

# CS 305: Computer Networks

## Fall 2023

### **Lecture 6: Transport Layer**

**Ming Tang**

Department of Computer Science and Engineering  
Southern University of Science and Technology (SUSTech)

# Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

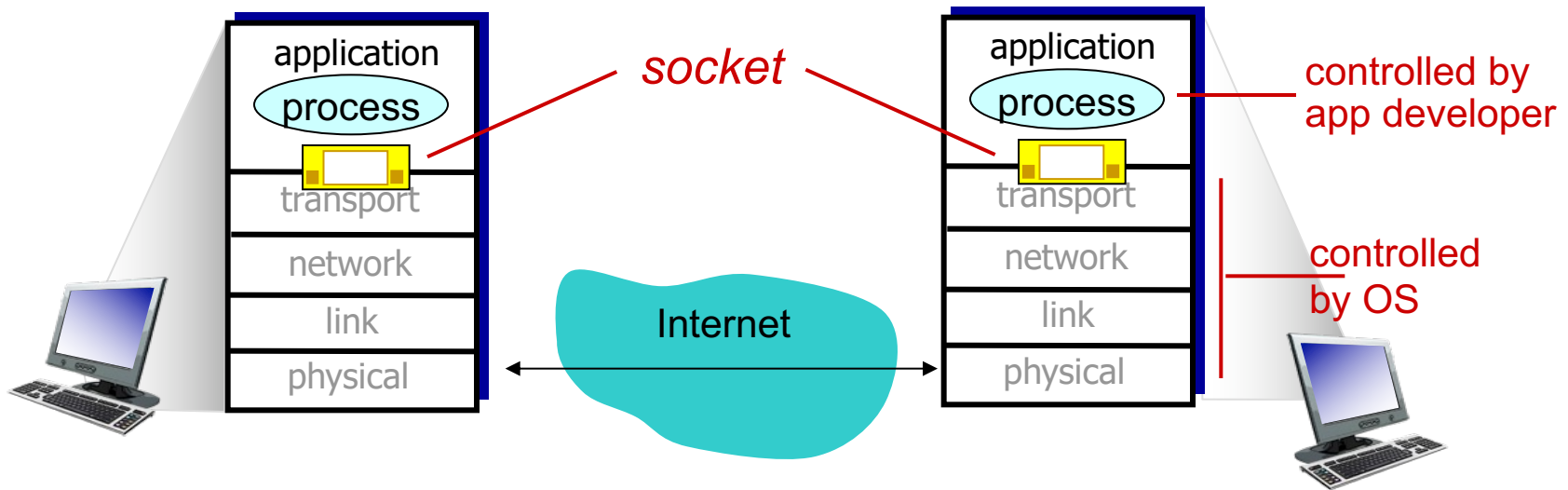
2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

# Socket programming

**Goal:** learn how to build client/server applications that communicate using sockets

**Socket:** door between application process and end-end-transport protocol



# Socket programming

**Socket programming:** how we can use socket API for creating communication between client and server processes.

Two socket types for two transport services:

- **UDP:** unreliable datagram
- **TCP:** reliable, connection-oriented

**Application Example:**

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

# Socket programming with UDP

UDP: no “connection” between client & server

- ❖ no handshaking before sending data
- ❖ sender explicitly attaches destination IP address and port number to each packet
- ❖ receiver extracts sender IP address and port number from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- ❖ UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server

# Client/server socket interaction: UDP

## server (running on serverIP)

create socket, port= x:  
`serverSocket =  
socket(AF_INET,SOCK_DGRAM)`

↓  
read UDP datagram from  
`serverSocket`

↓  
write reply to `serverSocket`  
specifying client address,  
port number

## client

create socket:  
`clientSocket =  
socket(AF_INET,SOCK_DGRAM)`

↓  
Create datagram with server IP and  
port=x; send datagram via  
`clientSocket`

↓  
read datagram from  
`clientSocket`

↓  
close  
`clientSocket`

# Example app: UDP server

## Python UDPServer

include Python's socket library → `from socket import *`

create UDP socket → `serverPort = 12000`

bind socket to local port number 12000 → `serverSocket = socket(AF_INET, SOCK_DGRAM)`  
*IPv4* (pointing to AF\_INET)  
*UDP socket* (pointing to SOCK\_DGRAM)  
*UDP socket is identified by destination IP address and port number* (pointing to SOCK\_DGRAM)

→ `serverSocket.bind(("", serverPort))`

→ `print ("The server is ready to receive")`

loop forever → `while True:`

Read from UDP socket into message, getting client's address (client IP and port) → `message, clientAddress = serverSocket.recvfrom(2048)`

→ `modifiedMessage = message.decode().upper()`

send upper case string back to this client → `serverSocket.sendto(modifiedMessage.encode(), clientAddress)`

## Example app: UDP client

# Python UDPClient

# We did not specify the client port number

either the IP address (e.g., “128.138.32.126”) or the hostname (e.g., “cis.poly.edu”)

```
from socket import *
serverName = 'hostname'
serverPort = 12000
```

## Create the client's socket

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

get user keyboard  
input \_\_\_\_\_

```
message = raw_input('Input lowercase sentence:')
```

Attach server name, port to message; send into socket

```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```

IP + portnumber

```
read reply characters from
socket into string
```

```
modifiedMessage, serverAddress =
    clientSocket.recvfrom(2048)
```

```
print out received string
and close socket
```

```
print modifiedMessage.decode()
clientSocket.close()
```



# Socket programming with TCP

## Client must contact server

- ❖ server process must first be running
- ❖ server must have created socket (door) that welcomes client's contact

## Client contacts server by:

- ❖ Creating TCP socket, specifying IP address, port number of server process
- ❖ Client TCP establishes connection to server TCP

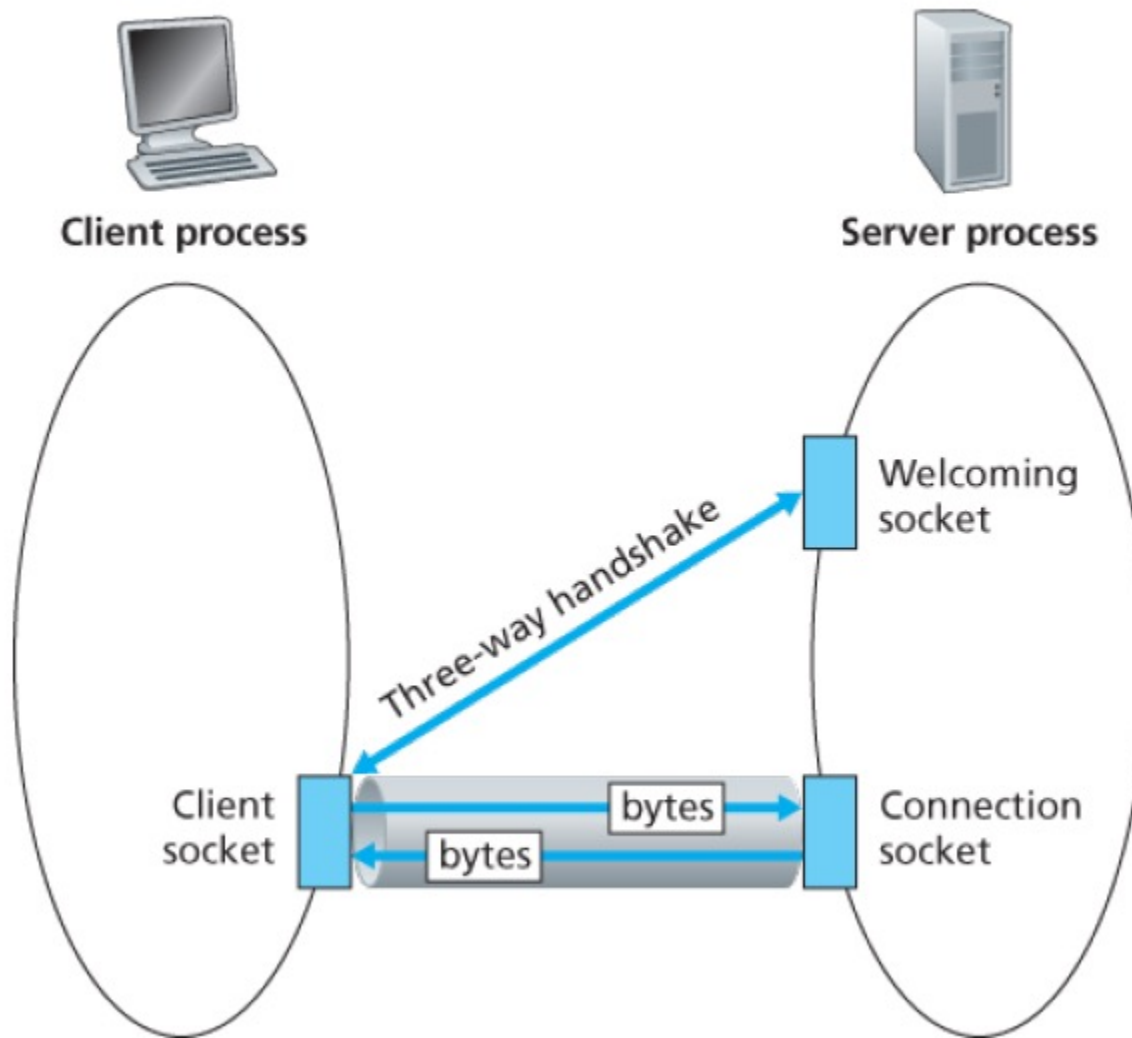
- ❖ when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

TCP socket is identified by (destination IP address, destination port number, source IP address, source port number)

## Application viewpoint:

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

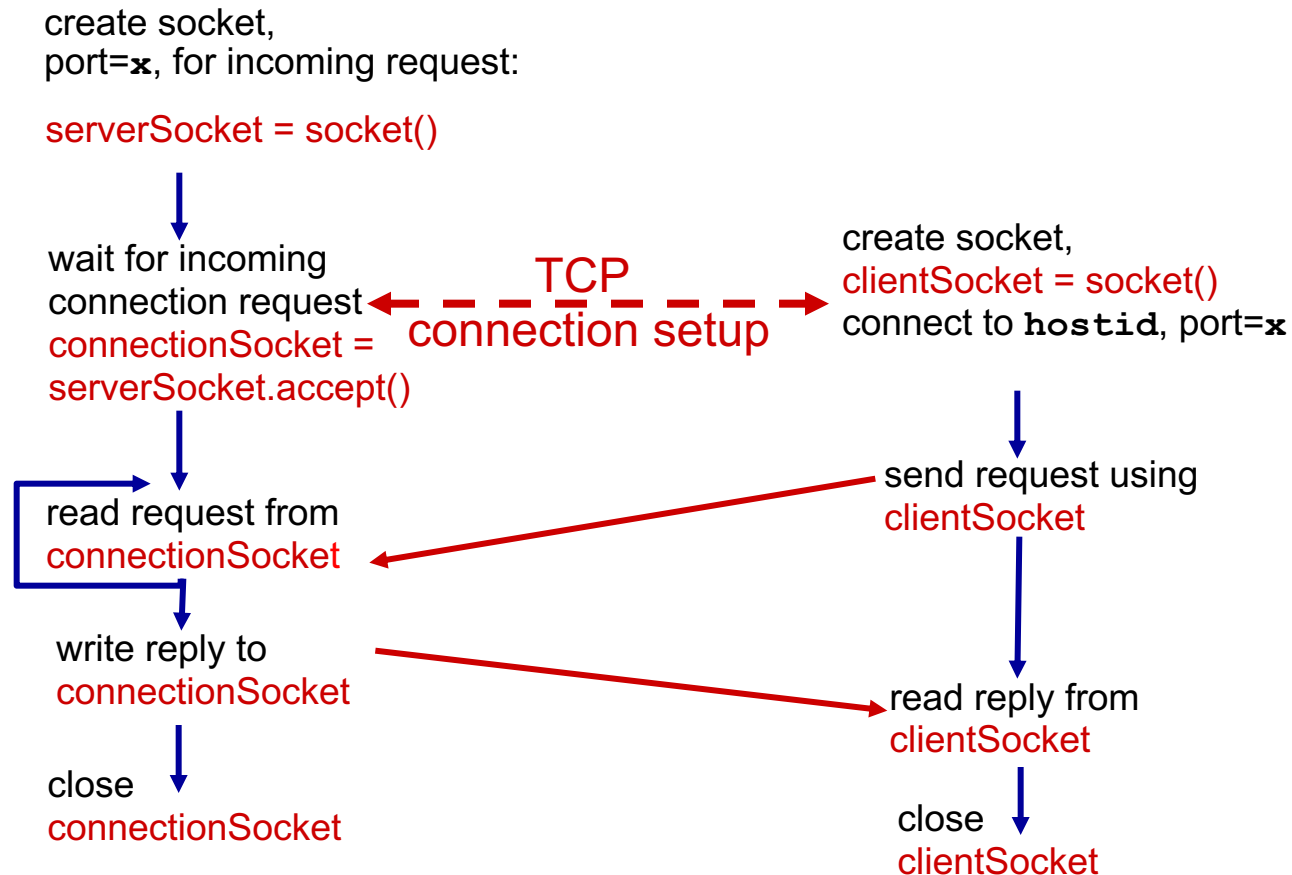
# Socket programming with TCP



# Client/server socket interaction: TCP

server (running on `hostid`)

client



# Example app: TCP server

## Python TCPServer

`connectionSocket` is identified by

- destination IP address and port number
- source IP address and port number

TCP socket

create TCP welcoming  
socket

```
from socket import *
```

```
serverPort = 12000
```

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

```
serverSocket.bind(('',serverPort))
```

server begins listening for  
incoming TCP requests

```
serverSocket.listen(1)
```

```
print 'The server is ready to receive'
```

loop forever

```
while True:
```

server waits on `accept()`  
for incoming requests, new  
socket created on return

```
connectionSocket, addr = serverSocket.accept()
```

read bytes from socket (but  
not address as in UDP)

```
sentence = connectionSocket.recv(1024).decode()
```

```
capitalizedSentence = sentence.upper()
```

```
connectionSocket.send(capitalizedSentence.
```

```
encode())
```

close connection to this client  
(but *not* welcoming socket)

