

# Chapter 3: Transport layer

- How can two entities reliably communicate over a channel in which messages can be corrupted or lost?
- How can two distributed entities synchronize and share state?
- How can a collection of different entities adjust their communication rates to prevent network congestion and resource exhaustion?
- The two main transport protocols UDP and TCP.

# 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

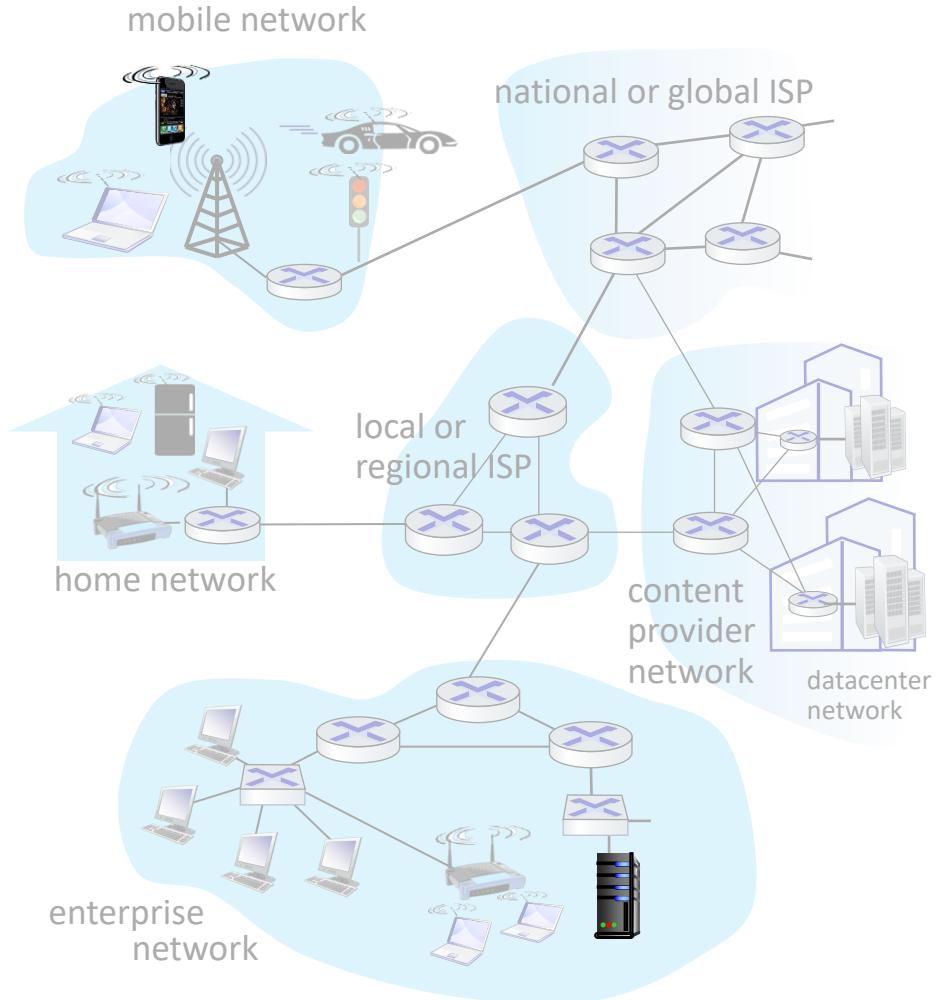
# Transport layer: roadmap

- 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



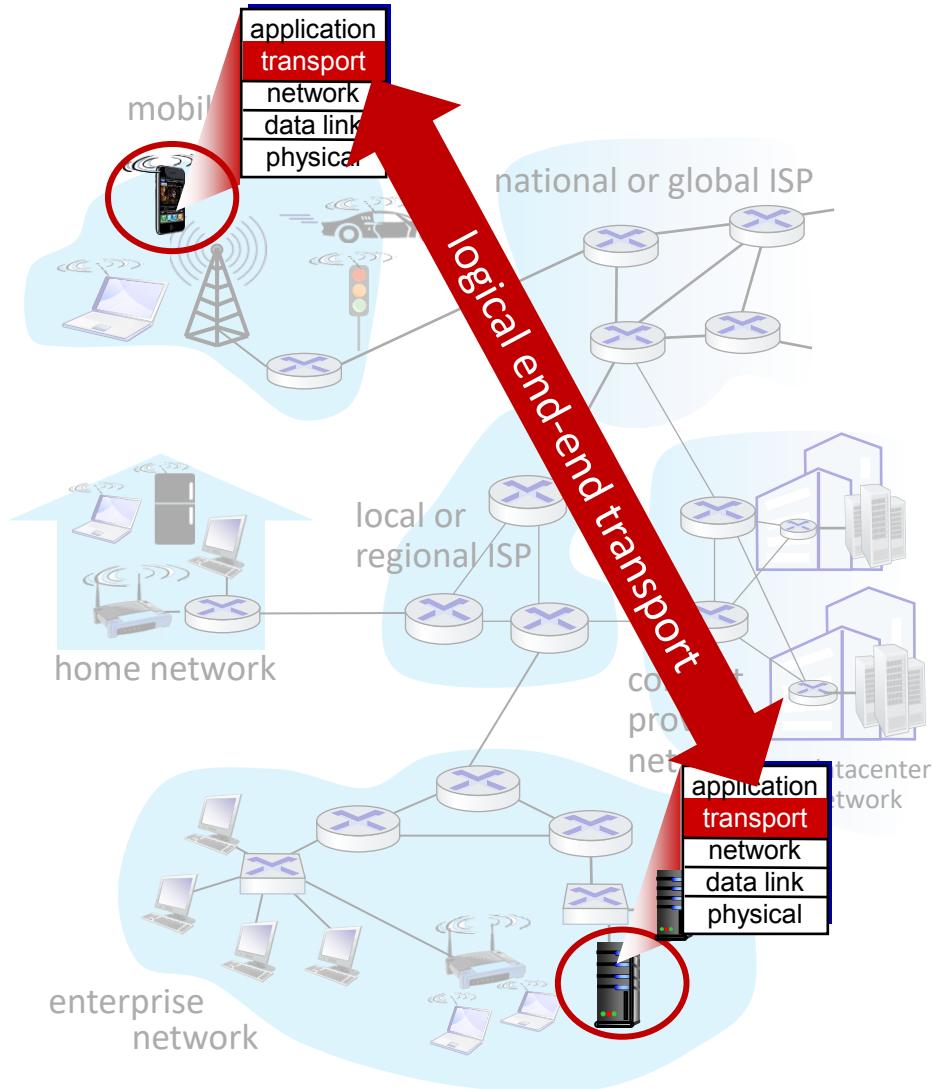
# Transport services and protocols

- 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

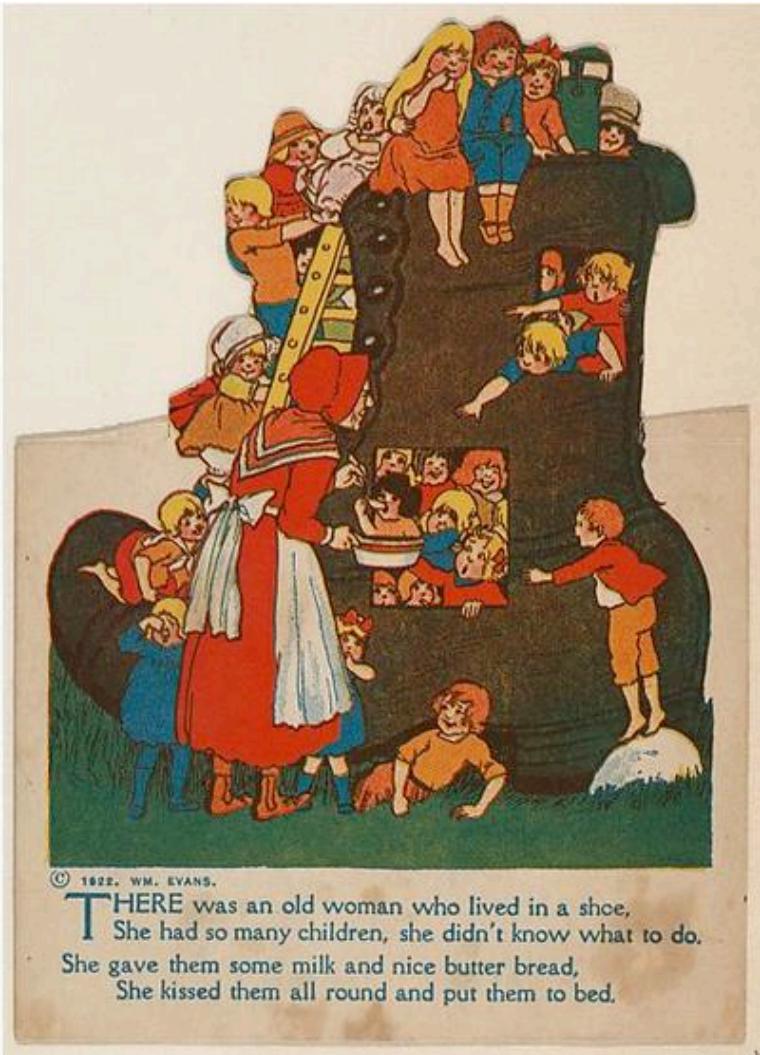


# Transport services and protocols

- 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



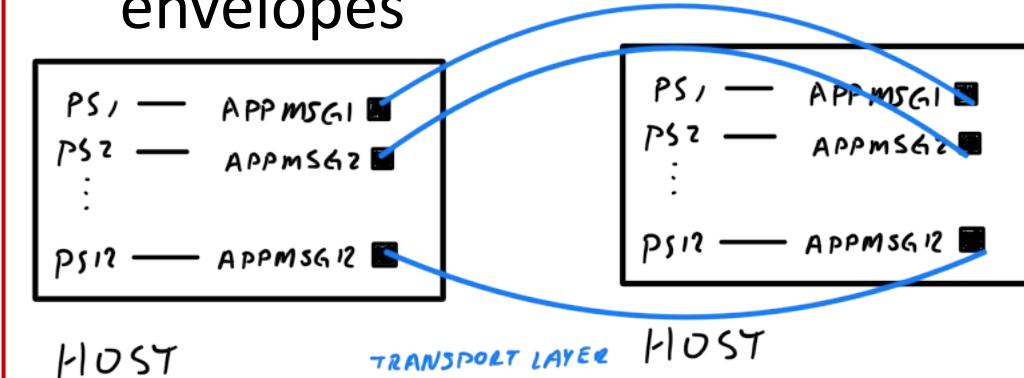
# Transport vs. network layer services and protocols



*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

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

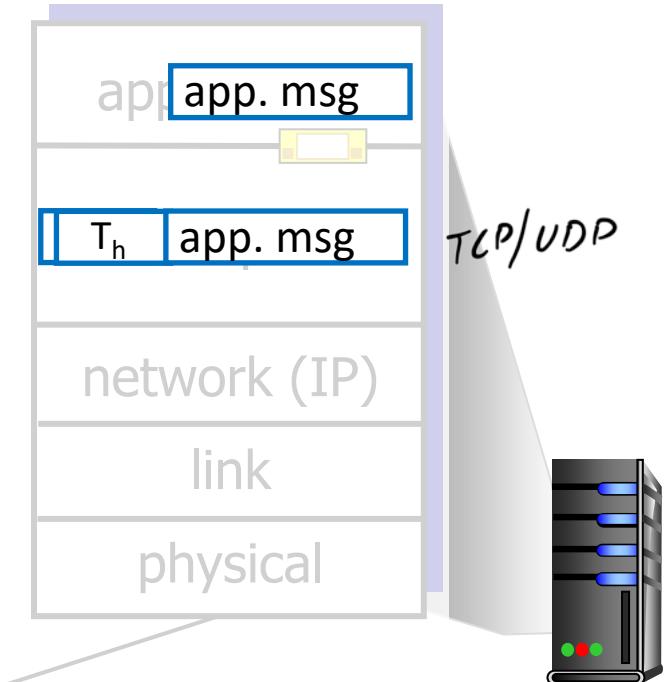
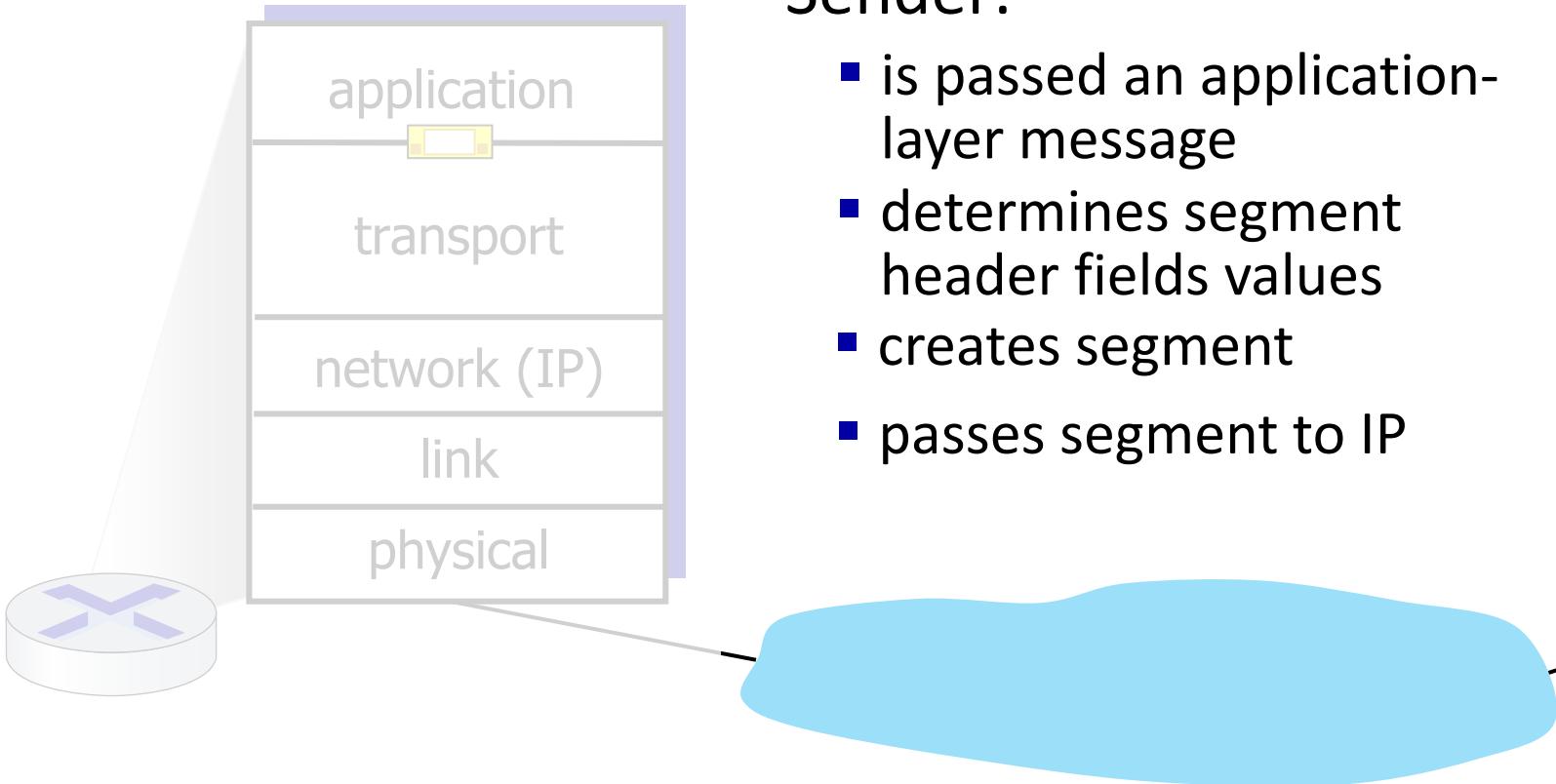
*household analogy:*

- 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:

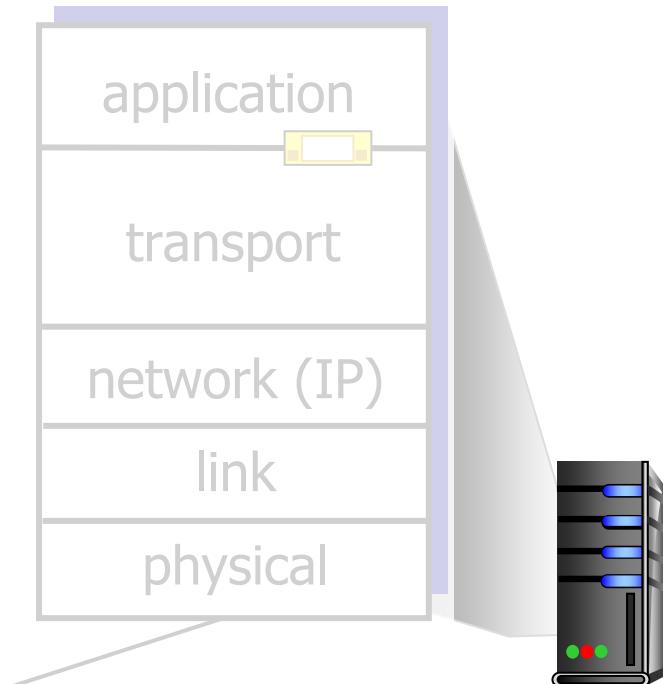
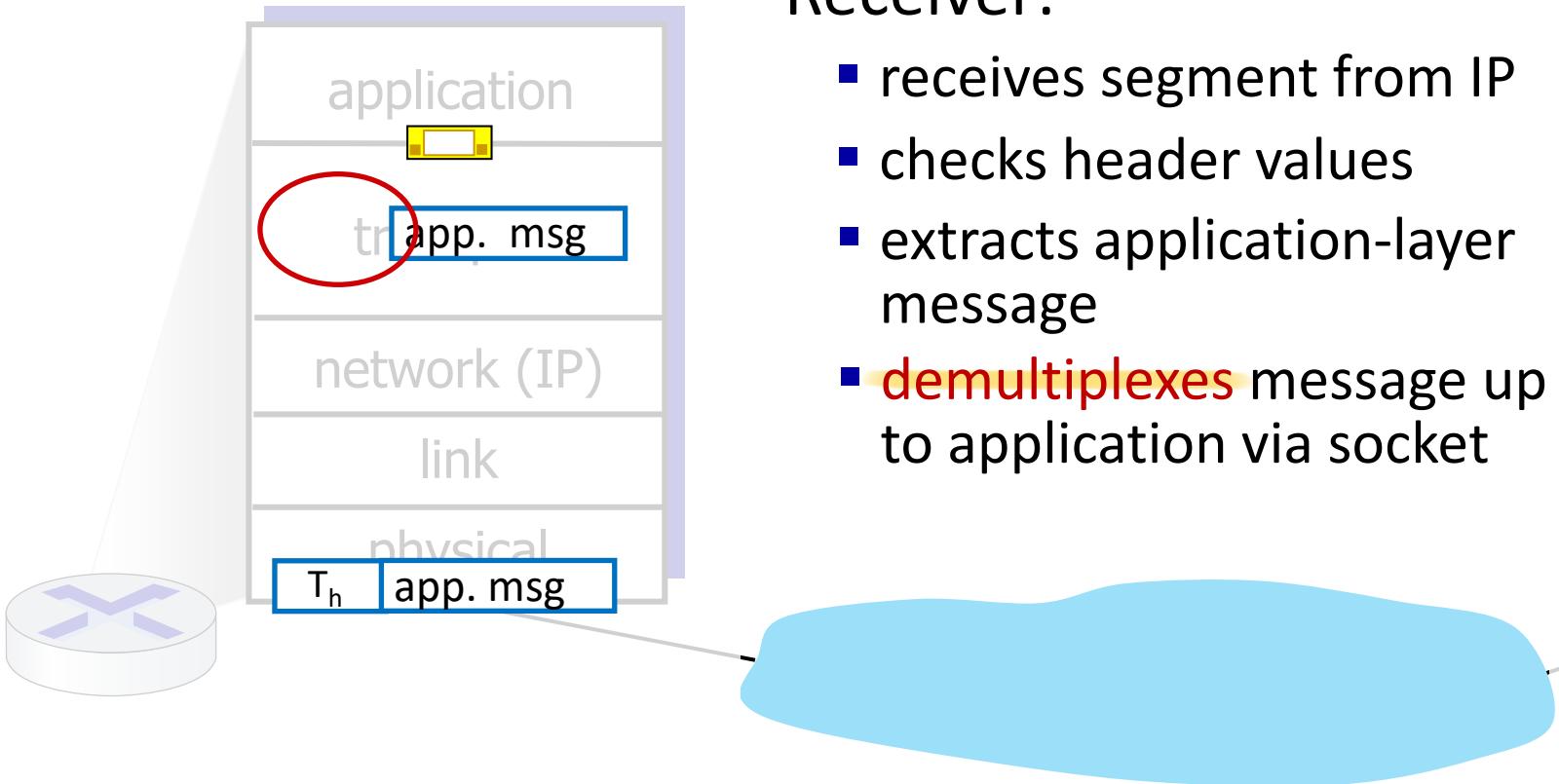
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP



# Transport Layer Actions

## Receiver:

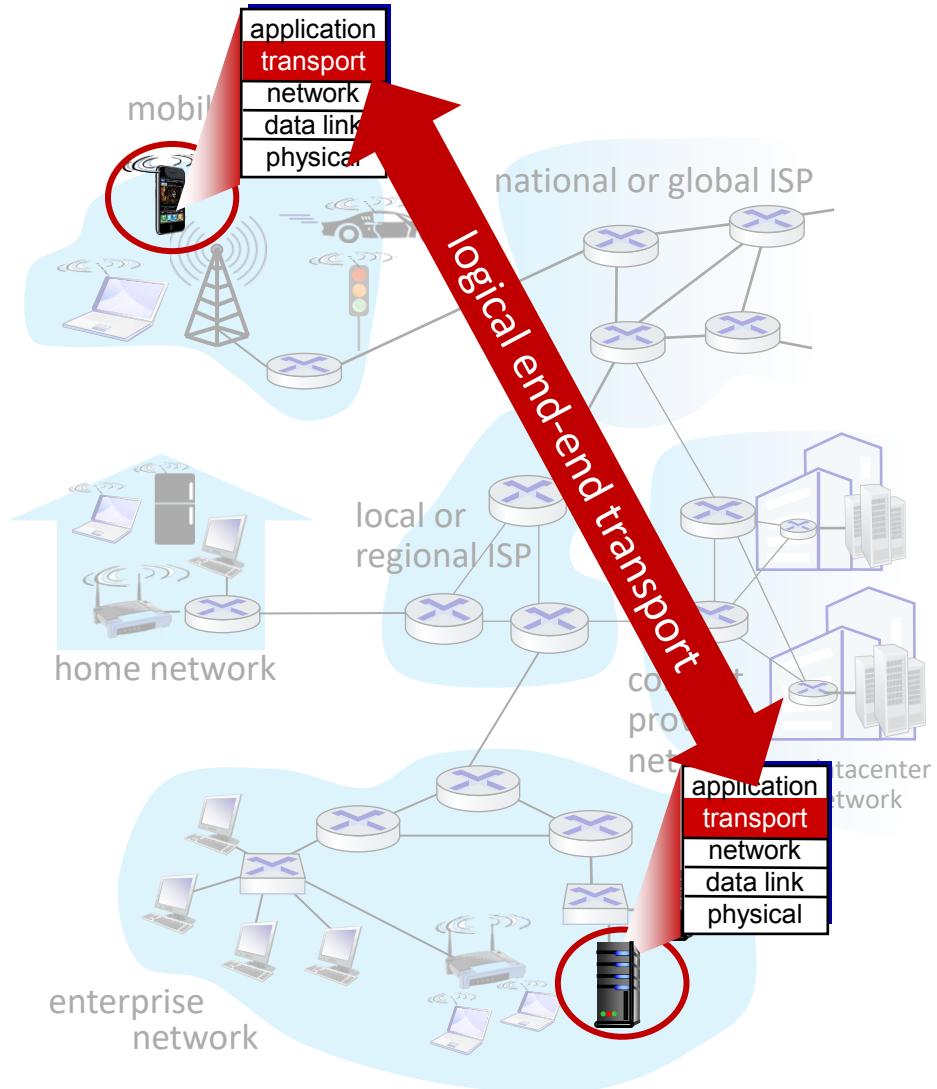
- receives segment from IP
- checks header values
- extracts application-layer message
- **demultiplexes** message up to application via socket



# Two principal Internet transport protocols

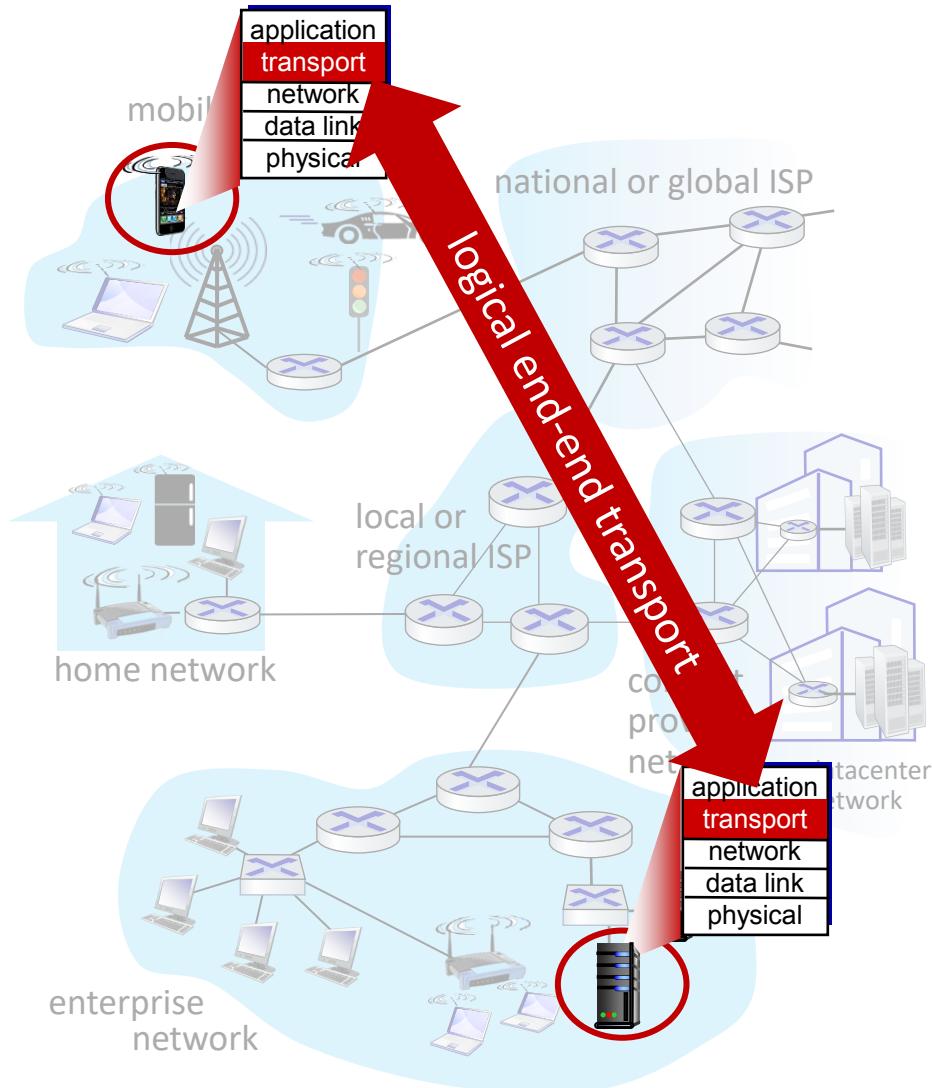
- **TCP:** Transmission Control Protocol

- reliable, in-order delivery
- congestion control
- flow control
- connection setup



# Two principal Internet transport protocols

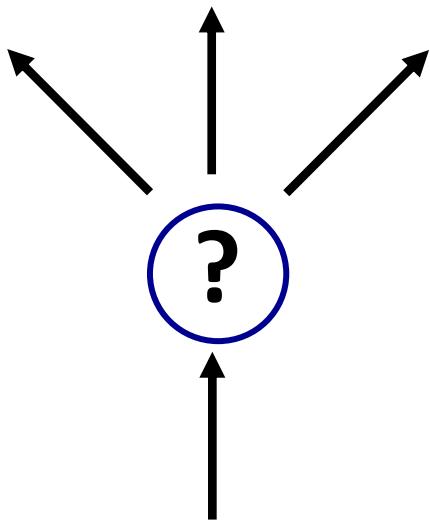
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services *not* available:
  - delay guarantees
  - bandwidth guarantees



