Computer Networks
@CS.NYTU

Lecture 3: Transport Layer (TCP/UDP)

Instructor: Kate Ching-Ju Lin (林靖茹)

Slides modified from
“Computer Networking: A Top-Down Approach” 7th Edition

Outline

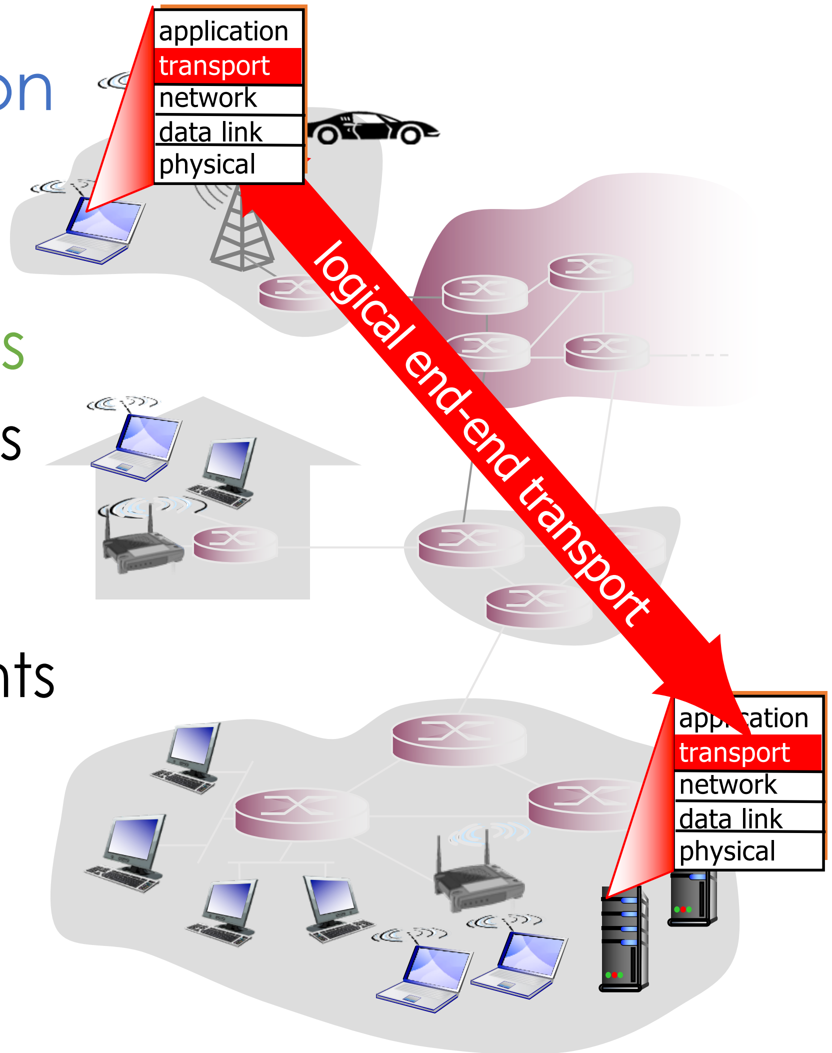
- Transport-layer services
- Multiplexing and demultiplexing
  - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Congestion Control

Outline

- **Transport-layer services**
- Multiplexing and demultiplexing
  - Socket programming
- Connectionless transport: UDP
- Reliable Data Transmission
- Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Congestion Control

Transport Services and Protocols

- Provide logical communication between app **processes** running on different hosts
- Transport protocols run in hosts
  - sender: breaks app messages into segments → passes to network layer
  - receiver: reassemble segments into messages → passes to app layer
- Available transport protocols
  - **TCP** and **UDP**



Transport vs. Network Layer

- **Network layer:**
  - logical communication between hosts
  - **Host-to-host**

- **Transport layer:**
  - logical communication relies on, enhances, network layer services
  - End-to-end (**process-to-process**)

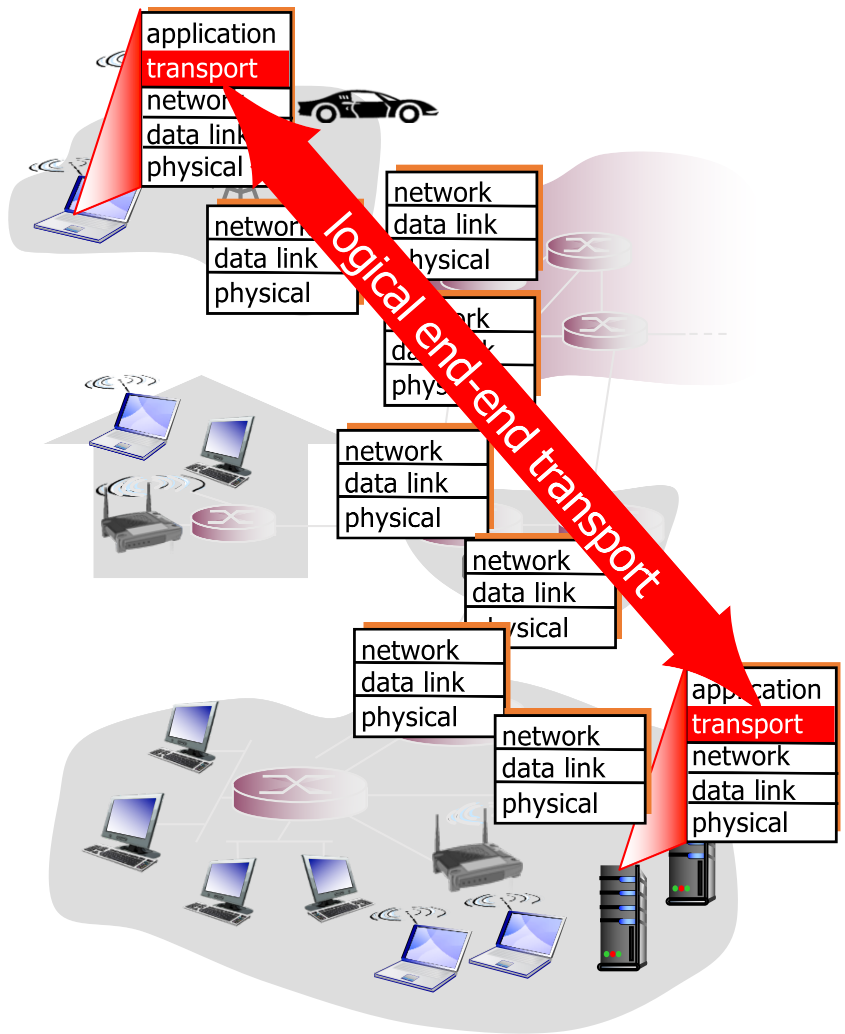
household analogy:

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 who demux to in-house siblings
- network-layer protocol = postal service

Internet Transport Protocols

- Reliable, in-order delivery: TCP
  - connection setup
  - congestion/flow control
  - acknowledgement
- Unreliable, unordered delivery: UDP
  - connectionless
  - Best effort: send as many as possible
- Services not available:
  - no delay guarantees
  - no bandwidth guarantees

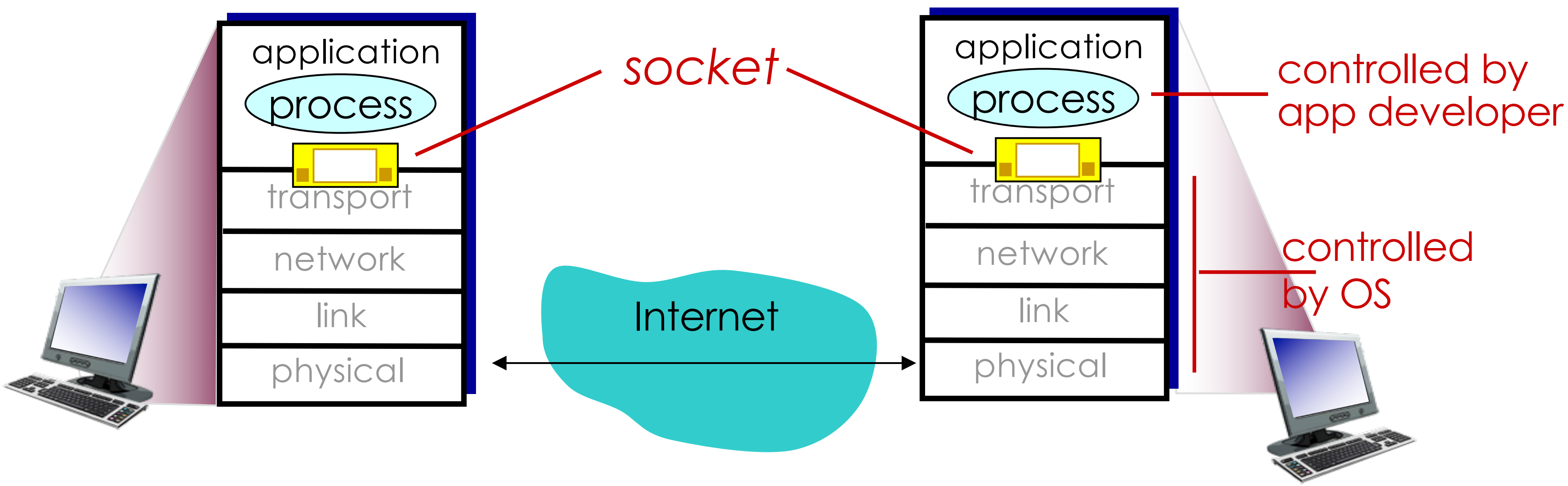


Outline

- Transport-layer services
- **Multiplexing and demultiplexing**
  - **Socket programming**
- Connectionless transport: UDP
- Reliable Data Transmission
- Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Congestion Control

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

- Two socket types for two transport services
  - **UDP:** unreliable datagram
  - **TCP:** reliable, byte stream-oriented
- **Port Number**
  - open protocols (FTP, HTTP, …): follow RFC
  - Proprietary applications: avoid using well-known ports
- 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

Transport Layer vs. Socket

- Each process can have one or more sockets
- Each socket has a unique ID
  - ID = (src IP, src port, dst IP, dst port)
- In a receiving host, the transport layer does not deliver data directly to a process, but to a socket instead
  - **Multiplexing:** pass the segments from different sockets to the network layer
  - **Demultiplexing:** deliver data in a transport-layer segment to the correct socket

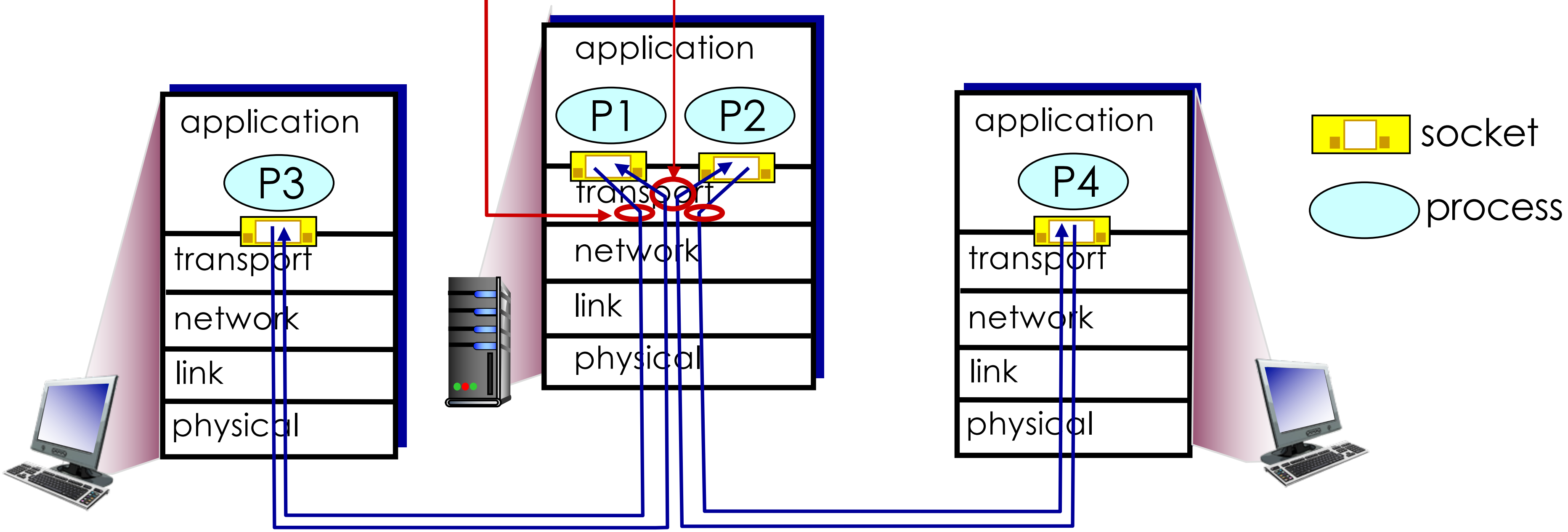
Multiplexing/Demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header