```
connectionSocket.close()
```

# Example app: TCP client

## Python TCPClient

```
from socket import *
```

```
serverName = 'servername'
```

```
serverPort = 12000
```

create TCP socket for server

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

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

```
sentence = raw_input('Input lowercase sentence:')
```

No need to attach server  
name, port

```
→ clientSocket.send(sentence.encode())
```

```
modifiedSentence = clientSocket.recv(1024)
```

```
print ('From Server:', modifiedSentence.decode())
```

```
clientSocket.close()
```

# Chapter 2: summary

*our study of network apps now complete!*

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming:  
TCP, UDP sockets

# Chapter 2: summary

*most importantly: learned about protocols!*

- ❖ typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- ❖ message formats:
  - *headers*: fields giving info about data
  - *data*: info being communicated

## important themes:

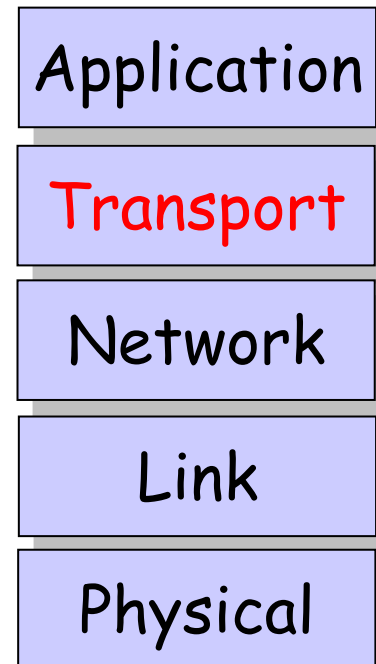
- centralized vs. distributed
- stateless vs. stateful
- reliable vs. unreliable message transfer

# Chapter 3: Transport Layer

## Our goals:

- ❖ 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





# Chapter 3 outline

## 3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

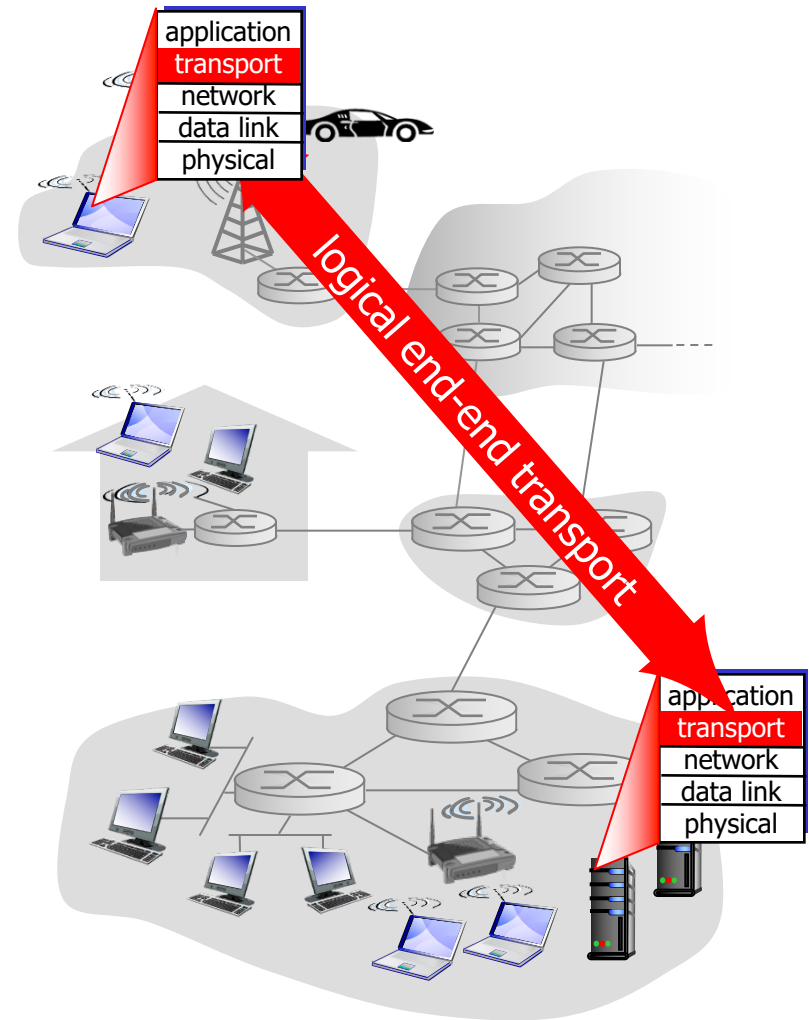
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Transport services and protocols

- ❖ Provide *logical communication* between app processes running on different hosts
- ❖ Transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- ❖ more than one transport protocol available to apps
  - Internet: TCP and UDP

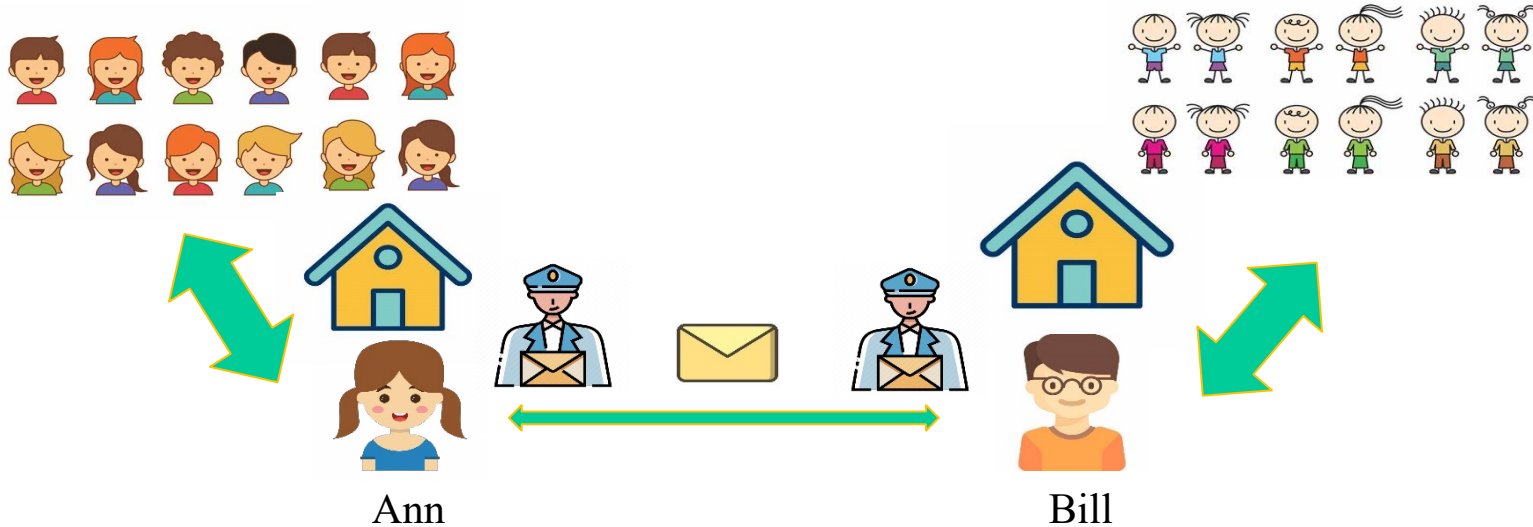


# Transport vs. network layer

- ❖ Network layer: logical communication between **hosts**
- ❖ Transport layer: logical communication between **processes**
  - relies on, enhances, network layer services



# Transport vs. network layer



## household analogy:

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

Susan and Harvey substitute for them and provide different delivery services?

# Transport vs. network layer

- ❖ The services that a transport protocol can provide are often **constrained by** the service model of the underlying network-layer protocol.
  - delay or bandwidth guarantees
- ❖ Certain services can be offered by a transport protocol **even when** the underlying network protocol **doesn't offer** the corresponding service at the network layer.
  - Reliable data transfer; security

# Internet transport-layer protocols

Network layer: Internet protocol (IP) is a best effort delivery service, unreliable

UDP: unreliable, unordered delivery:

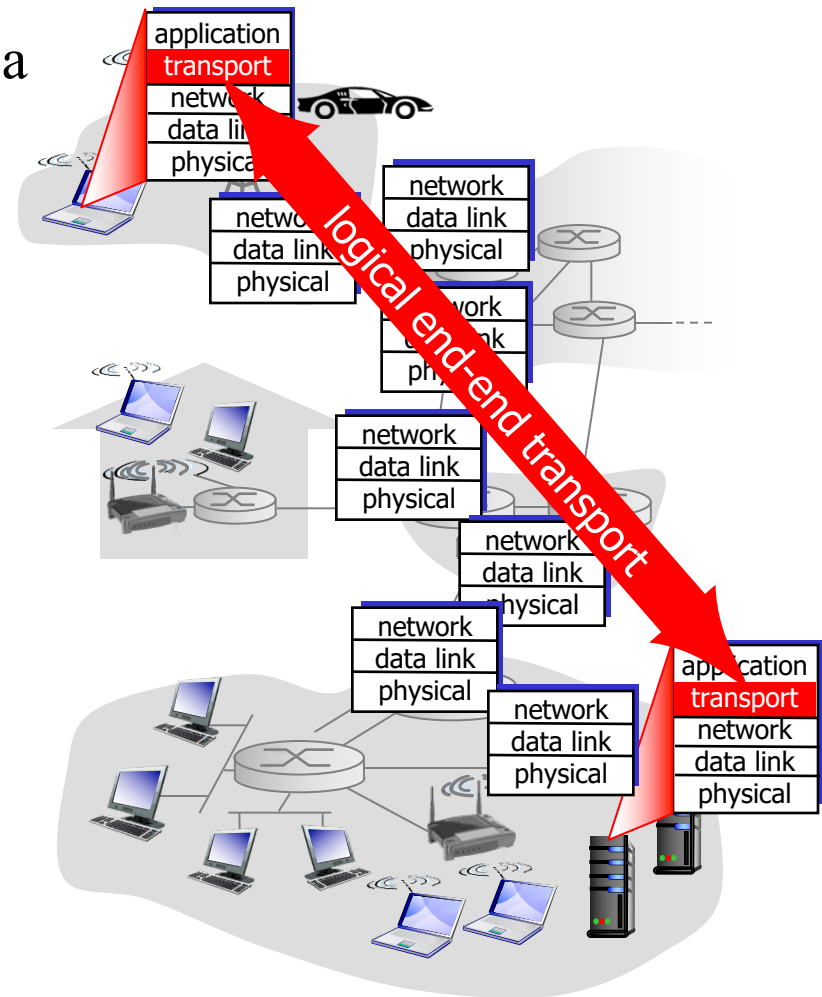
- ❖ no-frills extension
- ❖ process-to-process data delivery and error checking

TCP : reliable, in-order delivery

- ❖ congestion control
- ❖ flow control
- ❖ connection setup

Services not available:

- ❖ delay guarantees
- ❖ bandwidth guarantees



# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Multiplexing/demultiplexing

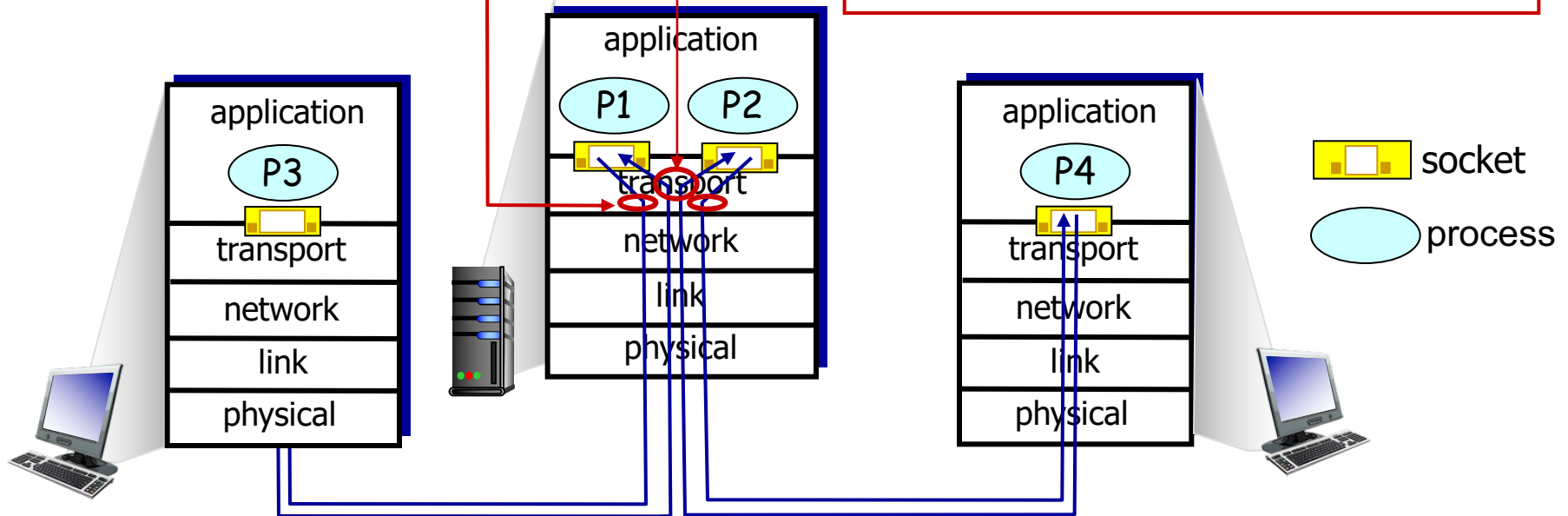
**Multiplexing and demultiplexing:** extending the host-to-host delivery service to a process-to-process delivery service for applications running on the hosts.

## **Multiplexing at sender:**

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

## **Demultiplexing at receiver:**

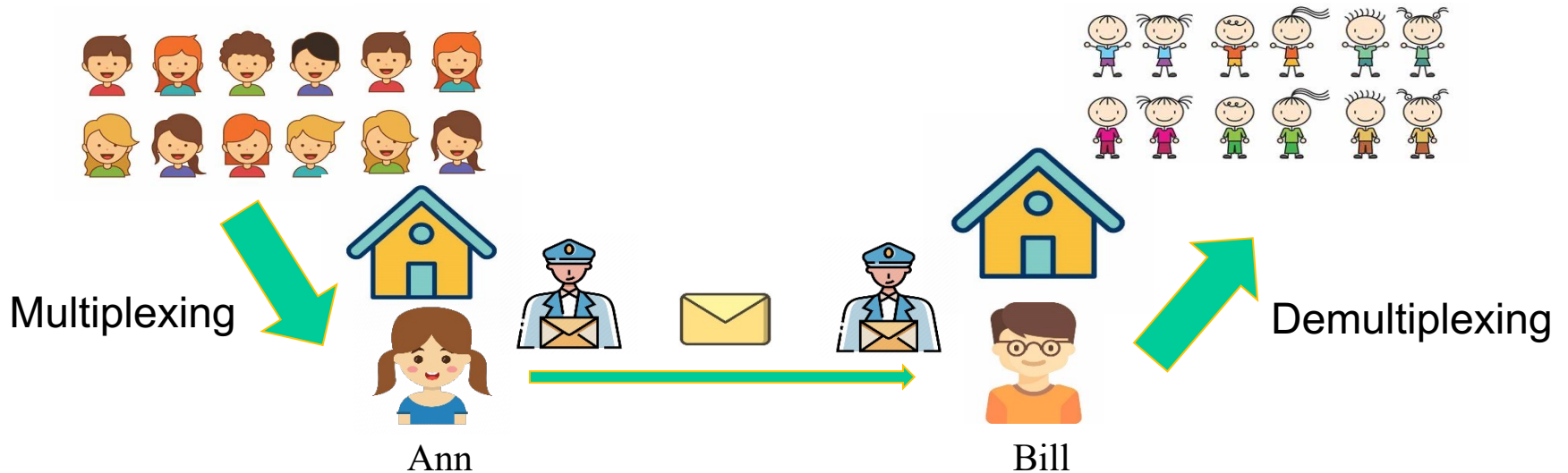
use header info to deliver received segments to correct socket



Ann and Bill example?



# Multiplexing/demultiplexing



How demultiplexing works ?

- Each child must have an identifier (e.g., name, ID)

# How demultiplexing works

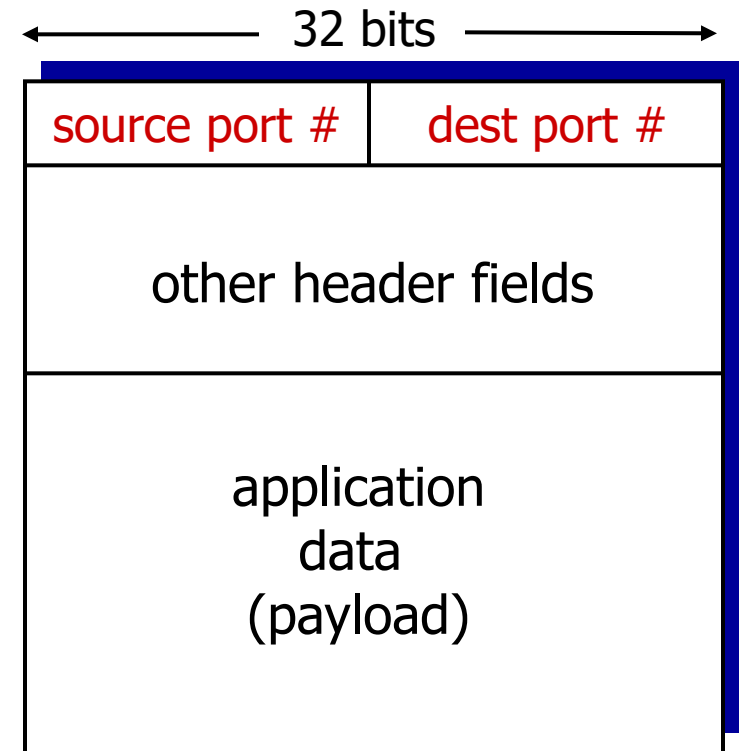
Multiplexing/demultiplexing:

- ❖ sockets have unique identifiers
- ❖ each segment have special fields that indicate the socket to which the segment is to be delivered.

Sending host: Host uses *IP addresses & port numbers* to direct segment to appropriate socket

Receiving host: Host receives IP datagrams from network layer

- ❖ each datagram has source IP address, destination IP address
- ❖ each datagram carries one transport-layer segment



TCP/UDP segment format

Port number: a 16-bit number, ranging from 0 to 65535:

- well-known port numbers: 0-1023

# Overview

- ❖ UDP: Connectionless demux
- ❖ TCP: Connection-oriented demux

# UDP: Connectionless demux

- ❖ *recall*: created socket has local port #:

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

← Automatically assigns a port number