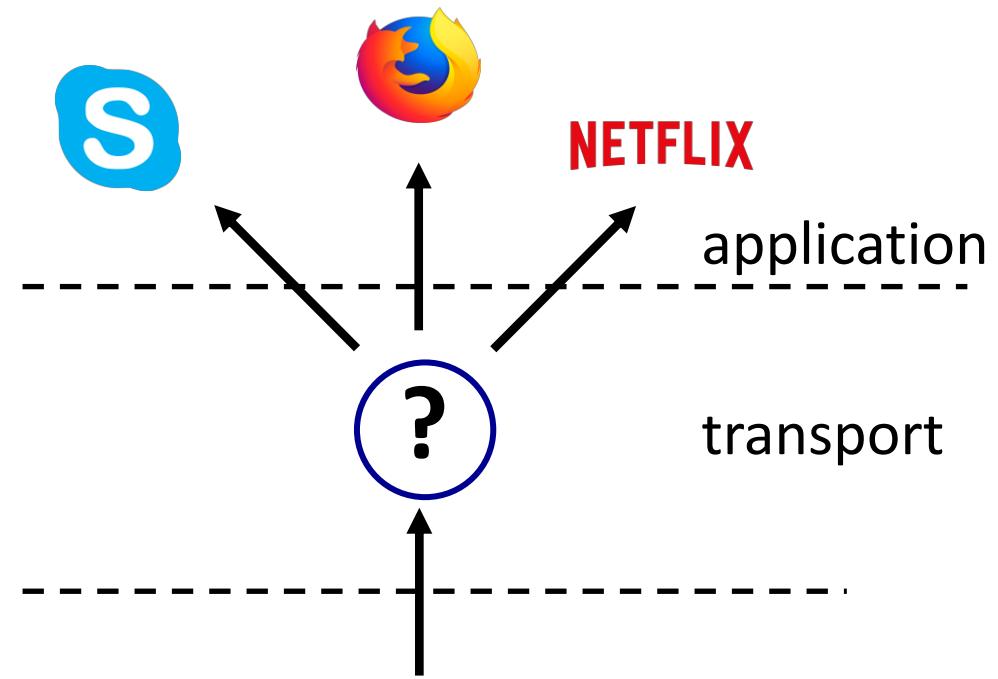
# Chapter 3: roadmap

- 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

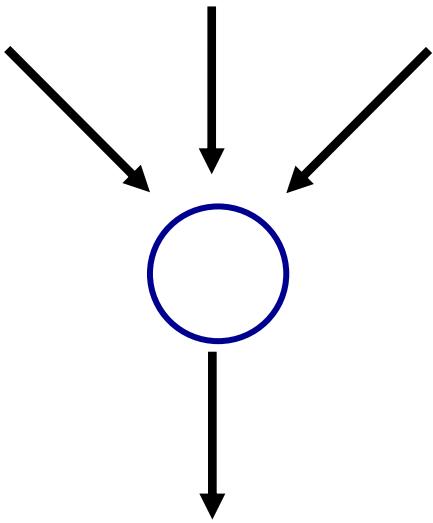




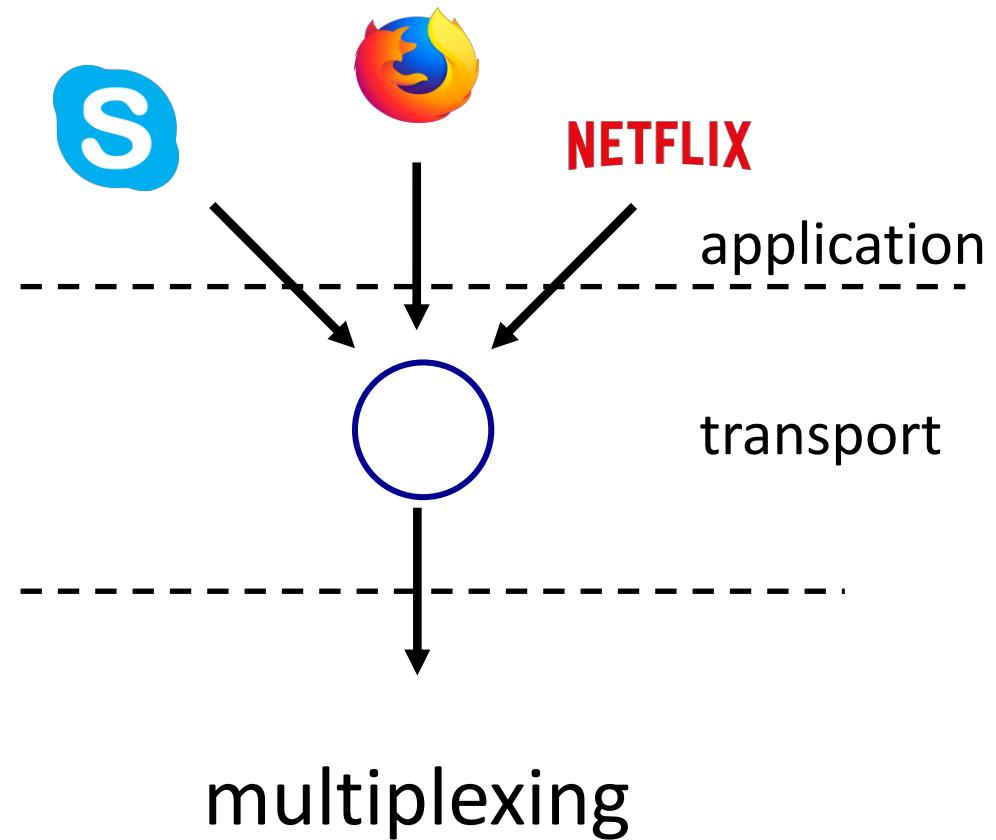
de-multiplexing

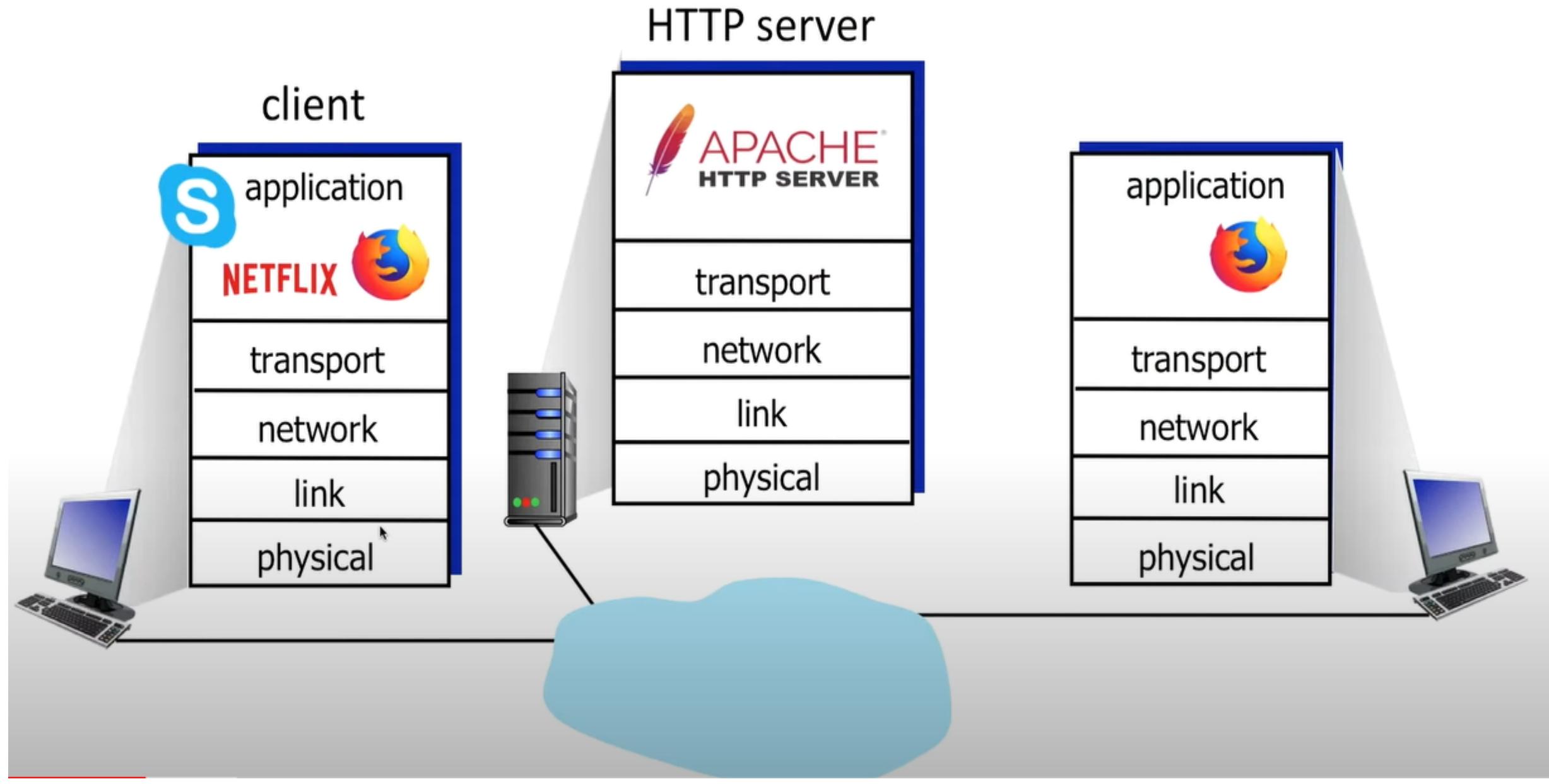


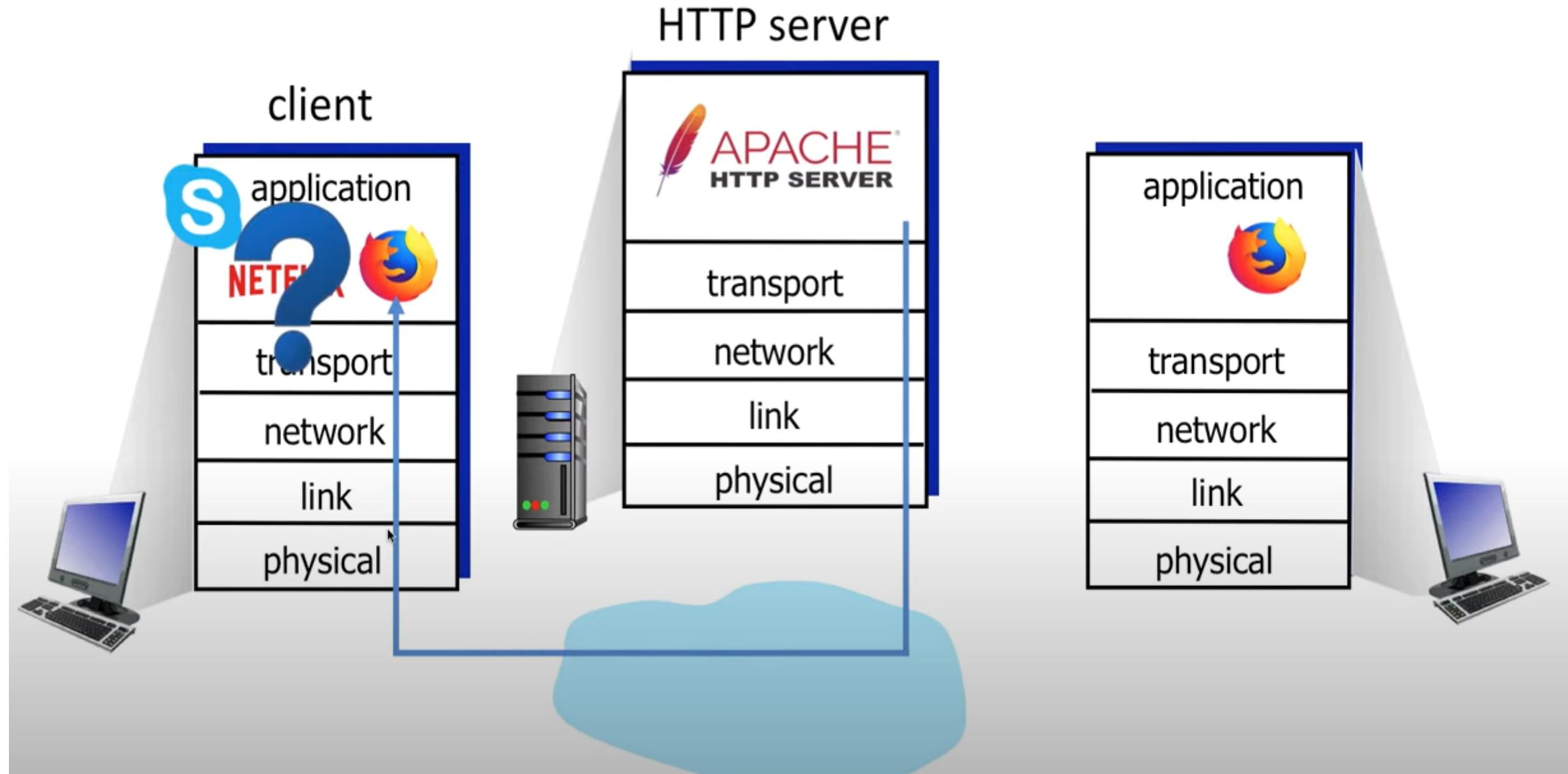
de-multiplexing

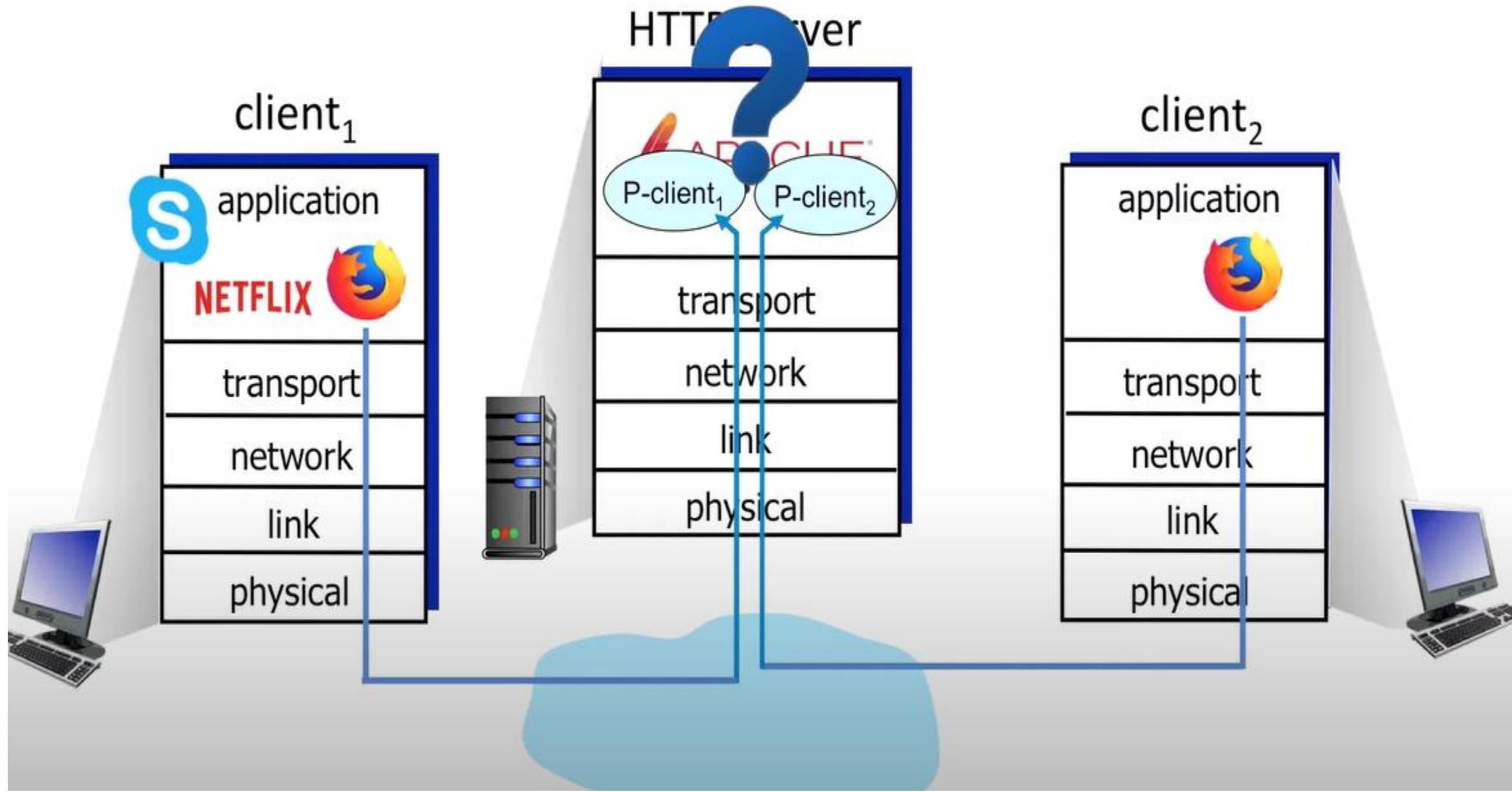


multiplexing











# Multiplexing



# Demultiplexing

AIRFRANCE /

ECONOMY /



SKYTEAM

AIRFRANCE /

SKY  
PRIORITY™



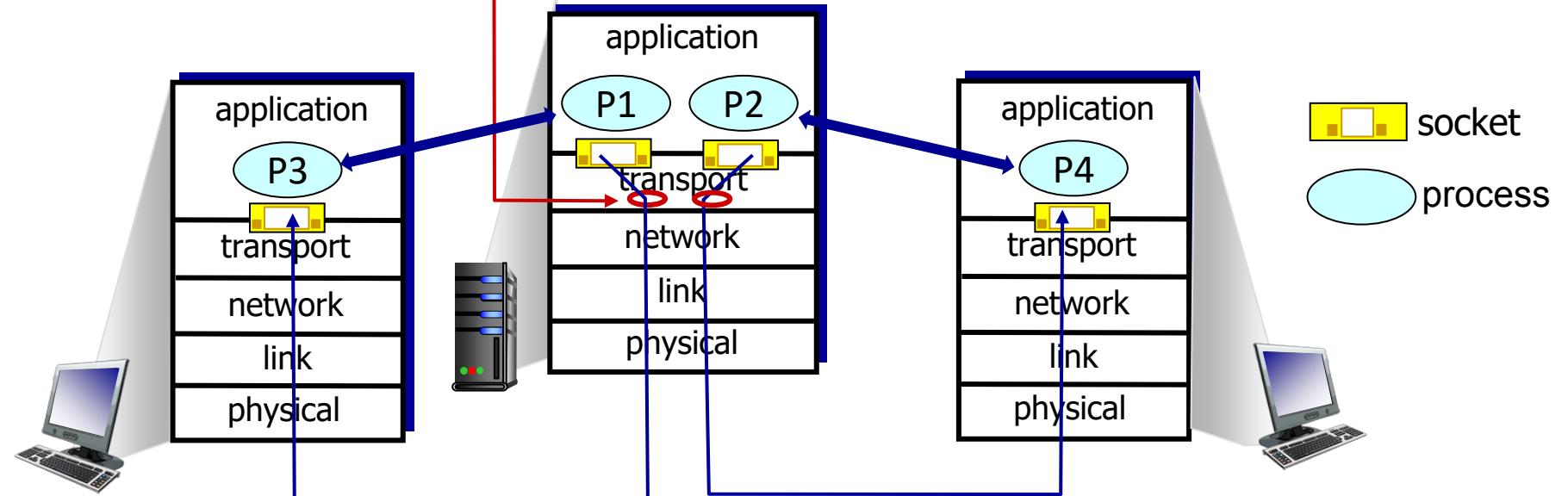
SKYTEAM



# Multiplexing/demultiplexing

*multiplexing as sender:*

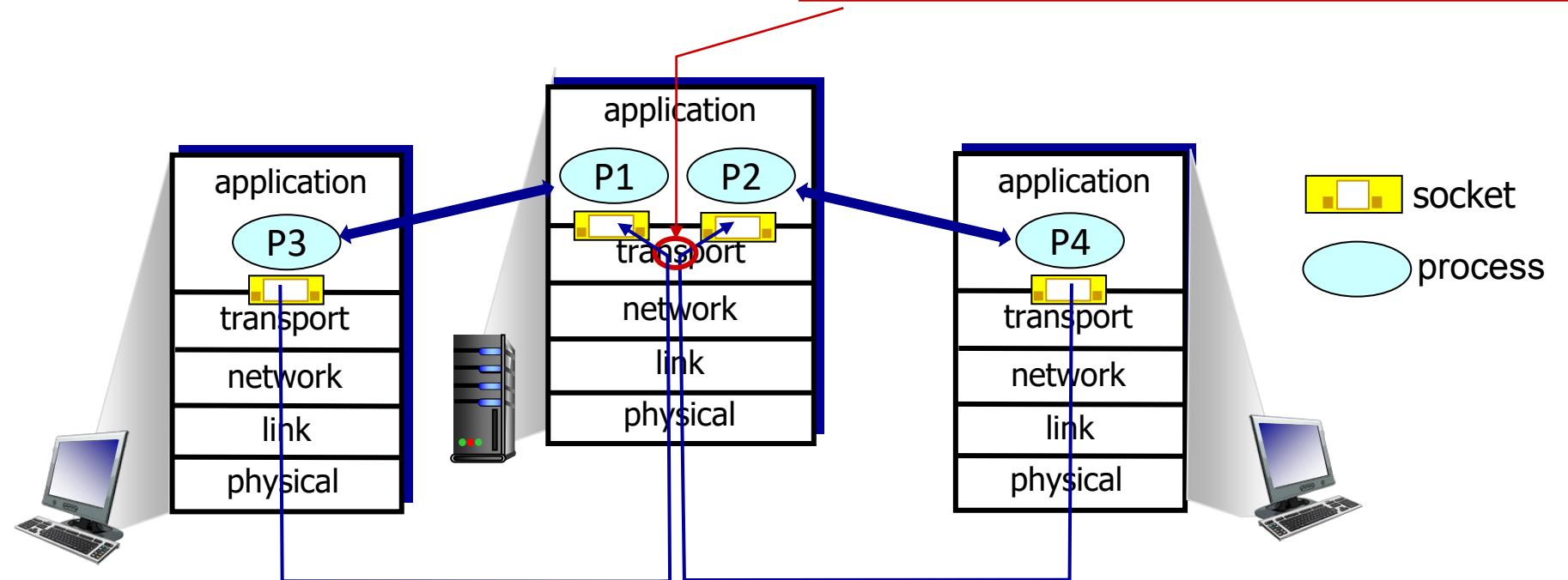
handle data from multiple  
sockets, add transport header  
(later used for demultiplexing)



# Multiplexing/demultiplexing

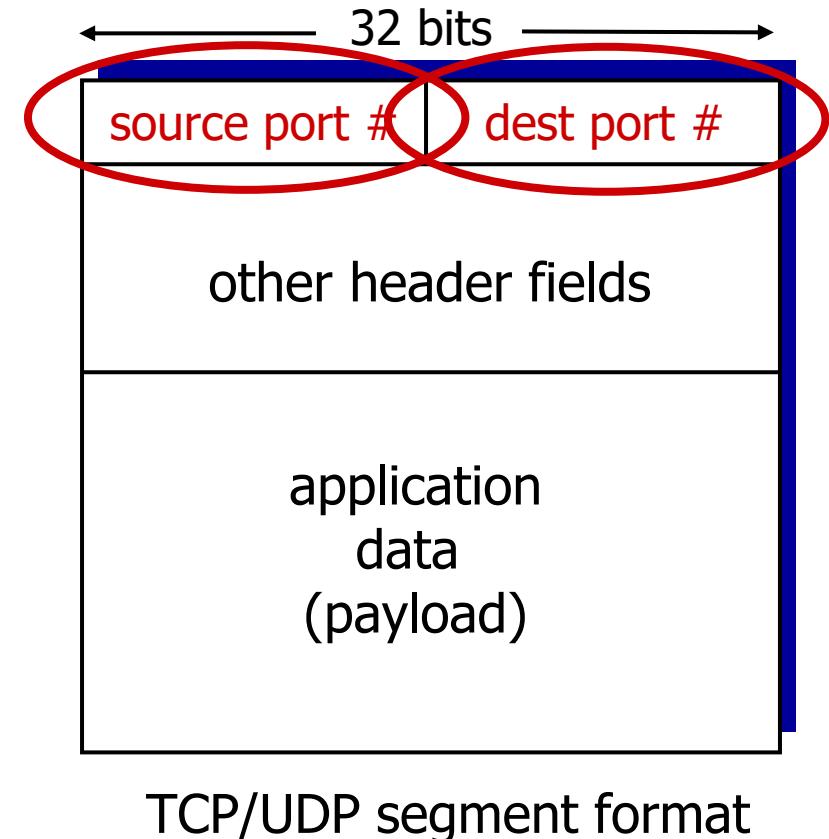
*demultiplexing as receiver:*

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(1234);
```

- 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 #

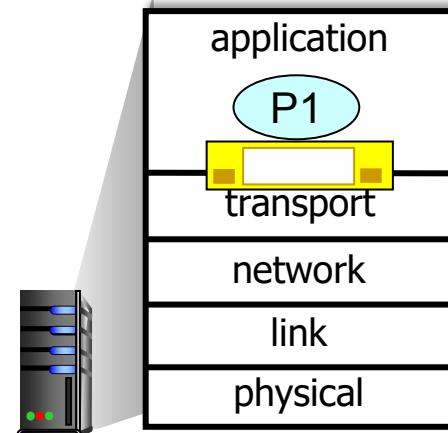
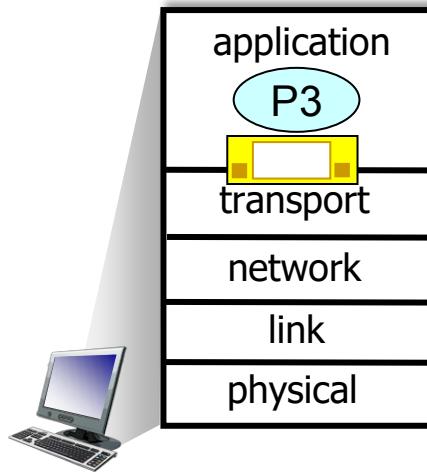


IP/UDP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

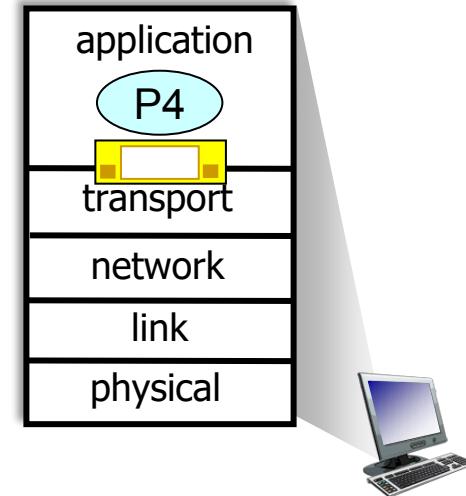
# Connectionless demultiplexing: an example

```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 6428);
```

```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 9157);
```



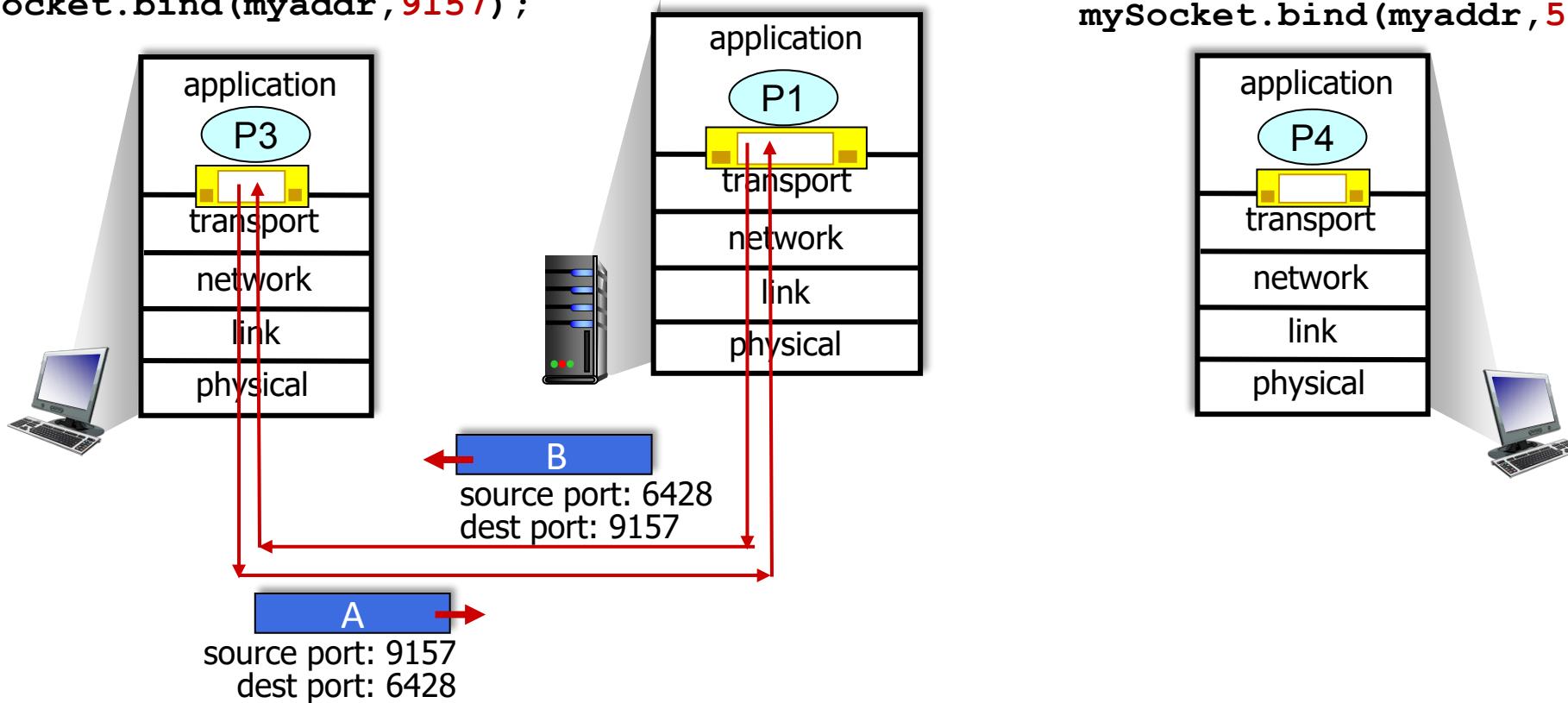
```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 5775);
```



# Connectionless demultiplexing: an example