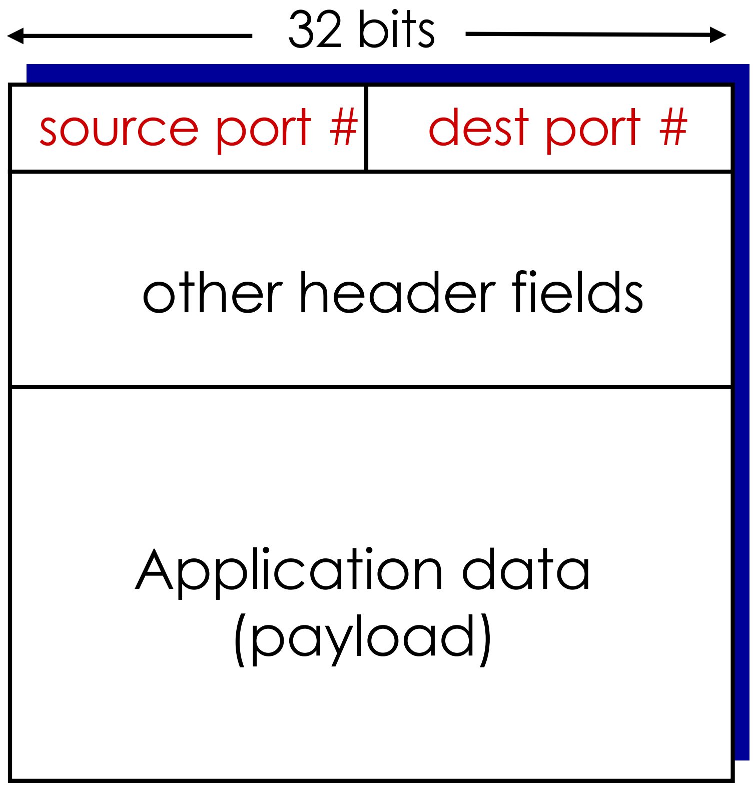
demultiplexing at receiver:

use header info to deliver received segments to correct **socket** *(unique proc ID)*



How Demultiplexing Works?

- Host receives IP datagrams
  - A datagram consists of multiple segments
- Each datagram has
  - source IP address
  - source port
  - destination IP address
  - destination port
- Host uses IP addresses & port numbers to direct segment to the appropriate socket



TCP/UDP segment format

Connectionless Demultiplexing

- Use UDP socket

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

- Identified by
  - (src IP, src port) or (dst IP, dst port)
- ID assignment?
  1. Transport layer automatically assigns a port number to the socket (typically used at receiver)
  2. Use bind() to specify a particular port

```
clientSocket.bind((‘’, given_port))
```

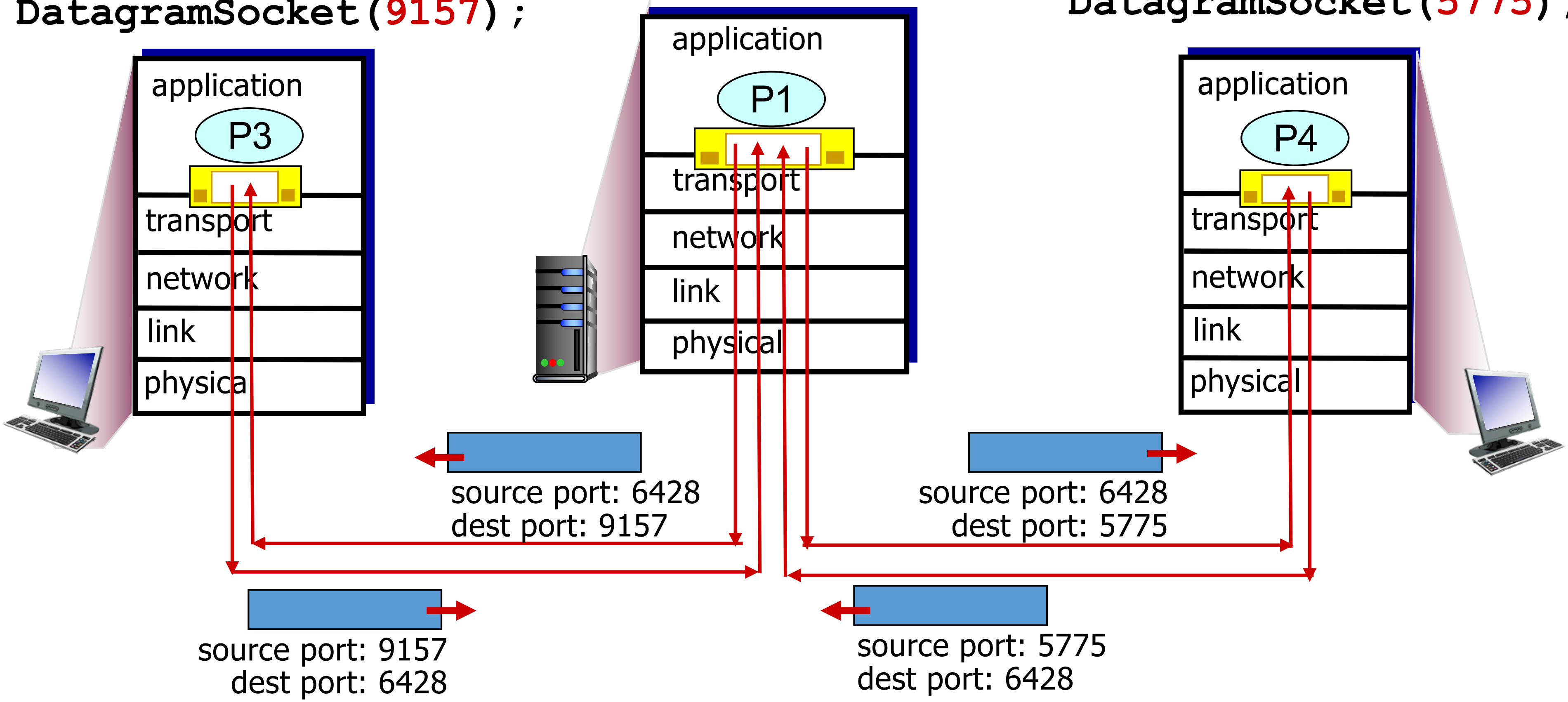
- Two UDP segments with different source IP and/or port numbers can be directed to the same destination process (socket)

Connectionless Demux: Example

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

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

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



Python Socket: UDP Client

include Python's socket library

create UDP socket for server

get user keyboard input

Attach server name, port to message; send into socket

read reply characters from socket into string

print out received string and close socket

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

Buffer size

Python Socket: UDP Server

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((' ', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

- create UDP socket → `serverSocket = socket(AF_INET, SOCK_DGRAM)`
- bind socket to local port number 12000 → `serverSocket.bind((' ', serverPort))`
- loop forever → `while True:`
- Read from UDP socket into message, getting client's address (client IP and port) → `message, clientAddress = serverSocket.recvfrom(2048)`
- send upper case string back to this client → `serverSocket.sendto(modifiedMessage.encode(), clientAddress)`

Connection-Oriented Demux

- Use TCP socket
- Identified by four-tuple
  - (src IP, src port, dst IP, dst port)
  - In contrast with UDP, two TCP segments with different source IP or ports will never be directed to the same destination socket
- Client-server TCP
  - Server has a "welcome socket", which as a well-known port number

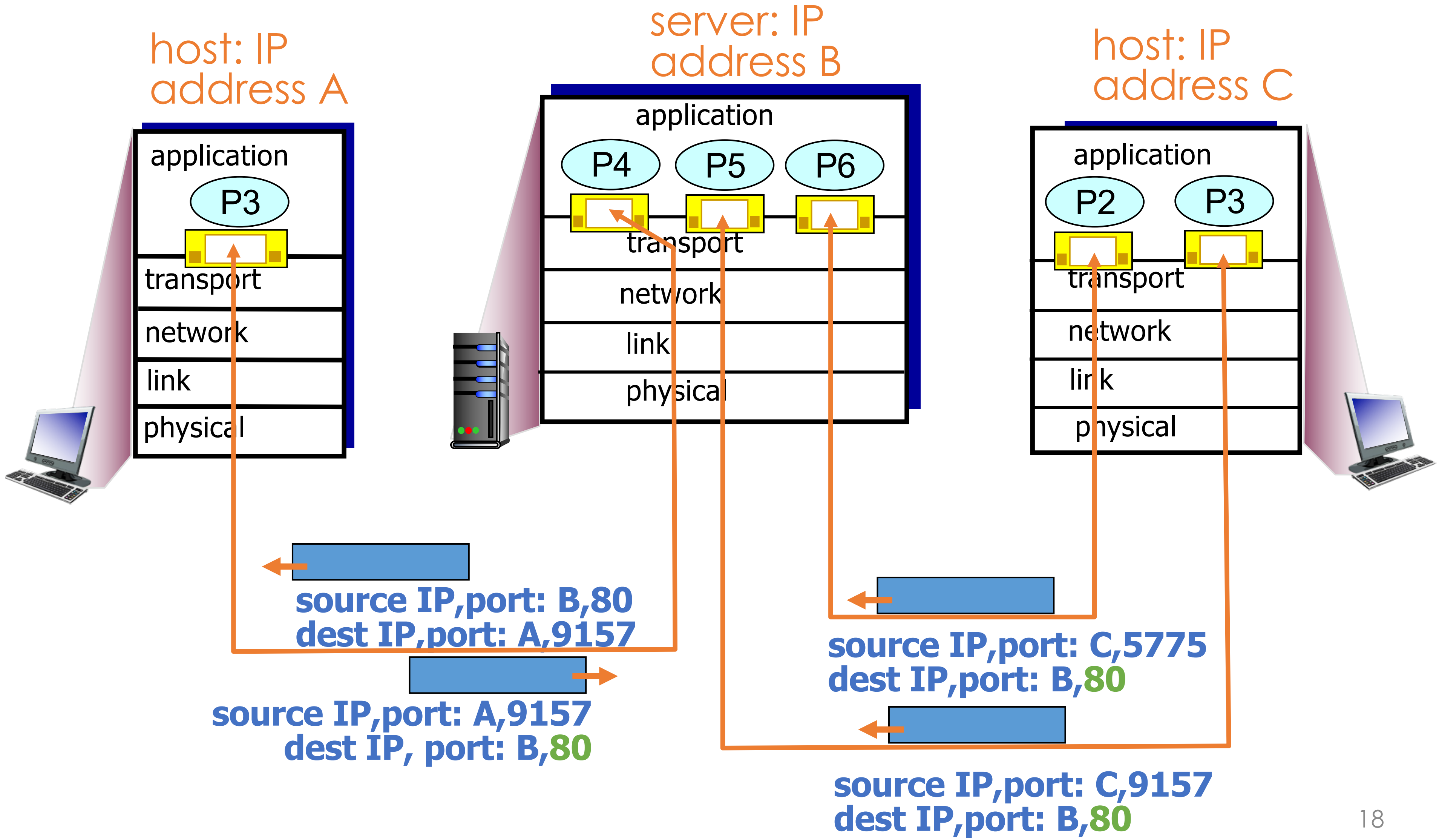
    ```
    clientSocket = socket(AF_INET, SOCK_STREAM)
    clientSocket.connect((serverName, welcome_port))
    ```

  - Client requests for a connection by sending to "welcome port" (including its source port)
  - Server accepts the request and creates a unique socket