```
clientSocket.bind(('', 19157))
```

← Optional; at the server side, usually assign port number

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

```
clientSocket.sendto(message.encode(), (serverName, serverPort))
```

UDP socket is fully identified by a **two-tuple** consisting of a destination IP address and a destination port number.

---

when host receives UDP segment:

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



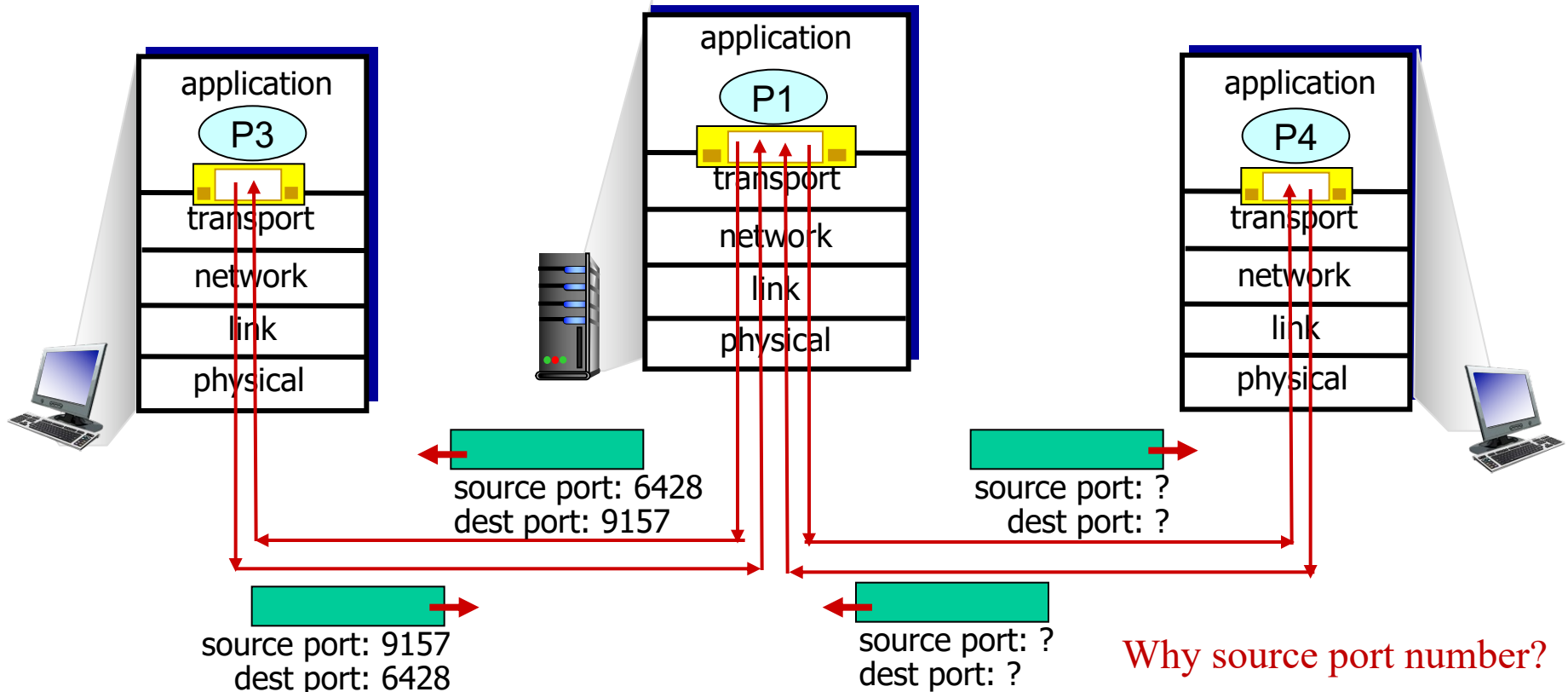
IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at destination

# Connectionless demux: example

```
mysocket2 =  
    socket(AF_INET,  
          SOCK_DGRAM)  
mysocket2.bind  
    ((' ', 9157))
```

```
serversocket =  
    socket(AF_INET,  
          SOCK_DGRAM)  
serversocket.bind  
    ((' ', 6428))
```