```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 6428);
```

```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 9157);
```

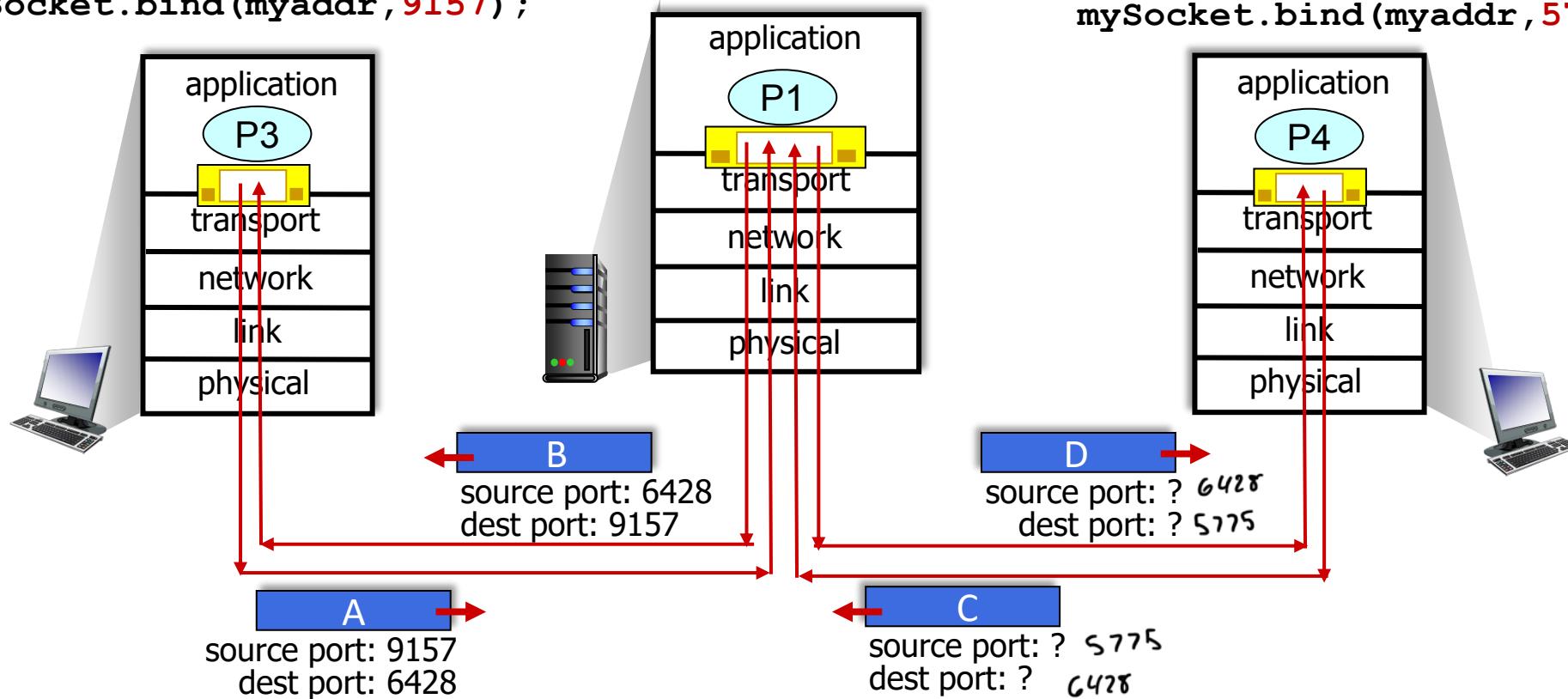


# Connectionless demultiplexing: an example