    ```
    ConnectionSocket, addr = clientSocket.accept()
    ```

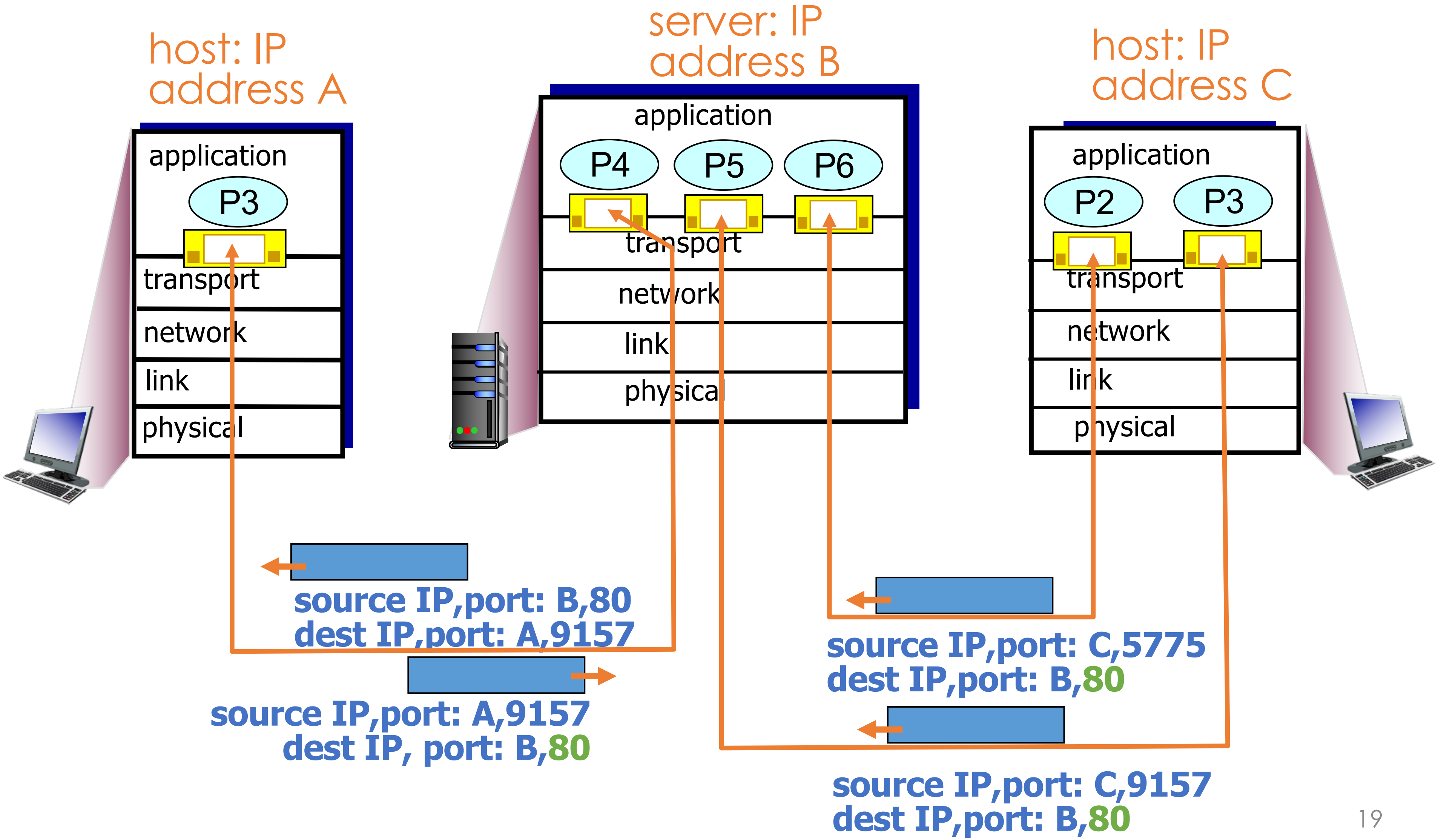
Connection-Oriented Demux: Example

Client C to server B has two connections!



C and A might pick the same random port

→ fine!!! Because they have different IP address



Python Socket: TCP Client

create TCP socket for server, remote port 12000

No need to attach server name, port

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Client's port is **randomly** picked
in the transport layer (implemented in OS)