```
mysocket1 =  
    socket(AF_INET,  
          SOCK_DGRAM)  
mysocket1.bind  
    ((' ', 5775))
```



Why source port number?  
❖ As the “return address”

# Overview

- ❖ UDP: Connectionless demux
- ❖ TCP: Connection-oriented demux

# TCP: Connection-oriented demux

- ❖ Server **creates a welcome socket** with port no.12000  
`serversocket = socket(AF_INET, SOCK_STREAM)`  
`serversocket.bind((' ', 12000))`
- ❖ Client **connects to the server**, the request is a TCP segment with a flag bit = 1  
`clientsocket = socket(AF_INET, SOCK_STREAM)`  
`clientsocket.connect((ServerName, 12000))`
- ❖ Server **creates a new socket** to accept the connection  
`connectionsocket, addr = serversocket.accept()`

The server may maintain TCP connections with multiple clients, each has a different **connectionsocket**. How to demux?

# Connection-oriented demux

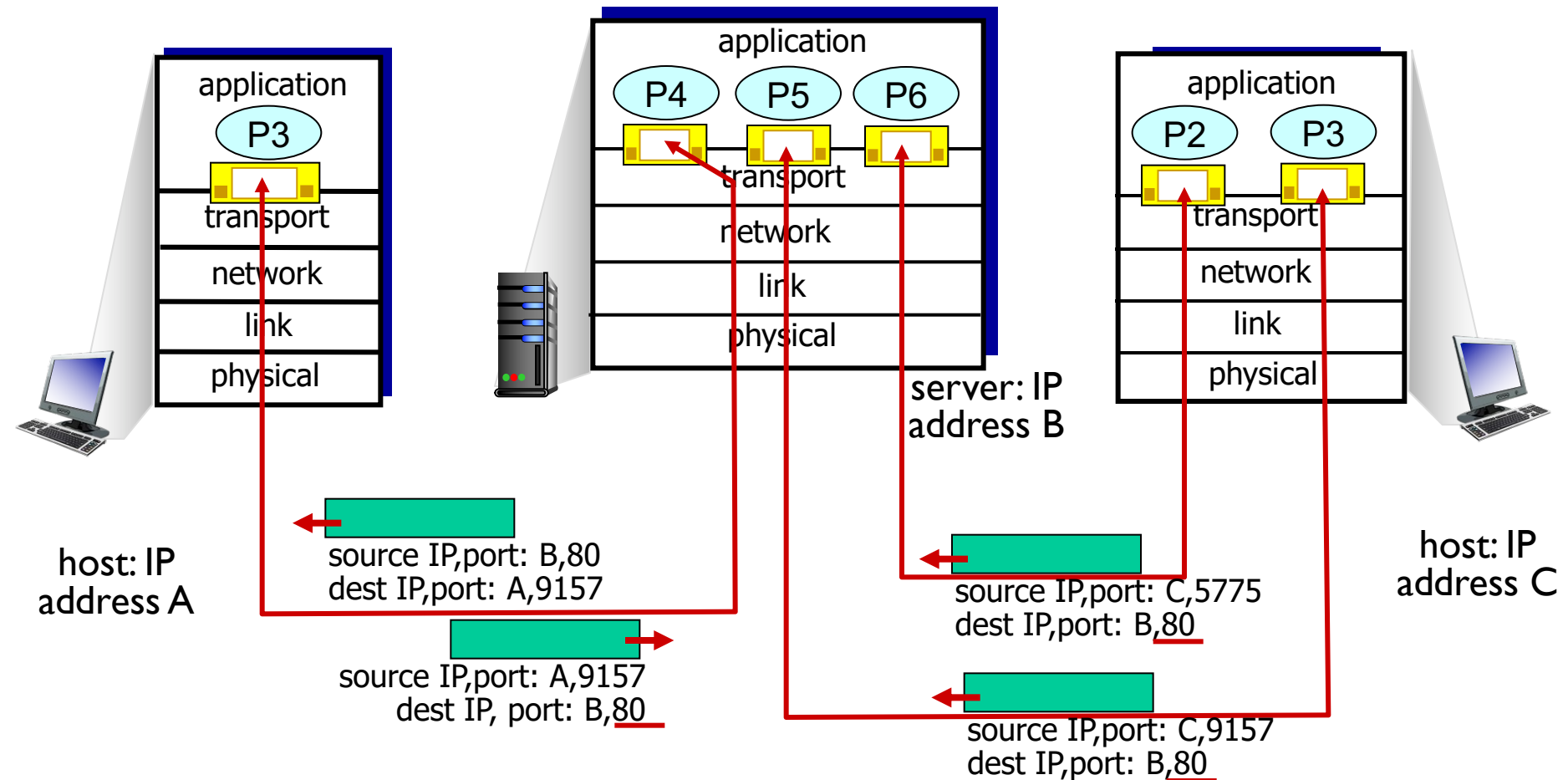
- ❖ TCP socket identified by 4-tuple:
  - source IP address, source port number, dest IP address, dest port number
- ❖ Demux: receiver uses all four values to direct segment to appropriate socket
- ❖ Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple

Web servers have different sockets for **each connecting client**

- Both the **initial connection-establishment segments** and the **segments carrying HTTP requests** will have destination port 80.
- non-persistent HTTP will have different socket for each request



# Connection-oriented demux: example

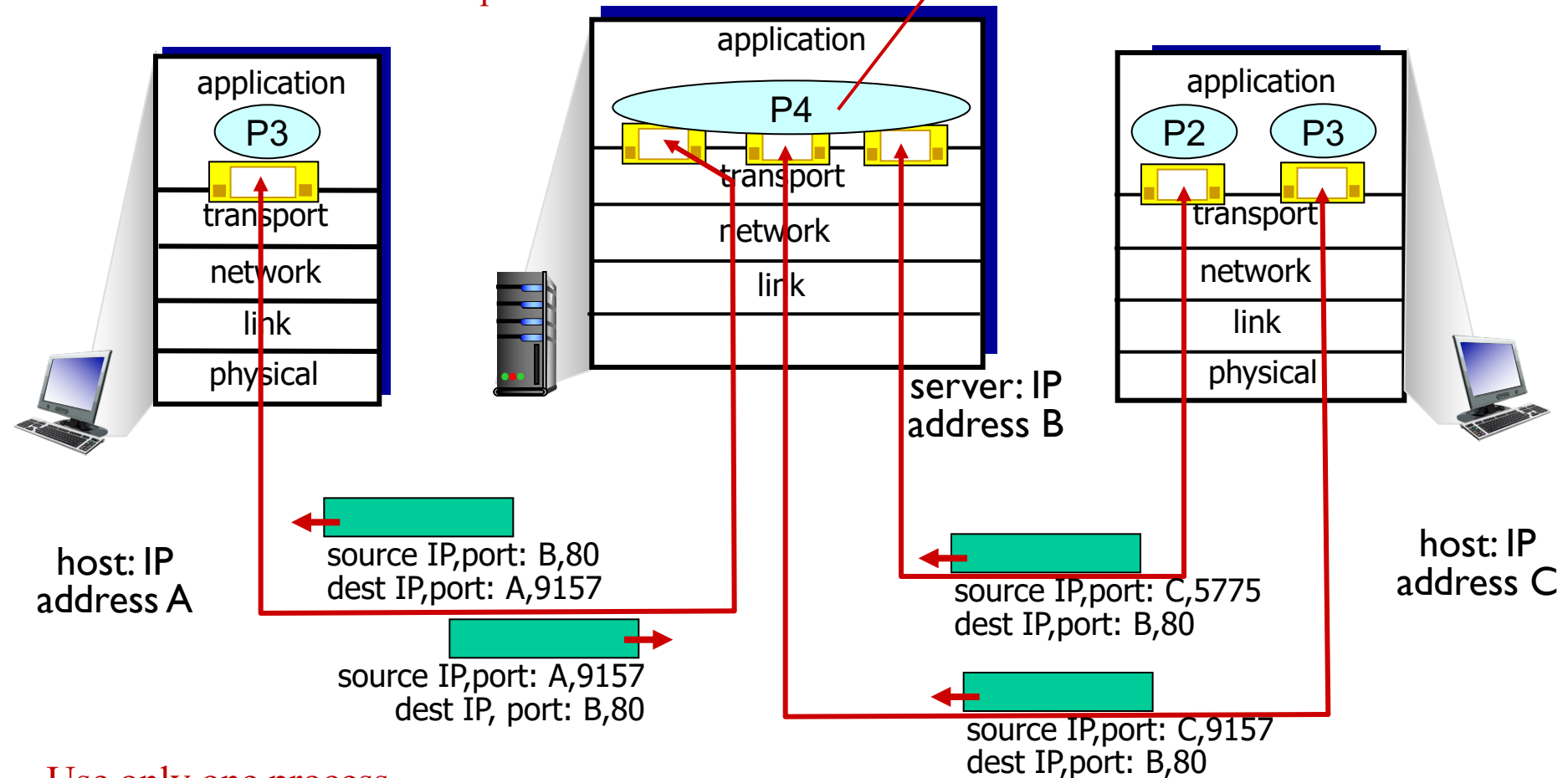


three segments, all destined to IP address: B, dest port: 80  
are demultiplexed to *different* sockets

# Connection-oriented demux: example

There is not always a one-to-one correspondence  
between connection sockets and processes

threaded server



Use only one process

- ❖ Create a new thread with a new connection socket for each new client connection.
- ❖ A thread can be viewed as a lightweight subprocess.

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# UDP: User Datagram Protocol

- ❖ “No frills,” “bare bones” Internet transport protocol
  - Multiplexing/demultiplexing; light error checking
- ❖ “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
  - No congestion control

## Advantage?

- No congestion control: Immediately pass the segment to network layer