```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 6428);
```

```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 9157);
```

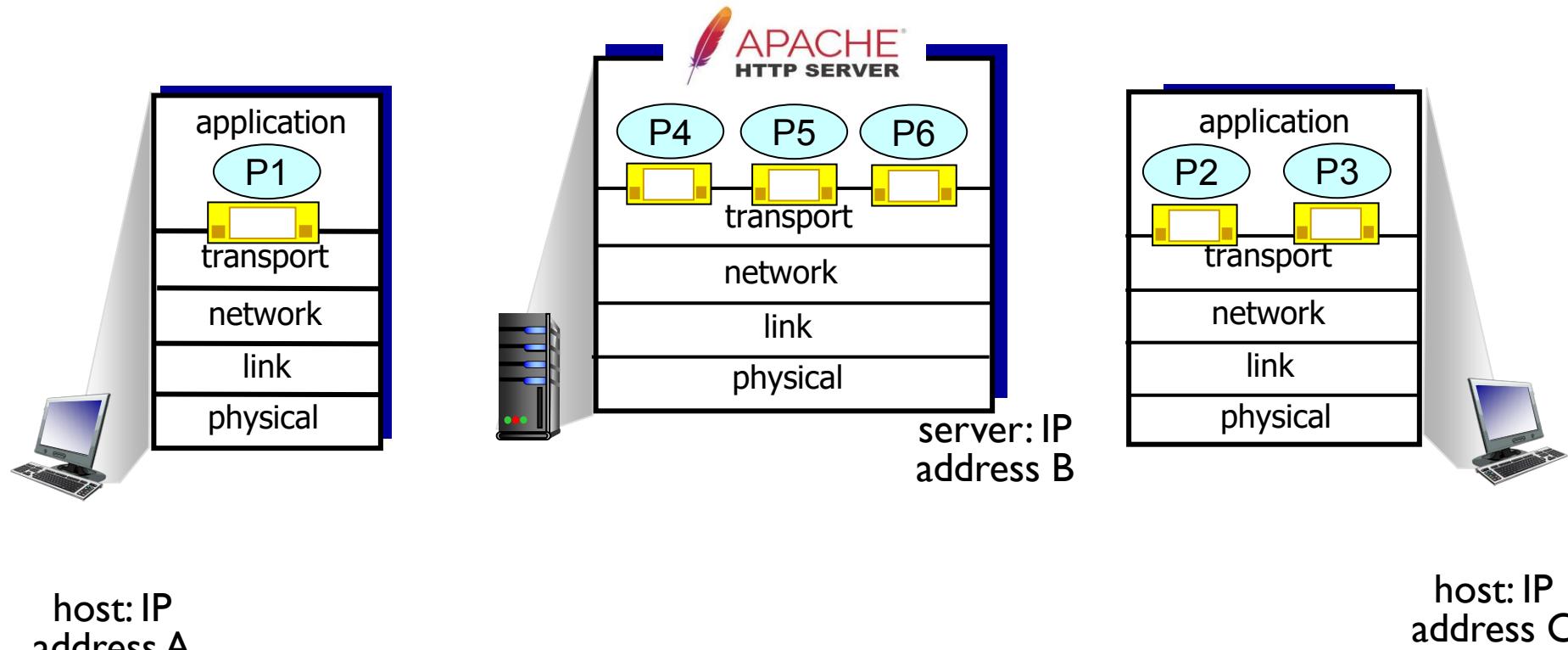
```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 5775);
```



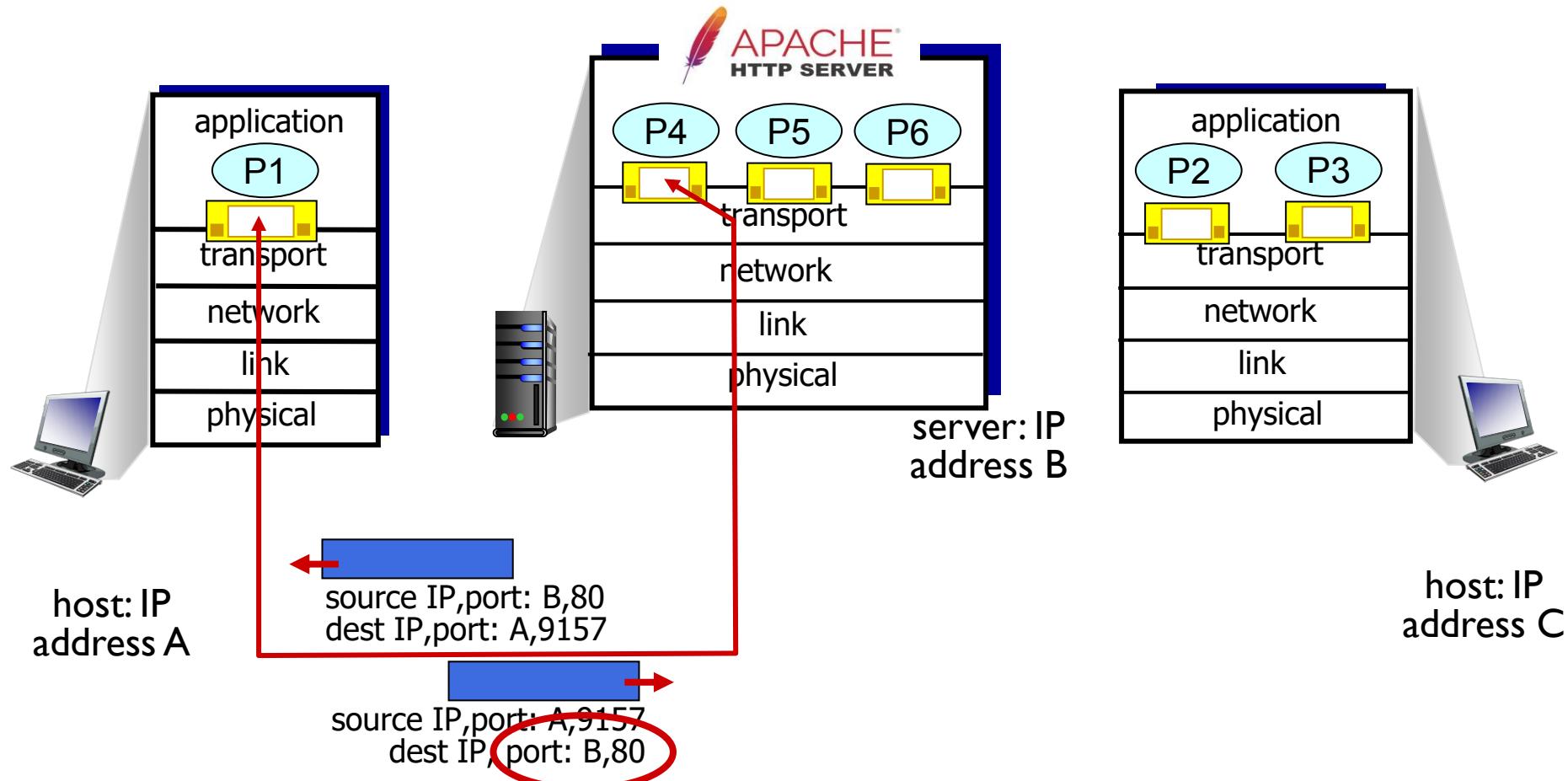
# Connection-oriented demultiplexing

- 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

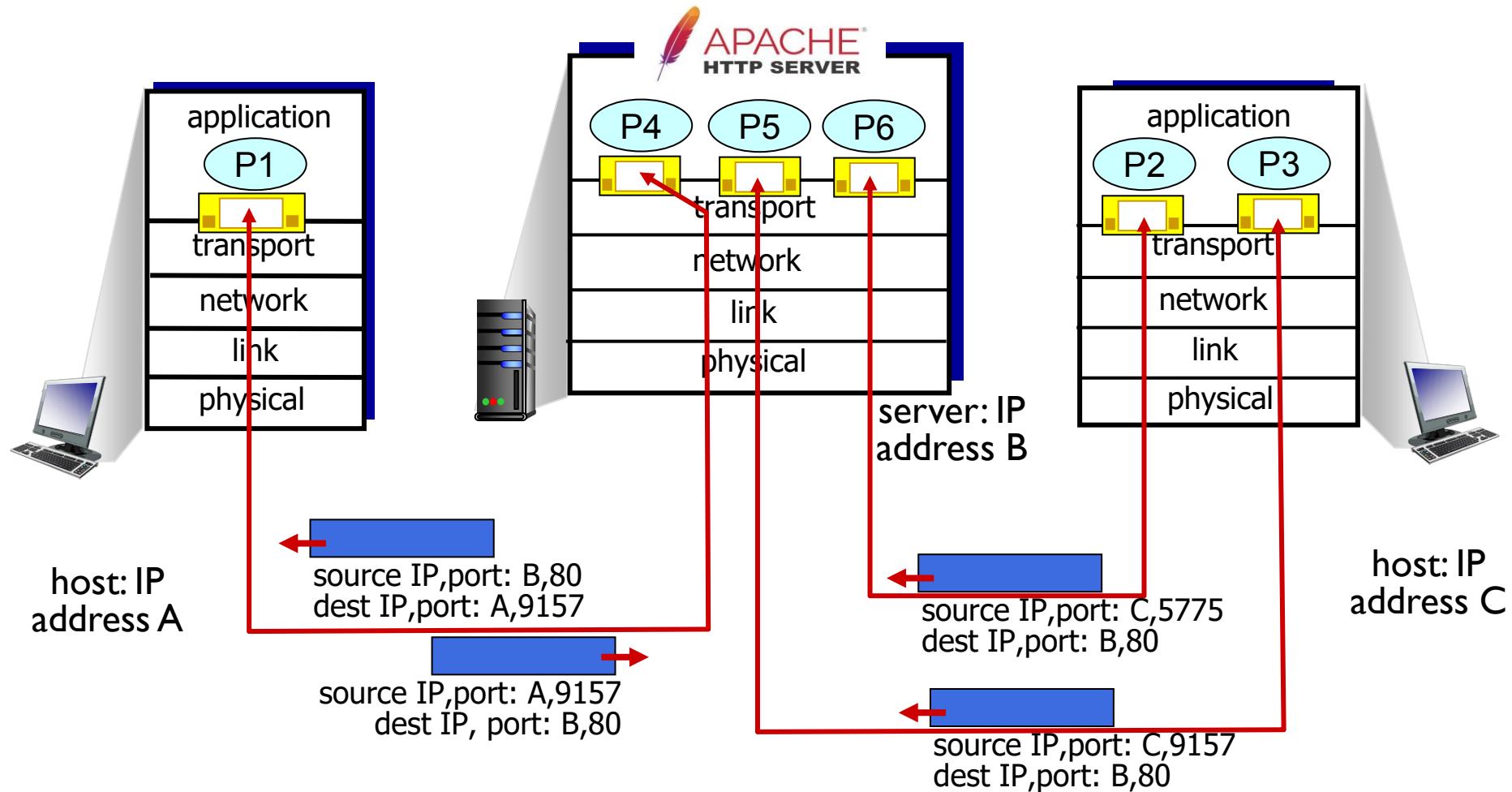
# Connection-oriented demultiplexing: example



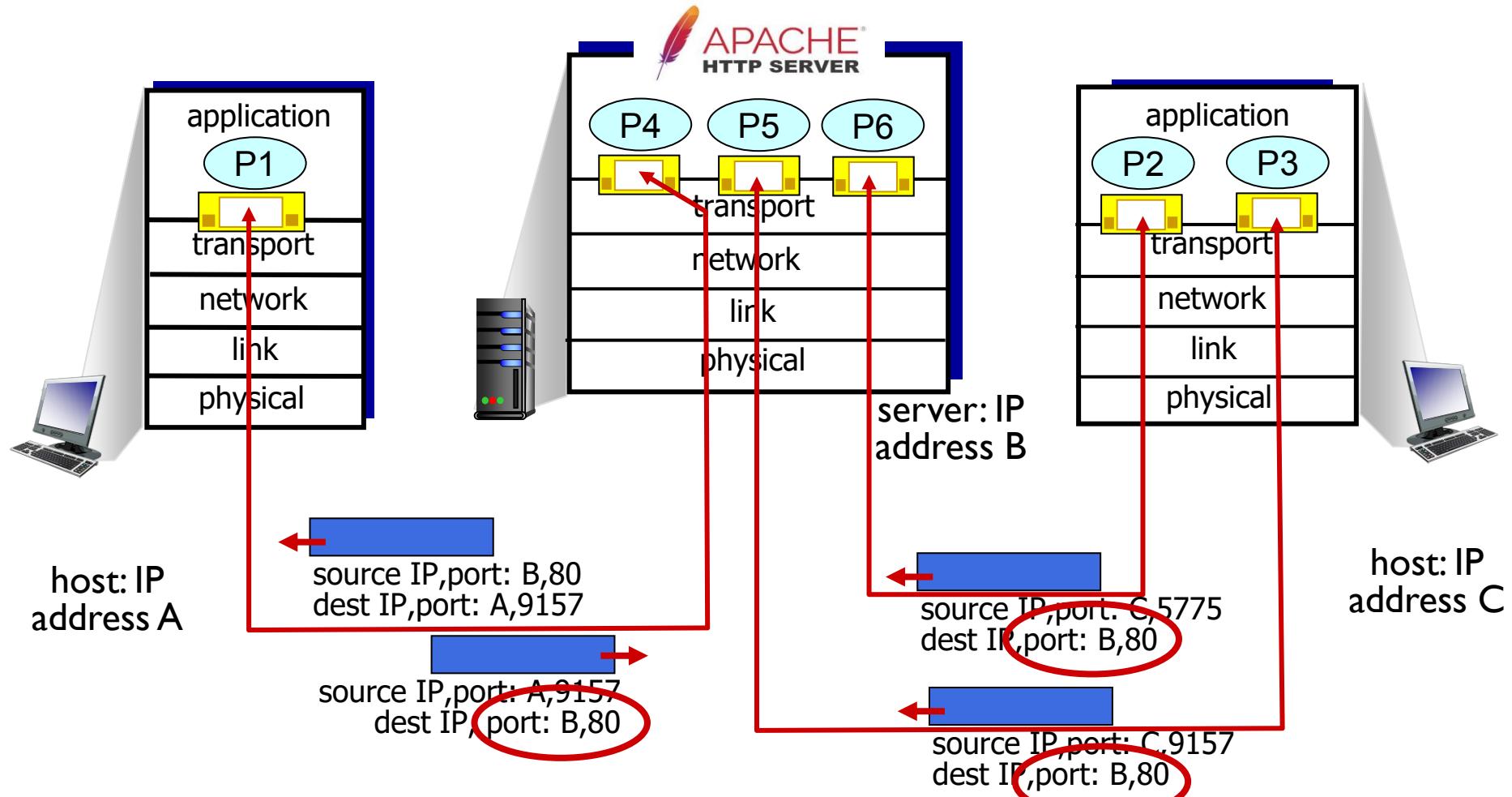
# Connection-oriented demultiplexing: example



# Connection-oriented demultiplexing: example



# Connection-oriented demultiplexing: example



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
- 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

- UDP sender/receiver actions
- UDP segment format
- Internet checksum

# 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

# UDP: User Datagram Protocol

## 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

# UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD  
RFC 768 J. Postel  
ISI  
28 August 1980

## User Datagram Protocol

---

### Introduction

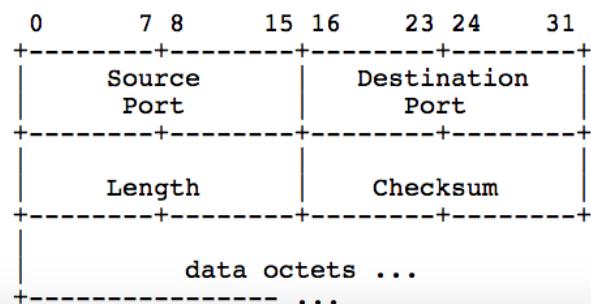
---

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.

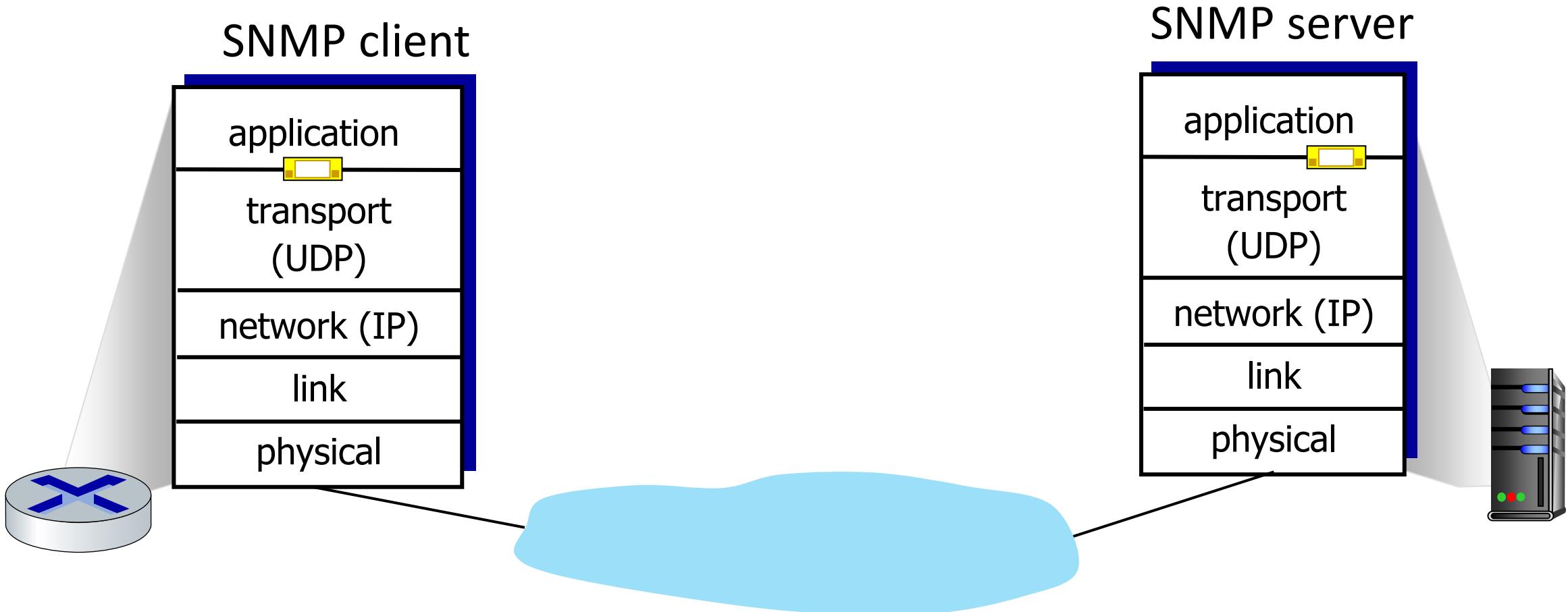
This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

### Format

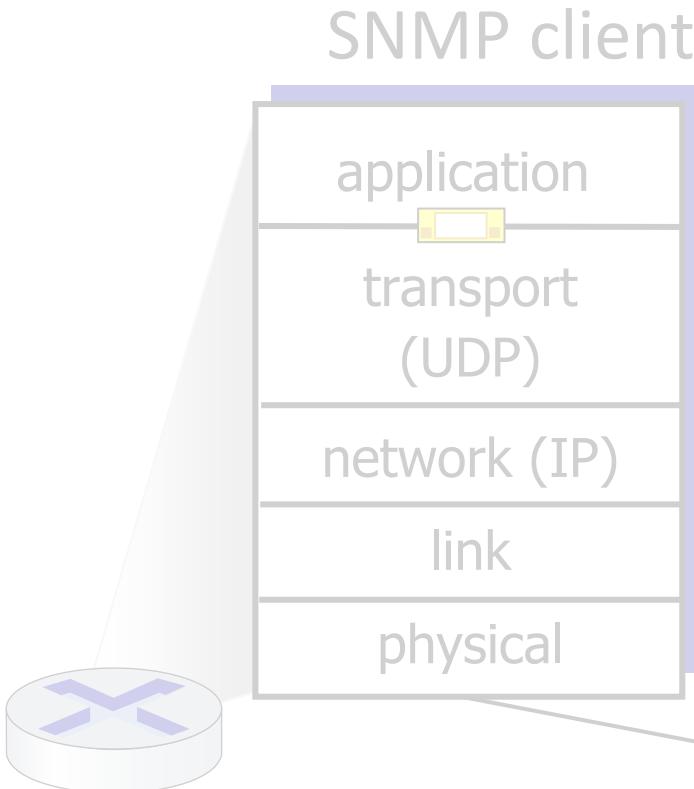
---



# UDP: Transport Layer Actions



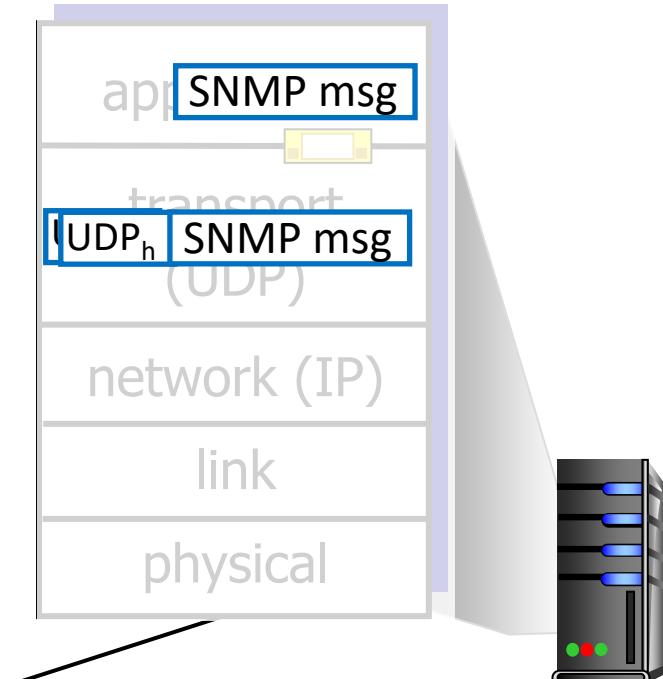
# UDP: Transport Layer Actions



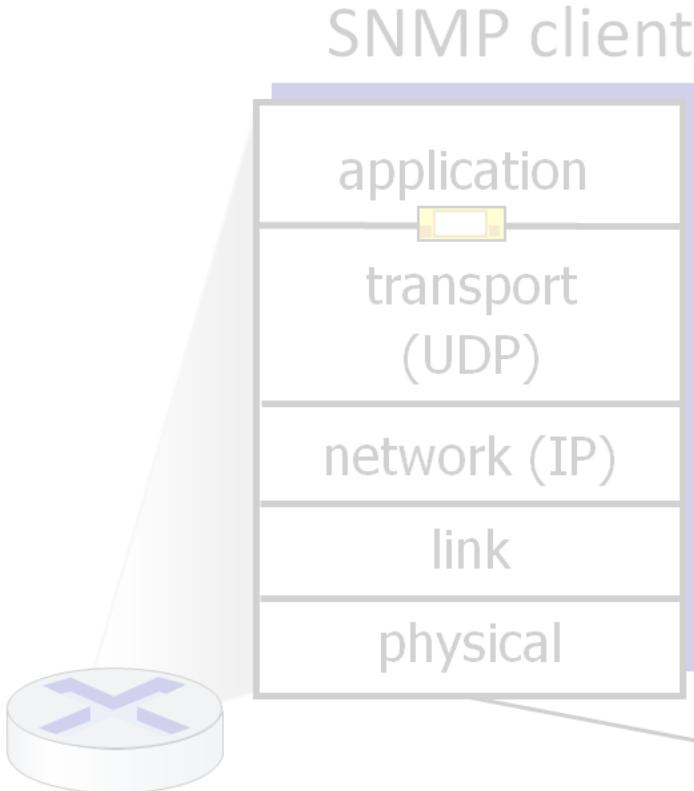
## UDP sender actions:

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

## SNMP server



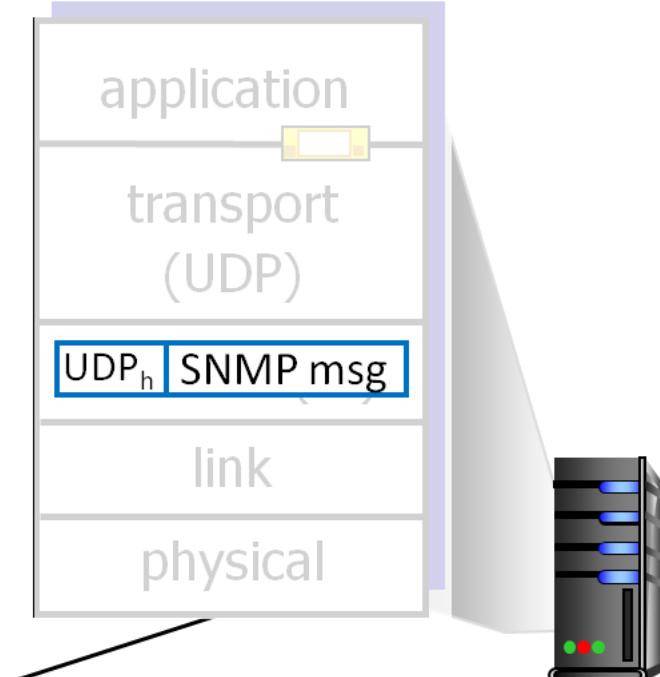
# UDP: Transport Layer Actions



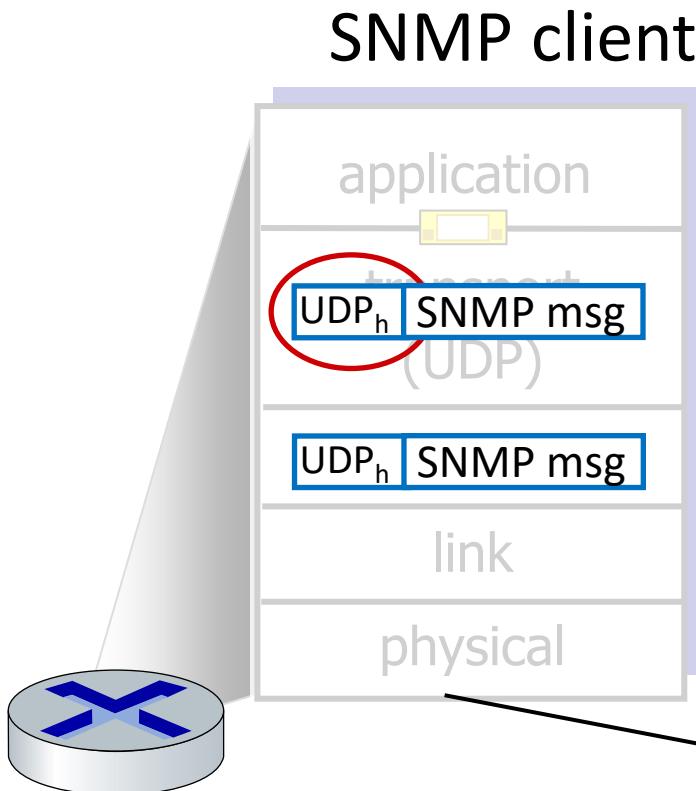
## UDP sender actions:

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

## SNMP server



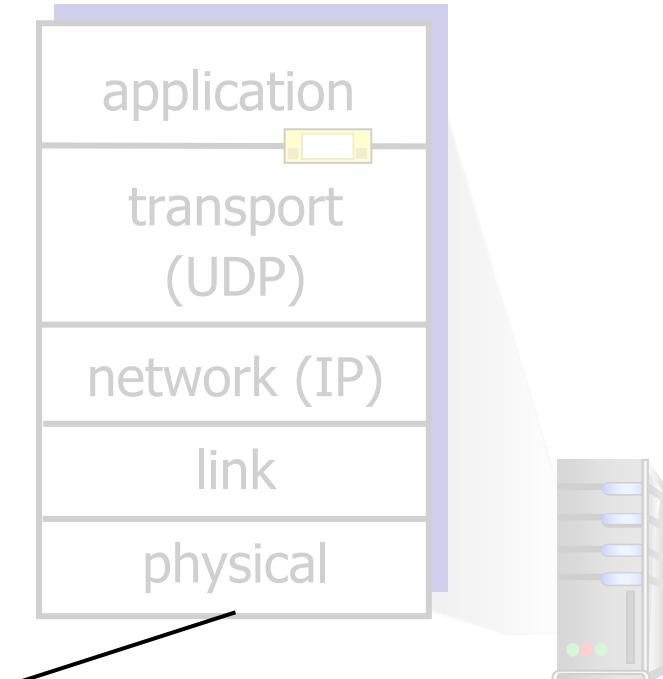
# UDP: Transport Layer Actions



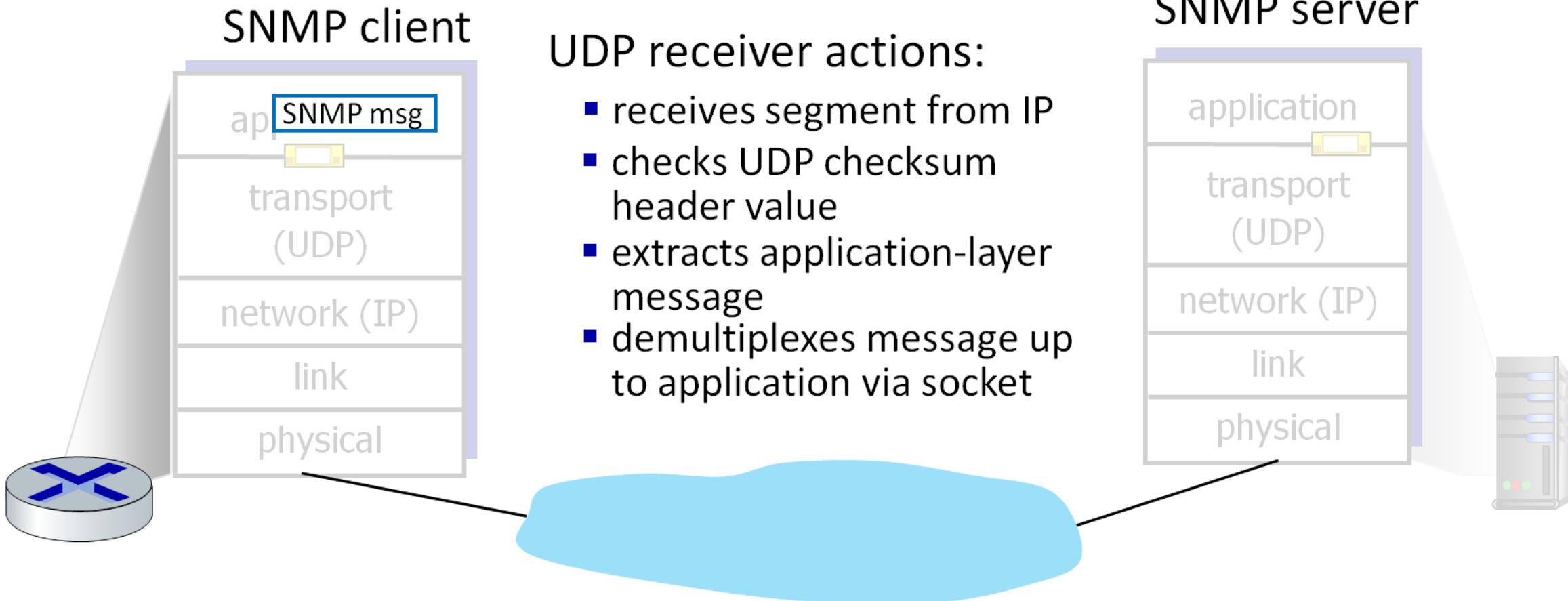
## UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message

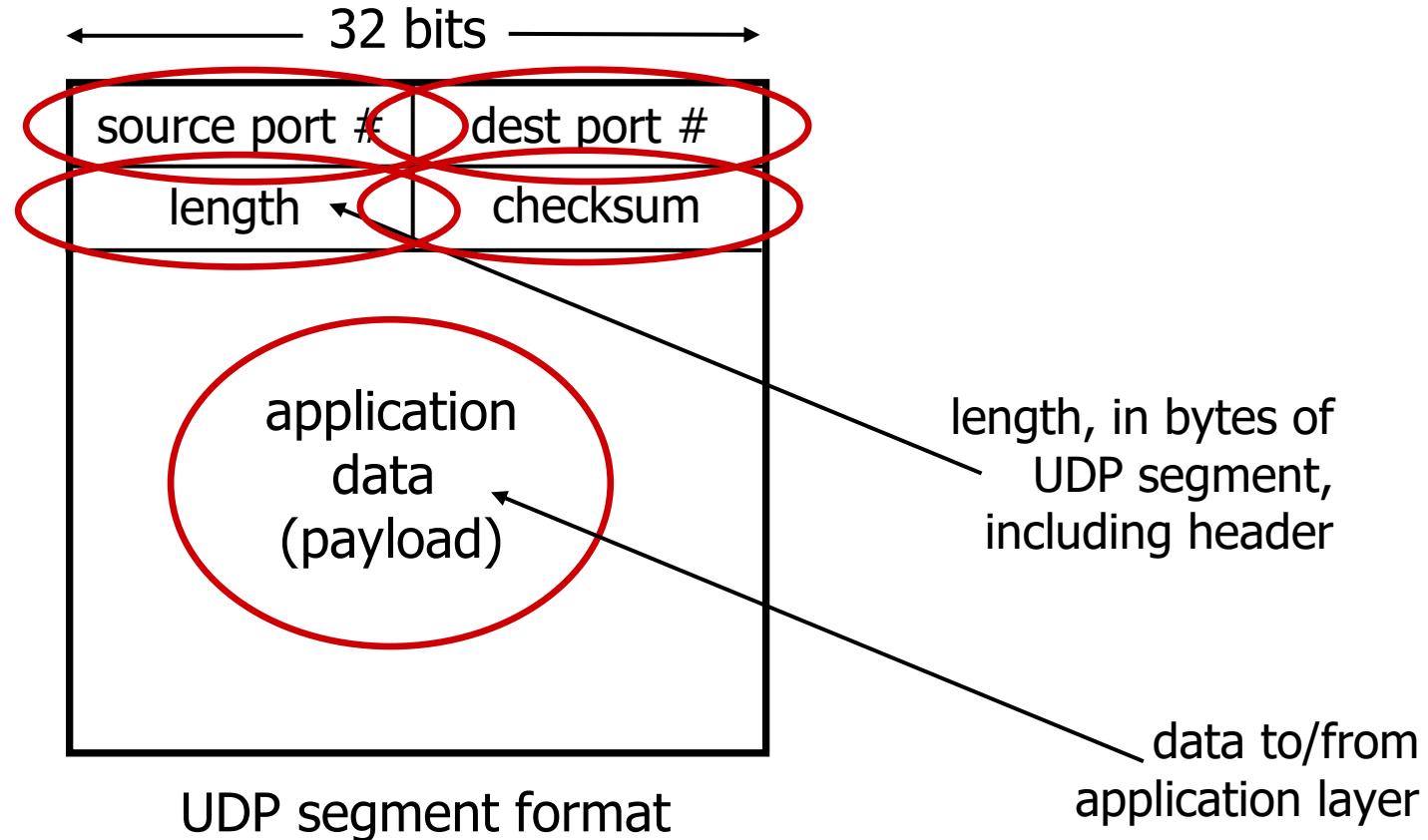
## SNMP server



# UDP: Transport Layer Actions

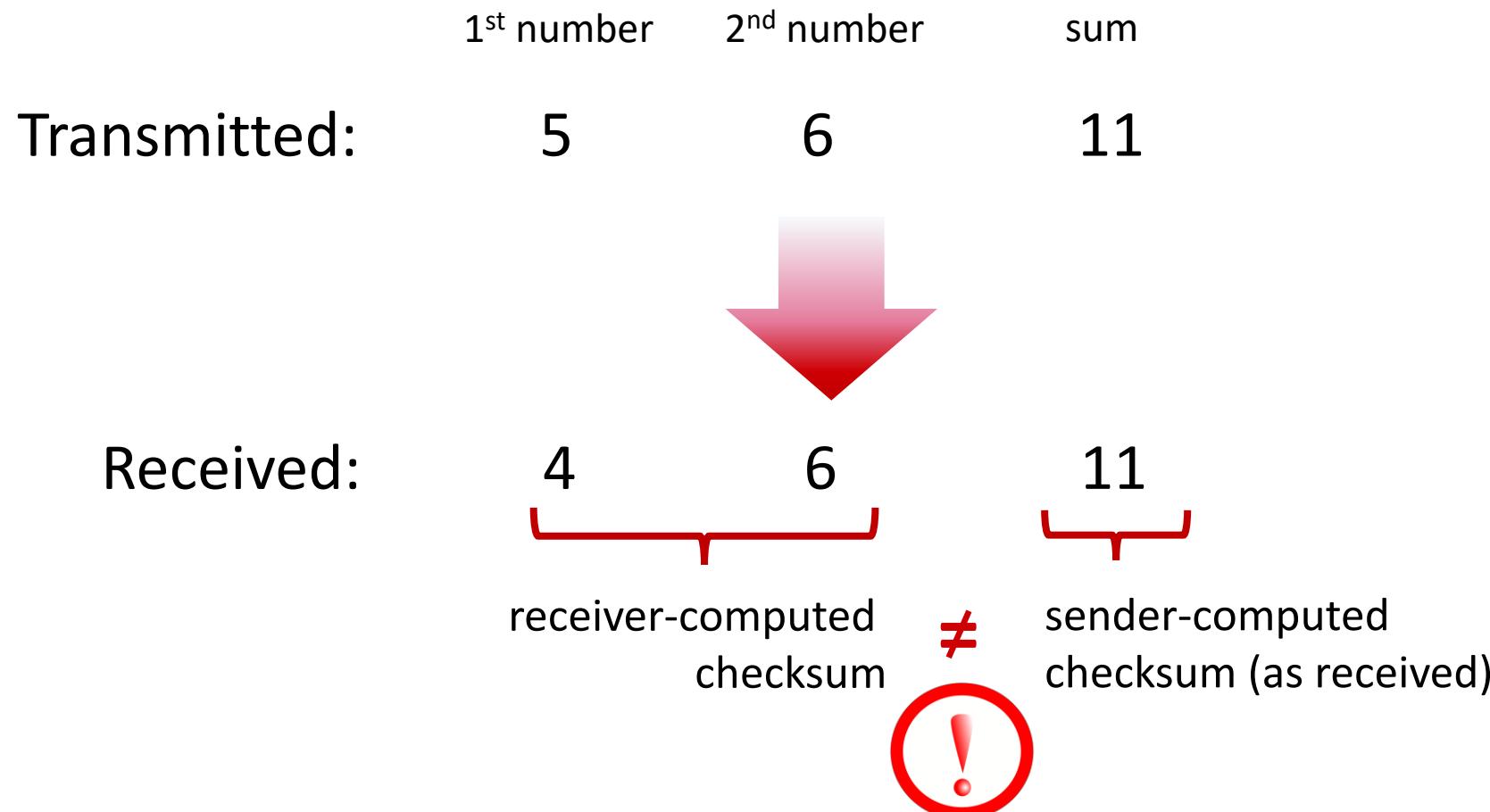


# UDP segment header



# UDP checksum

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



# 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 sequence of 16-bit integers
- **checksum:** addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

# Internet checksum

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

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