Python Socket: TCP Server

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

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

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

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

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((' ',serverPort))
serverSocket.listen(1) // max # of clients, at least 1
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
     // connectionSocket dedicated to a particular user
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                      encode())
    connectionSocket.close()
```

Outline

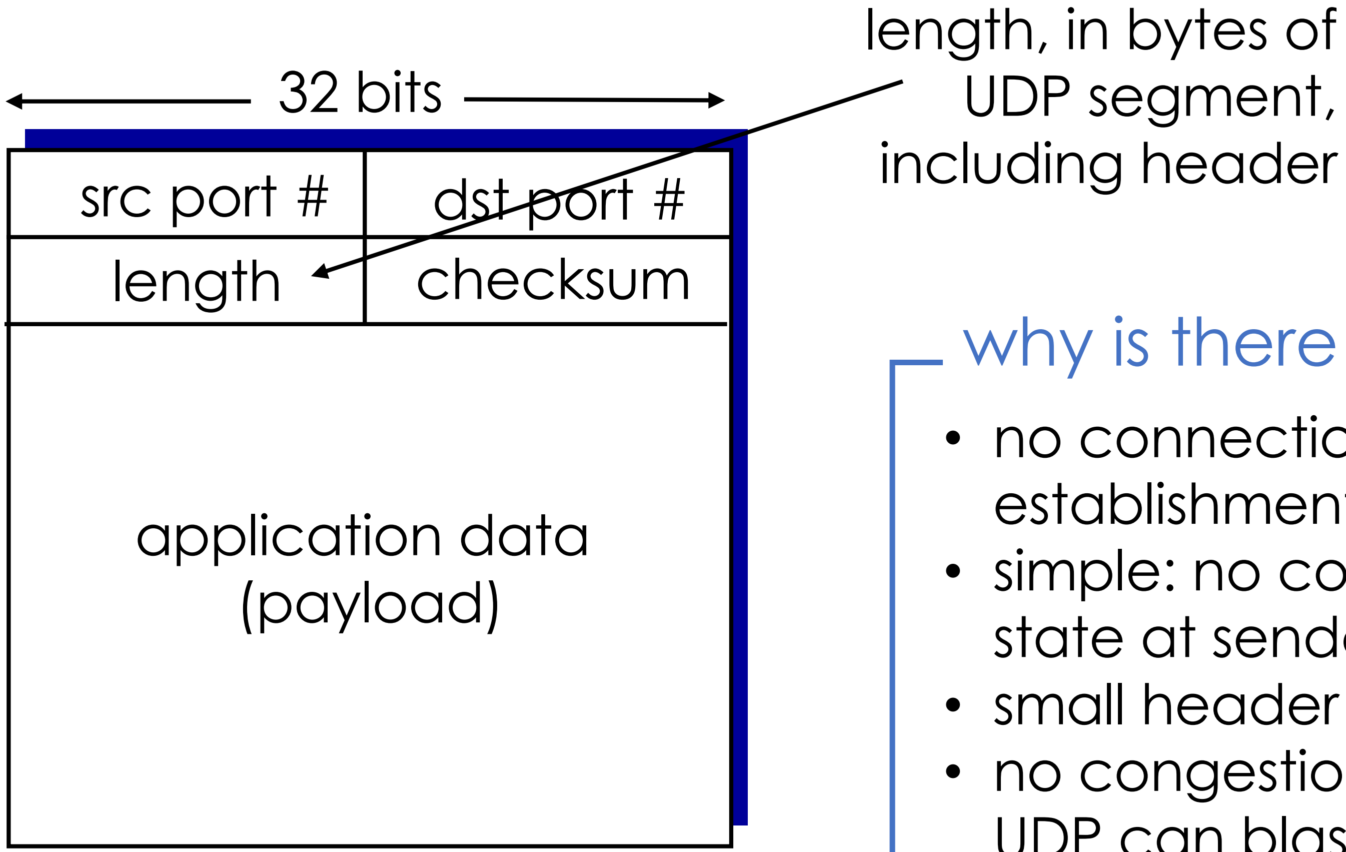
- Transport-layer services
- Multiplexing and demultiplexing
  - Socket programming
- **Connectionless transport: UDP**
- Reliable Data Transmission
- Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Congestion Control

UDP: User Datagram Protocol [RFC 768]

- "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

- Pros:
  - low latency
  - no state (stateless)
    → support more users
  - smaller packet header
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- Reliable transfer over UDP:
  - add reliability at application layer via error recovery

UDP: Segment Header



UDP segment format

why is there a UDP?

- no connection establishment (no delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP Checksum

Goal: detect "errors" in transmitted segment

- **Sender**
  - treat segment contents, including header fields, as sequence of 16-bit integers
  - checksum: addition (one's complement sum) of segment contents
  - sender puts checksum value into UDP checksum field
- **Receiver**