- No connection-establish delay
- No connection state: server can support more clients
- Smaller packet overhead

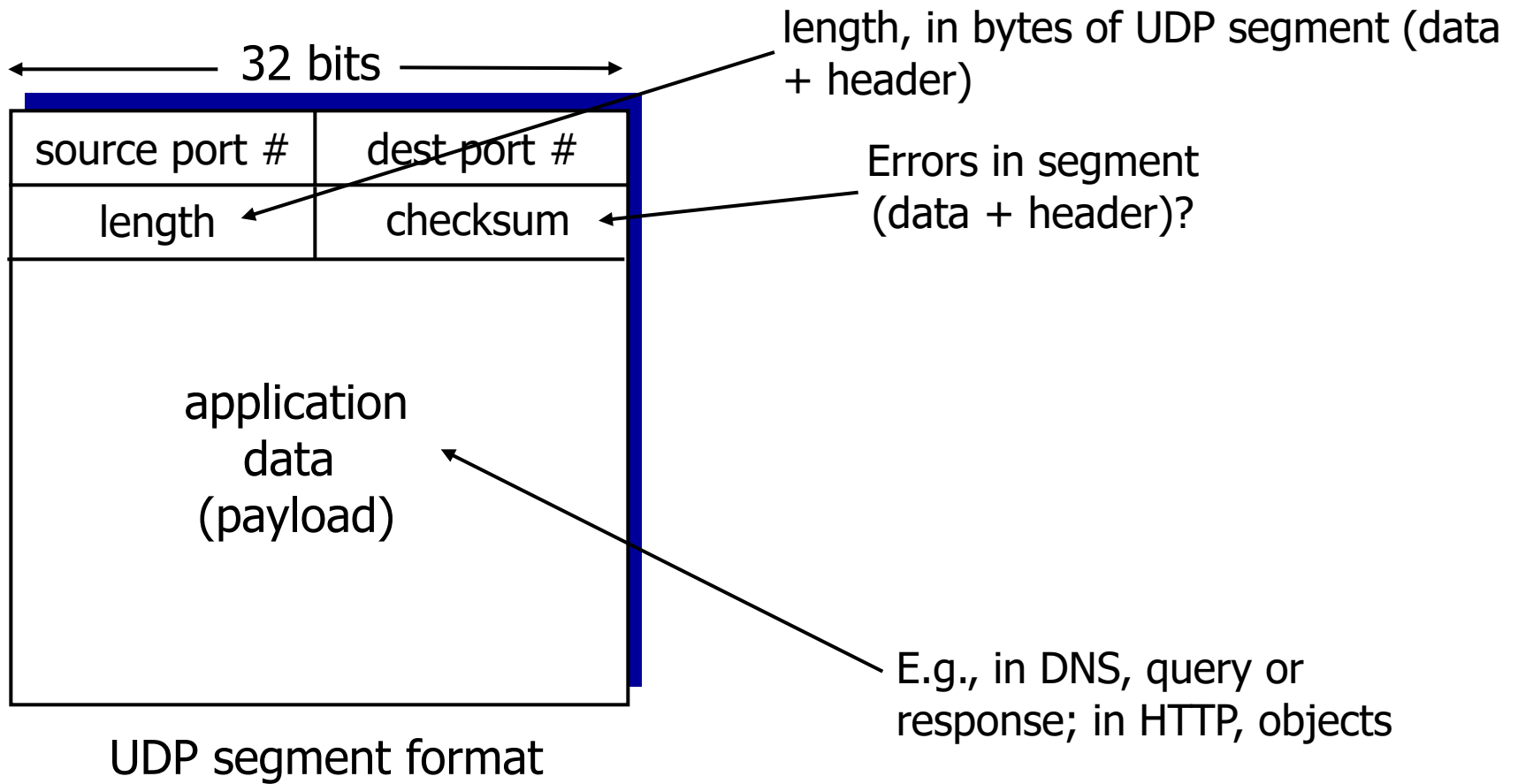
## Disadvantage?

- No congestion control: congestion, overflow, fairness
- Not reliable

# UDP: User Datagram Protocol

- ❖ UDP is used in:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
- ❖ reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

# UDP: segment header



# UDP checksum

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment  
(from source to destination)

## **Sender:**

- ❖ treat segment contents, including header fields, as sequence of 16-bit integers
- ❖ checksum: **1s complement of the sum** of segment contents
- ❖ sender puts checksum value into UDP checksum field

## **Receiver:**

- ❖ check the sum of the segment
  - All bits are equal to 1 - no error detected. *But maybe errors nonetheless?* More later ....
  - Otherwise: error detected

# Internet checksum: example

At the sender side, **determine the check sum** of the following three 16-bit words


0110011001100000  
0101010101010101  
1000111100001100

**The sum of first two** of these 16-bit words is

0110011001100000  
0101010101010101  
                      
1011101110110101

**Adding the third word** to the above sum gives

1011101110110101  
1000111100001100  
                      
0100101011000001  
10100101011000001  
                      
0100101011000010

 **Wrap around**

**Note:** when adding numbers, a carryout from the most significant bit needs to be added to the result

The 1s complement is obtained **by converting all the 0s to 1s and converting all the 1s to 0s**.

**Checksum: 1011010100111101**

**Receiver Side ?**



# Internet checksum: example

At the receiver side, all four 16-bit words are added, including the checksum:

- ❖ If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111.
- ❖ If one of the bits is a 0, then we know that errors have been introduced into the packet.

Check the sum of the segment

- All bits are equal to 1 - **no error detected**. *But maybe errors nonetheless? More later ....*
- Otherwise: error detected

# Internet checksum: example

## Why UDP provides a checksum?

- no guarantee that all the links provide error checking
- bit errors could be introduced when segments are in memory