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
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

# 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	0	1
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1
<hr/>																			
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	1	0	0	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	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 (no RTT incurred)
  - 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



# Interactive Problem

- Compute the Internet checksum value for these two 16-bit words:

11100010 11000111

*this binary number is 58055  
decimal (base 10)*

11011110 11111010

*this binary number is 57082  
decimal (base 10)*

- What is the sum of these two 16 bit numbers?
- Using the sum from question 1, what is the checksum?

# Answer

1.

11100010 11000111

11011110 11111010

---

1 11000001 11000001

1

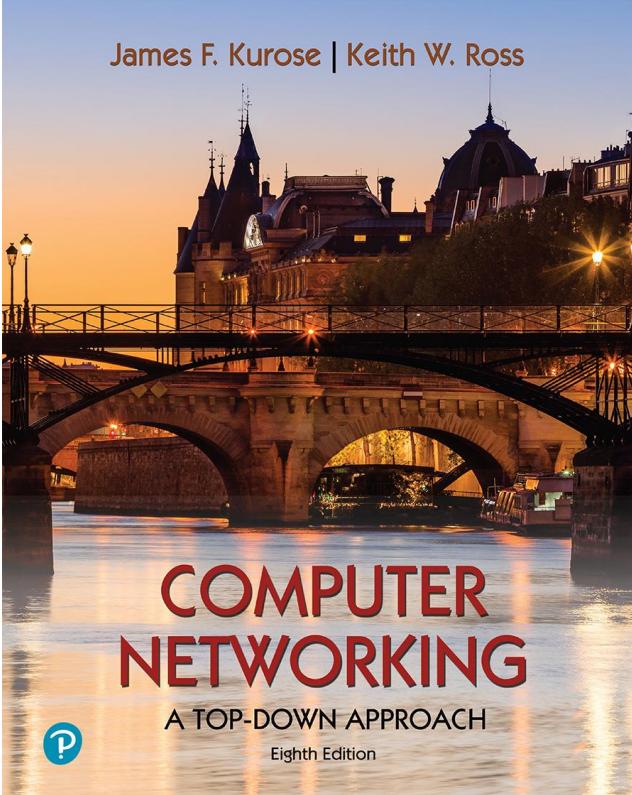
---

11000001 11000010

2.

00111110 00111101

# Copyright Information



## *Computer Networking: A Top-Down Approach*

8<sup>th</sup> edition

Jim Kurose, Keith Ross  
Pearson, 2020

The Slides are adapted from,

All material copyright 1996-2023  
J.F Kurose and K.W. Ross, All Rights Reserved

# 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

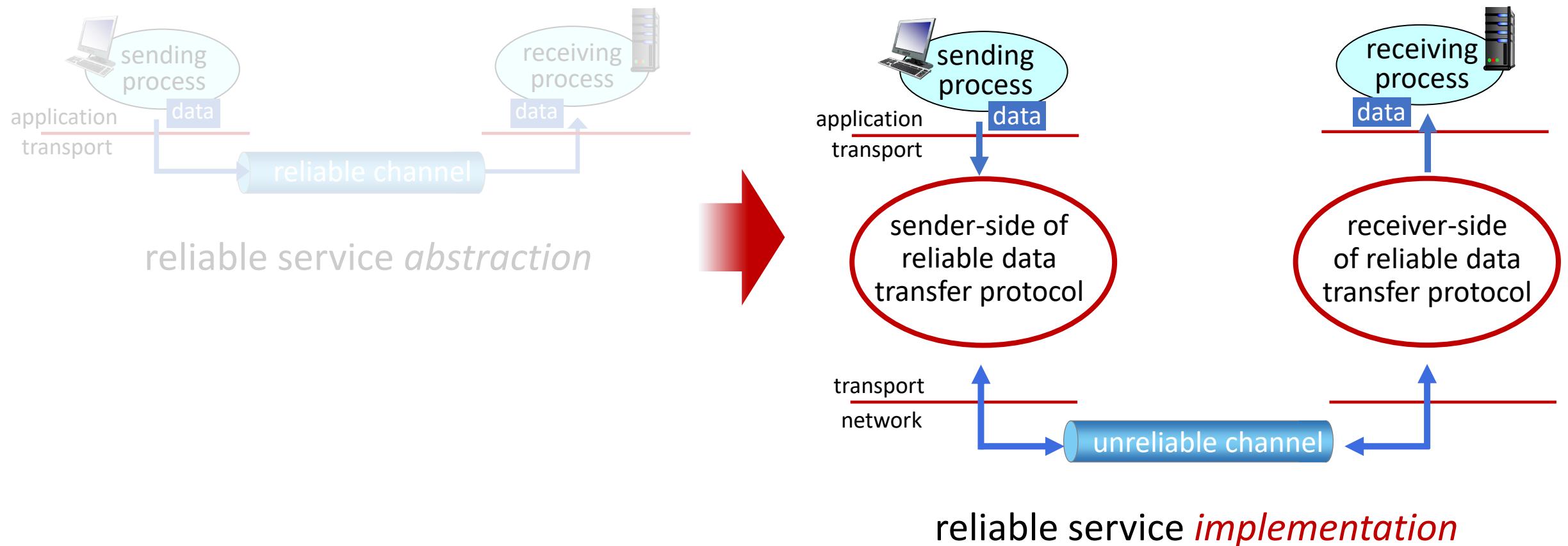
- Channel model
- Realistic assumptions
- Protocol mechanisms

# Principles of reliable data transfer



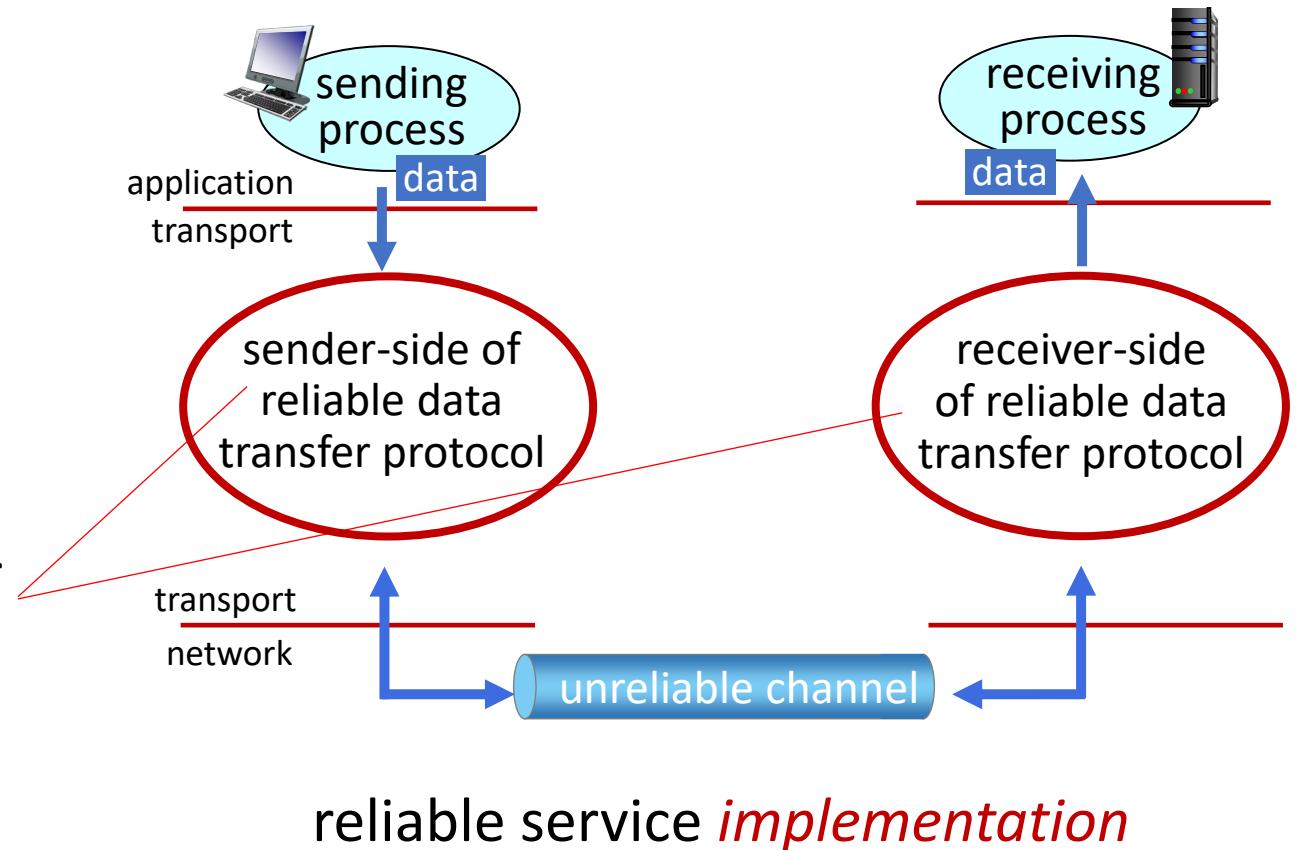
reliable service *abstraction*

# Principles of reliable data transfer



# Principles of reliable data transfer

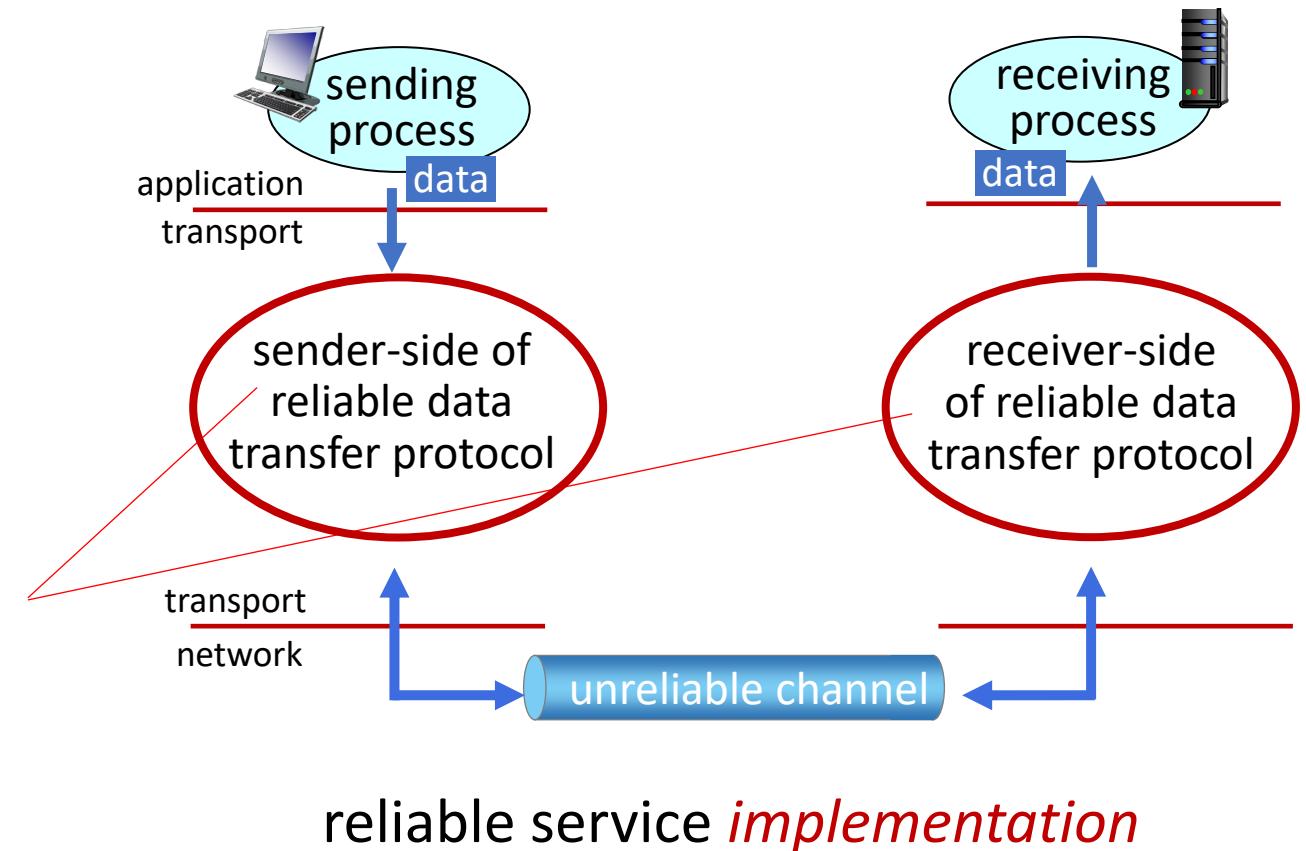
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



# Principles of reliable data transfer

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

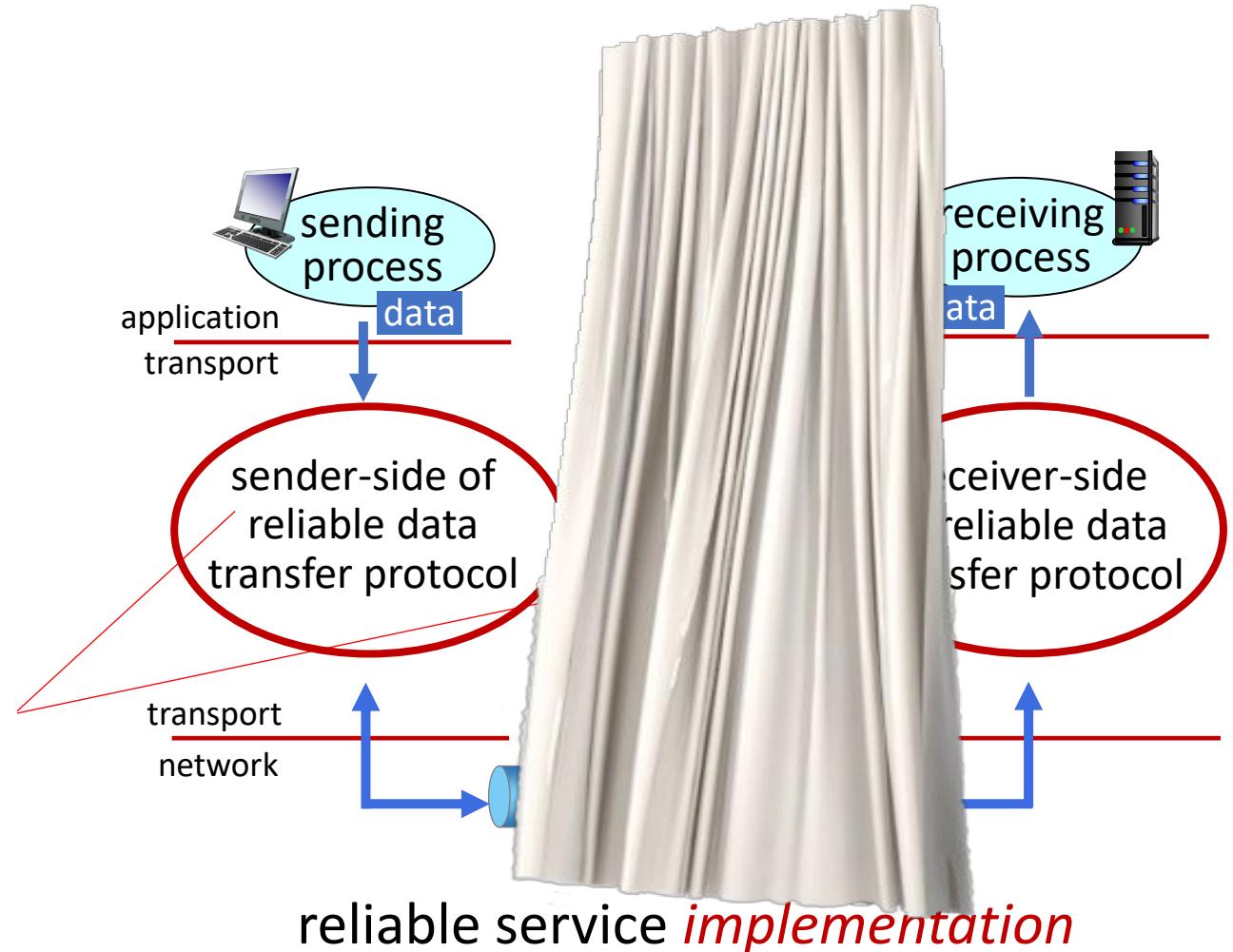
- unless communicated via a message



# Principles of reliable data transfer

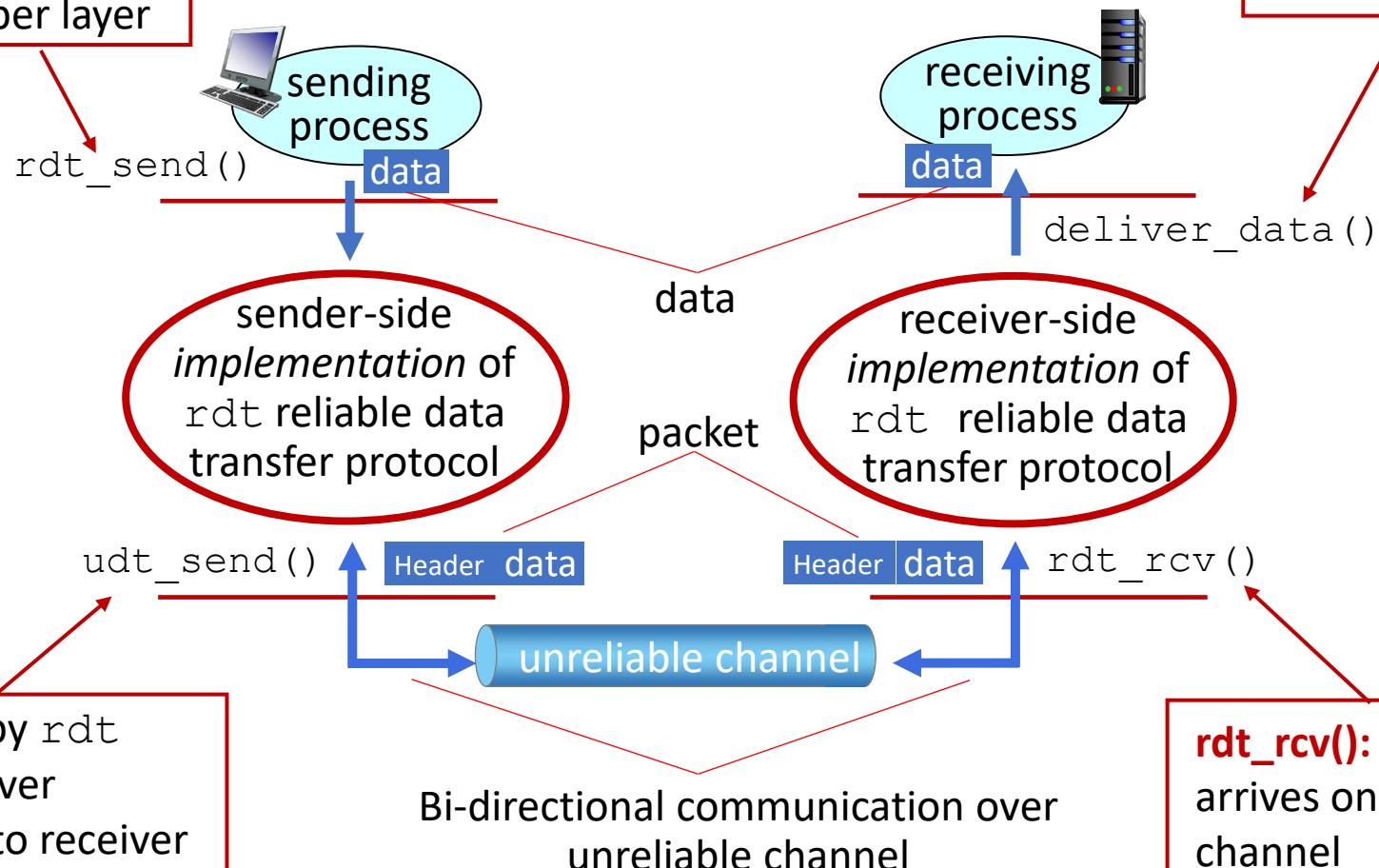
Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

- unless communicated via a message



# Reliable data transfer protocol (rdt): interfaces

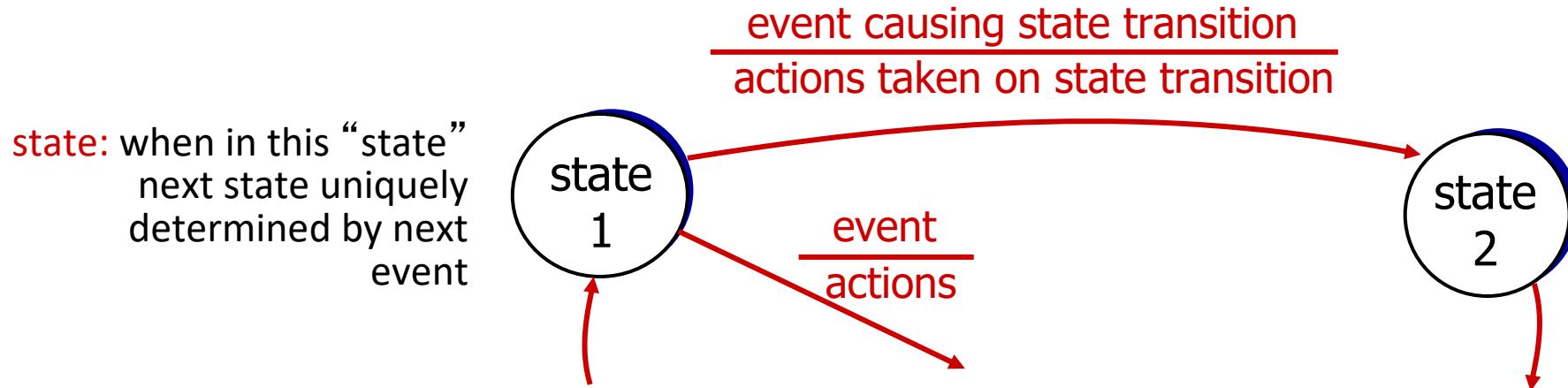
**rdt\_send():** called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



# 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



# Reliable data transfer: getting started

- What do we really mean when we say RDT senders in a given state or a receiver is in a given state?
- Think about a link being in a transmitting state or in an idle State.
- Think about notion of there being transitions between states.
- Transitions happening because of an event that takes place.