  - all 16-bit integers (including the checksum) are added
  - packet error if the sum != 1111…11111

Checksum: Example

example: add two 16-bit integers

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

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

UDP Checksum

Link-layer protocols usually also provide error checking

- Why UDP needs checksum?
  - Not all the link layer protocols provide error checking
  - Bit errors could be introduced during I/O (not the networking problem)

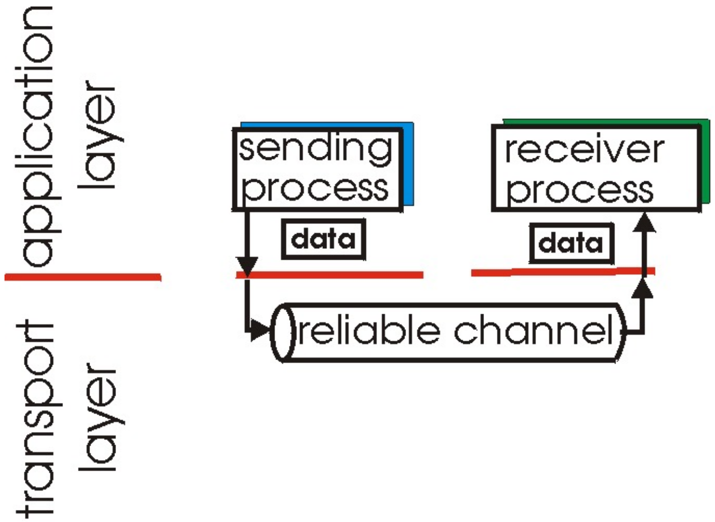
- Error detection vs. error recovery
  - UDP only supports error detection (via checksum)
  - UDP directly discard erroneous segments
  - Does not try to recover an error → lossy transportation
    - Recovery could be done in the application (optional)

Outline

- Transport-layer services
- Multiplexing and demultiplexing
  - Socket programming
- Connectionless transport: UDP
- **Reliable Data Transmission**
- Connection-oriented transport: TCP
  - Segment structure
  - Connection management
  - Flow control
  - Congestion Control

What is Reliable Data Transfer?

- Important in application, transport, link layers
  - top-10 list of important networking topics!

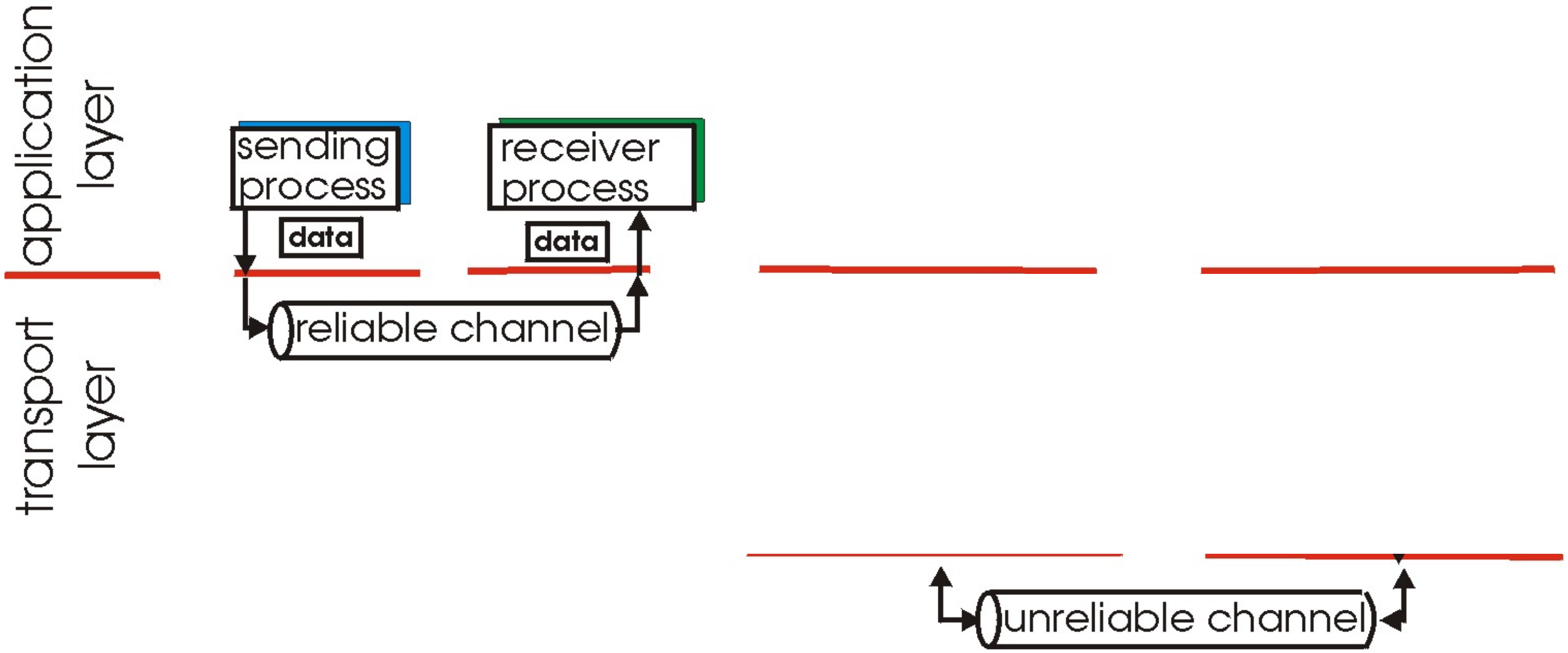


(a) provided service

- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

What is Reliable Data Transfer?

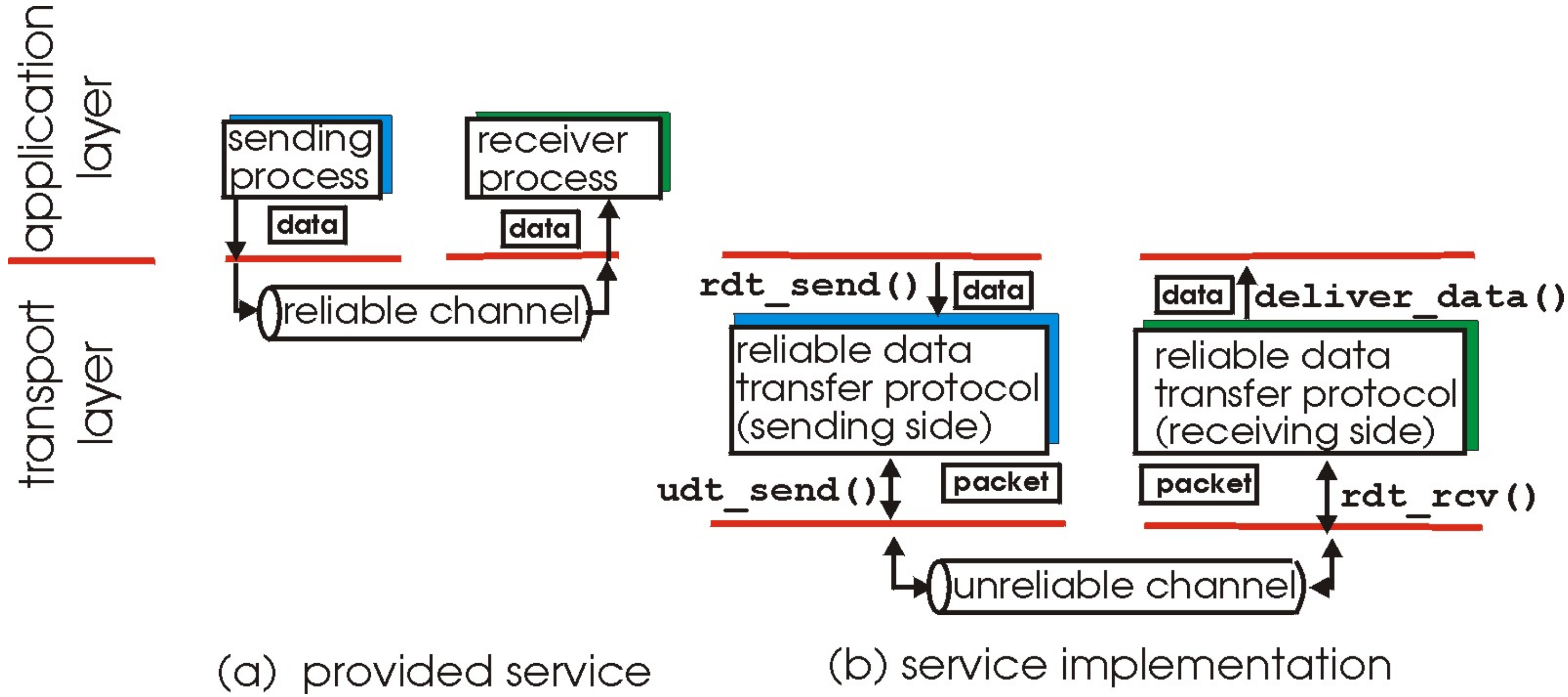
- Provide error free transfer over unreliable links
- Applications do not need to worry about how to recover errors



(a) provided service
(b) service implementation

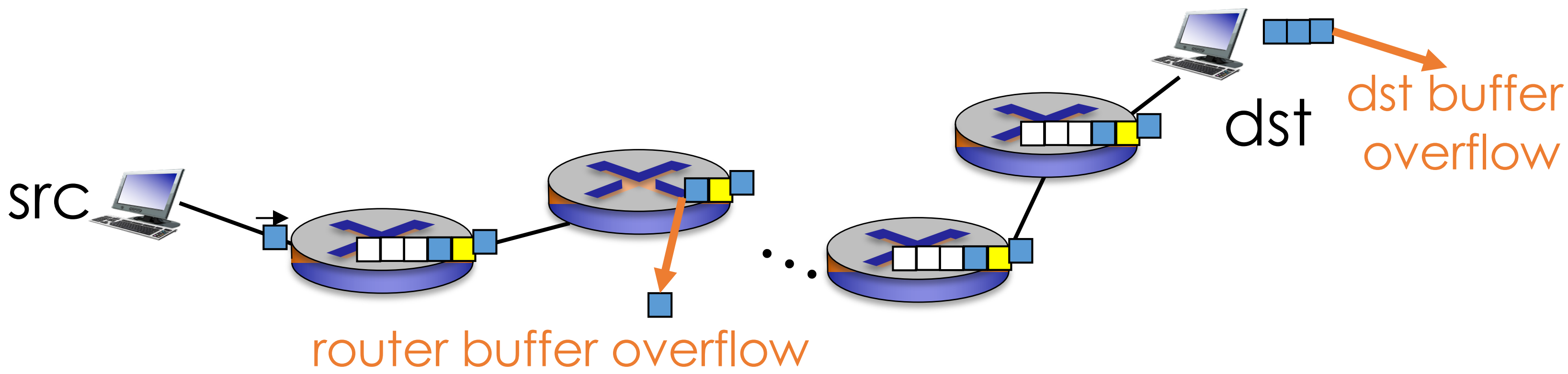
What is Reliable Data Transfer?