- “functions placed at the lower levels may be redundant or of little value when compared to the cost of providing them at the higher level.”

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

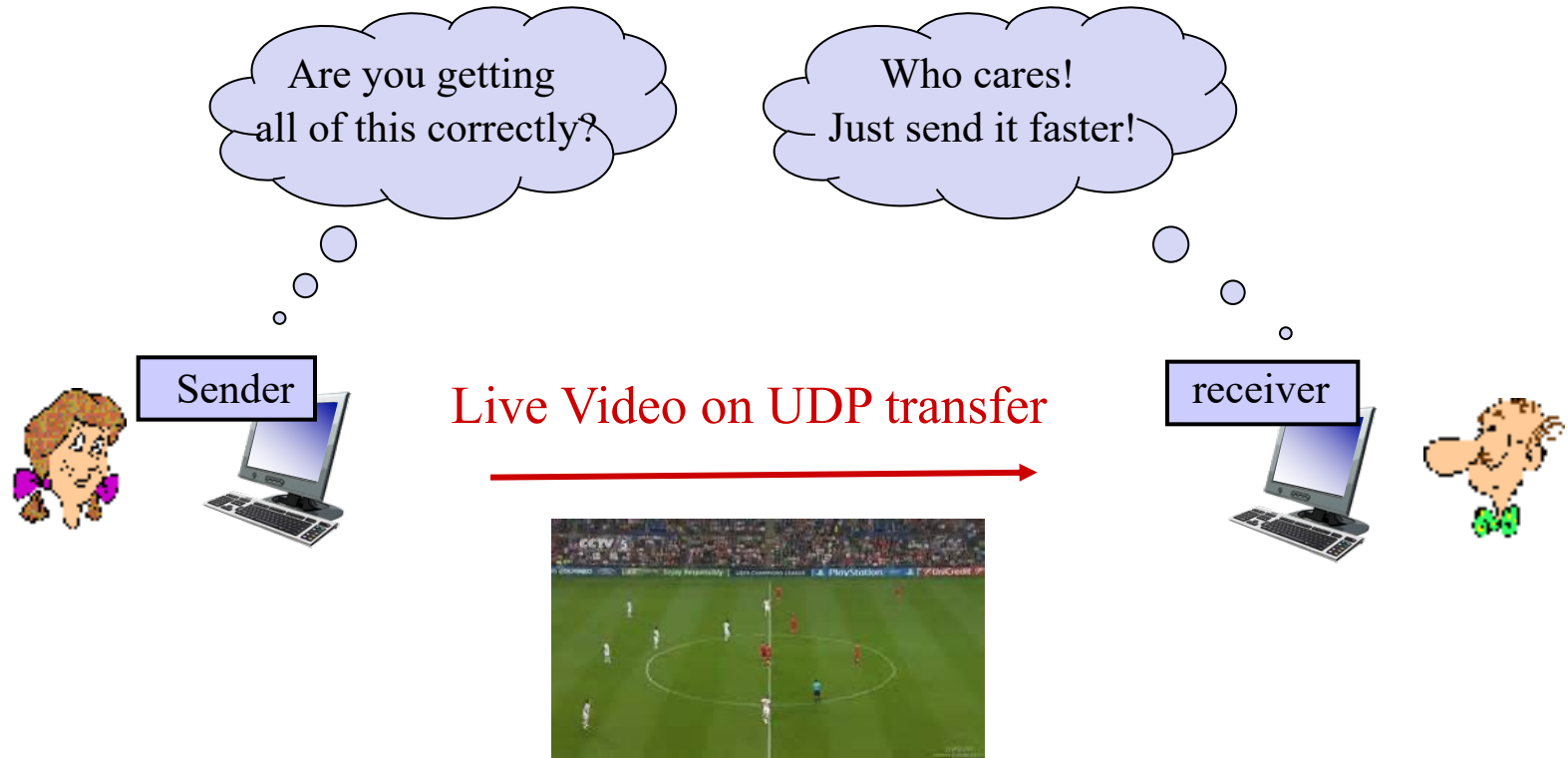
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

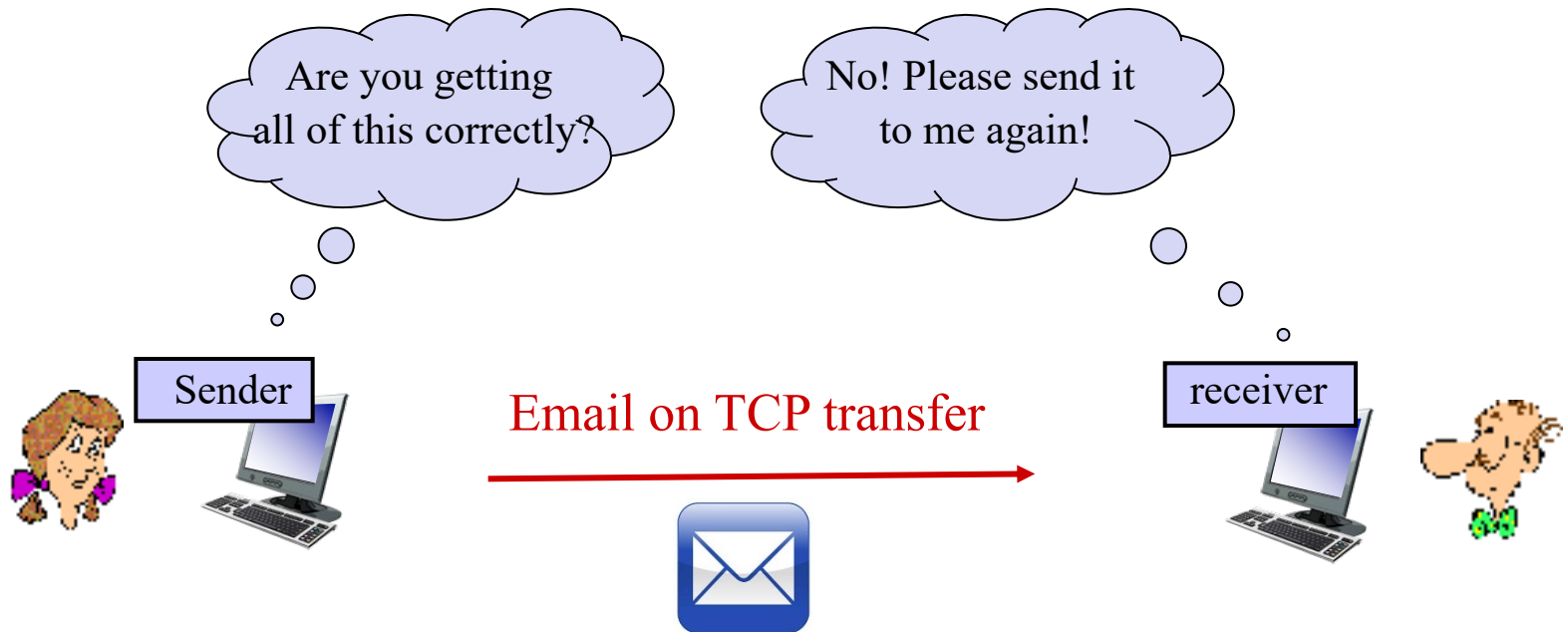
# UDP Transfer

- ❖ UDP cannot guarantee reliable data transfer
- ❖ But, it's faster!



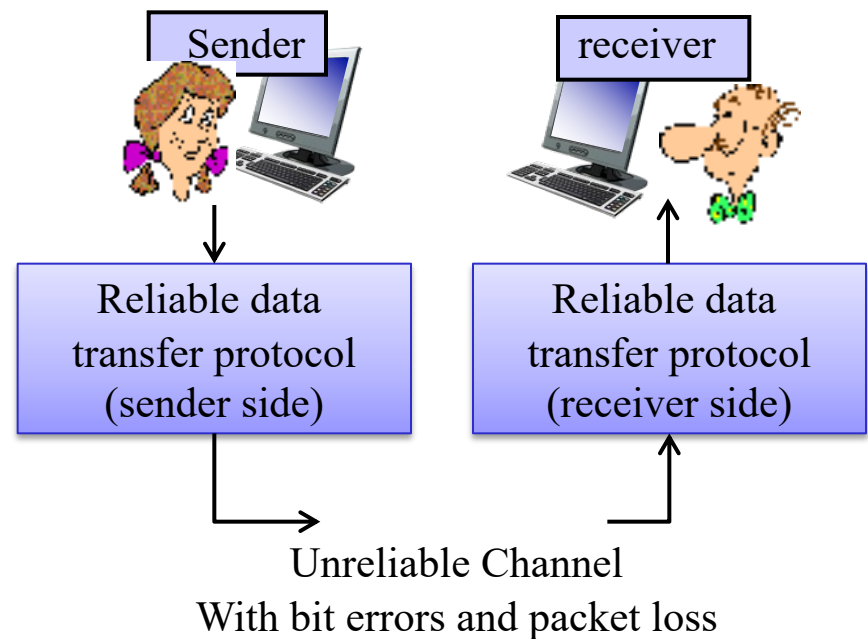
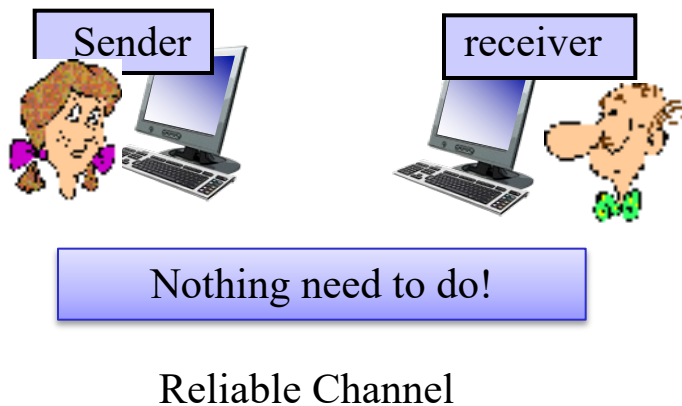
# TCP Transfer

- ❖ TCP can guarantee reliable data transfer
- ❖ But, it's slower and more complex!

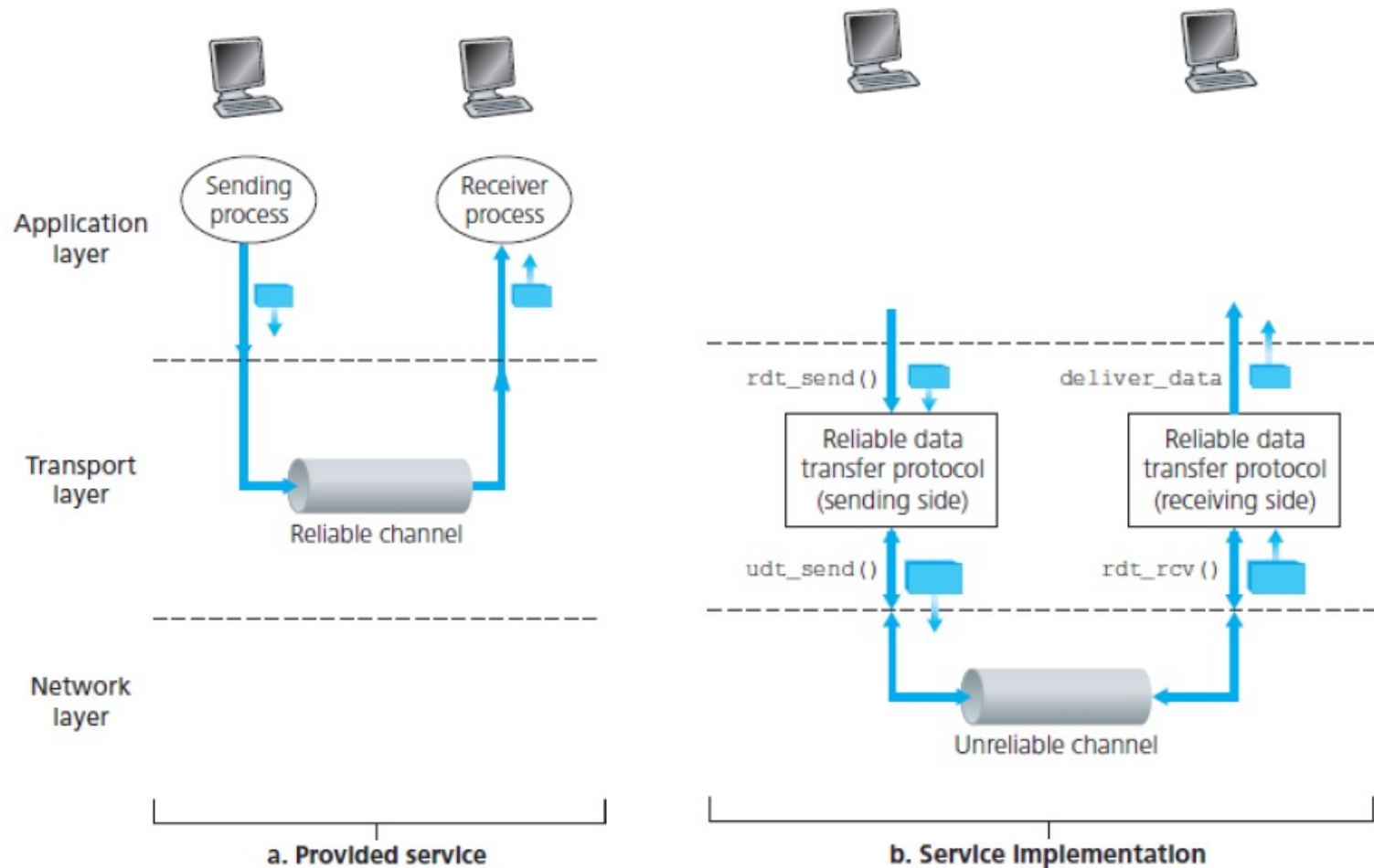


# Reliable Data Transfer (rdt)

- ❖ In top-10 list of important networking topics!
- ❖ Reliable data transfer over **unreliable channel**:
  - Bit flip, lost, out-of-order
  - In this section, assume unreliable channel not reorder packets



# Reliable Data Transfer (rdt)

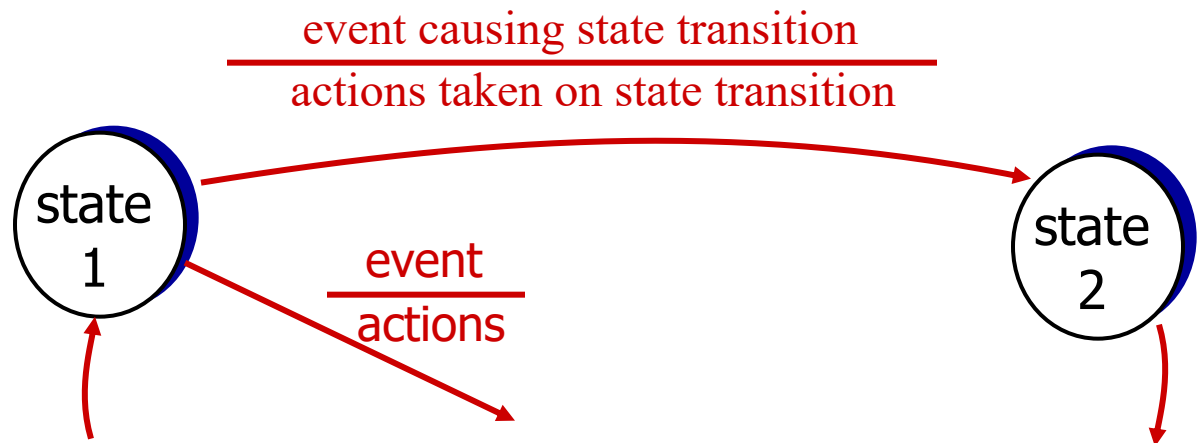


# Reliable data transfer: getting started

We'll:

- ❖ **Incrementally** develop sender, receiver sides of **reliable data transfer protocol (rdt)**
- ❖ Consider only **unidirectional data transfer**
  - but control info will flow on both directions!
- ❖ Use finite state machines (FSM) to specify sender, receiver

**state:** when in this “state”  
next state uniquely  
determined by next  
event





# Overview

---

## Roadmap:

- ❖ Perfectly reliable channel: rdt1.0
- ❖ Channel with bit error:
  - bit error in packet: rdt 2.0
  - bit error in ACK: 2.1
  - NAK-free: 2.2
- ❖ Lossy channel: rdt 3.0

## Summary of Techniques

- Checksum
- Sequence number
- ACK packets
- Retransmission
- Timeout

# rdt1.0: reliable transfer over a reliable channel

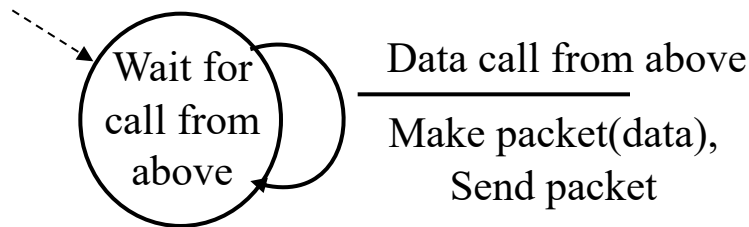
## ❖ Underlying channel **perfectly reliable**

- no bit errors
- no loss of packets

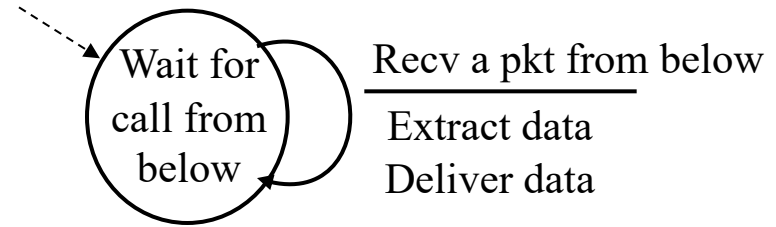


## ❖ **Rdt 1.0:**

- sender sends data into underlying channel
- receiver reads data from underlying channel
- Reliable channel, no need for feedback (no control message)

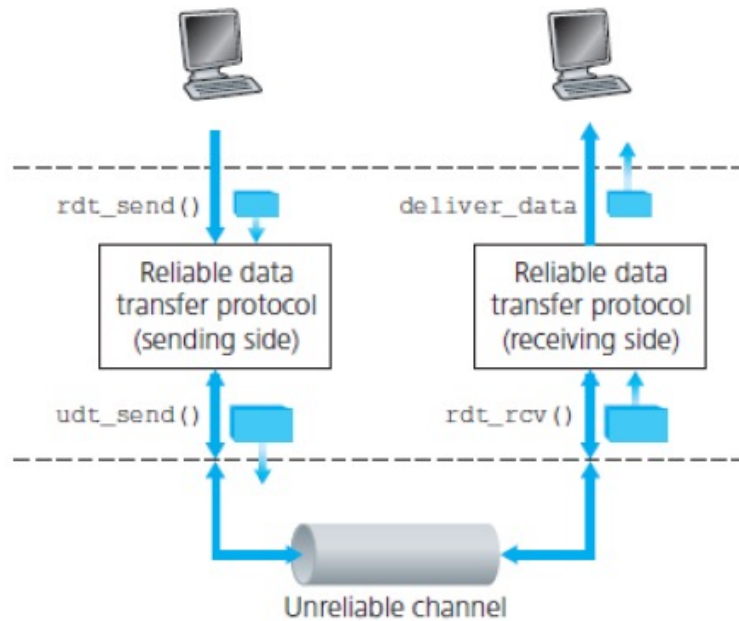


sender

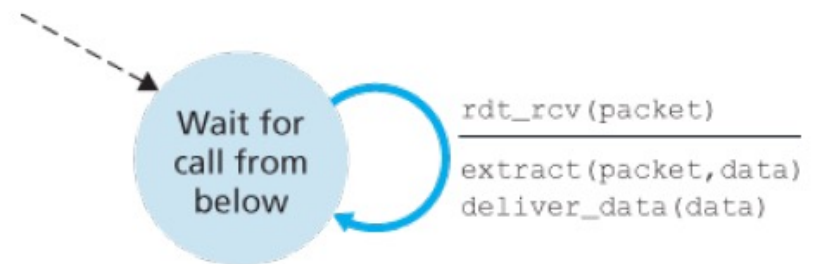


receiver

# rdt1.0: reliable transfer over a reliable channel



sender

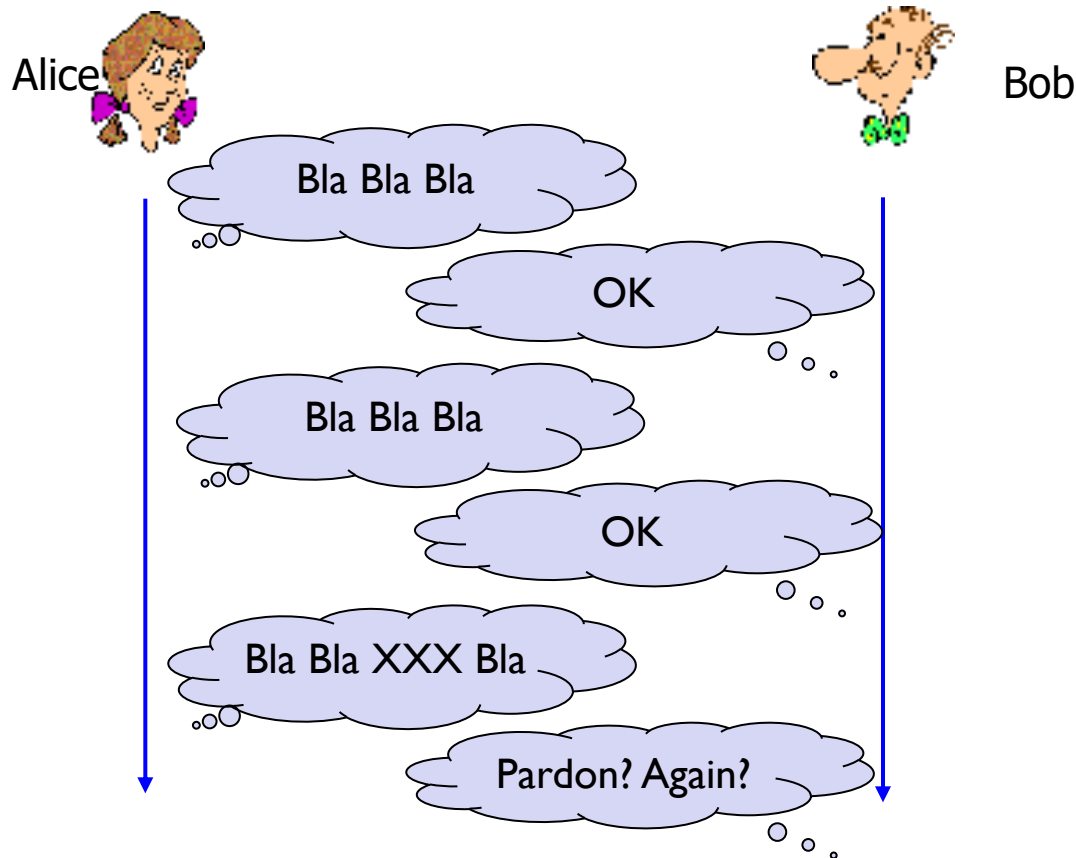


receiver

