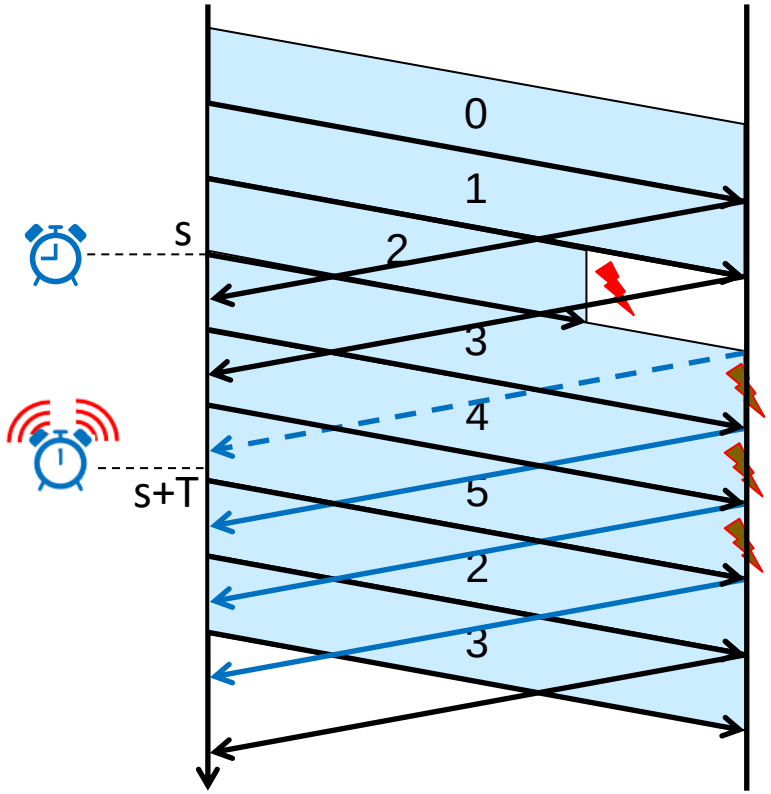
- The condition for a continuous transmission is that the window will not close before the arrival of the first ack

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

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

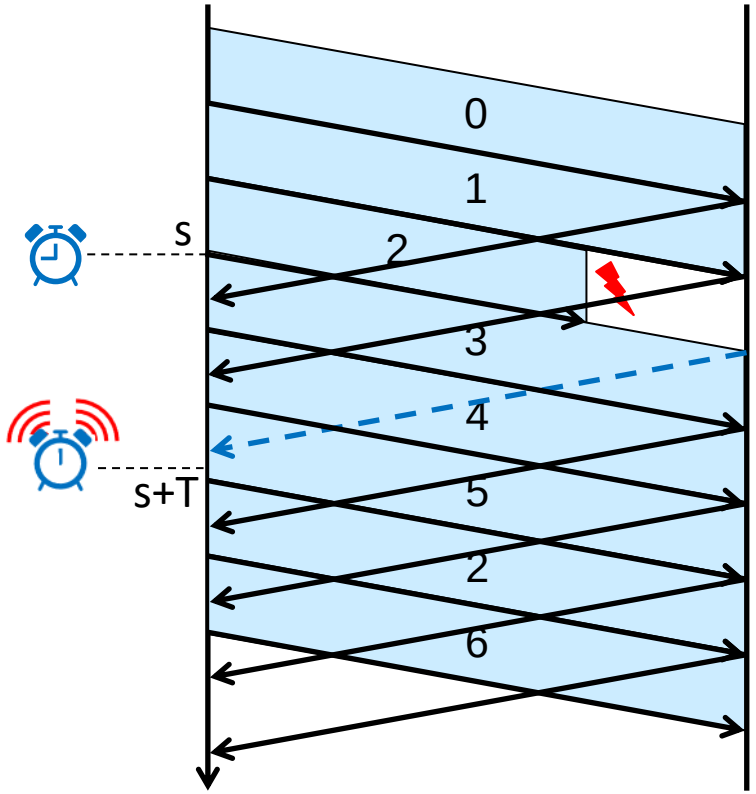
- In the example of two slides ago: $W \geq 1251$

Reliable data transfer



- What is retransmitted when a segment is lost?
- First possibility: **Go-Back-N** protocol
  - Transmit and slide the window after any ack reception
  - If s+T is reached or repeated ack is received, retransmit all the segments from the last acknowledged segment
- There is no need of buffering at the receiver
  - Duplicate segments received after a loss are simply discarded
  - No further duplications once the segment in sequence is retransmitted
- Acks are cumulative
  - It is possible to recover from multiple ack losses

Reliable data transfer



- Second possibility: **Selective Repeat** protocol
  - Acks are individual
  - 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

Reliable data transfer

- Observations:
  - There are not only two possibilities (i.e., pure Go-Back-N and pure Selective Repeat): real protocols as TCP rely on hybrid solutions
  - Sequence numbers of segments are represented with a finite number of bits
    - Some rules are defined to bind the maximum sequence number to the window's size, so that confusion is avoided when sequence numbers start back from zero:
      - Go-Back-N: $N \geq W + 1$
      - Selective Repeat: $N \geq 2W$