- Provide error free transfer over unreliable links
- Applications do not need to worry about how to recover errors



Types of Errors

- **Bit errors**
  - Receiver gets the packet, but some bits are in error
- **Packet losses**
  - Router buffer overflow: need congestion control
  - Destination buffer overflow: need flow control



- **Packet disordered**
  - Packets are not arrived in the correct order

> - How to detect errors at sender?
> - How to retransmit packets?

Reliable Data Transfer

Single direction transportation

- Learn step-by-step

1. rdt over reliable channel

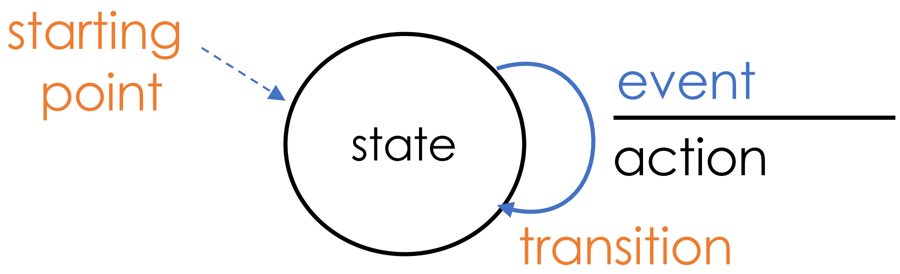
2. rdt over a forward channel with bit errors, but
   (a) no feedback channel losses, disorder
   (b) no packet losses, packets in order

3. rdt over a forward channel with bit errors and feedback channel may be unreliable, but
   no packet losses, packets in order

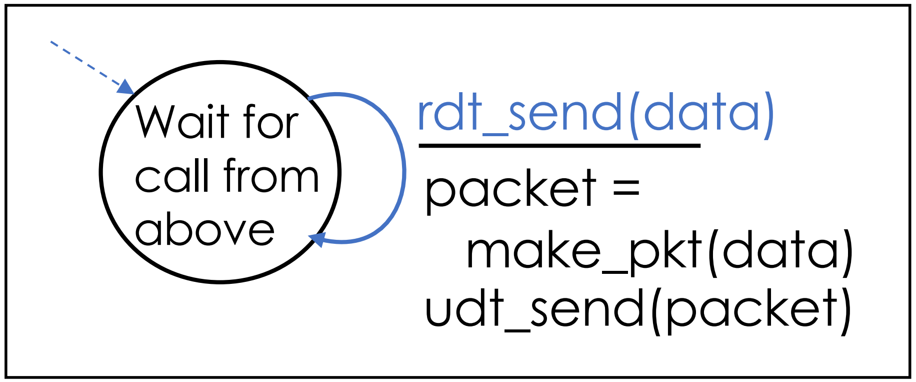
4. rdt over a lossy channel with bit errors

1. rdt over reliable channel

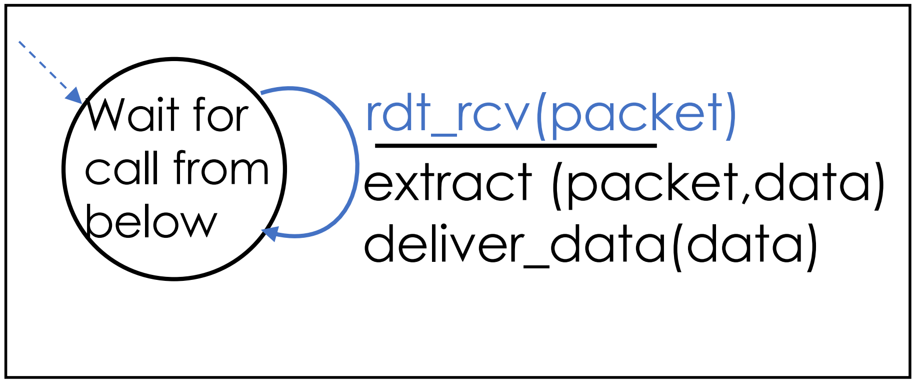
- Use **finite-state machine (FSM)** to describe the behavior of sender and receiver



- rdt over reliable channel (no error, no loss)



sender



receiver

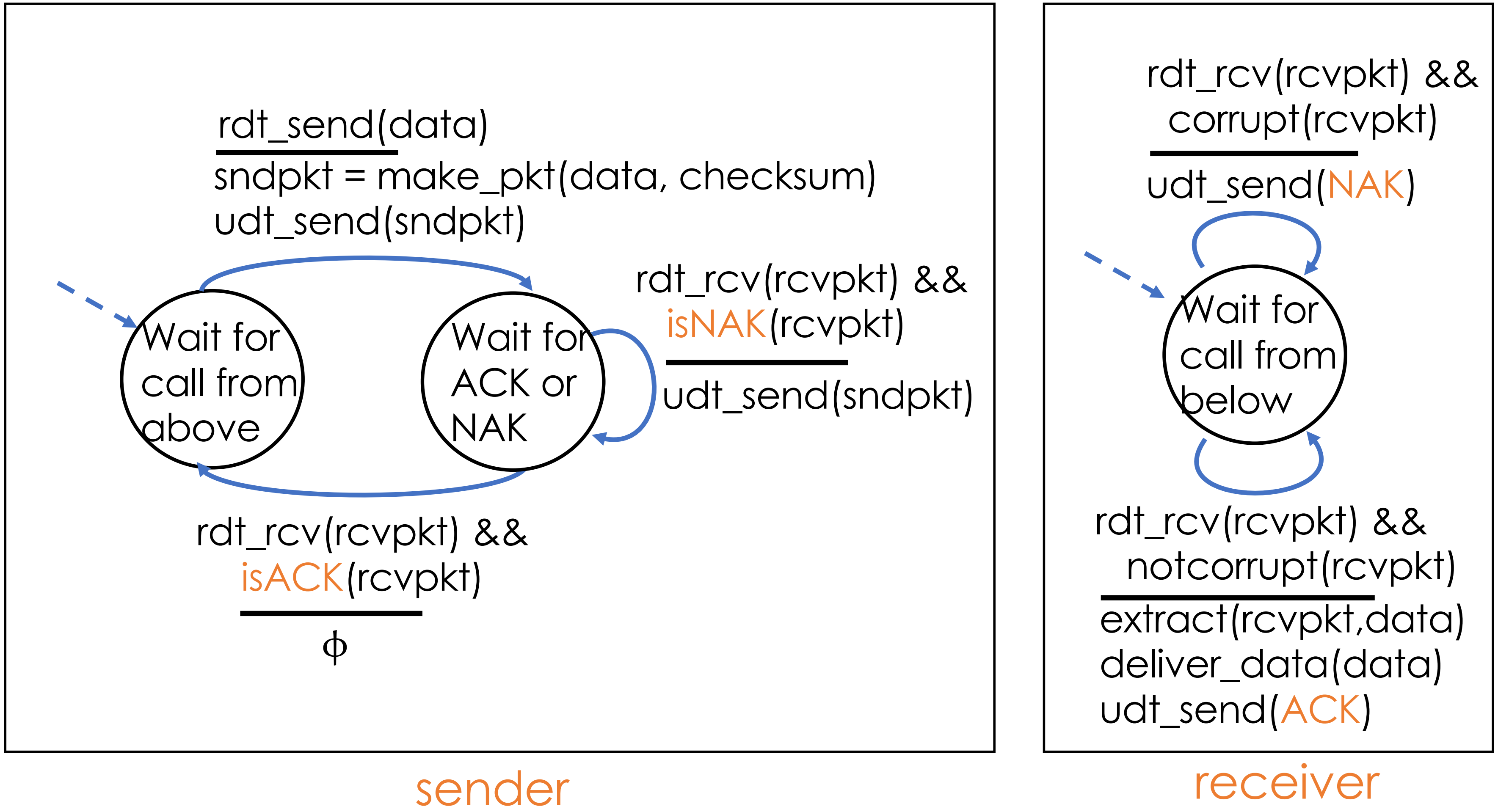
2. rdt over a channel with bit errors

- Example: when you make a call but does not hear clearly, how do we do?
  - Could you repeat again?
- In rdt, when a receiver does not receive correctly
  - Could you send again? → **retransmission!**

- How to realize this idea?
  - **Error detection**: use checksum (similar to that in UDP)
  - **Receiver feedback**:
    - **ACK**: positive acknowledgments ("OK")