# rdt2.0: channel with bit errors

- ❖ Underlying channel may **flip bits** ( $0 \rightarrow 1$ ) in packet

How do humans recover from “errors” during conversation?



# rdt2.0: channel with bit errors

- ❖ Underlying channel may **flip bits** ( $0 \rightarrow 1$ ) in packet

How do humans recover from “errors” during conversation?

- ❖ **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
- ❖ new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender
  - retransmission

# rdt2.0: channel with bit errors

## ❖ Key mechanisms:

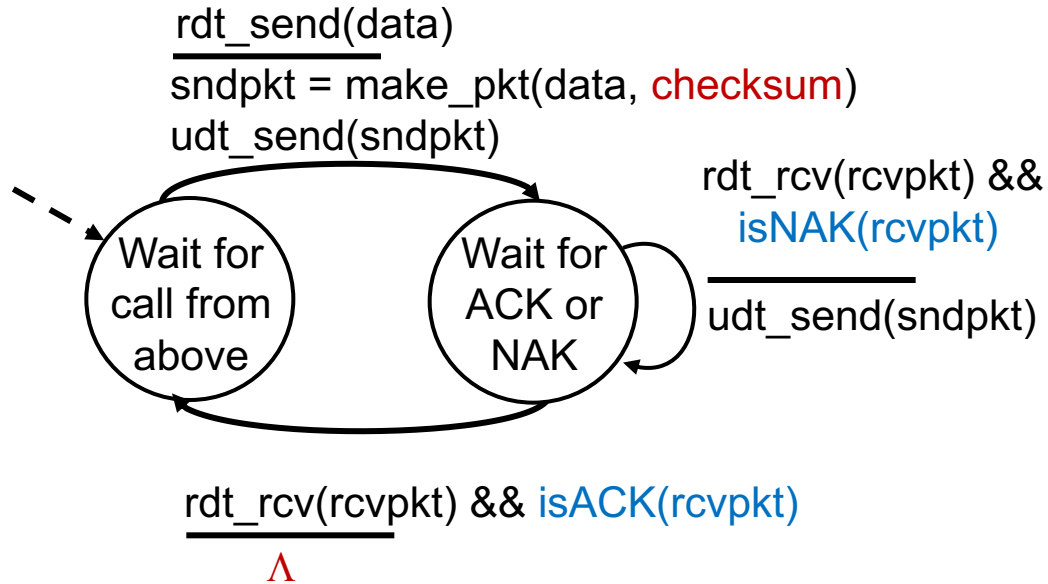
- **error detection**
- **feedback**: control msgs (ACK, NAK) from receiver to sender
- retransmission

## ❖ Error detection: checksum

## ❖ Feedback messages:

- *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

# rdt2.0: FSM specification

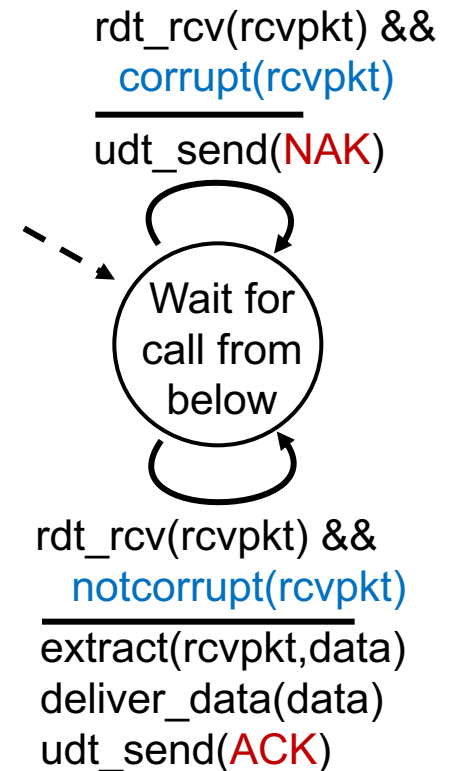


sender

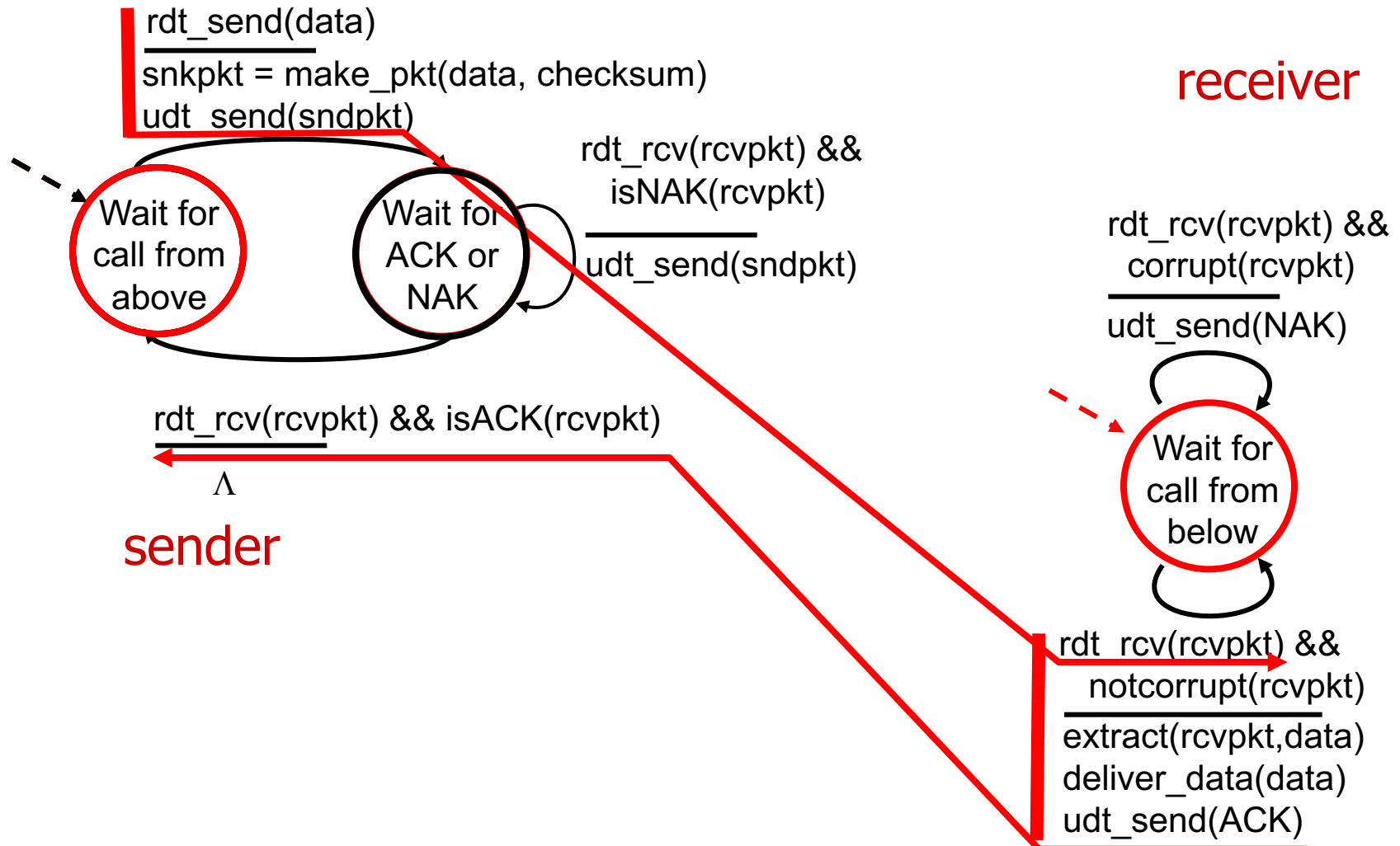
## Stop and wait

Sender sends one packet,  
then waits for receiver  
response

receiver

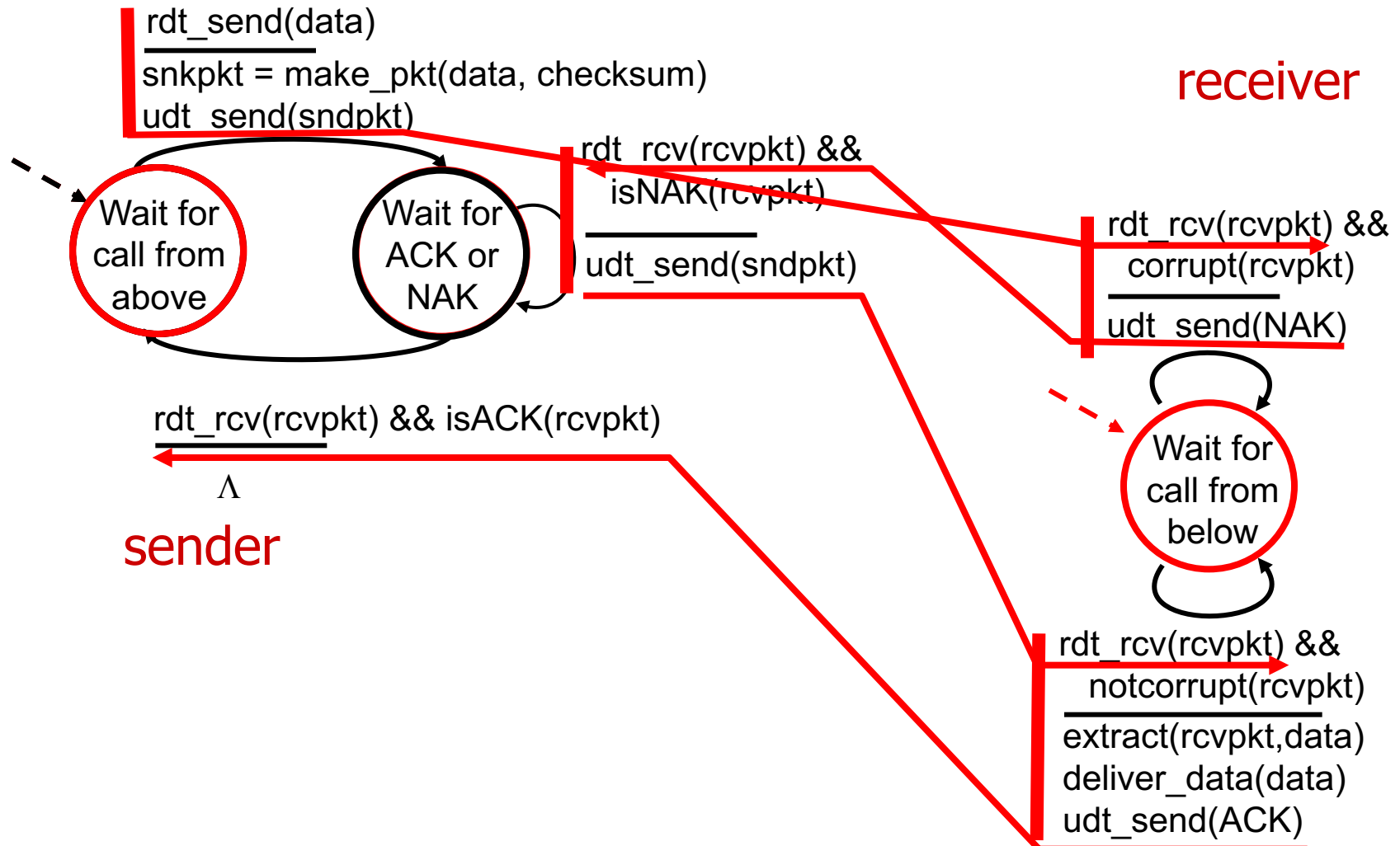


# rdt2.0: operation with no errors





# rdt2.0: error scenario



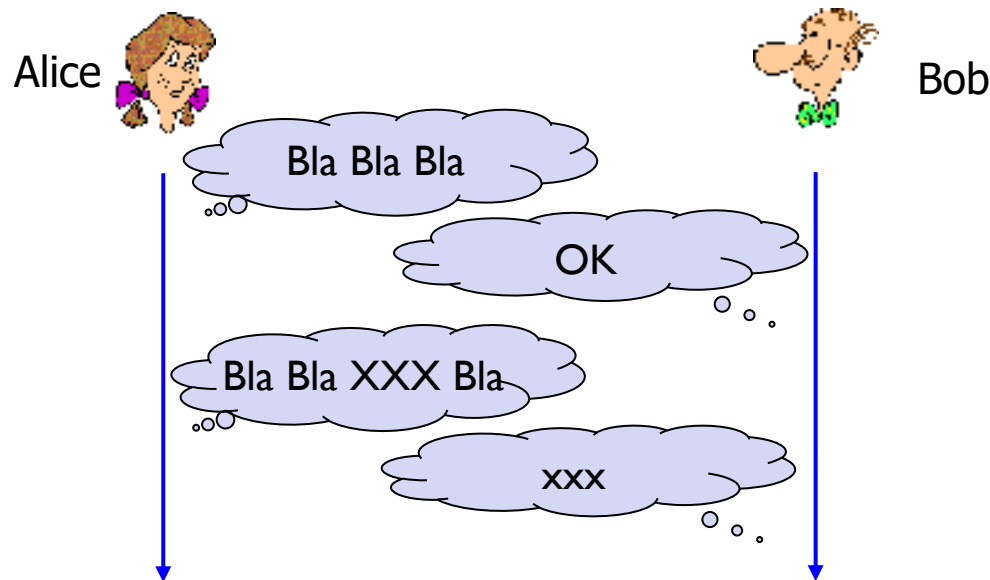
# rdt2.0 has a fatal flaw!

The possibility that ACK or NAK packet could be corrupted:

- ❖ Checksum bits

Handling corrupted ACKs or NAKs:

- ❖ Option 1: “blabla...”, “OK”, “What did you say?”, “OK”
  - “What did you say?”, “What did you say?”, ...
- ❖ Option 2: add enough checksum to recover
- ❖ Option 3: when garbled ACK or NAK, retransmit



# rdt2.0 has a fatal flaw!

The possibility that ACK or NAK packet could be corrupted:

- ❖ Checksum bits

Handling corrupted ACKs or NAKs:

- ❖ Option 1: “blabla...”, “OK”, “What did you say?”, “OK”
  - “What did you say?”, “What did you say?”, ...
- ❖ Option 2: add enough checksum to recover
- ❖ **Option 3: when garbled ACK or NAK, retransmit**

**Problem:** can't just retransmit: new data or retransmission?  
possible duplicate

**Handling duplicates:**

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

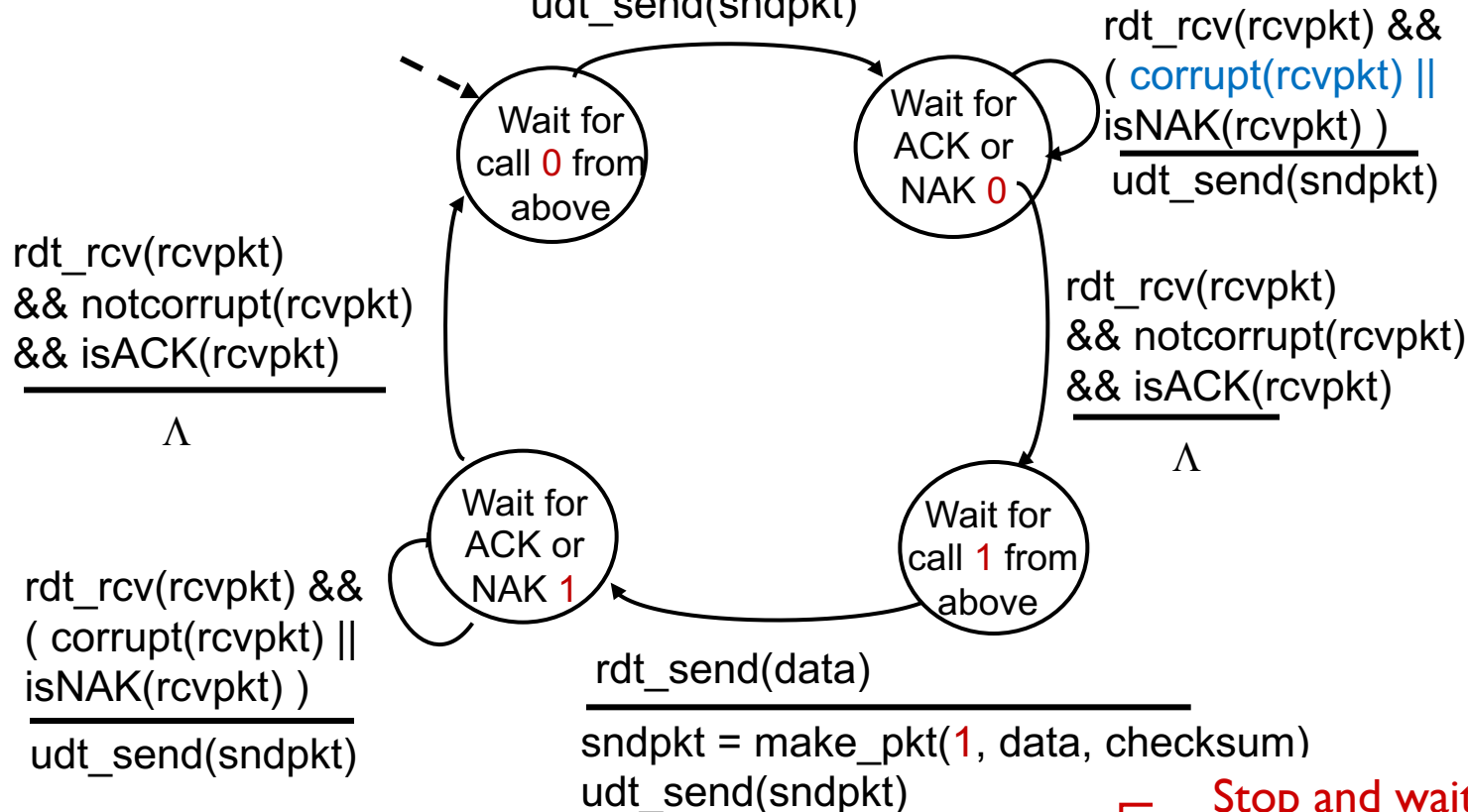
# rdt2.1: sender, handles garbled ACK/NAKs

sender

rdt\_send(data)

sndpkt = make\_pkt(0, data, checksum)

udt\_send(sndpkt)



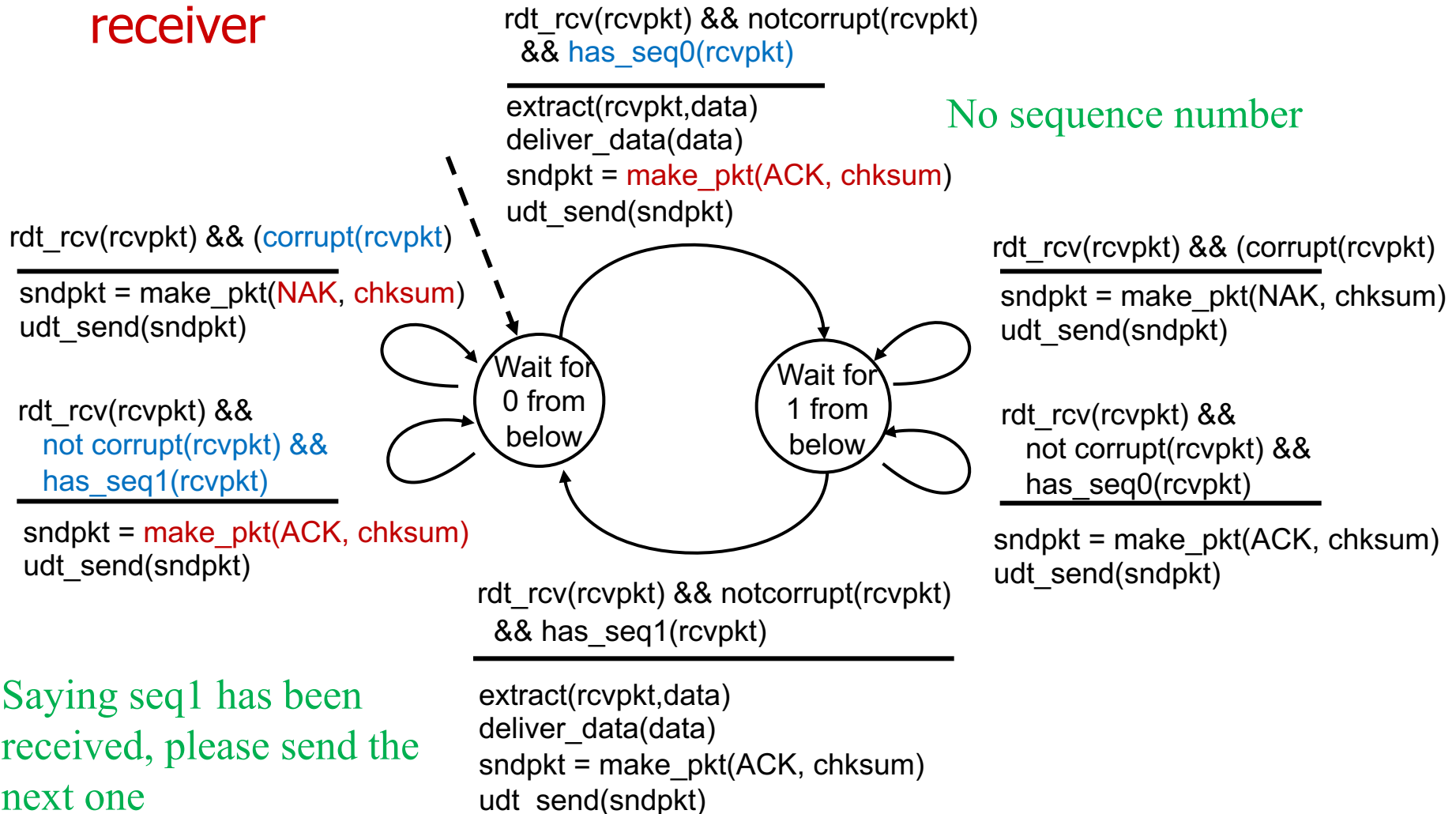
**Stop and wait**

Sender sends one packet, then waits for receiver response