- think about actions that are taken by the system.

# 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

# rdt1.0: reliable transfer over a reliable channel



# 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?*

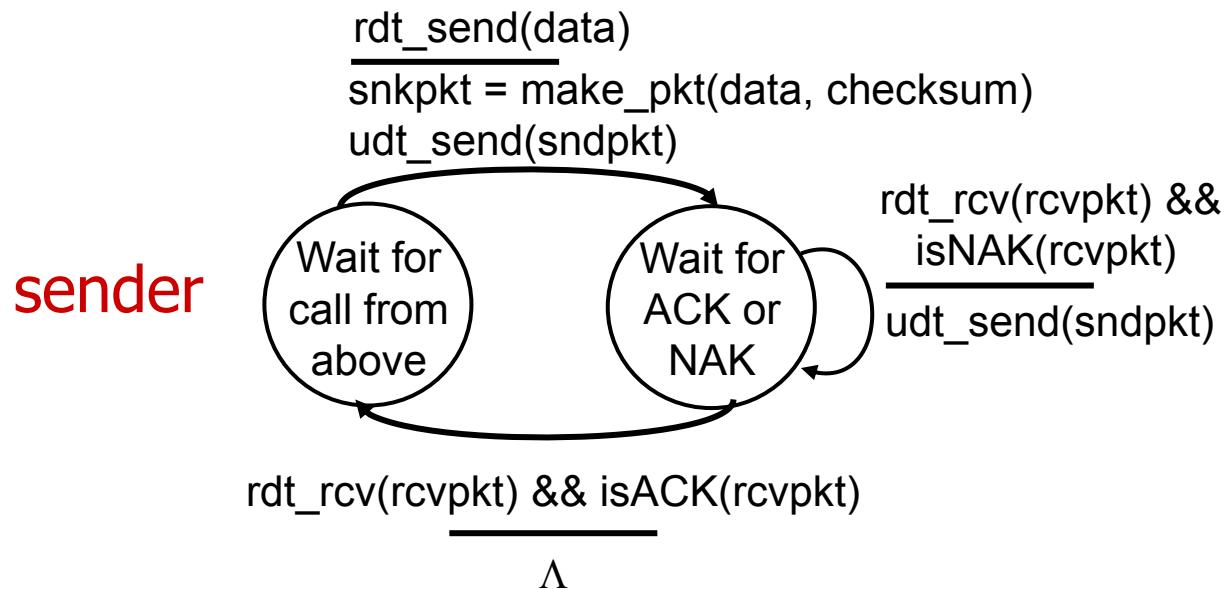
# rdt2.0: channel with bit errors

- 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

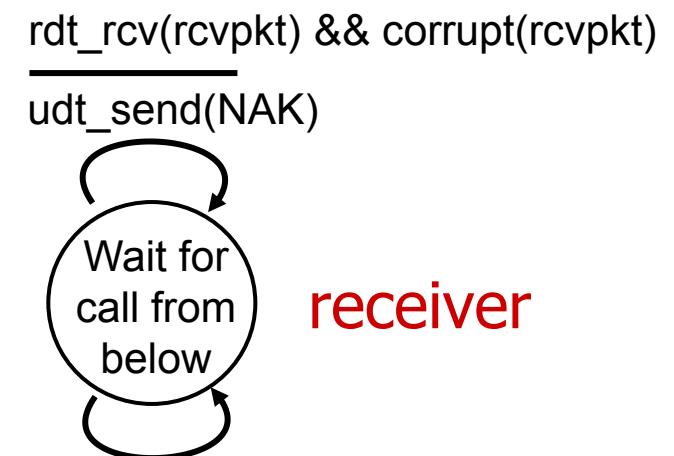
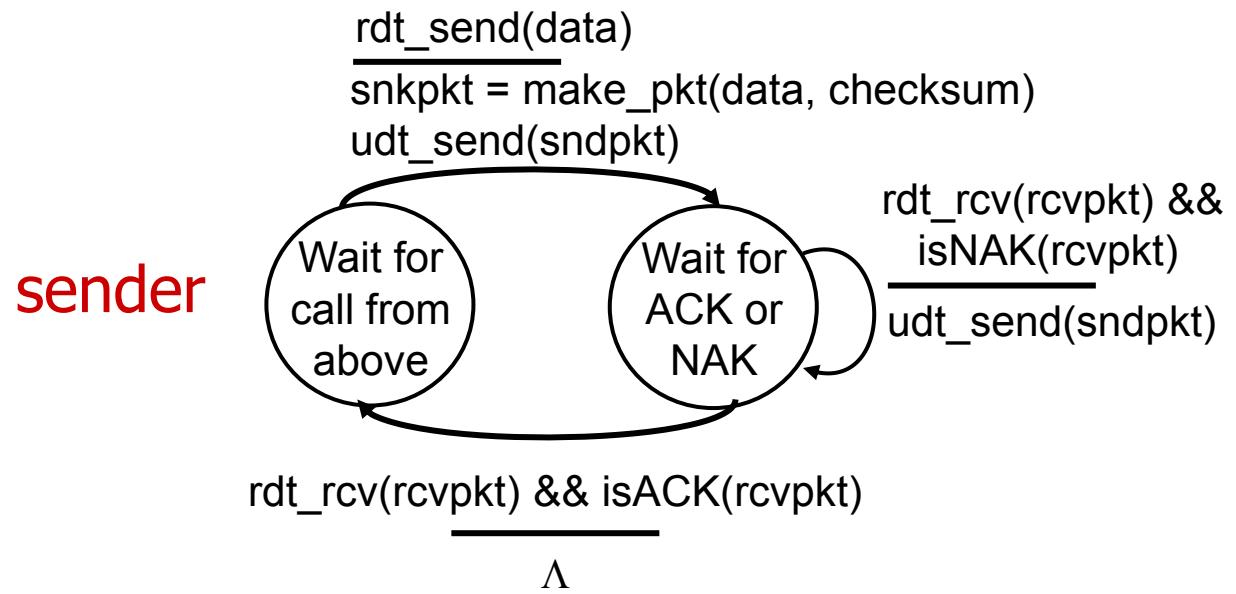
stop and wait

sender sends one packet, then waits for receiver response

# rdt2.0: FSM specification



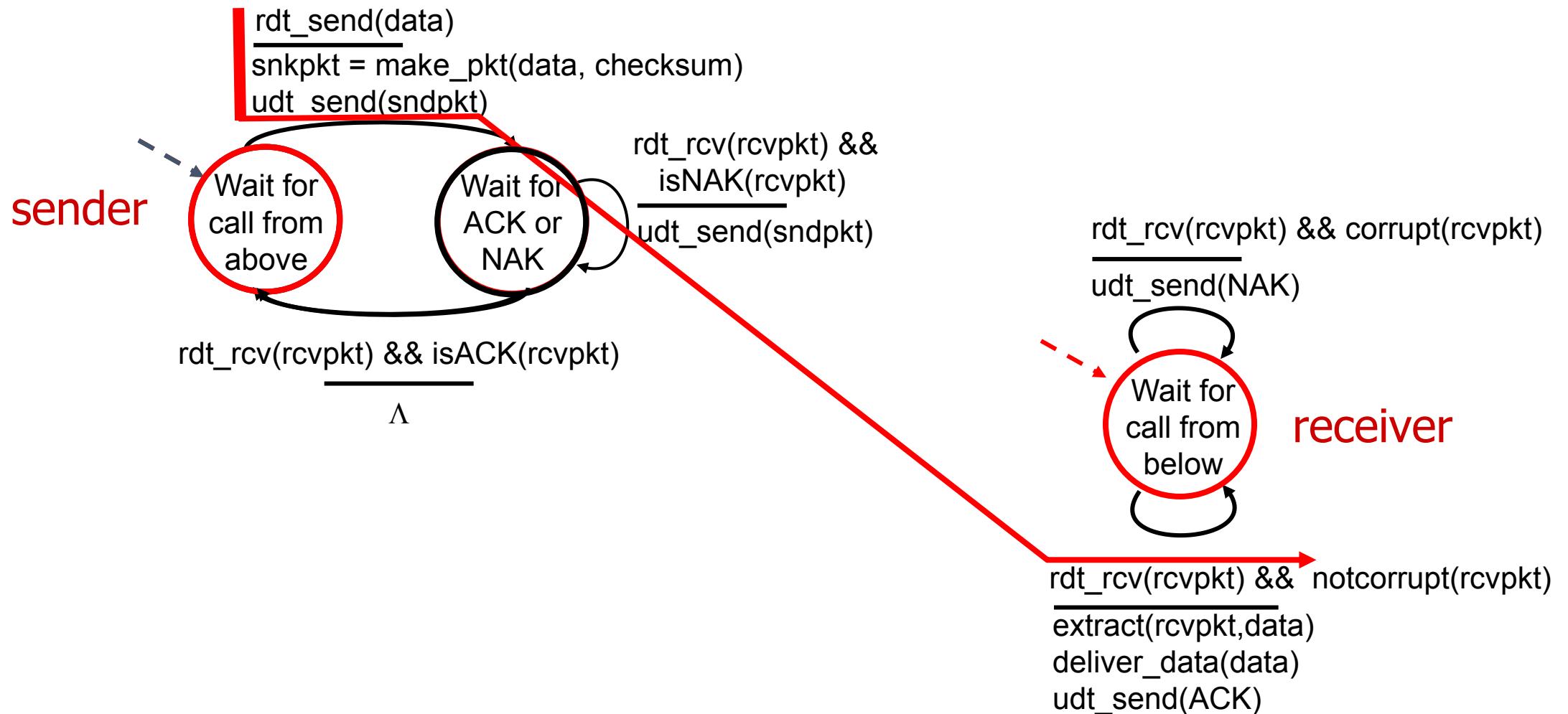
# rdt2.0: FSM specification



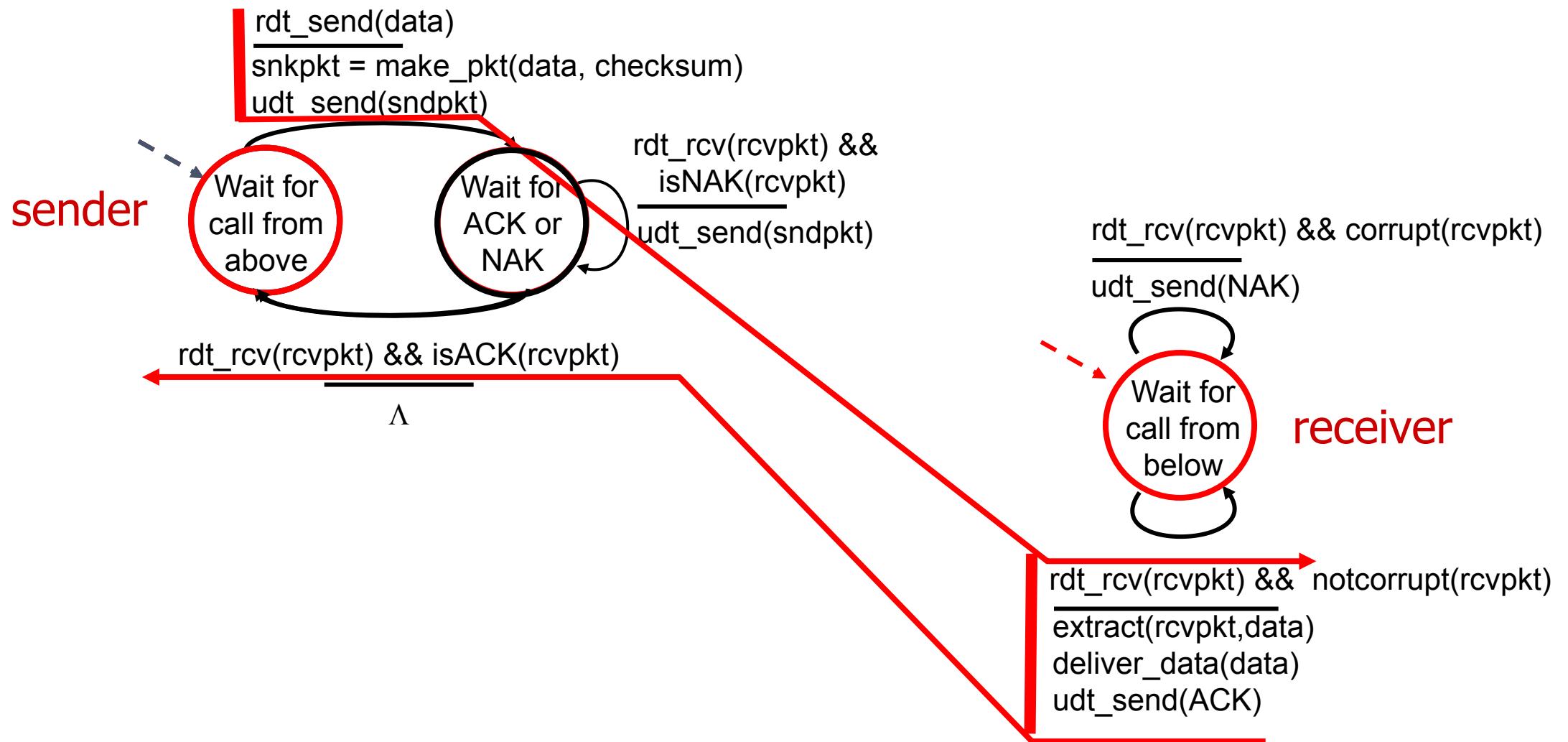
**Note:** “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender

- that’s why we need a protocol!

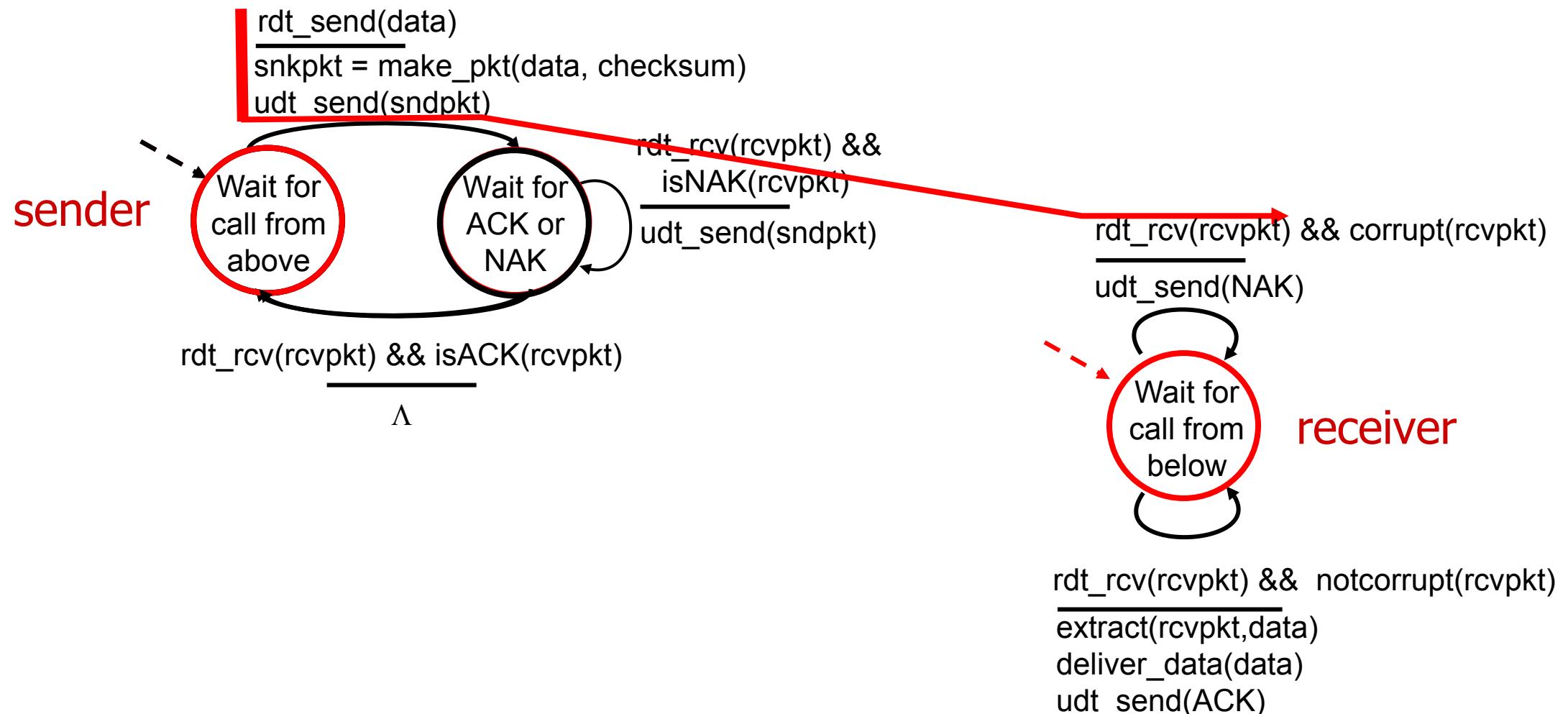
# rdt2.0: operation with no errors



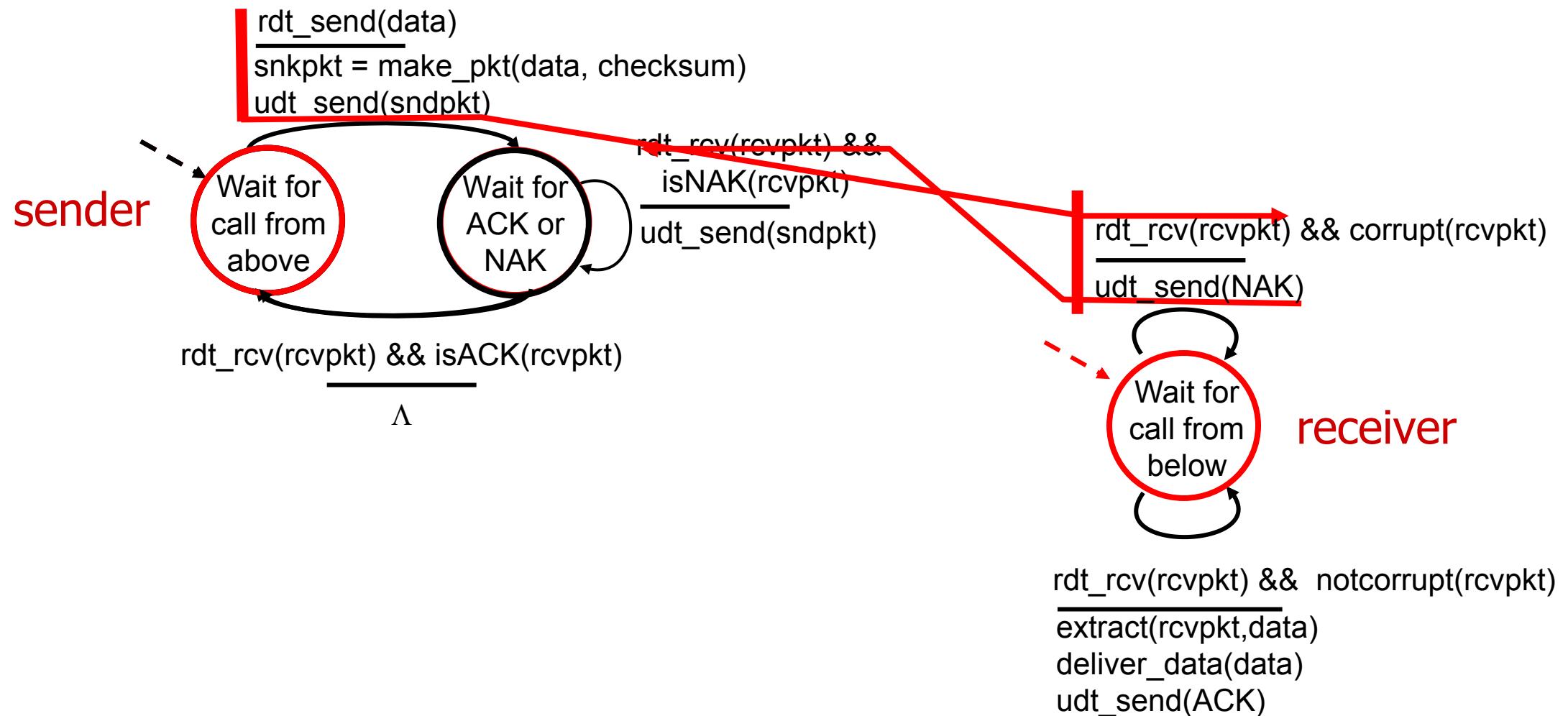
# rdt2.0: operation with no errors



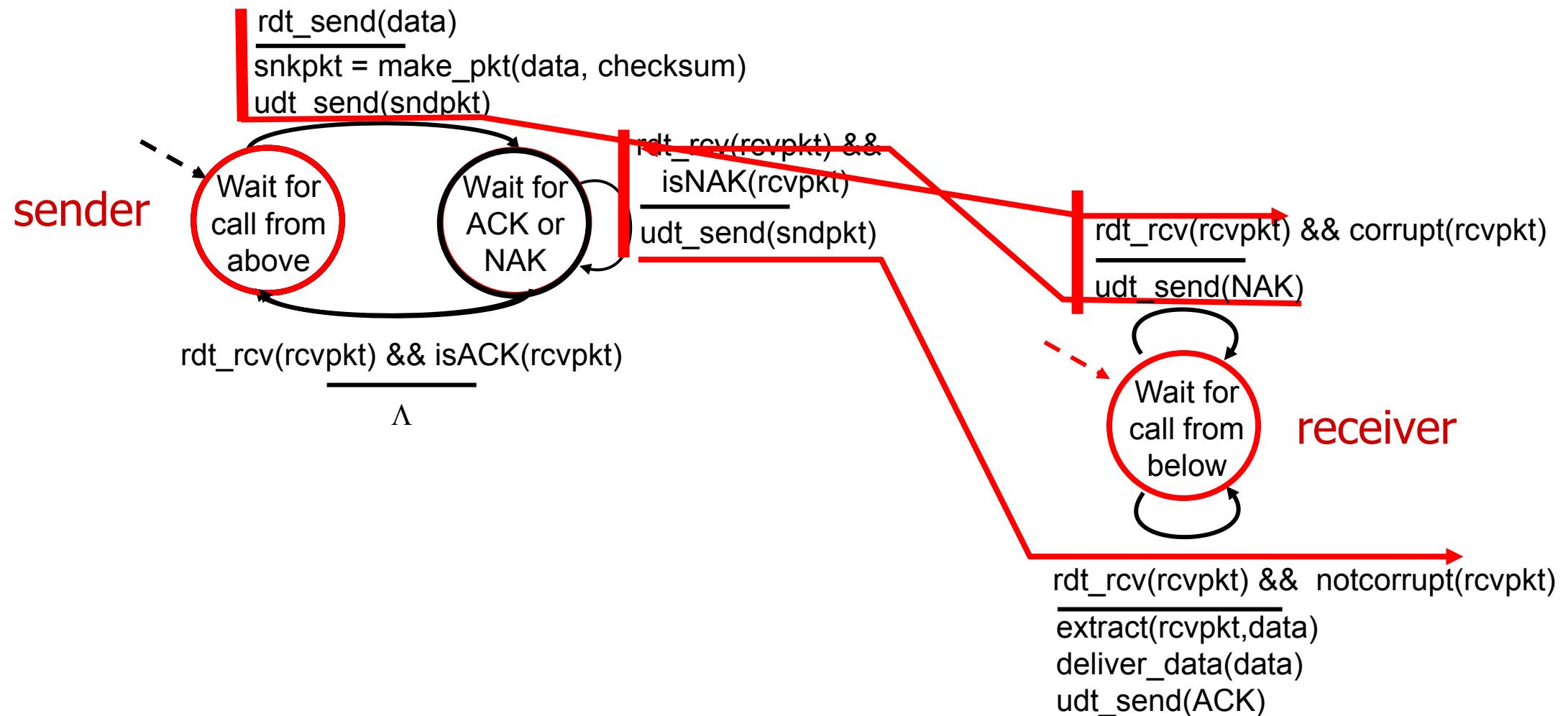
# rdt2.0: corrupted packet scenario



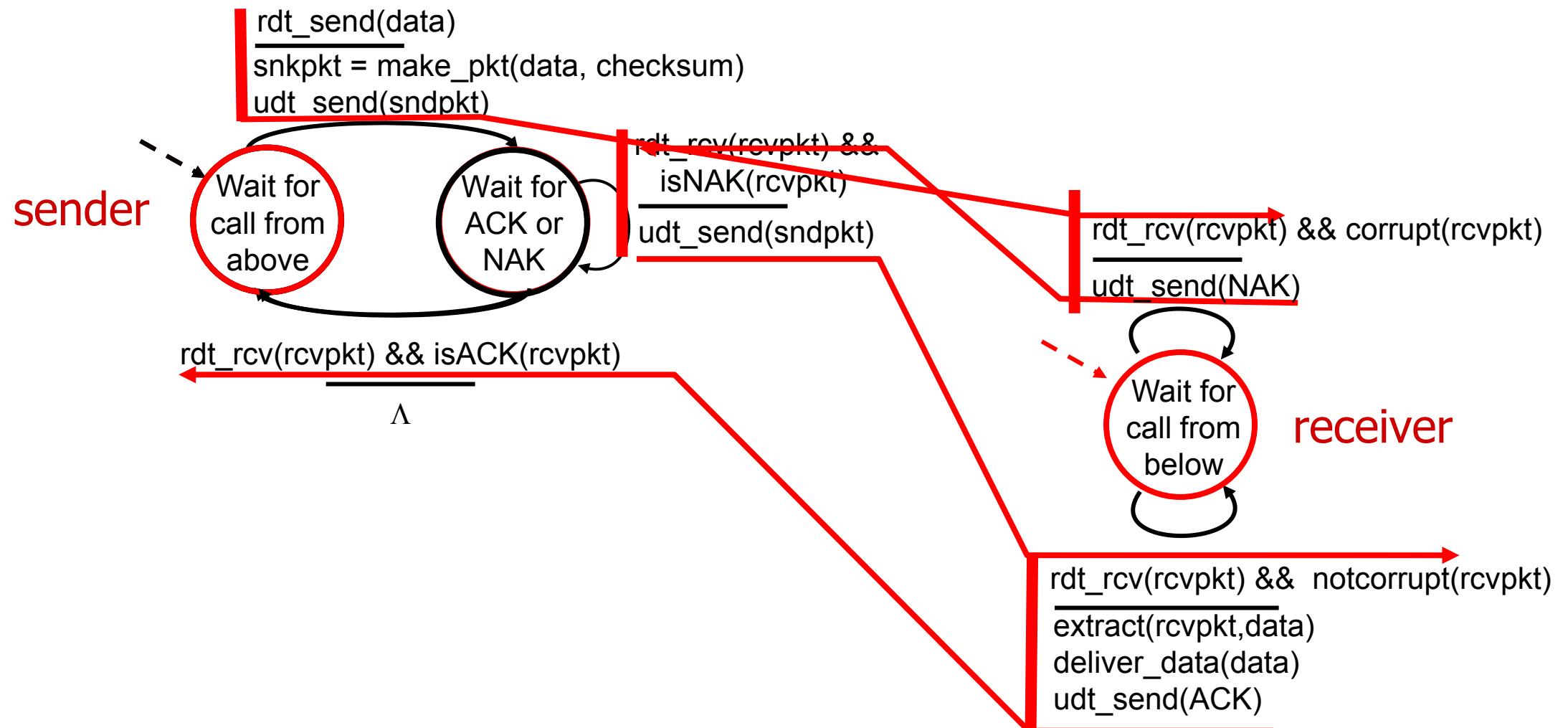
# rdt2.0: corrupted packet scenario



# rdt2.0: corrupted packet scenario



# 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

# rdt2.0 has a fatal flaw!

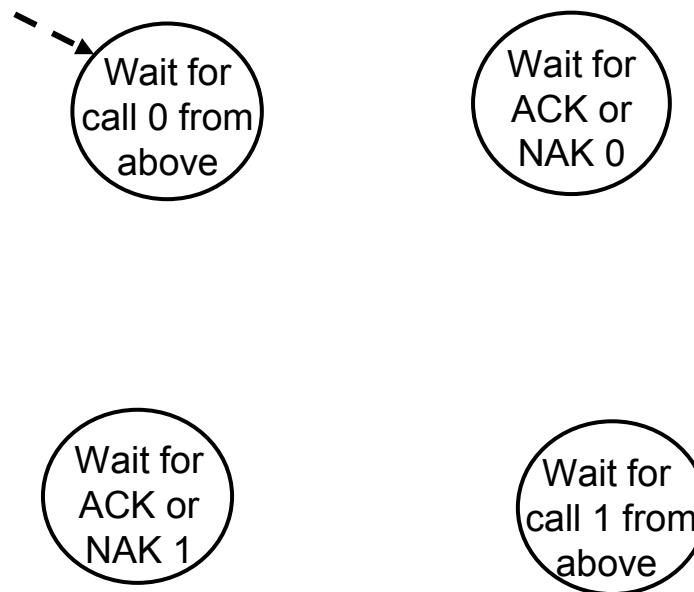
## handling duplicates:

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

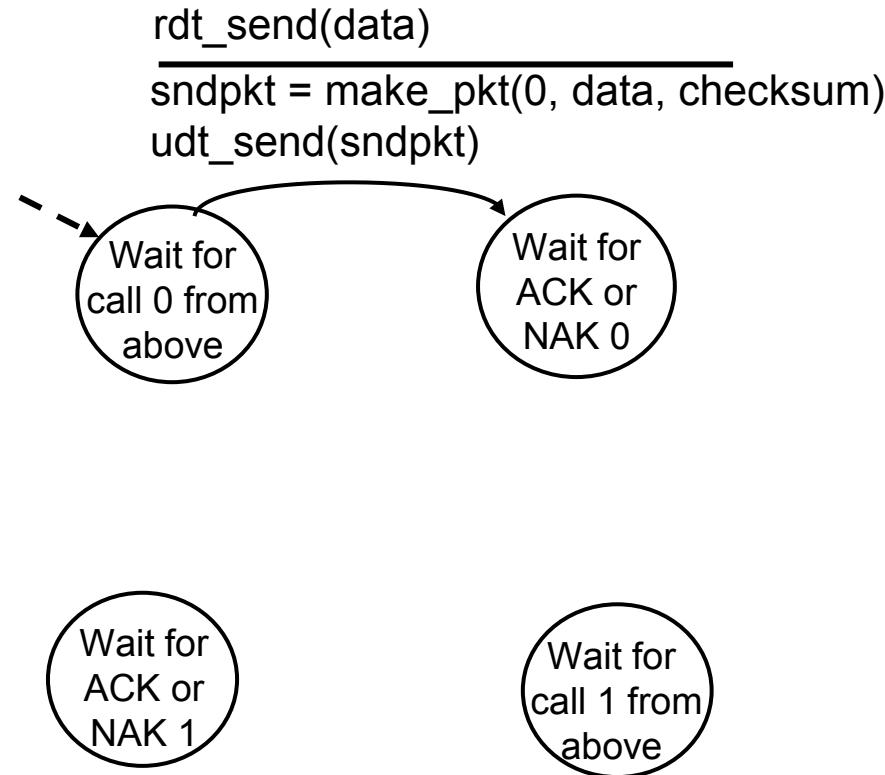
stop and wait

sender sends one packet, then waits for receiver response

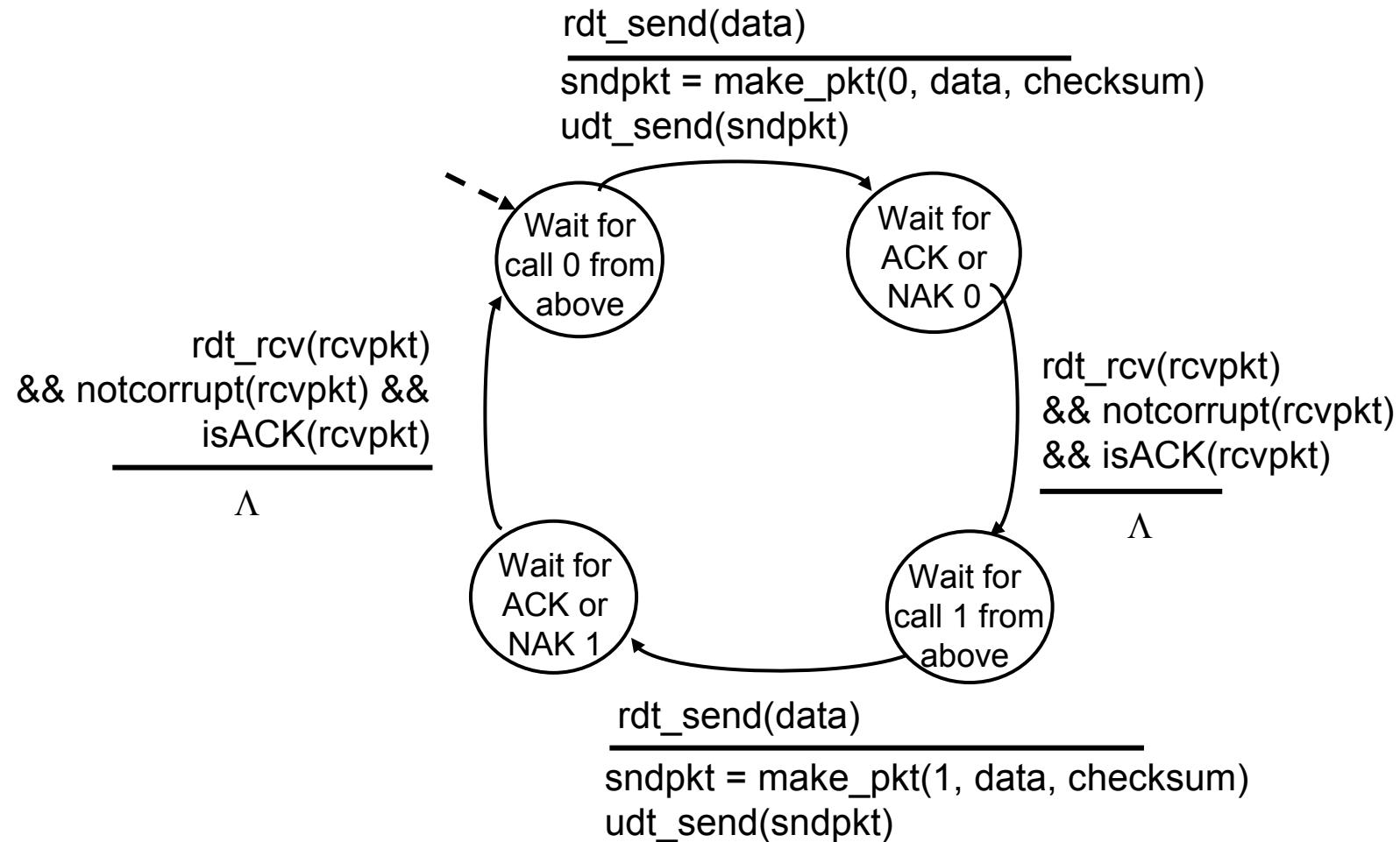
# rdt2.1: sender, handling garbled ACK/NAKs



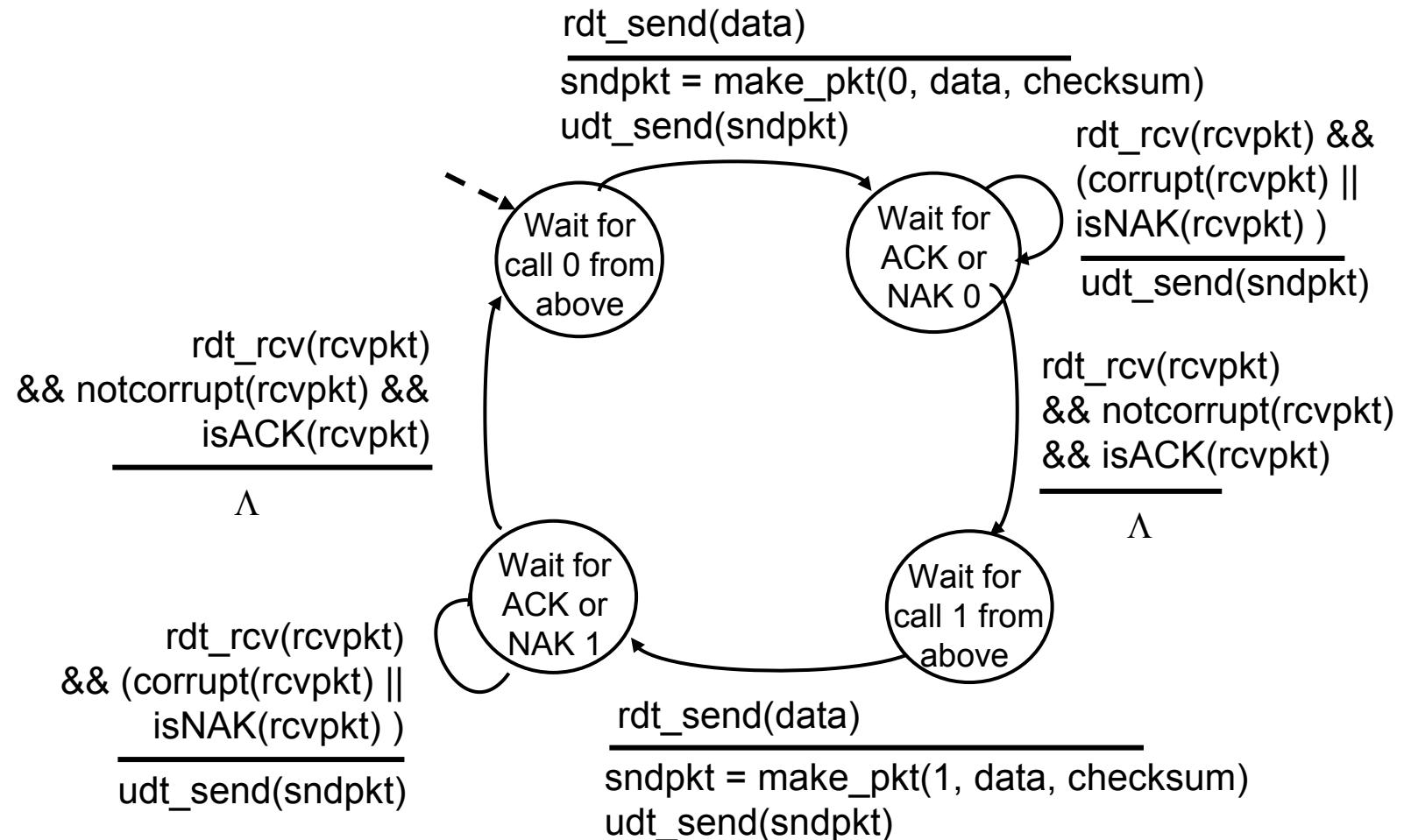
# rdt2.1: sender, handling garbled ACK/NAKs



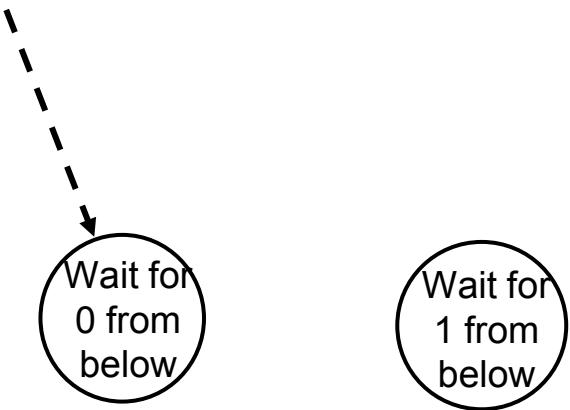
# rdt2.1: sender, handling garbled ACK/NAKs



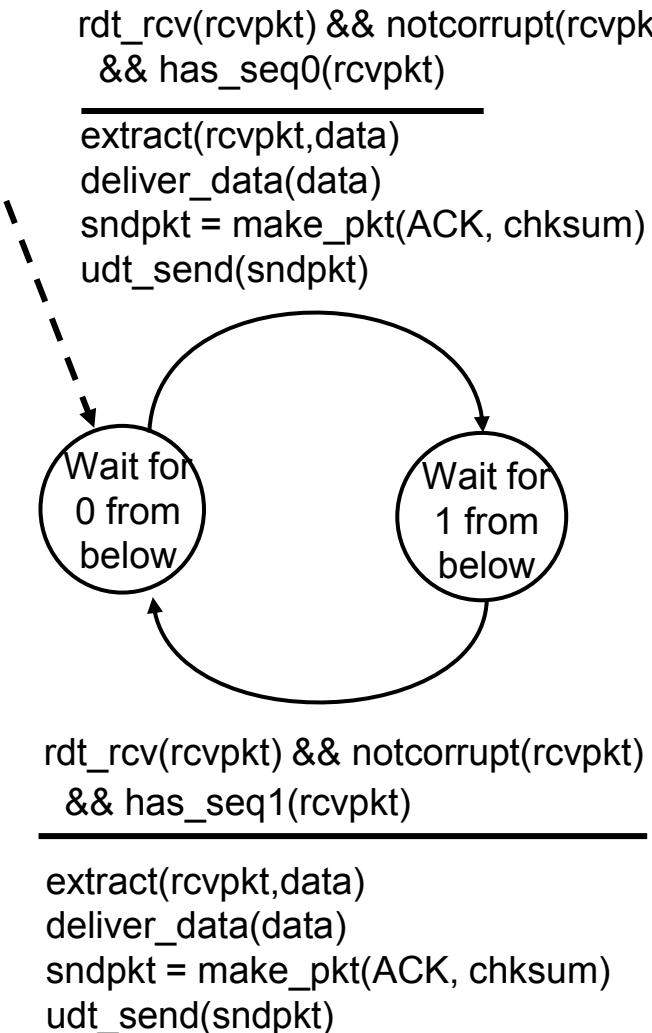
# rdt2.1: sender, handling garbled ACK/NAKs



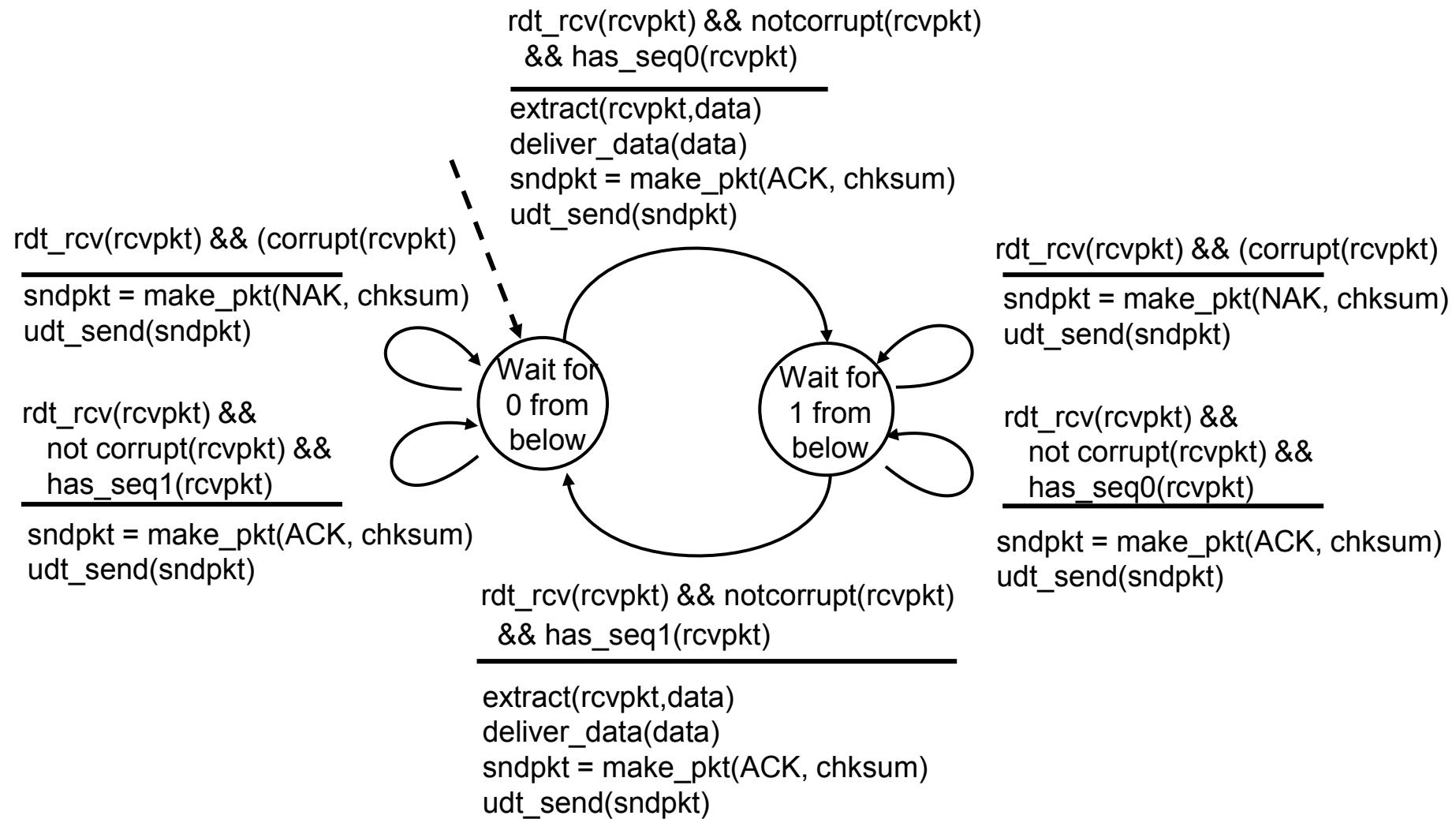
# rdt2.1: receiver, handling garbled ACK/NAKs



# rdt2.1: receiver, 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

# rdt protocol mechanisms

- Error detection (e.g., checksum)
- ACKs, NAKs
- Retransmission
- Sequence numbers (duplicate detection)

## Important Dates

06-21-2024(Friday) – HW 2

06-24-2024(Monday) – Quiz 1

# Quiz 1 Guidelines

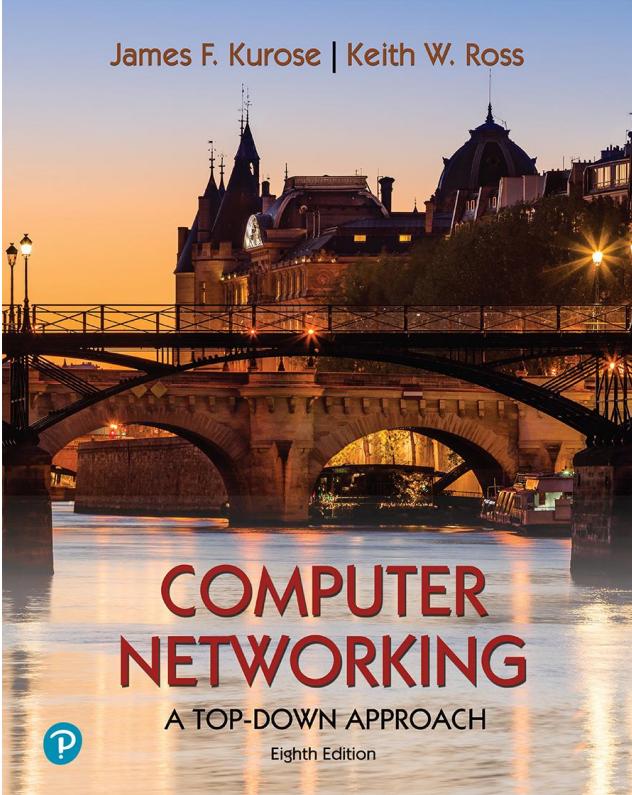
- 06-24-2024(Monday)– 11:00 AM-12:15 PM
- Please arrive by 11 AM and leave at least two seats vacant between you and other students while taking the quiz.
- The quiz is closed-book, but one side of an 8  $\frac{1}{2}$ " by 11" sheet may be used, and the quiz is worth 100 points.(Only formulas are allowed)