    - **NAK**: negative acknowledgments ("failed, repeat again")
  - **Retransmission**: sender re-send the erroneous packets

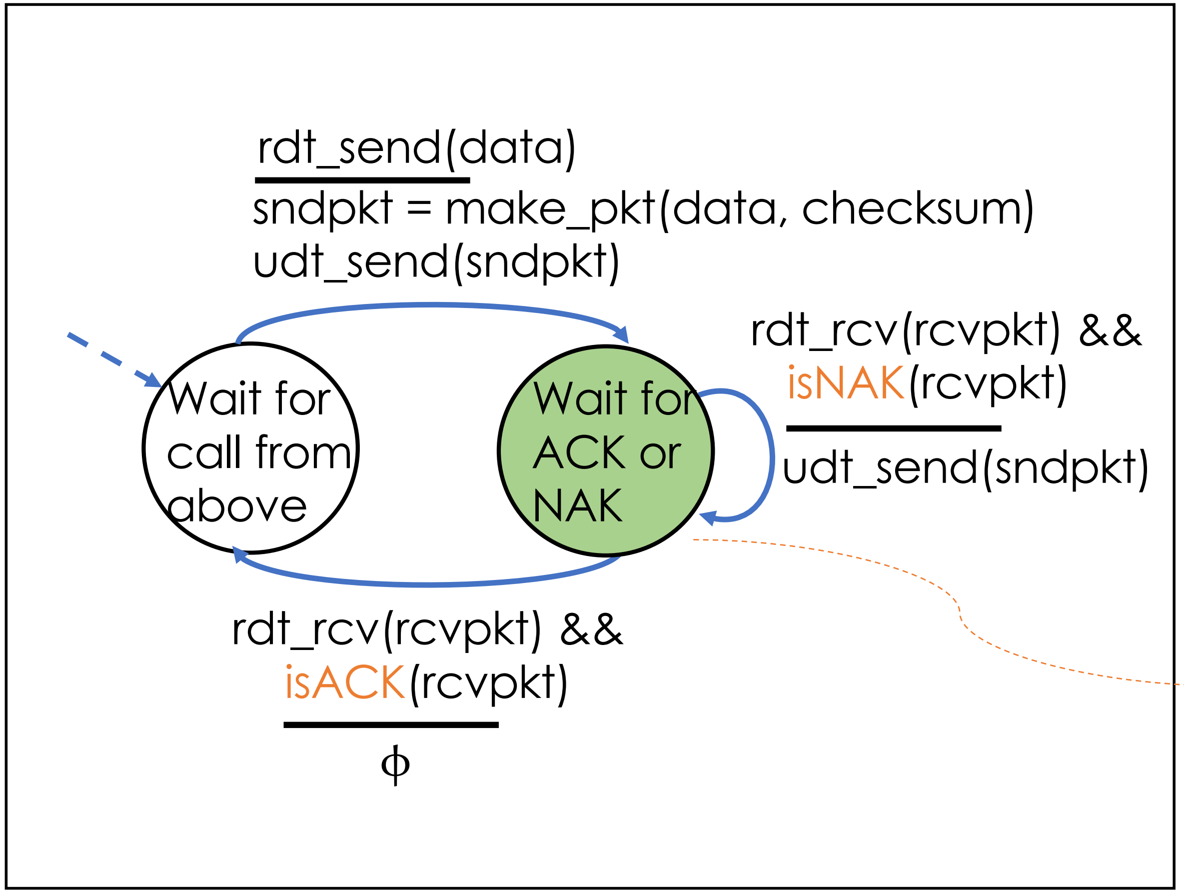
2. rdt over a channel with bit errors

- Assume no loss, no disorder



2. rdt over a channel with bit errors

- Issue: what if ACK/NAK is in error (corrupted)?
  - Sender does not know what happen at receiver

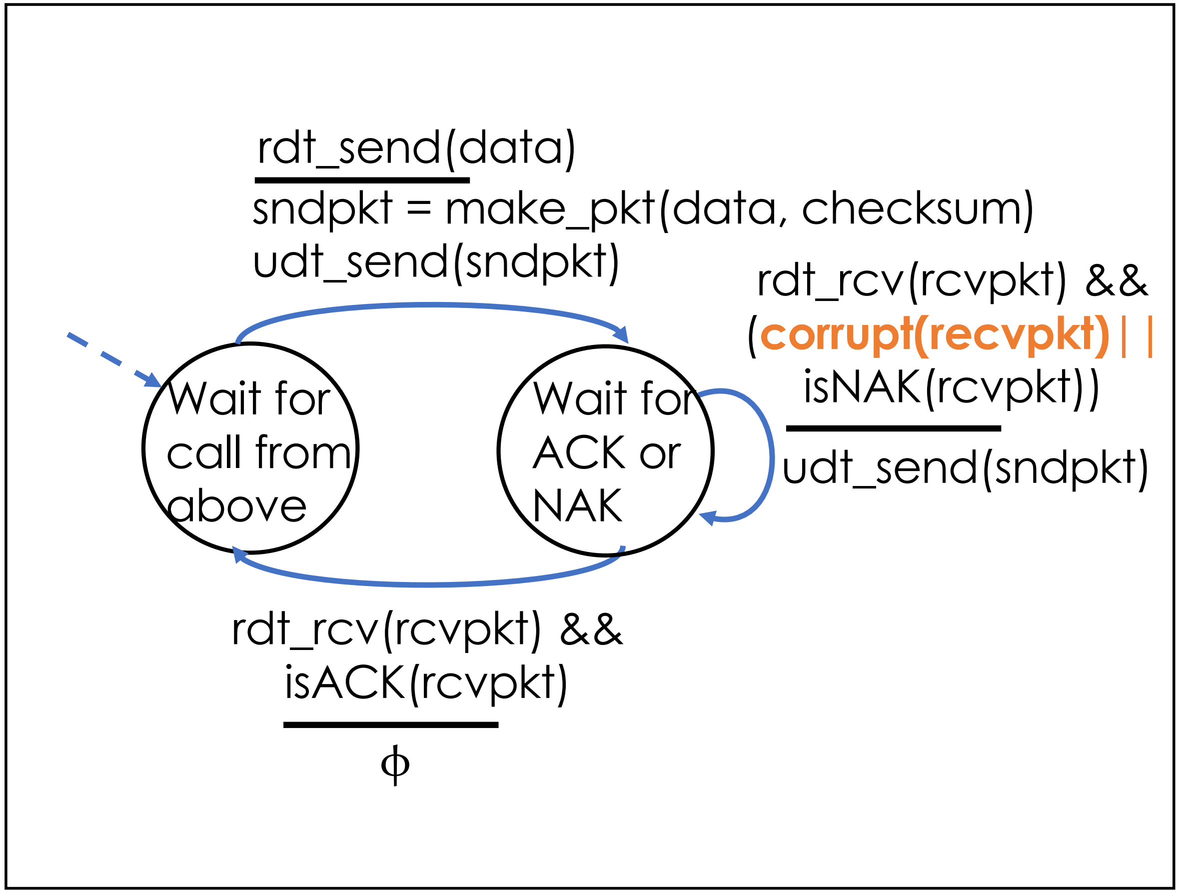


sender

keep waiting here!
➔ **stop-and-wait** protocol

3. rdt over a channel with feedback errors

- Possible solution: retransmission if ACK/NAK corrupted!



sender

✗ Does not work!! Why?

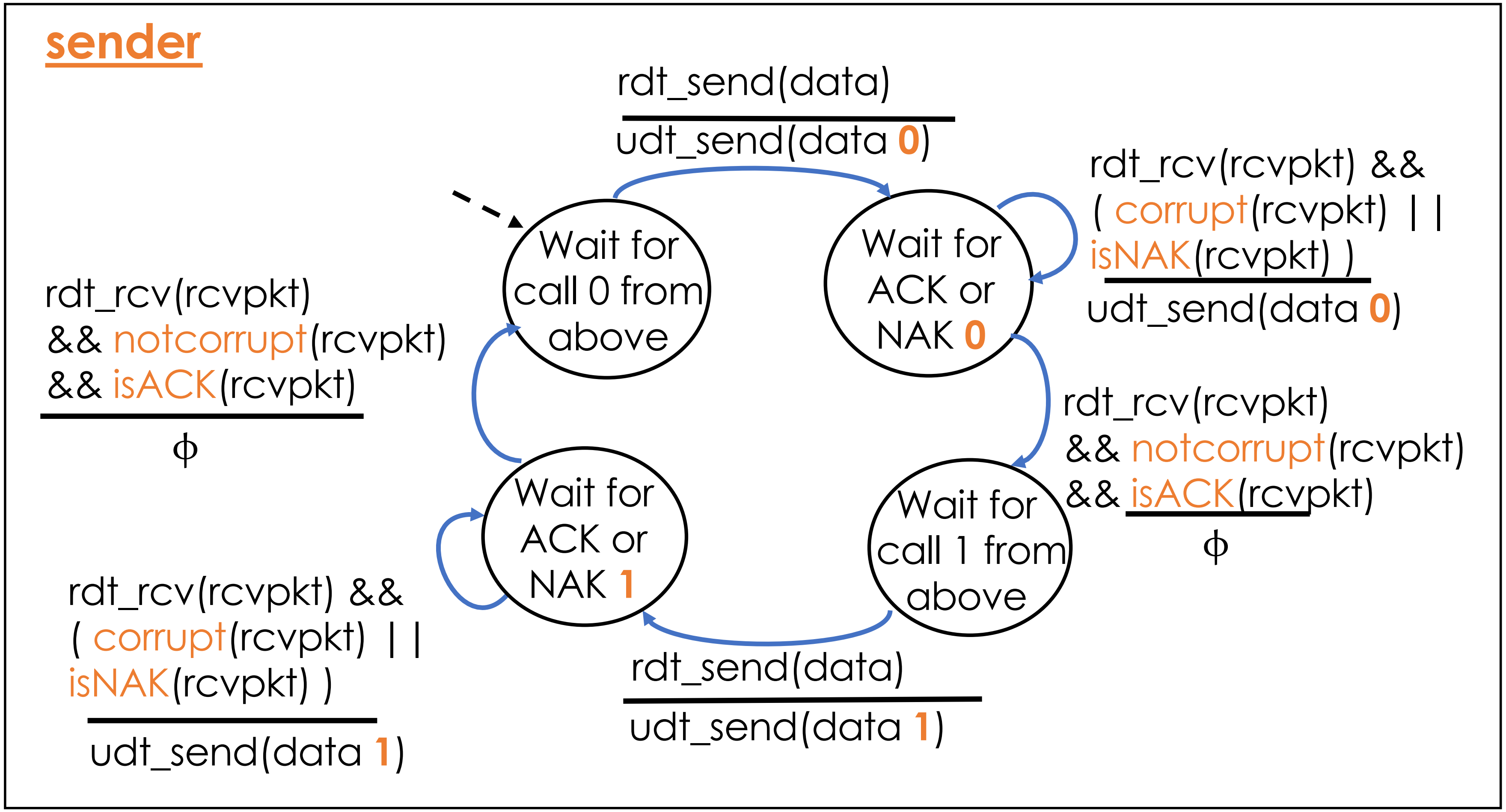
Problem: if the corrupted feedback is ACK
→ duplicate packet

Q: reTx without dup?

Trick: add **sequence number!**

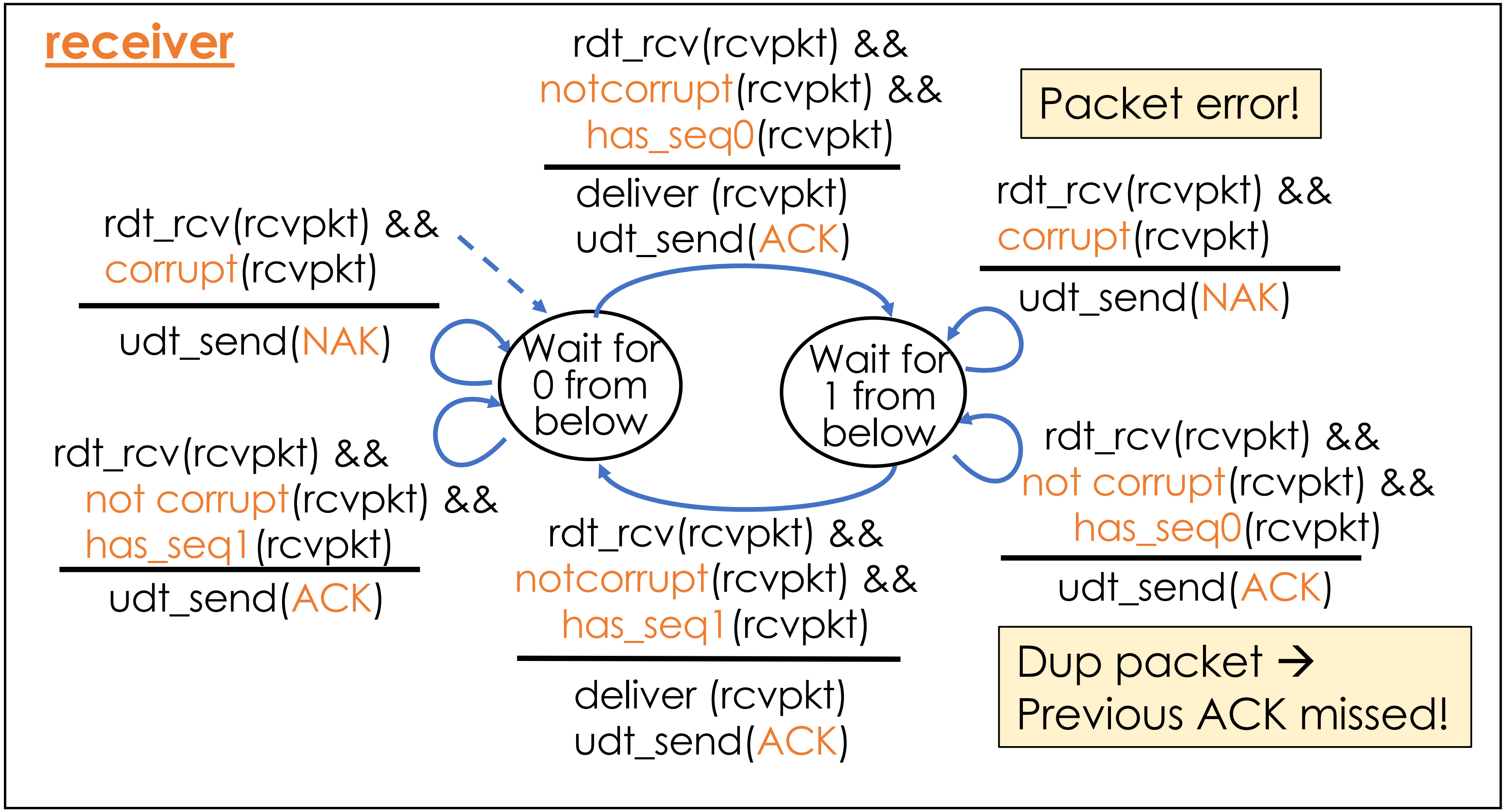
3. rdt over a channel with feedback errors

- Simple implementation
  → 1-bit seq: 0, 1, 0, 1, 0, 1, …, 0, 1



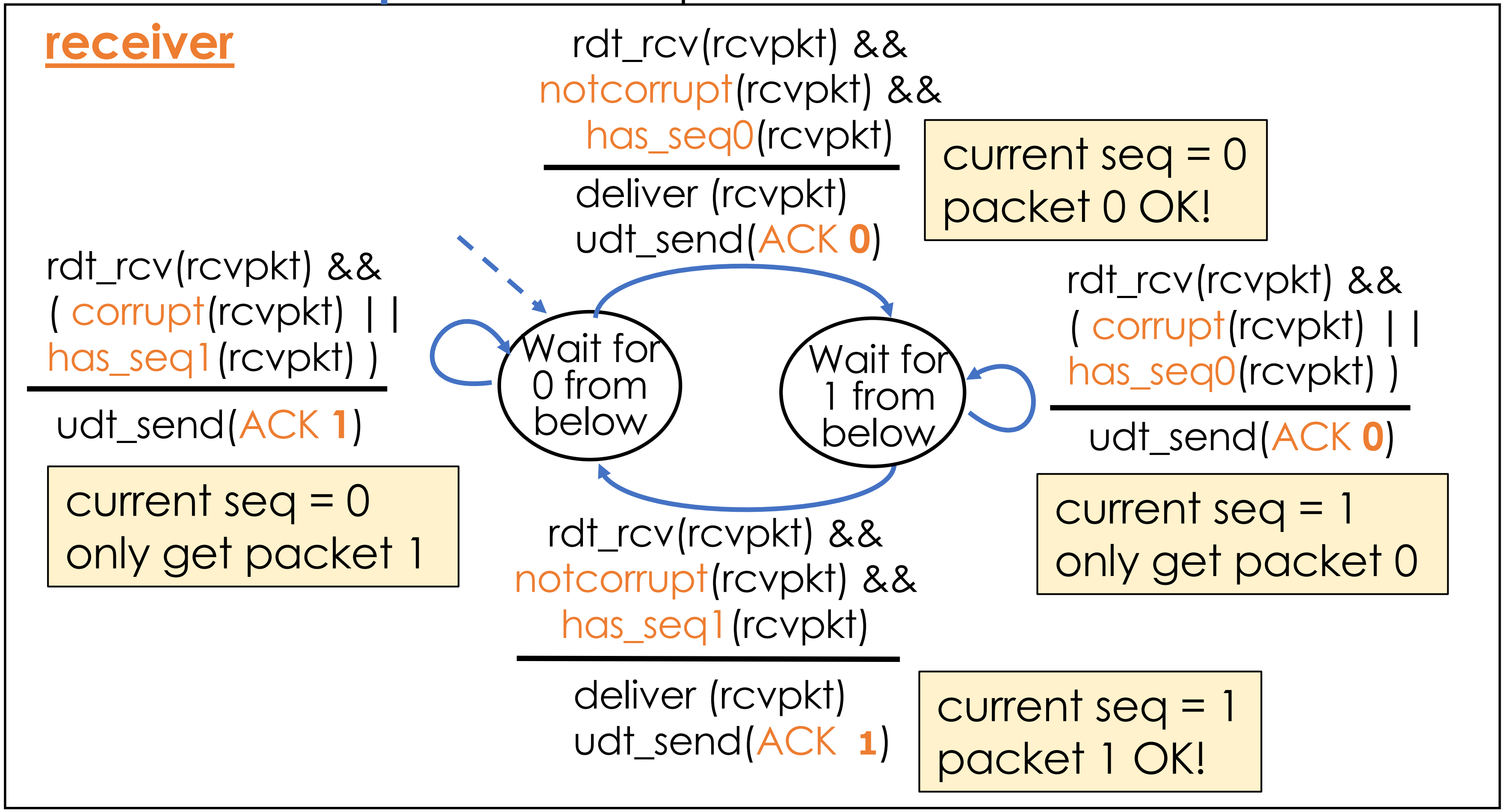
3. rdt over a channel with feedback errors

- Receiver can detect whether a transmission is
  - new packet or retransmission



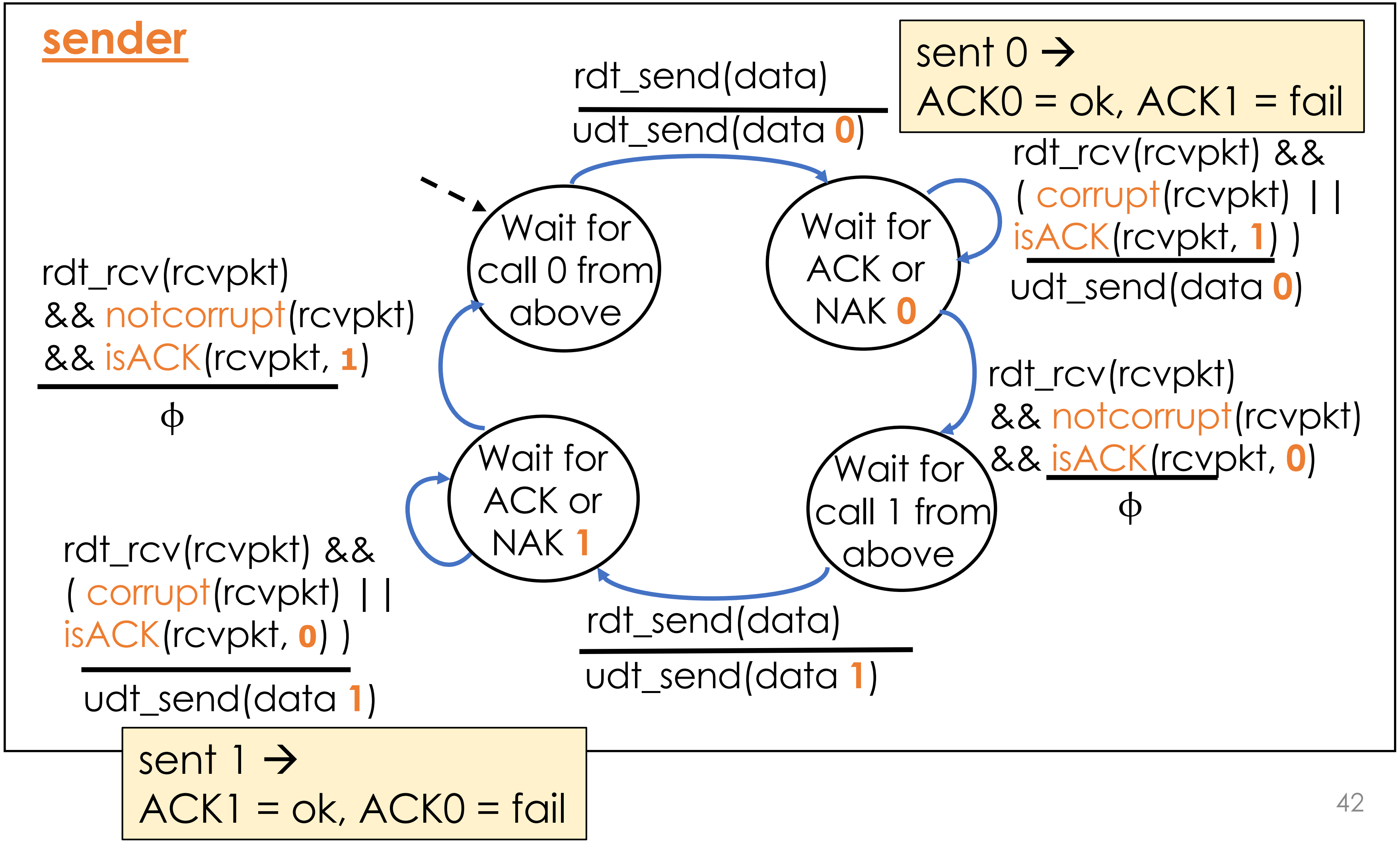
3. rdt over a channel with feedback errors

- Use only ACK, no NAK → put sequence number in the ACK!
  - In receiver, no error: ACK **“current”** seq; error: ACK **“previous”** seq



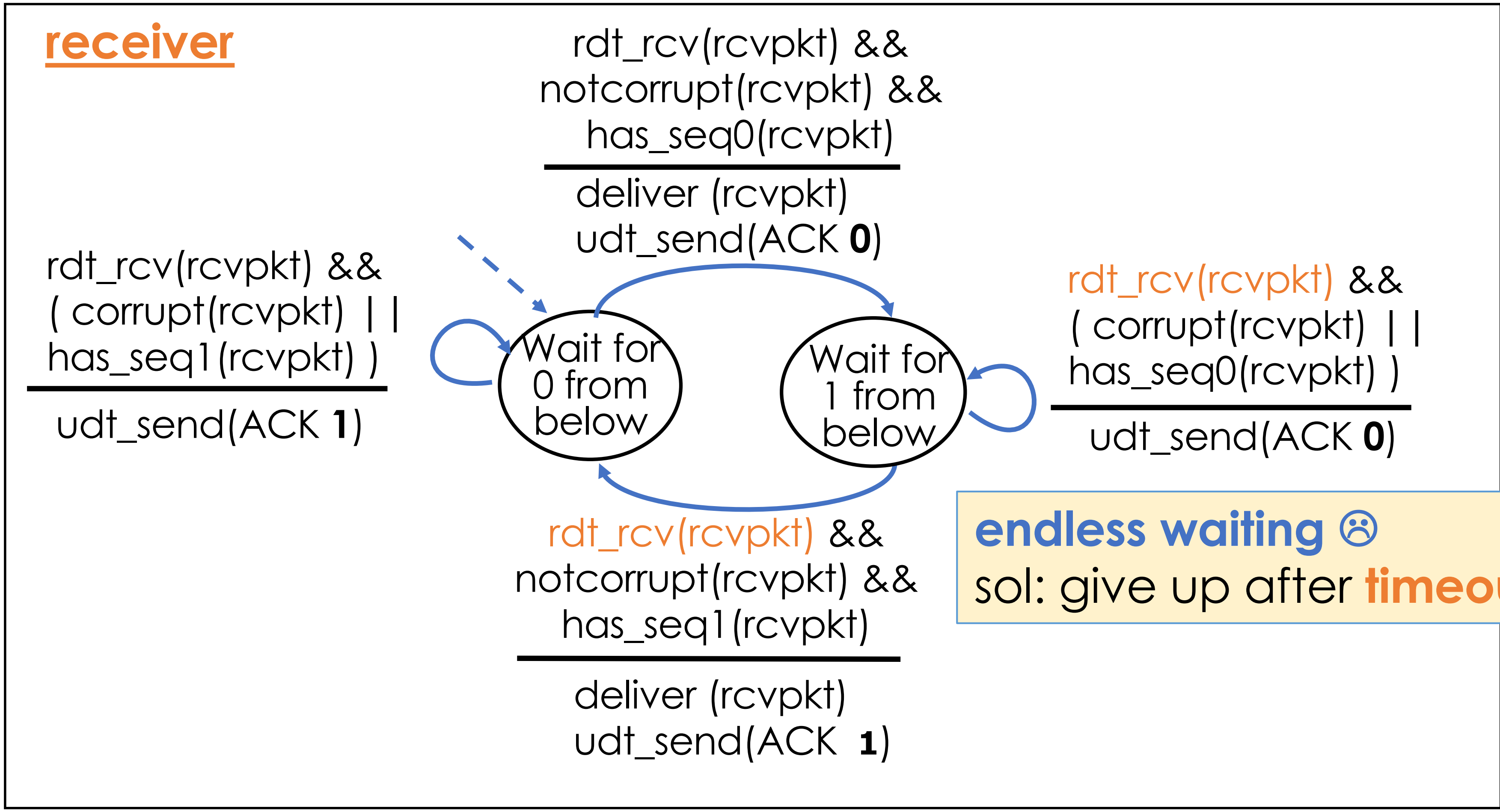
3. rdt over a channel with feedback errors

- Use only ACK, no NAK → put sequence number in the ACK!
  - In sender, **duplicate ACK** implies failure!



4. rdt over lossy channel

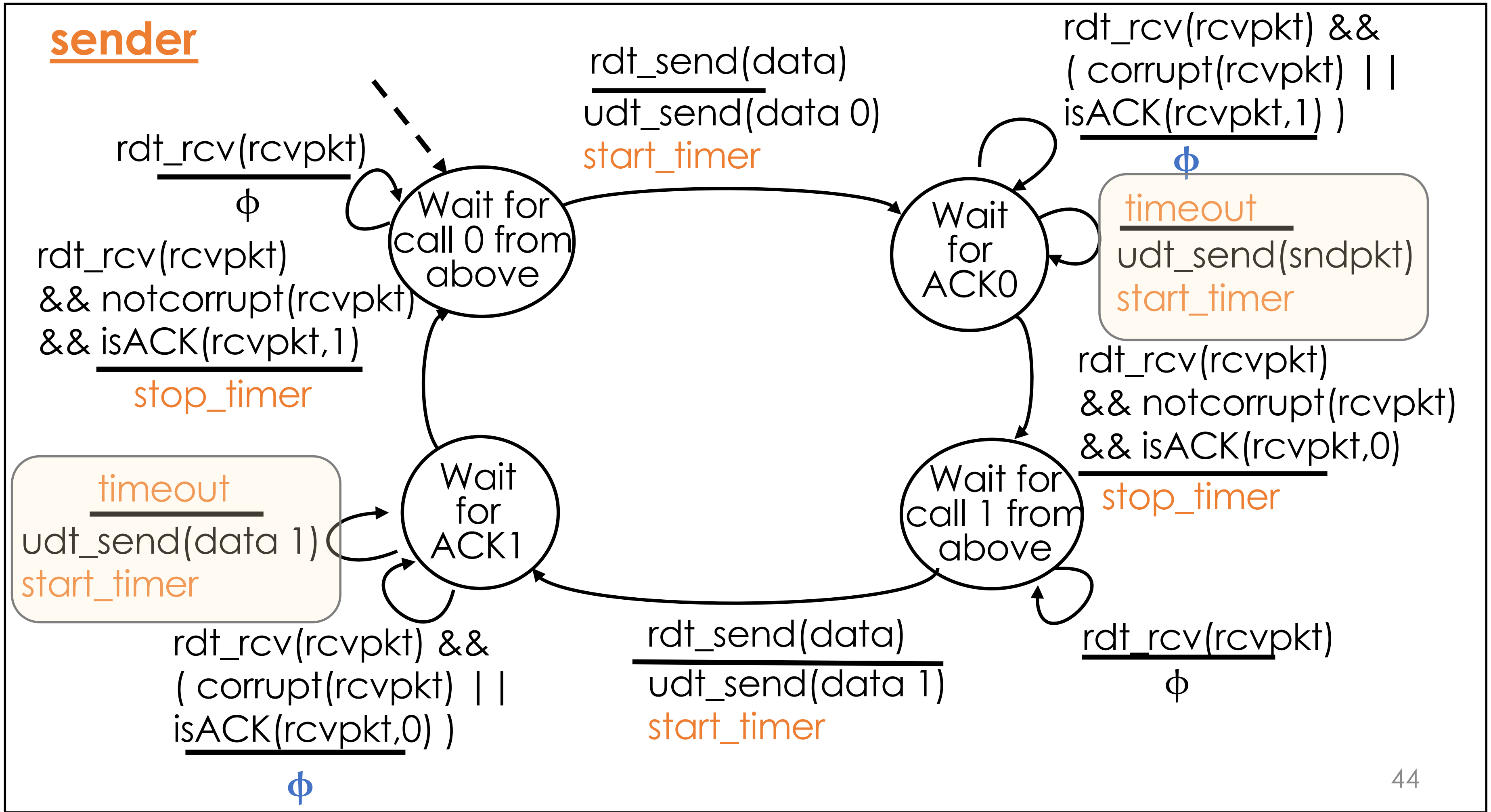
- What if a packet is lost (dropped before reaching the receiver)?



endless waiting ☹