Two sequence number would be sufficient!

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

## receiver



# rdt2.1: discussion

## sender:

- ❖ seq # added to pkt
- ❖ 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

Two seq. #'s (0,1) will suffice. Why?

# 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*

# rdt2.2: sender, receiver fragments

sender

rdt\_send(data)

sndpkt = make\_pkt(0, data, checksum)

udt\_send(sndpkt)

Wait for  
call 0 from  
above

sender FSM  
fragment

Wait for  
ACK  
0

rdt\_rcv(rcvpkt) &&  
( corrupt(rcvpkt) ||  
isACK(rcvpkt,1) )  
udt\_send(sndpkt)

rdt\_rcv(rcvpkt)  
&& notcorrupt(rcvpkt)  
&& isACK(rcvpkt,0)

$\Lambda$

rdt\_rcv(rcvpkt) &&  
( corrupt(rcvpkt) ||  
has\_seq1(rcvpkt) )  
udt\_send(sndpkt)

Wait for  
0 from  
below

receiver FSM  
fragment

rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)  
&& has\_seq1(rcvpkt)

extract(rcvpkt,data)

deliver\_data(data)

sndpkt = make\_pkt(ACK, 1, chksum)

udt\_send(sndpkt)

receiver



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

New assumption: underlying channel can also lose packets (data, ACKs)

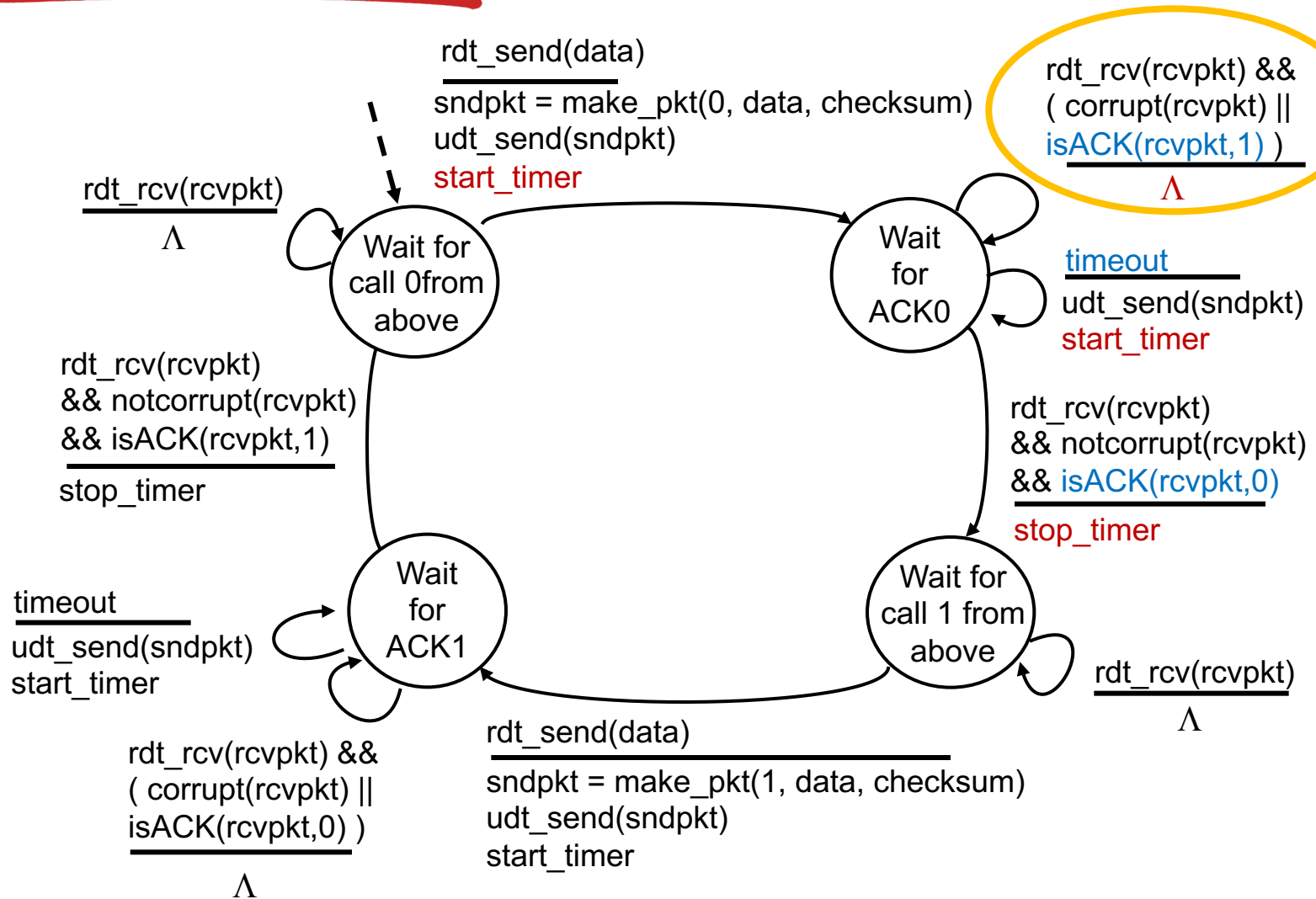
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

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

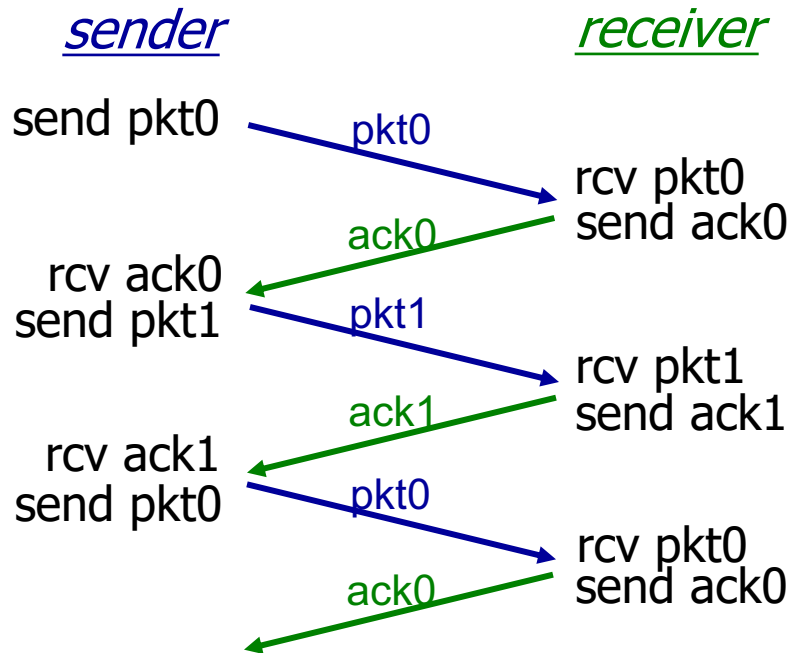
- ❖ retransmits if no ACK received in this time
- ❖ if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- ❖ requires countdown timer
  - start timer, timer interrupt, stop timer

How long should the sender wait?

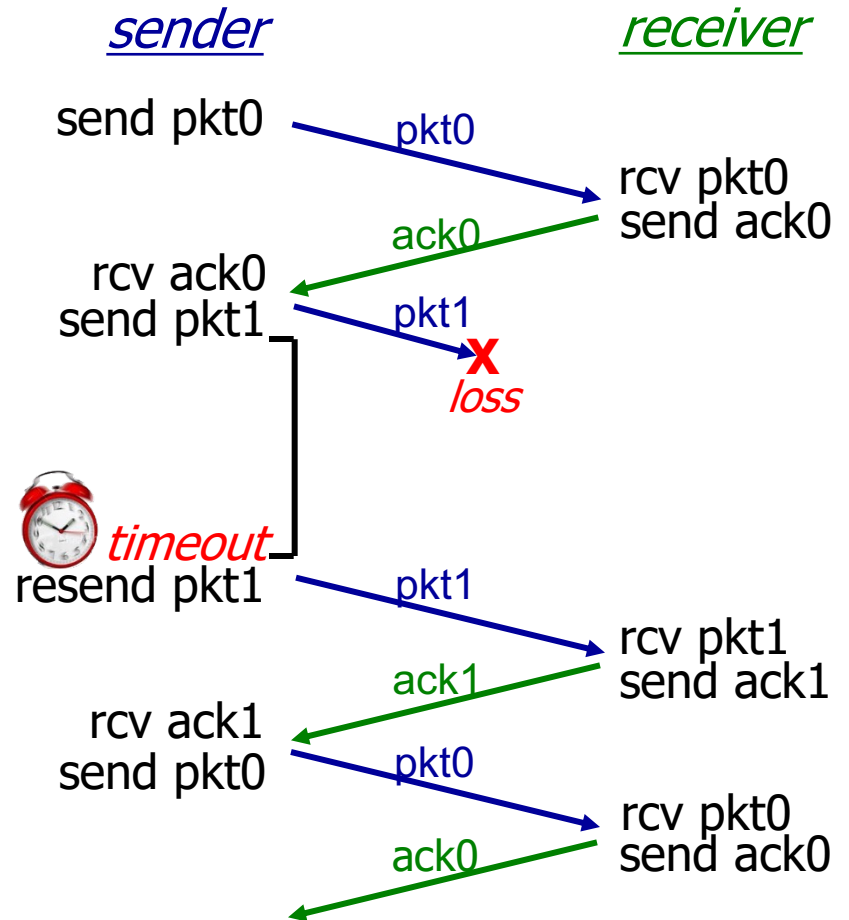
# rdt3.0 sender



# rdt3.0 in action

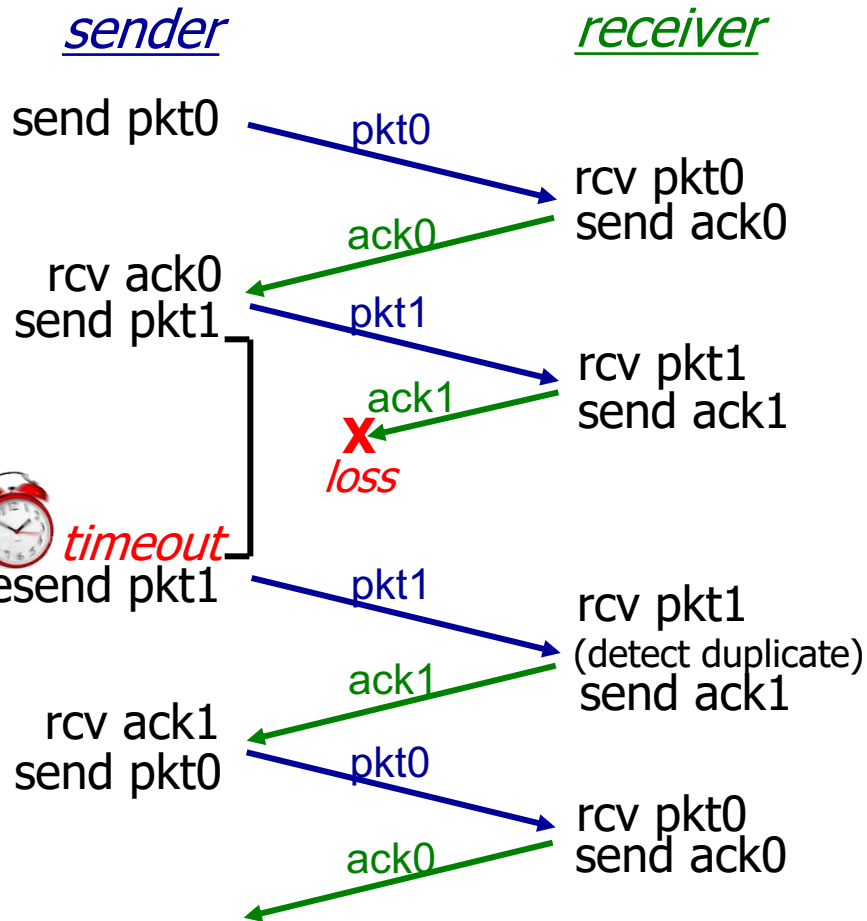


(a) no loss

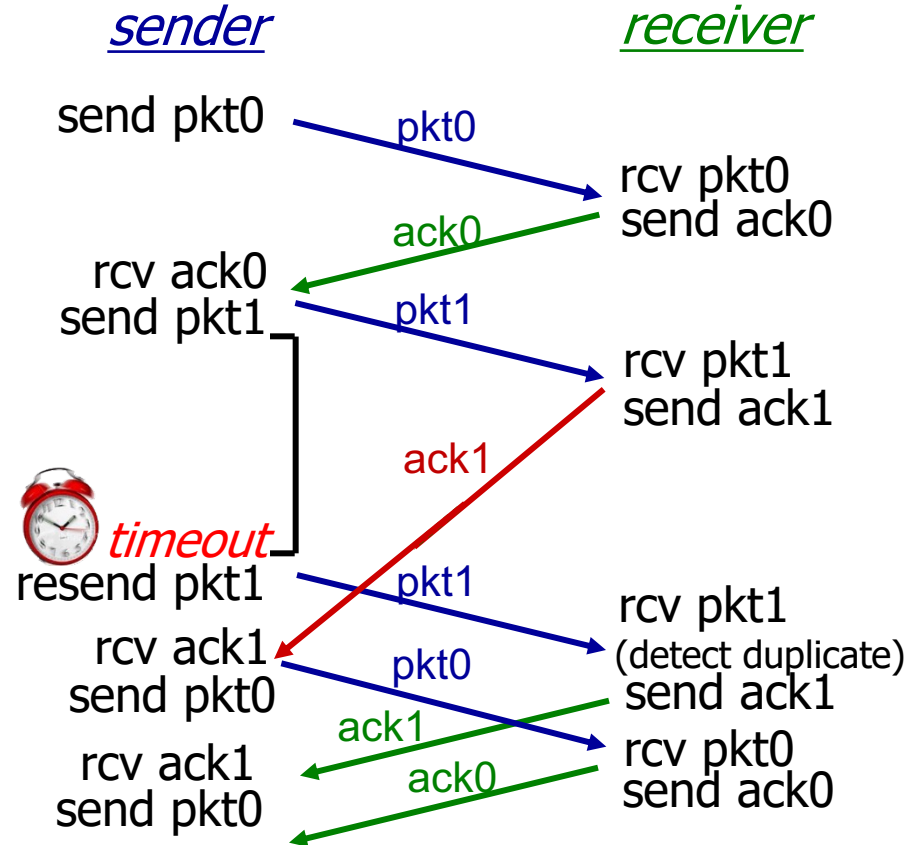


(b) packet loss

# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Summary

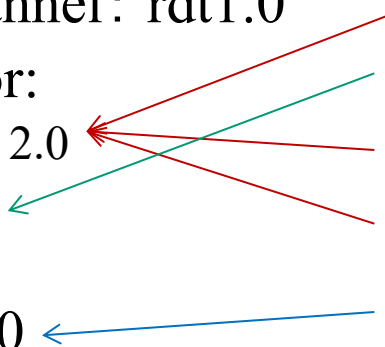
---

## Roadmap:

- ❖ Perfectly reliable channel: rdt1.0
- ❖ Channel with bit error:
  - bit error in packet: rdt 2.0
  - bit error in ACK: 2.1
  - NAK-free: 2.2
- ❖ Lossy channel: rdt 3.0

## Summary of Techniques

- Checksum
- Sequence number
- ACK packets
- Retransmission
- Timeout



# Next Lecture

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer (continue)

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control