- The quiz will consist of three main questions, each with different sub-questions.
- The quiz will cover all the concepts

From LectureD\_1\_Chapter\_1

To LectureD\_8\_Chapter\_3

- No other electronic devices are allowed except a calculator for the quiz.

# Copyright Information



## *Computer Networking: A Top-Down Approach*

8<sup>th</sup> edition

Jim Kurose, Keith Ross  
Pearson, 2020

The Slides are adapted from,

All material copyright 1996-2023  
J.F Kurose and K.W. Ross, All Rights Reserved

# 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



# 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?

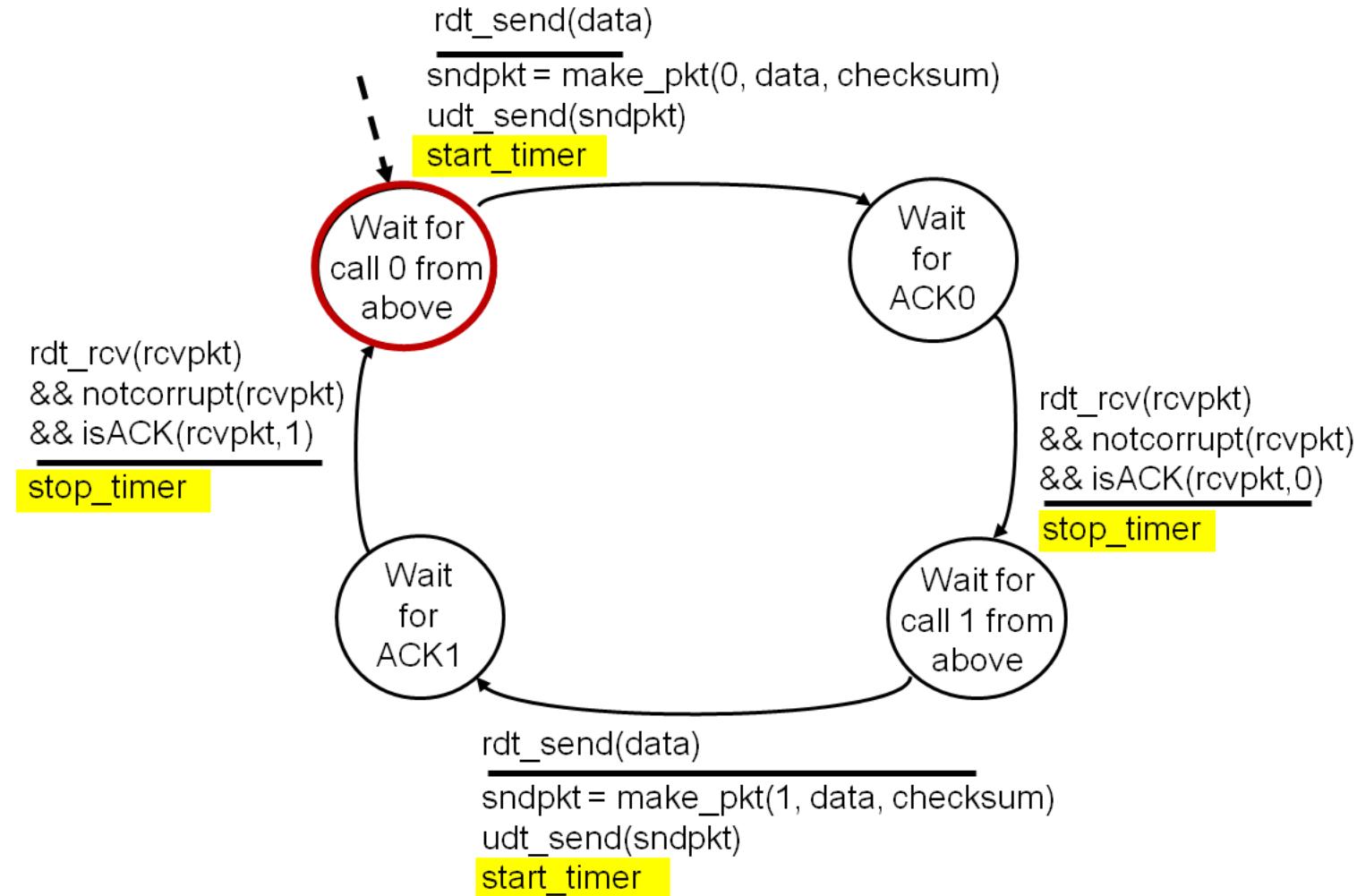
# rdt3.0: channels with errors *and* loss

**Approach:** sender waits “reasonable” amount of time for ACK

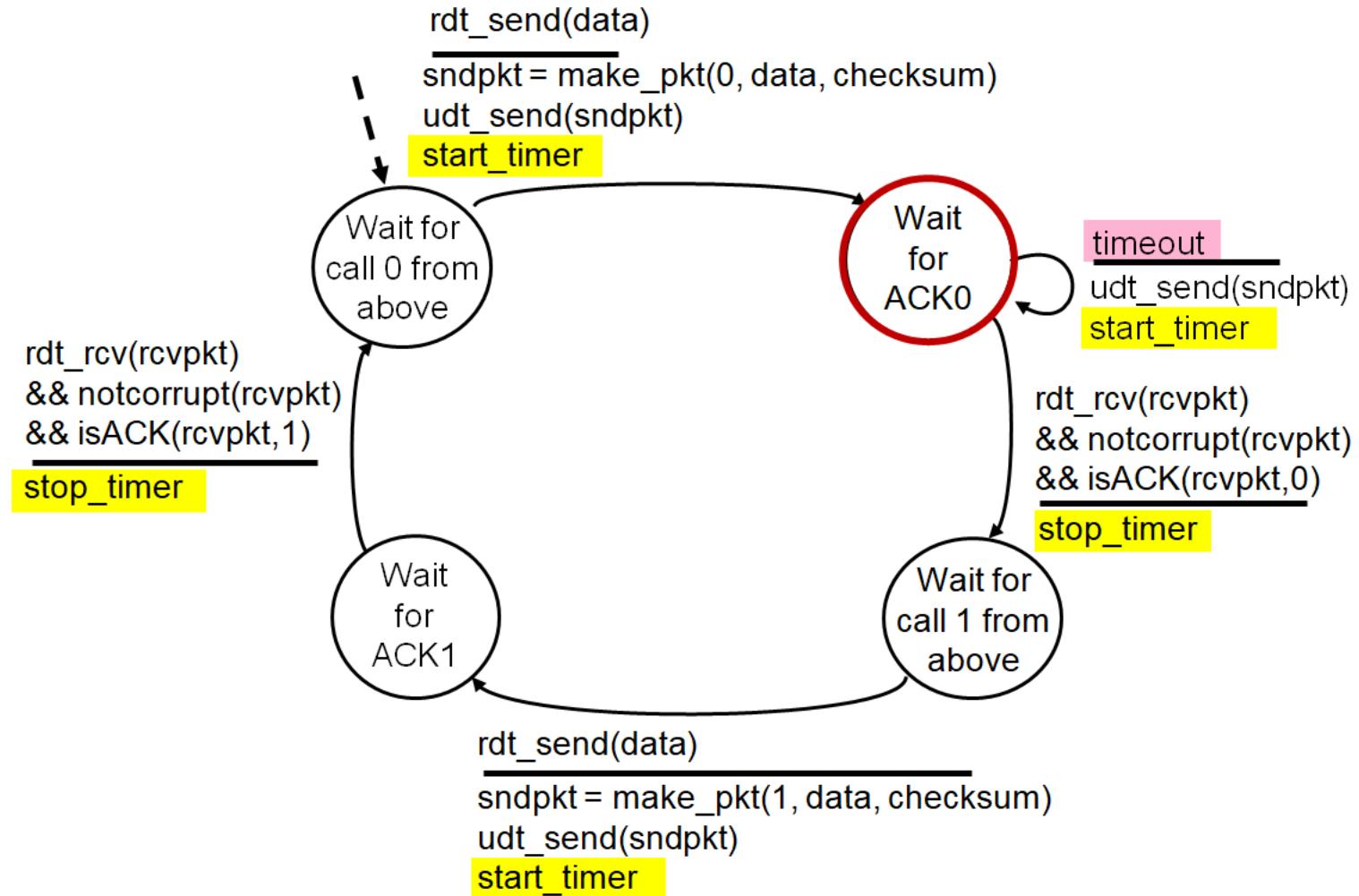
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time



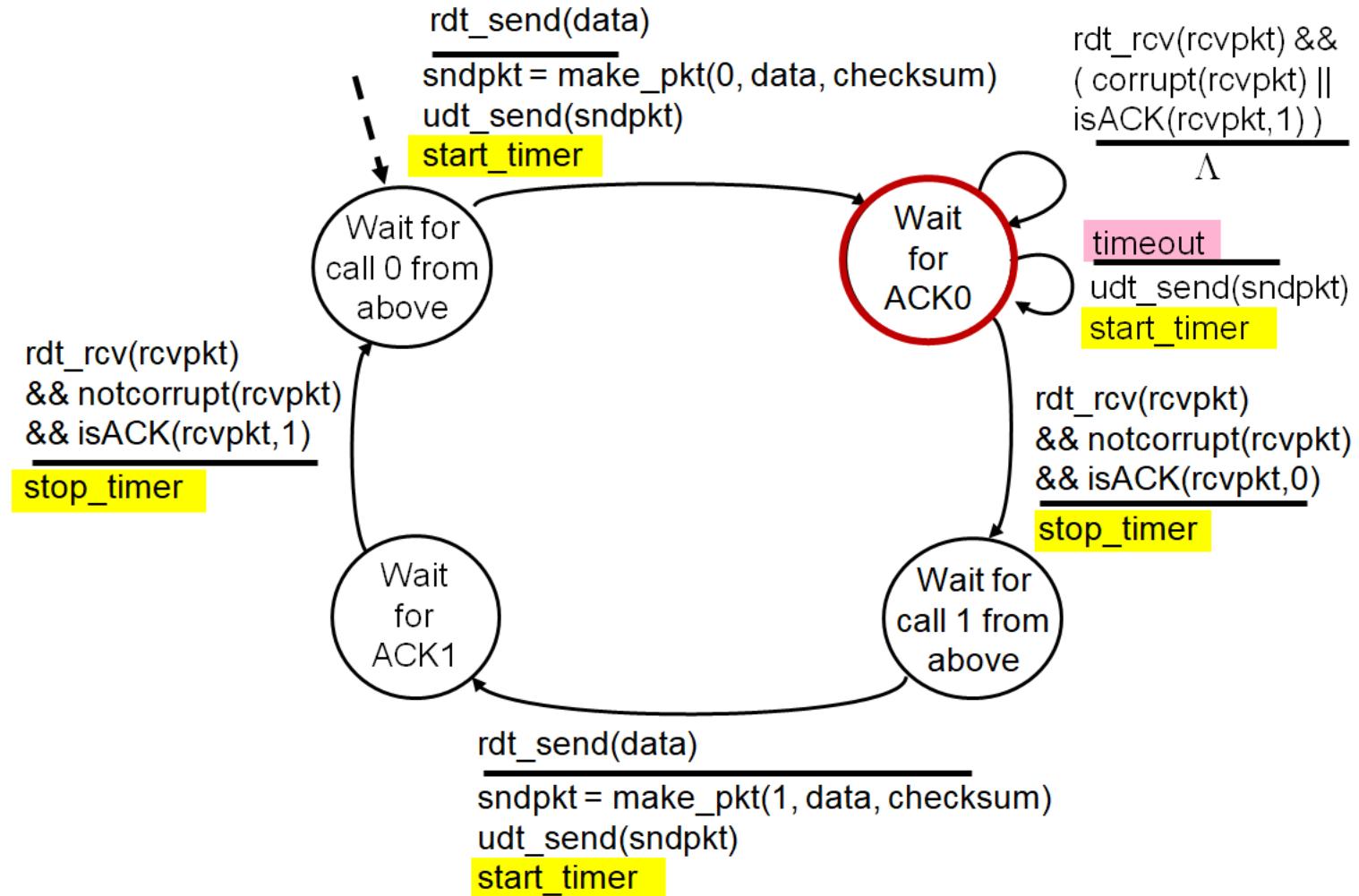
# rdt3.0 sender



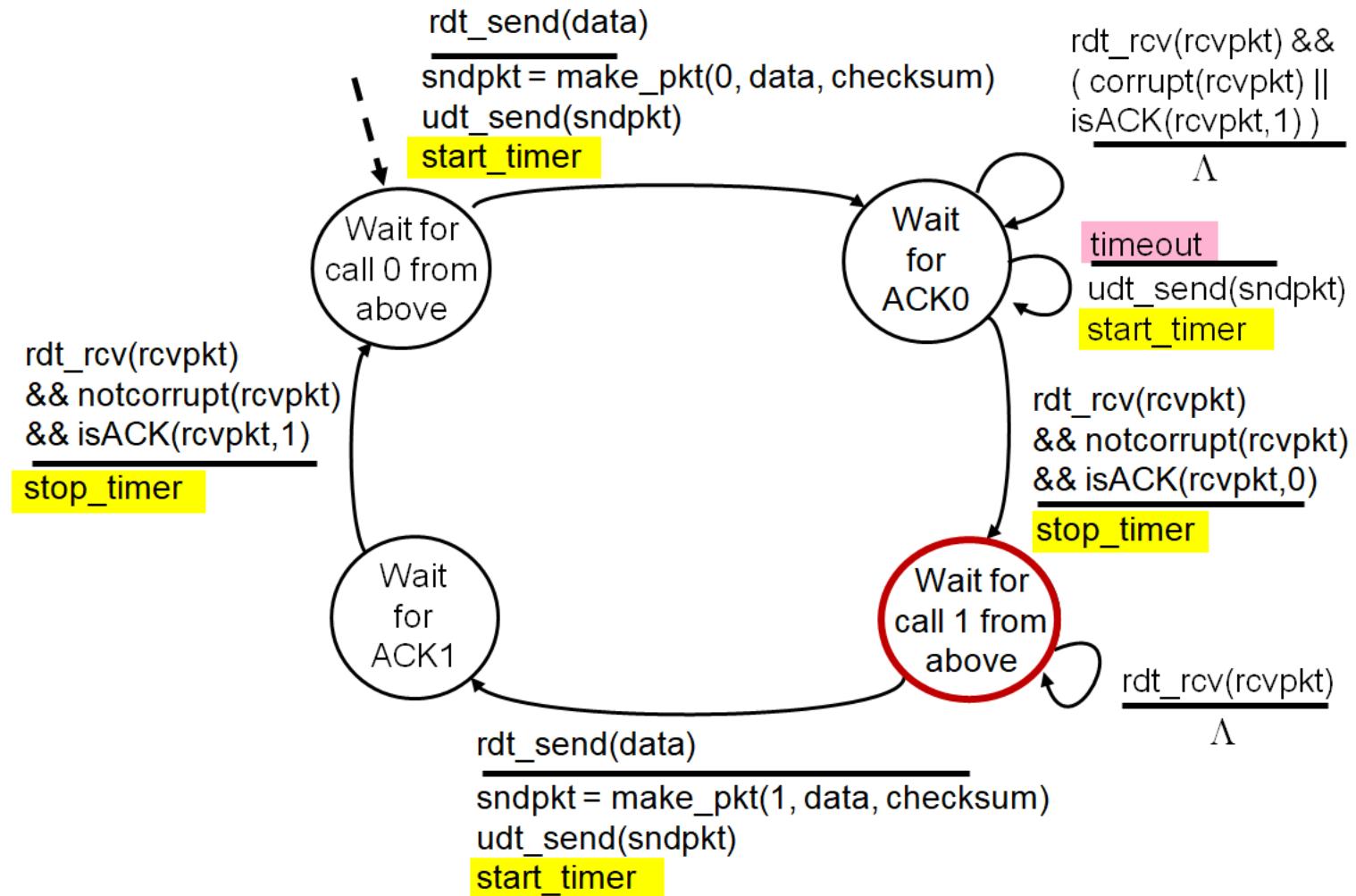
# rdt3.0 sender



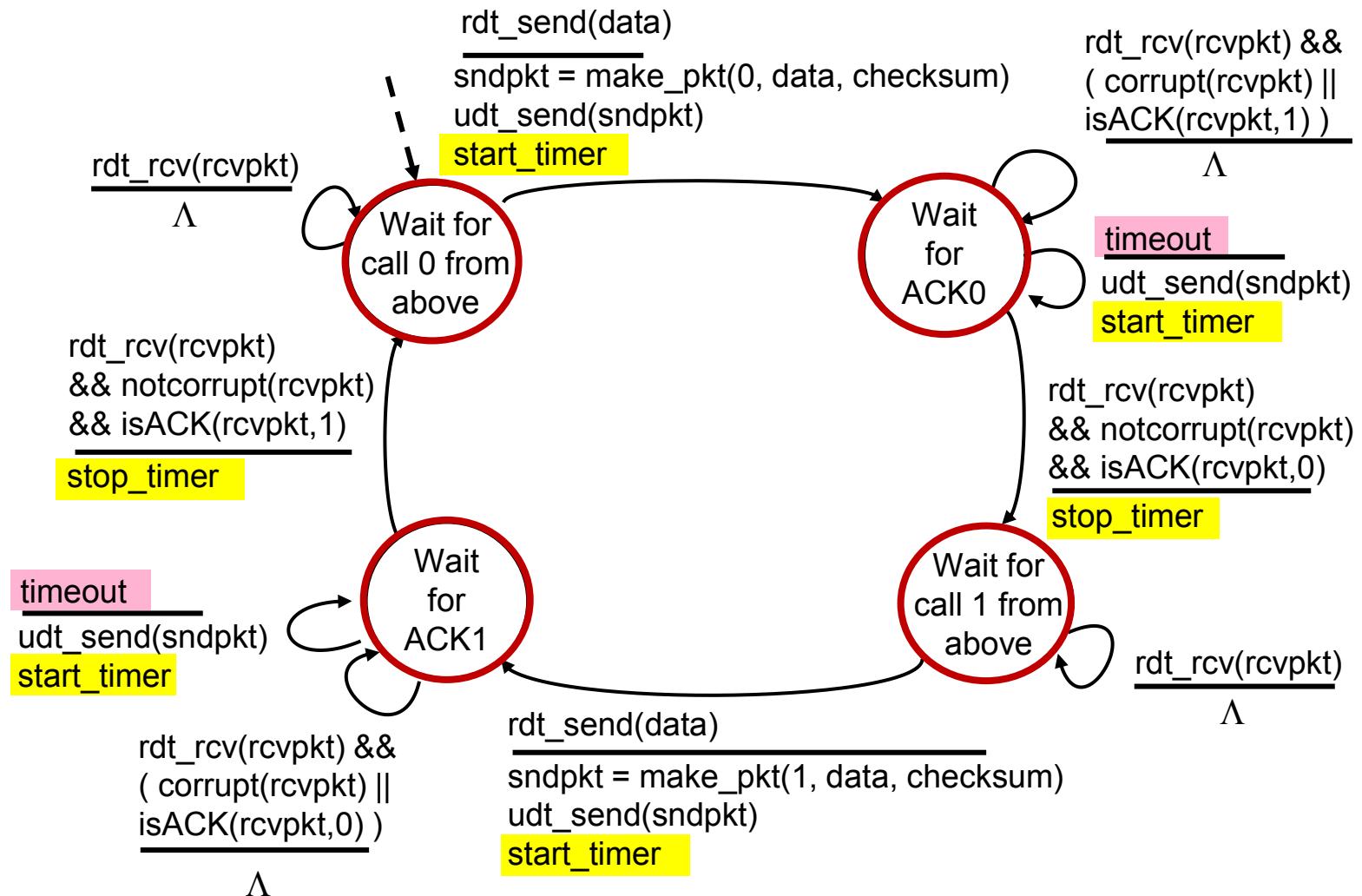
# rdt3.0 sender



# rdt3.0 sender



# rdt3.0 sender



# rdt3.0 in action

sender

send pkt0

receiver

sender

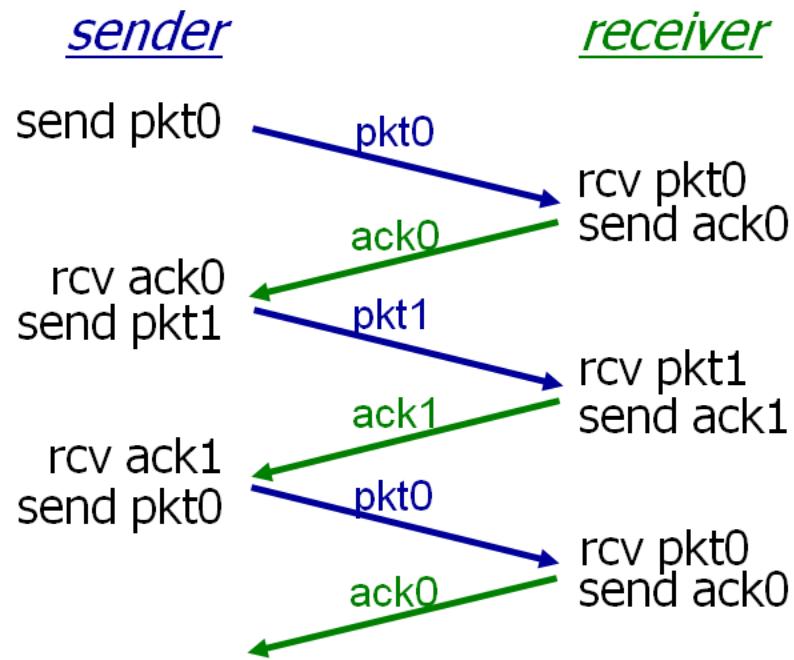
send pkt0

receiver

(a) no loss

(b) packet loss

# rdt3.0 in action

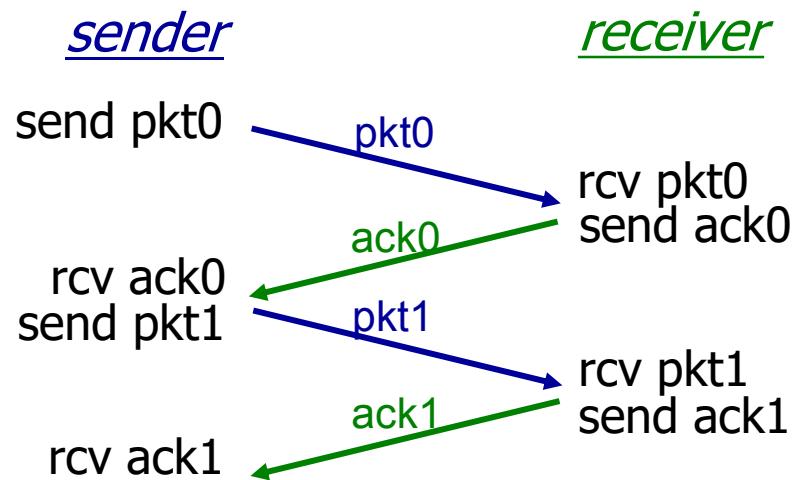


(a) no loss

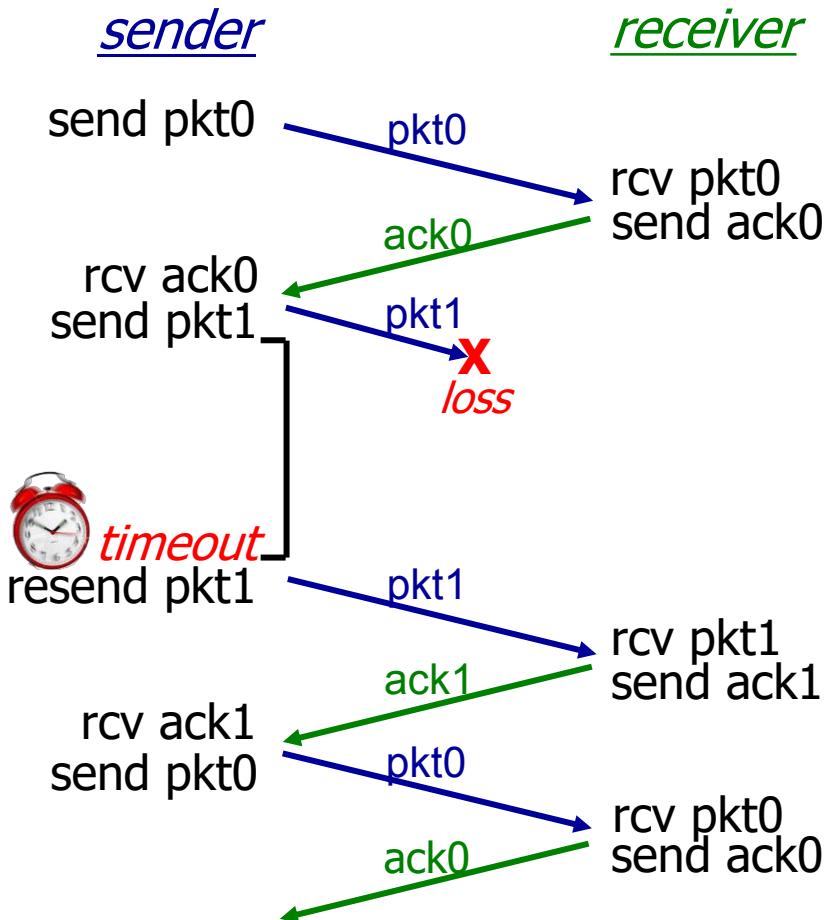


(b) packet loss

# rdt3.0 in action

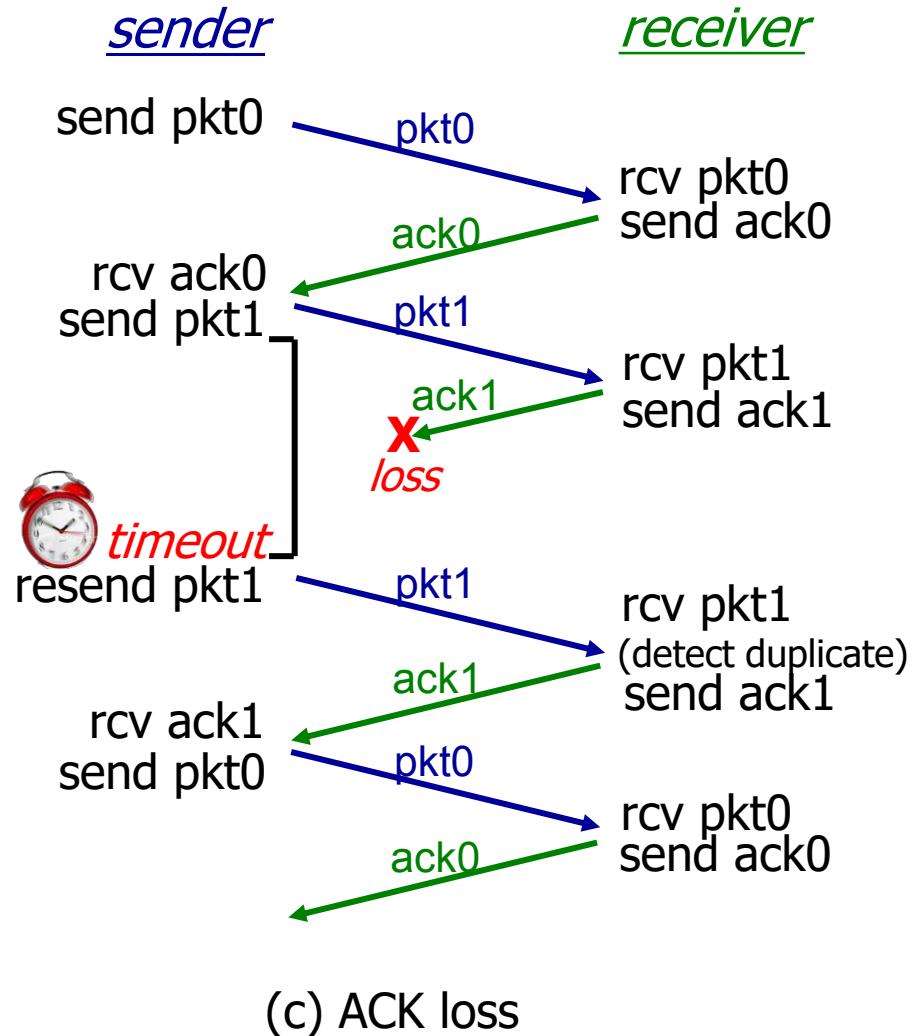


(a) no loss

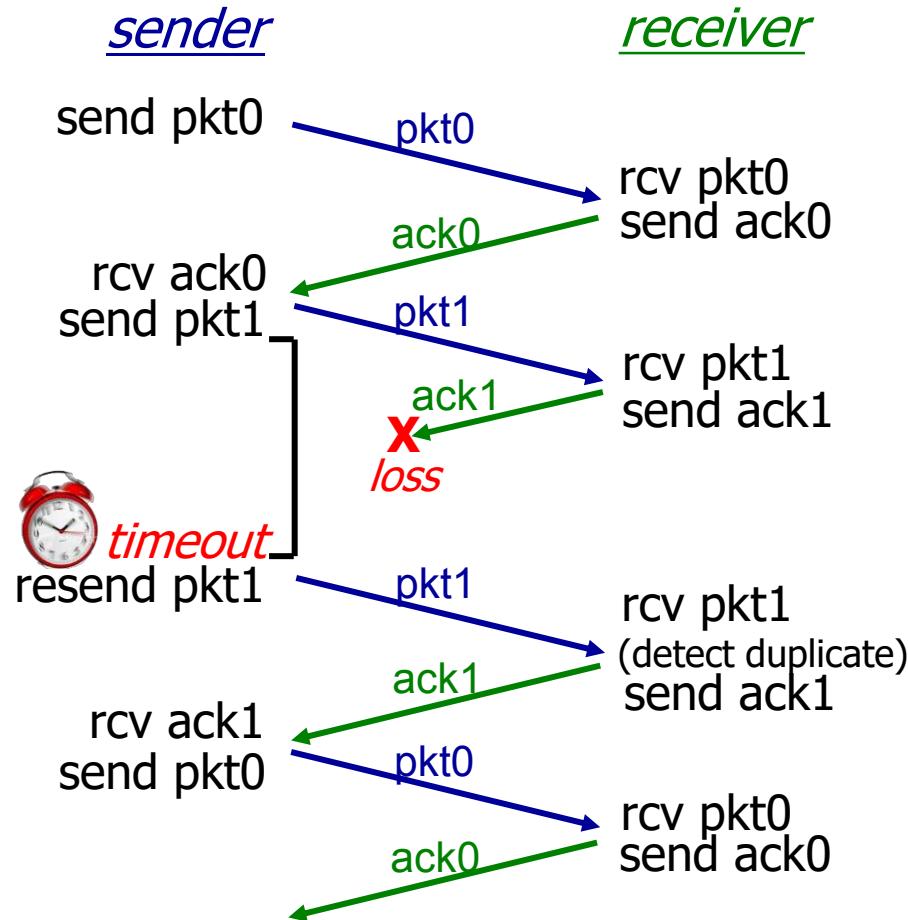


(b) packet loss

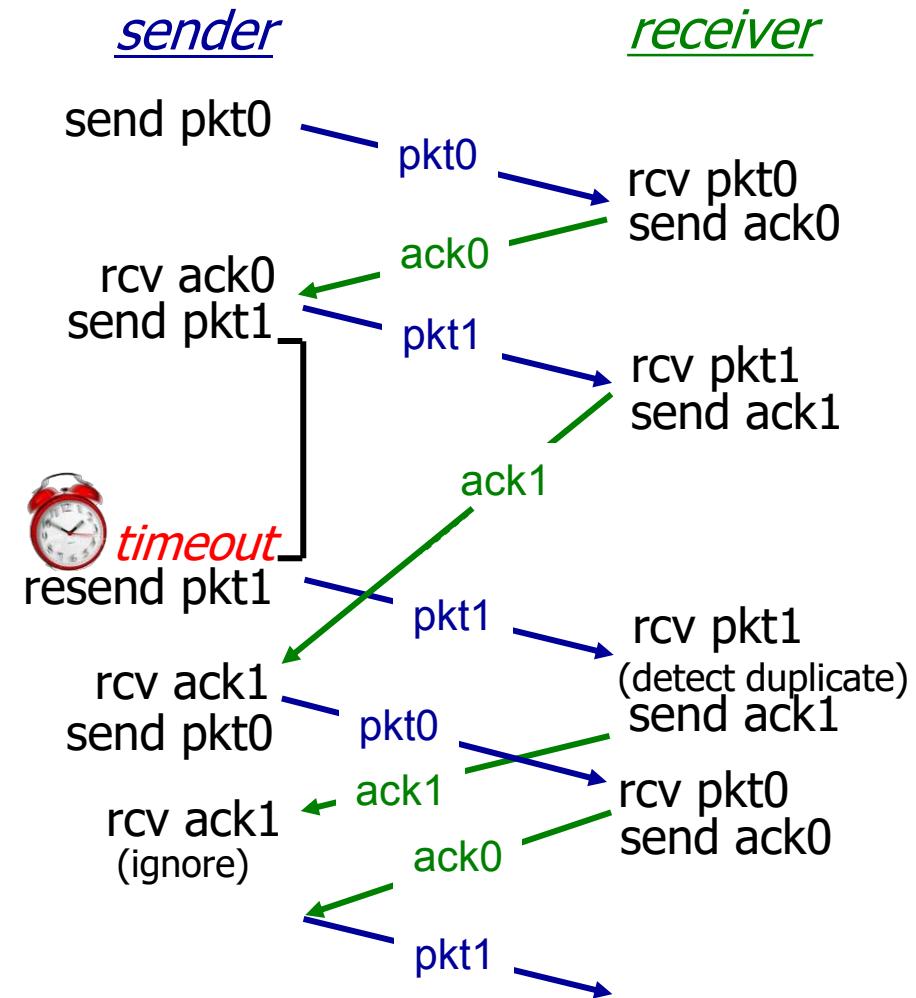
# rdt3.0 in action



# rdt3.0 in action



(c) ACK loss



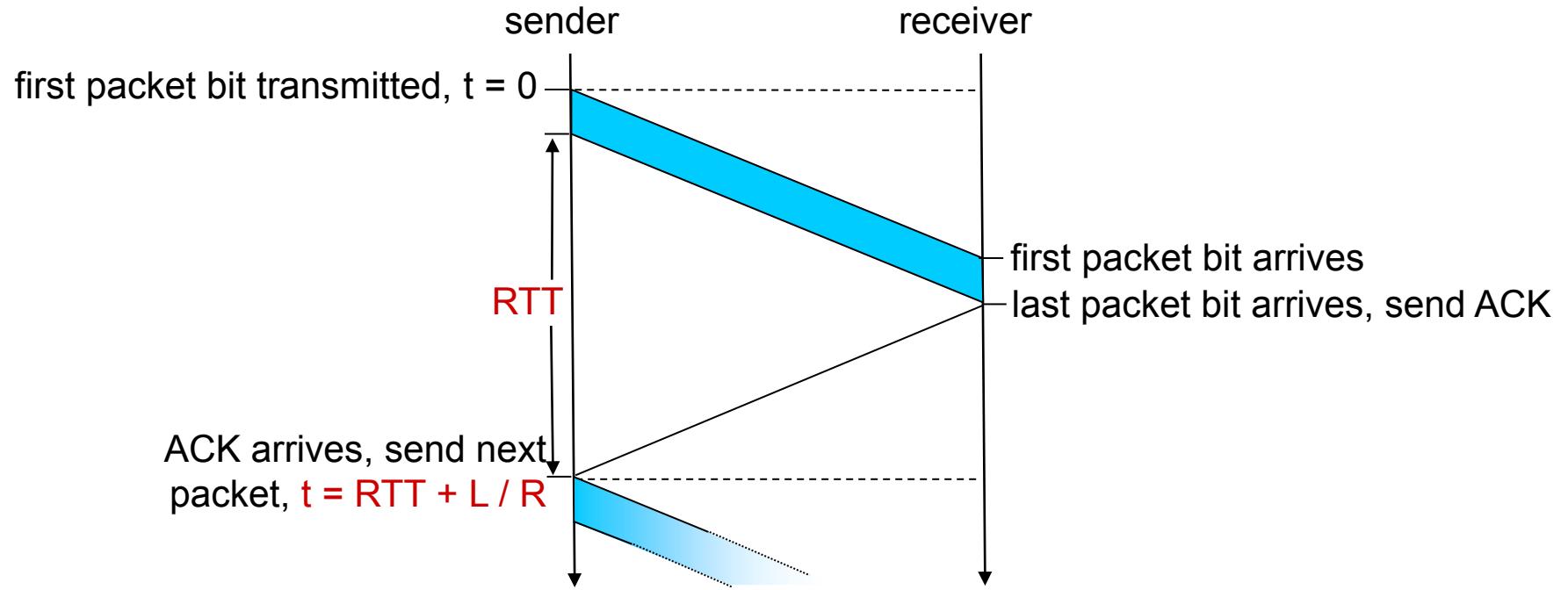
(d) premature timeout/ delayed ACK

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

- $U_{\text{sender}}$ : *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

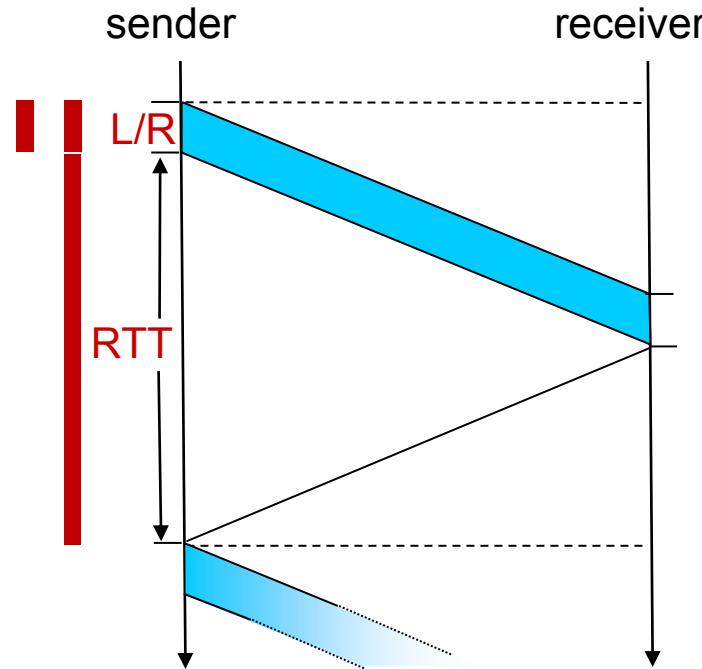
$$D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

# rdt3.0: stop-and-wait operation



# rdt3.0: stop-and-wait operation

$$\begin{aligned} U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\ &= \frac{.008}{30.008} \\ &= 0.00027 \end{aligned}$$

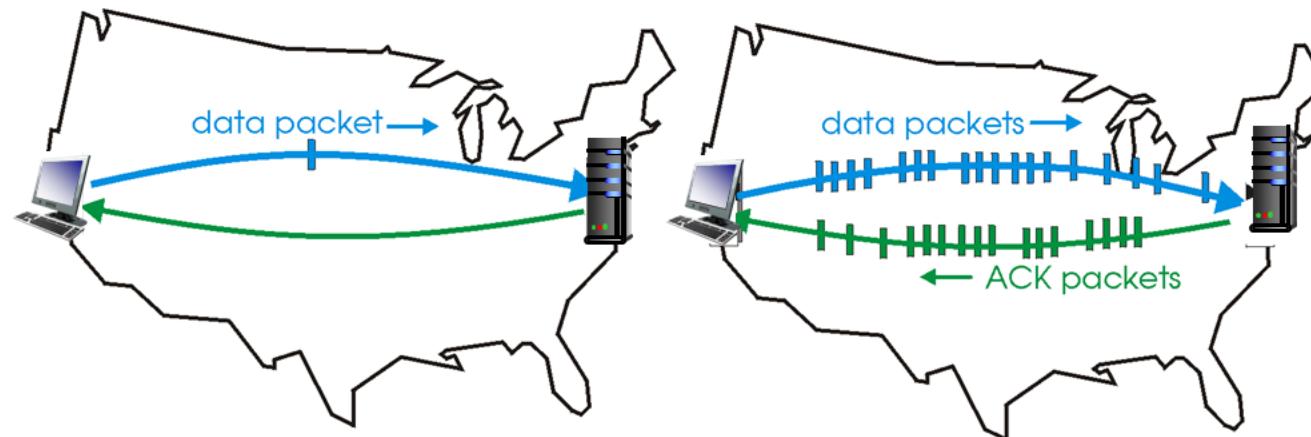


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

# rdt3.0: pipelined protocols operation

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

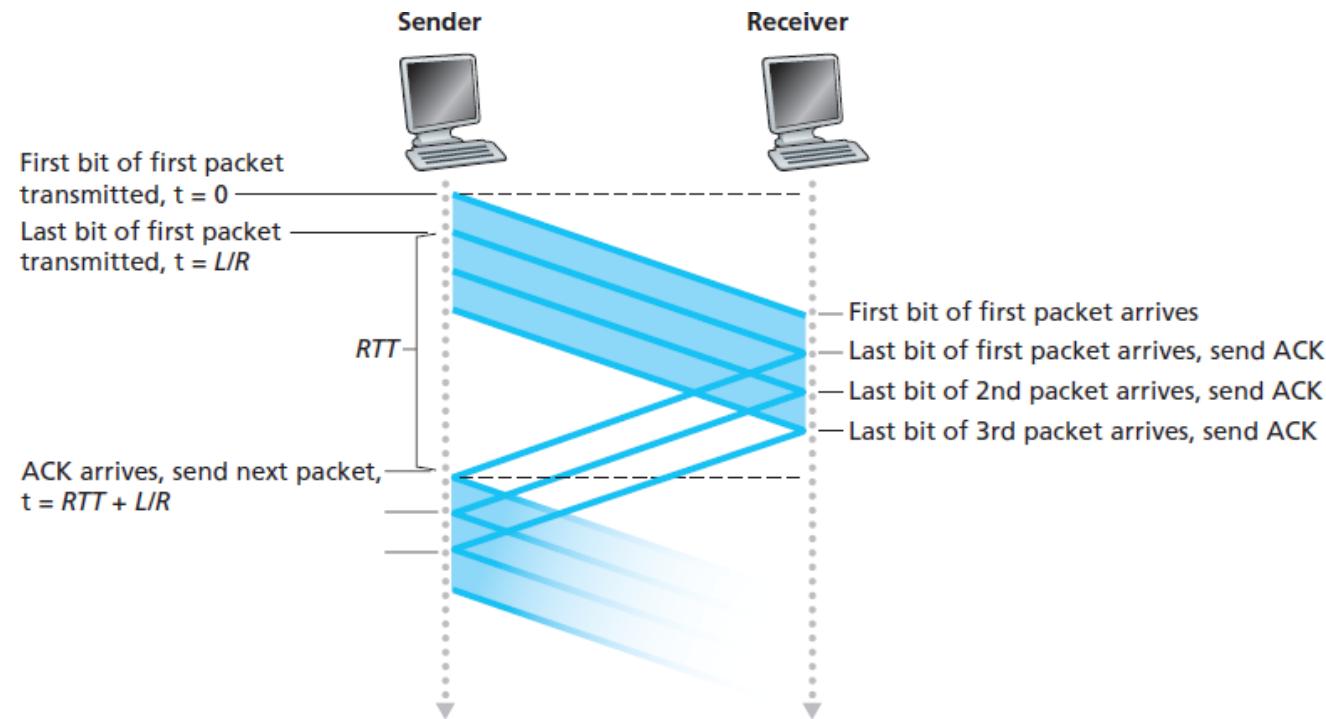
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

# Pipelining: Increased Utilization

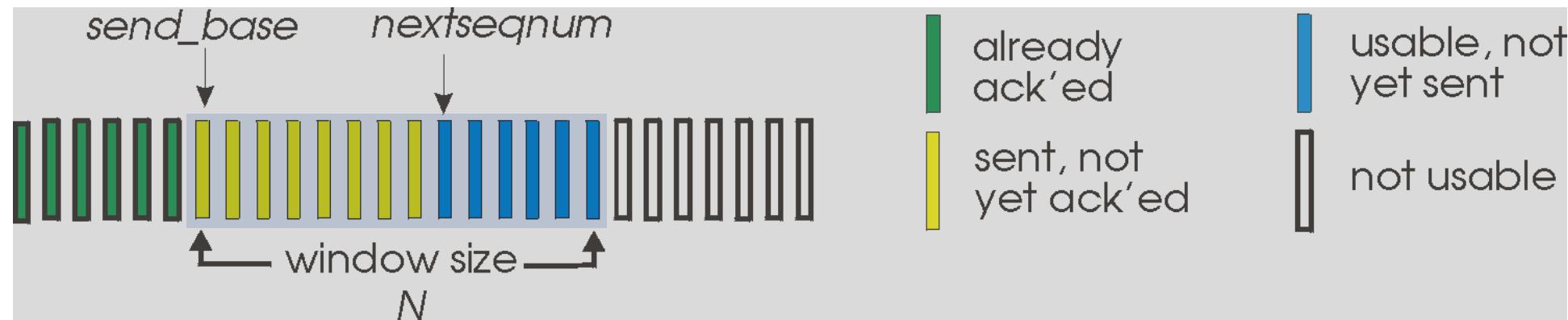


$$U_{\text{sender}} = \frac{\frac{3L}{R}}{RTT + \frac{L}{R}} = \frac{.0024}{30.008} = 0.00081$$