sol: give up after **timeout**!

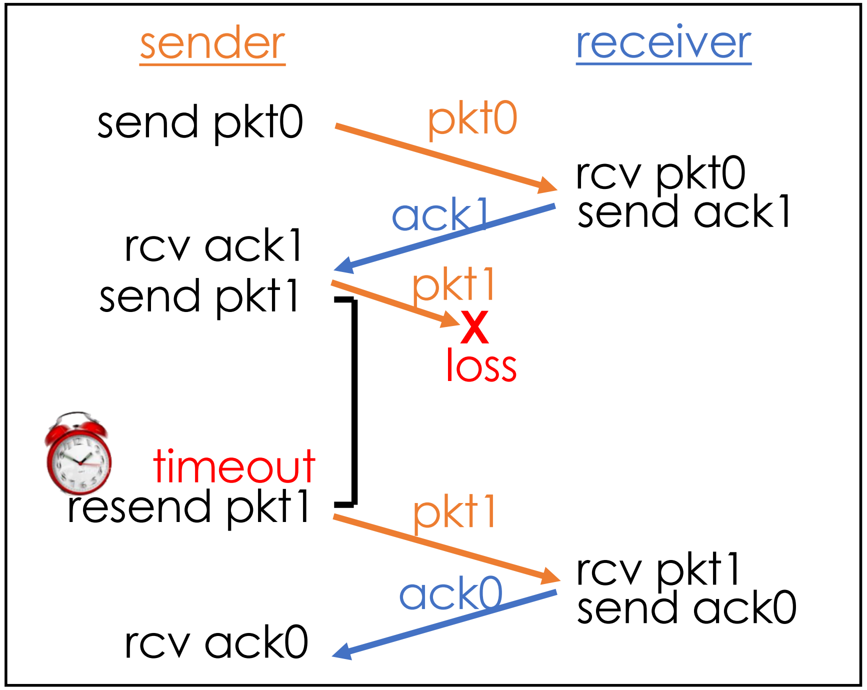
4. rdt over lossy channel

- Sender automatically retransmit after timeout
  - Namely, no ACK after a given time interval
  - Restart timer when sending new packet or re-tx

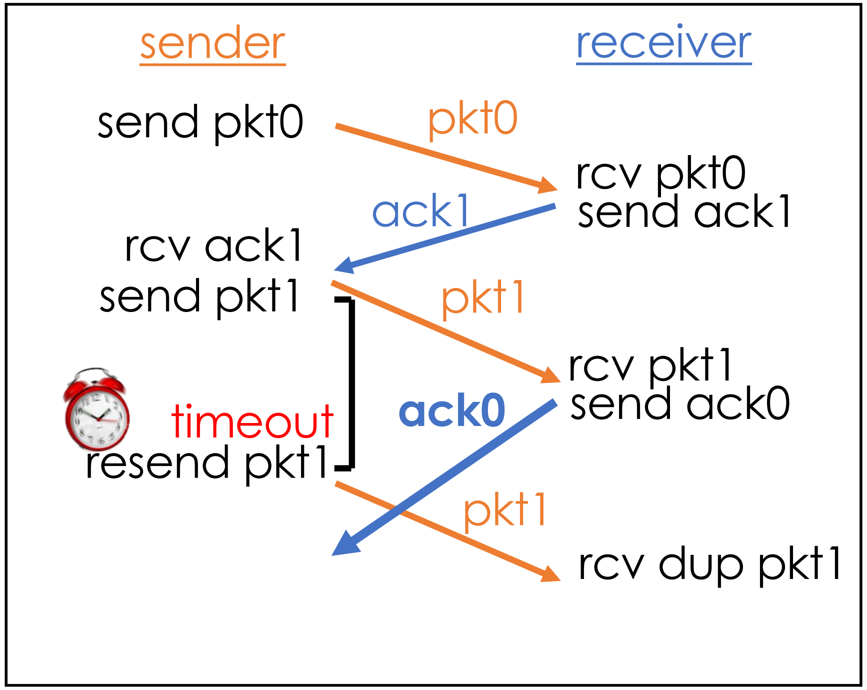


4. rdt over lossy channel

- Non optimal, but work! Why?
  - Either packet loss or ACK loss triggers retransmission
  - Re-tx even if the packet might eventually reach the destination but just experience longer delay
    - That is, some retransmissions might be unnecessary!



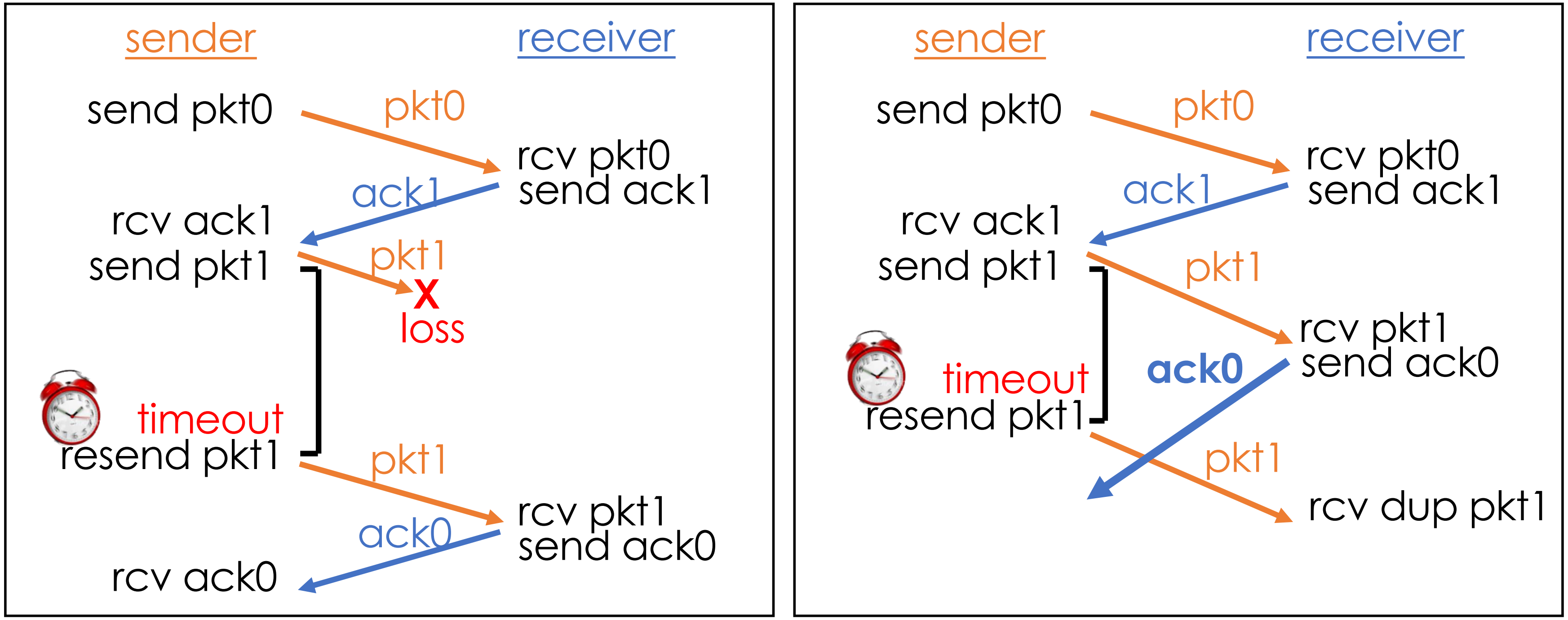
effective retransmission



unnecessary retransmission

4. rdt over lossy channel

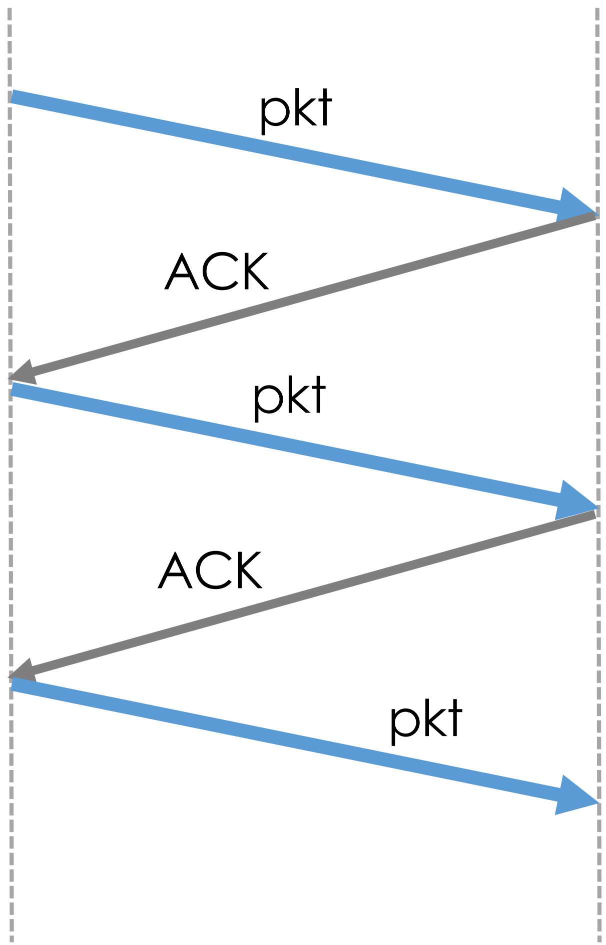
- How to setup timer?
  - Large timer: long waiting time → low utilization
  - Short timer: more unnecessary retransmissions



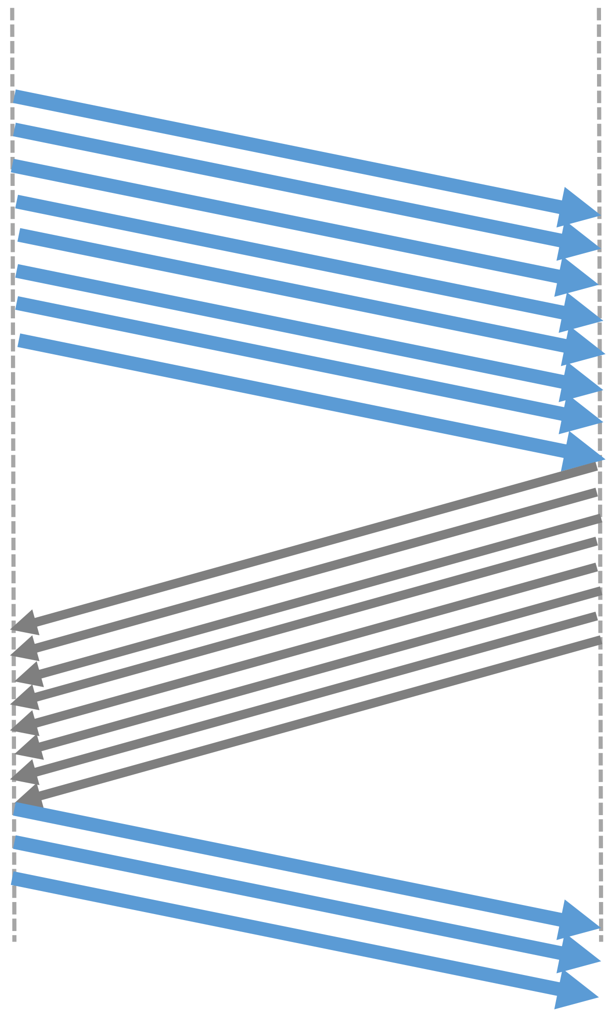
effective retransmission　　unnecessary retransmission

So far, we can transmit reliably!

But, is it good enough?



stop-and-wait
low utilization/throughput



pipeline
high utilization/throughput

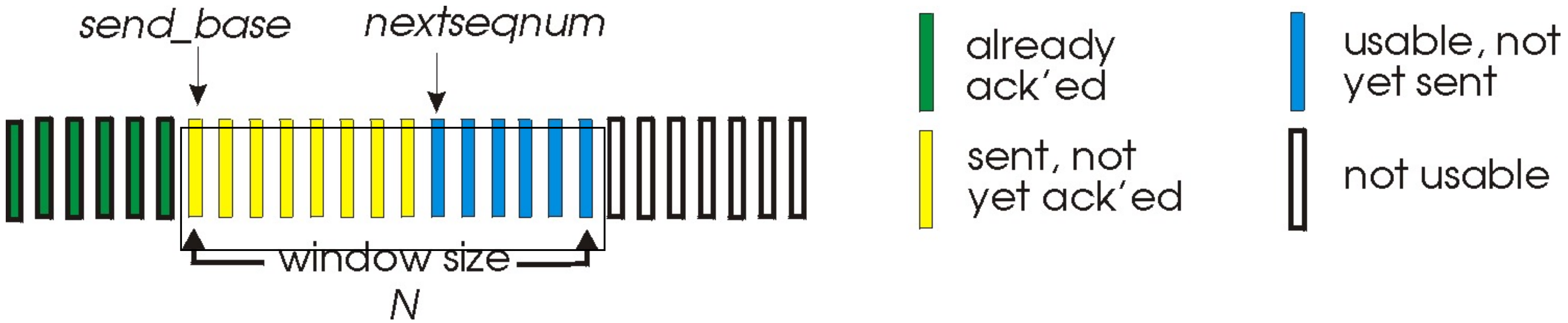
Pipeline rdt

Two implementation options

- **Go-back-N**
  - Sender transmits a batch of N packets at a time
  - sender only maintain a single timer
  - Receiver only send cumulative ACK (an ACK for several packets)

- **Selective Repeat**
  - Sender has at most N un-ACKed packets
  - Sender maintains a timer for every un-ACKed packet
  - Receiver sends an individual ACK for each packet

GBN's Cumulative ACK



- Receiver ACKs the last received sequence number
- Sender knows that everything before that has been correctly received