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

# Go-Back-N: sender

- sender: “window” of up to  $N$ , consecutive transmitted but unACKed pkts
  - $k$ -bit seq # in pkt header

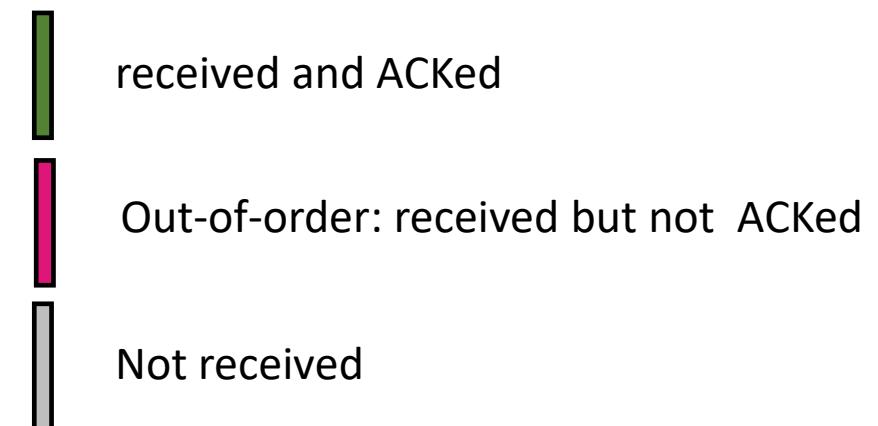
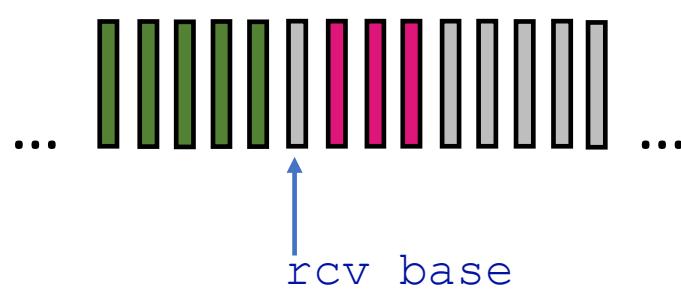


- *cumulative ACK*:  $\text{ACK}(n)$ : ACKs all packets up to, including seq #  $n$ 
  - on receiving  $\text{ACK}(n)$ : move window forward to begin at  $n+1$
- timer for oldest in-flight packet
- $\text{timeout}(n)$ : retransmit packet  $n$  and all higher seq # packets in window

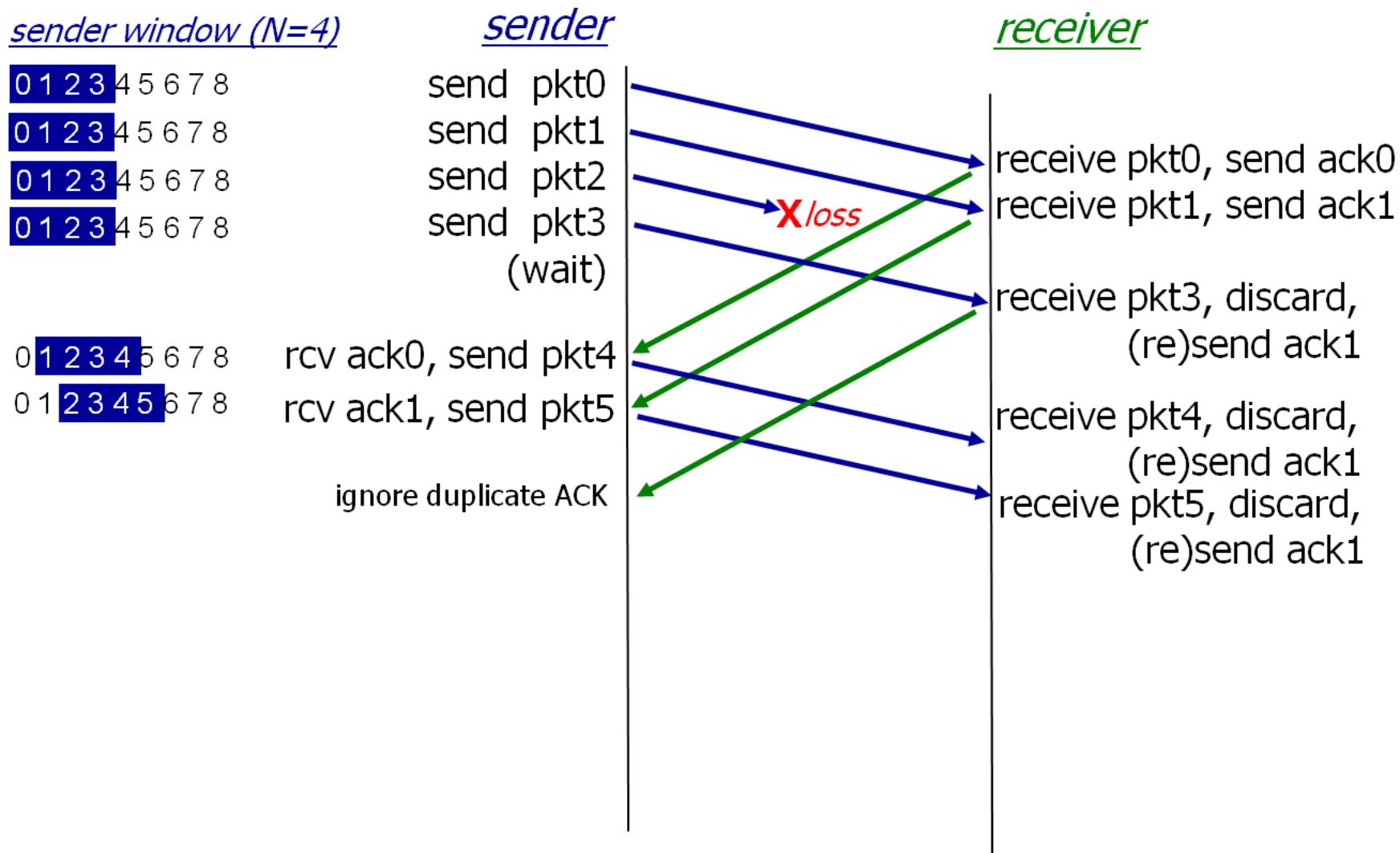
# Go-Back-N: receiver

- 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 (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

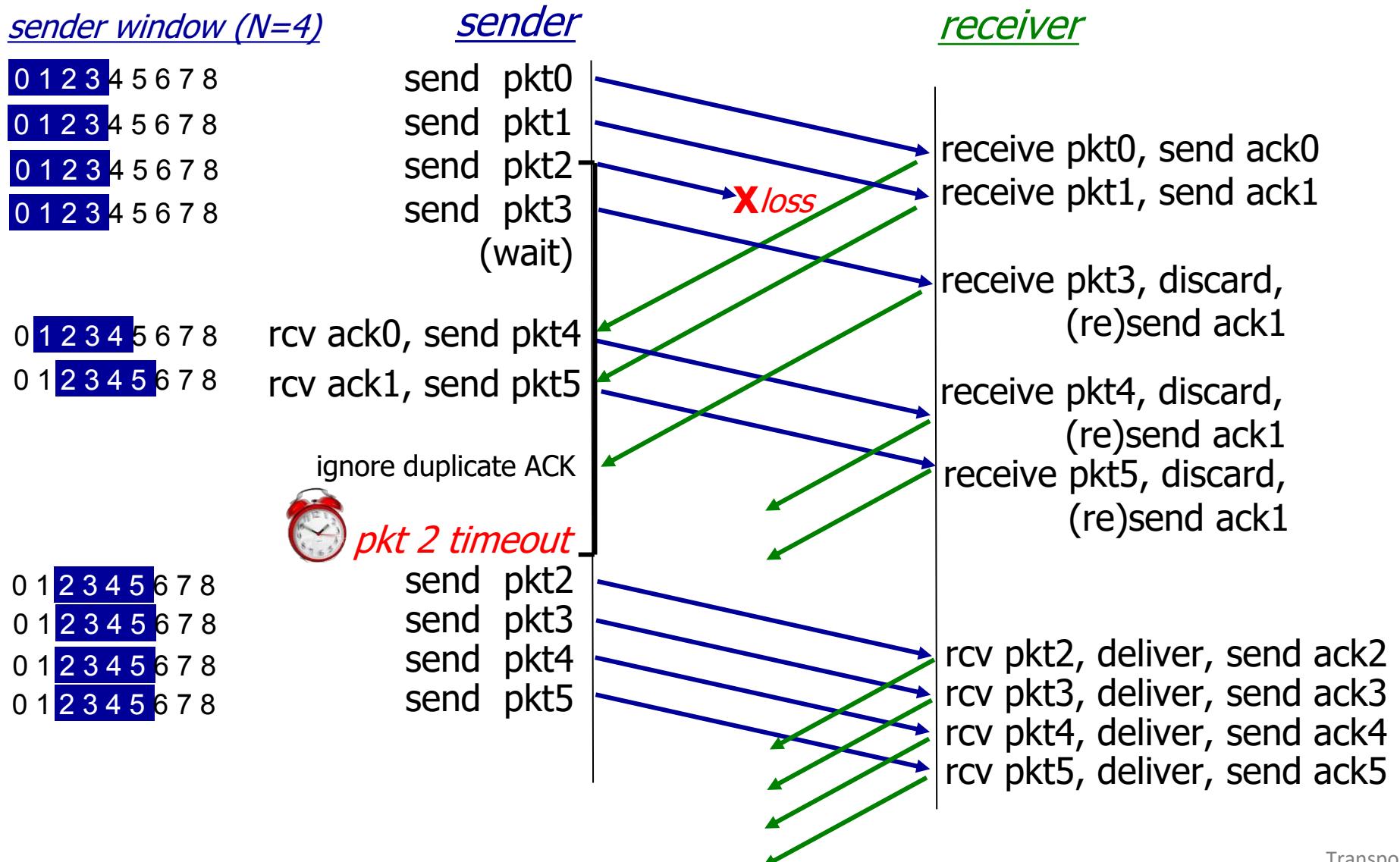
Receiver view of sequence number space:



# Go-Back-N in action



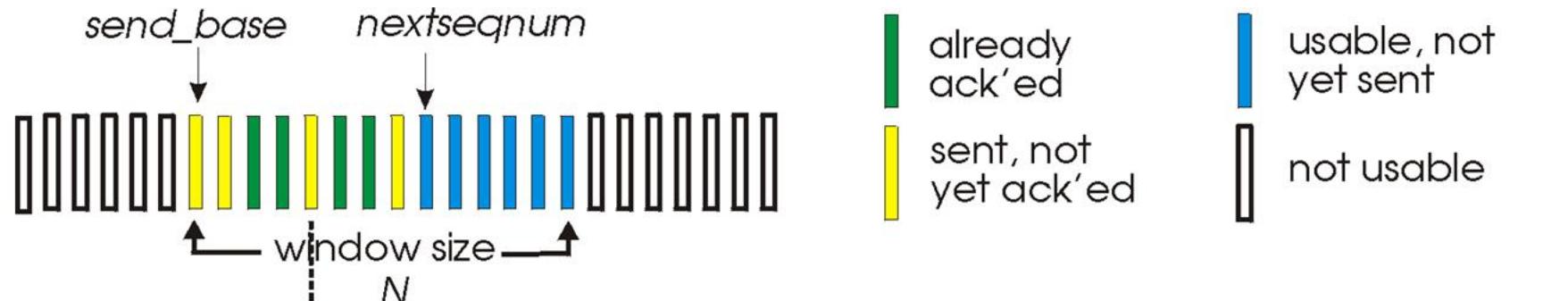
# Go-Back-N in action



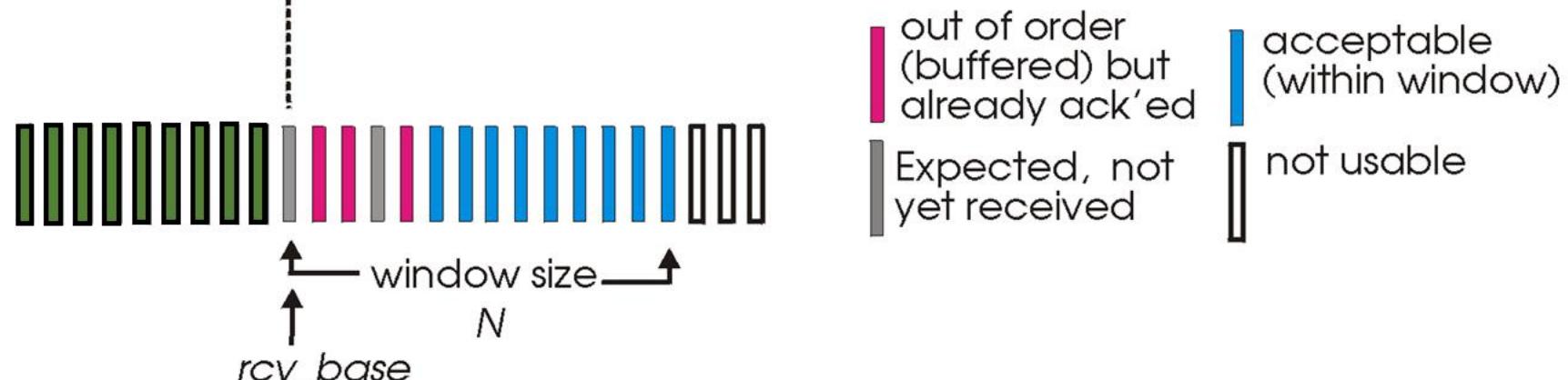
# Selective repeat: the approach

- *pipelining*: *multiple* packets in flight
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer
- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) “window” over  $N$  consecutive seq #s
    - limits pipelined, “in flight” packets to be within this window

# 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

$\text{timeout}(n)$ :

- resend packet  $n$ , restart timer

$\text{ACK}(n)$  in  $[\text{sendbase}, \text{sendbase}+N-1]$ :

- mark packet  $n$  as received
- if  $n$  smallest unACKed packet, advance window base to next unACKed seq #

# Selective repeat: sender and receiver

## receiver

**packet  $n$  in  $[rcvbase, rcvbase+N-1]$**

- send  $ACK(n)$
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

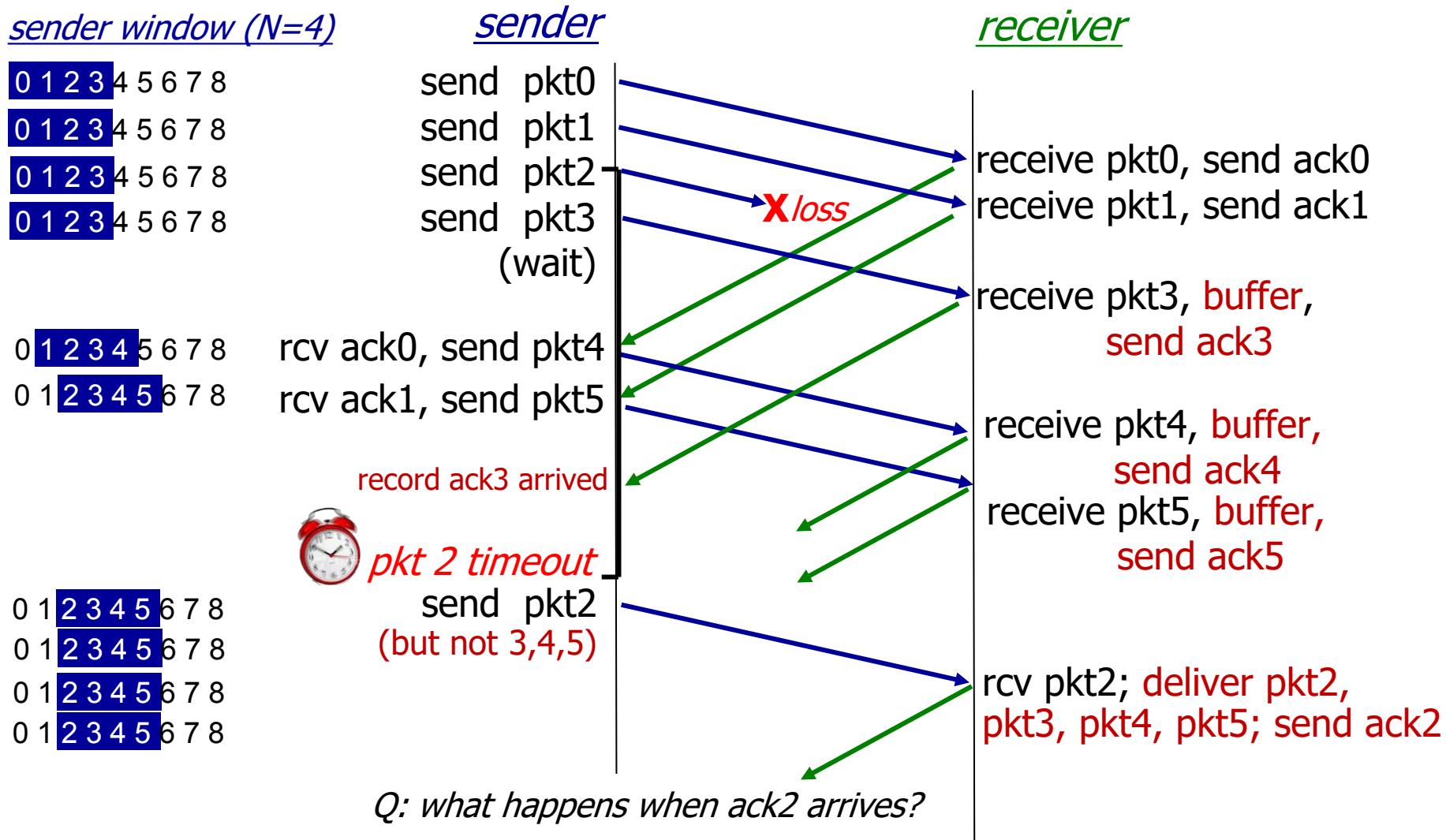
**packet  $n$  in  $[rcvbase-N,rcvbase-1]$**

- $ACK(n)$

**otherwise:**

- ignore

# Selective Repeat in action



# RDT conclusion

- We saw the use of a common set of techniques, acknowledgments, checksums, and sequence numbers, timeout and retransmit mechanisms.
- primarily in the context of the transport layer but happens at all the layers.
- This is a top 5 networking problem .

# Chapter 3: roadmap

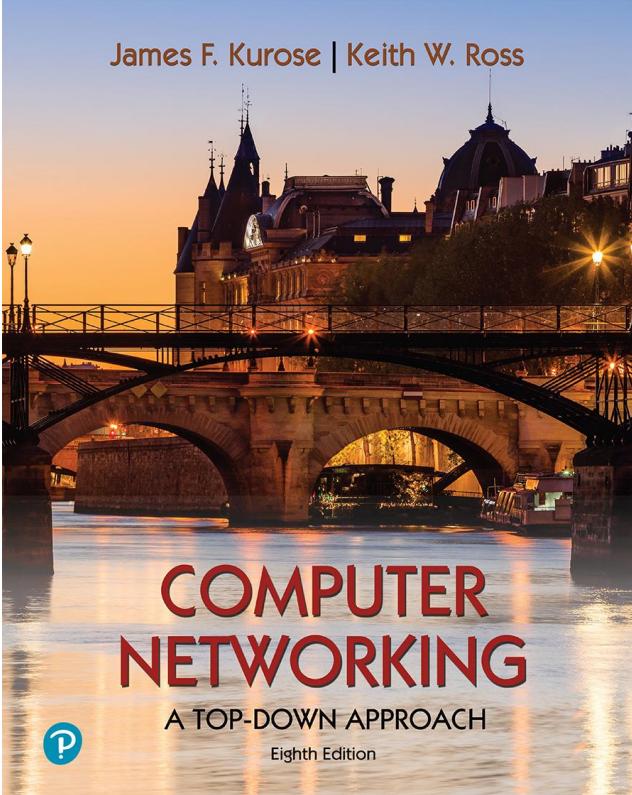
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
-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



# Important Dates

07-05-2024(Friday) – Exam 1

# Copyright Information



## *Computer Networking: A Top-Down Approach*

8<sup>th</sup> edition

Jim Kurose, Keith Ross  
Pearson, 2020

The Slides are adapted from,

All material copyright 1996-2023  
J.F Kurose and K.W. Ross, All Rights Reserved