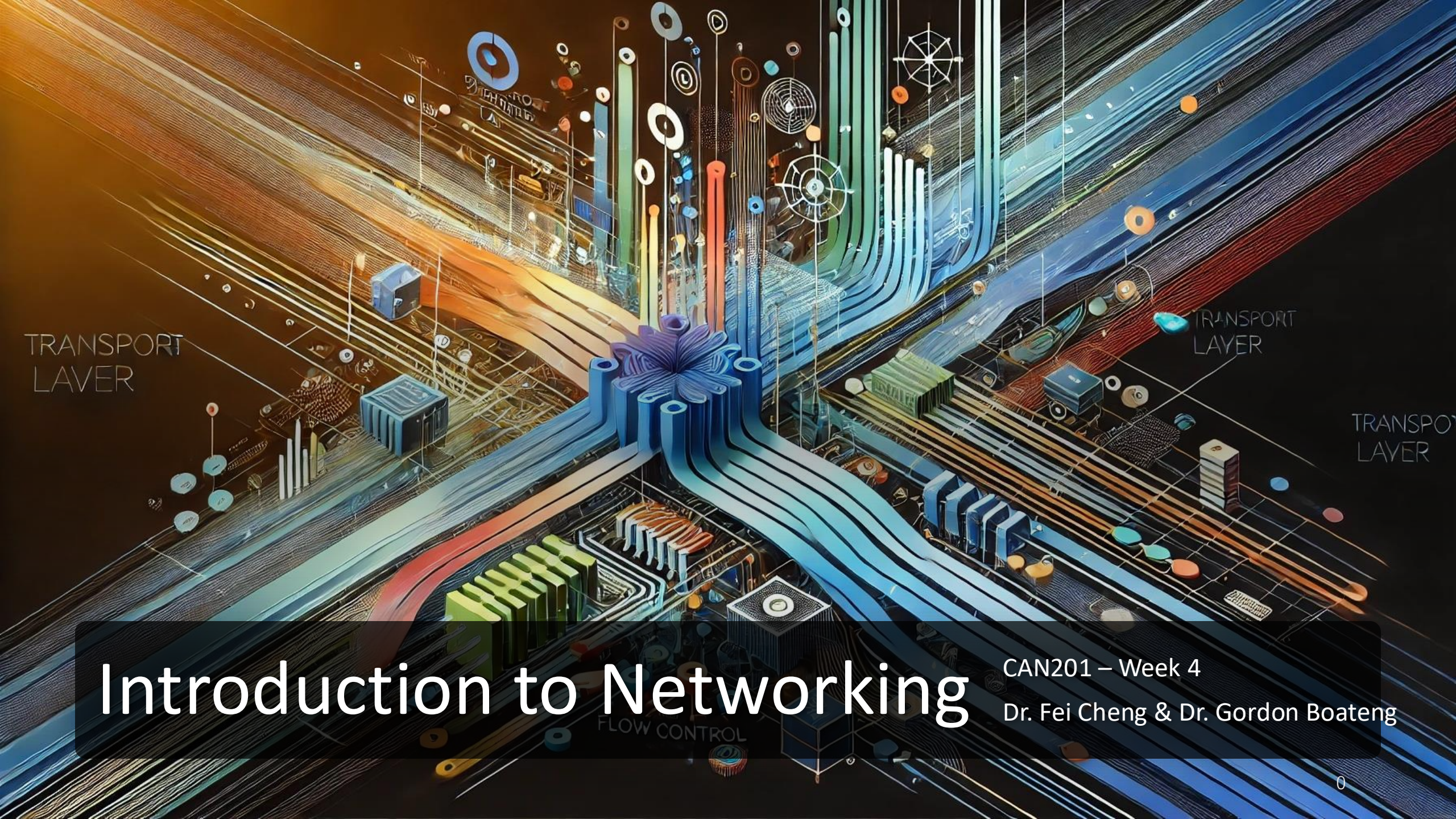


Introduction to Networking

CAN201 – Week 4

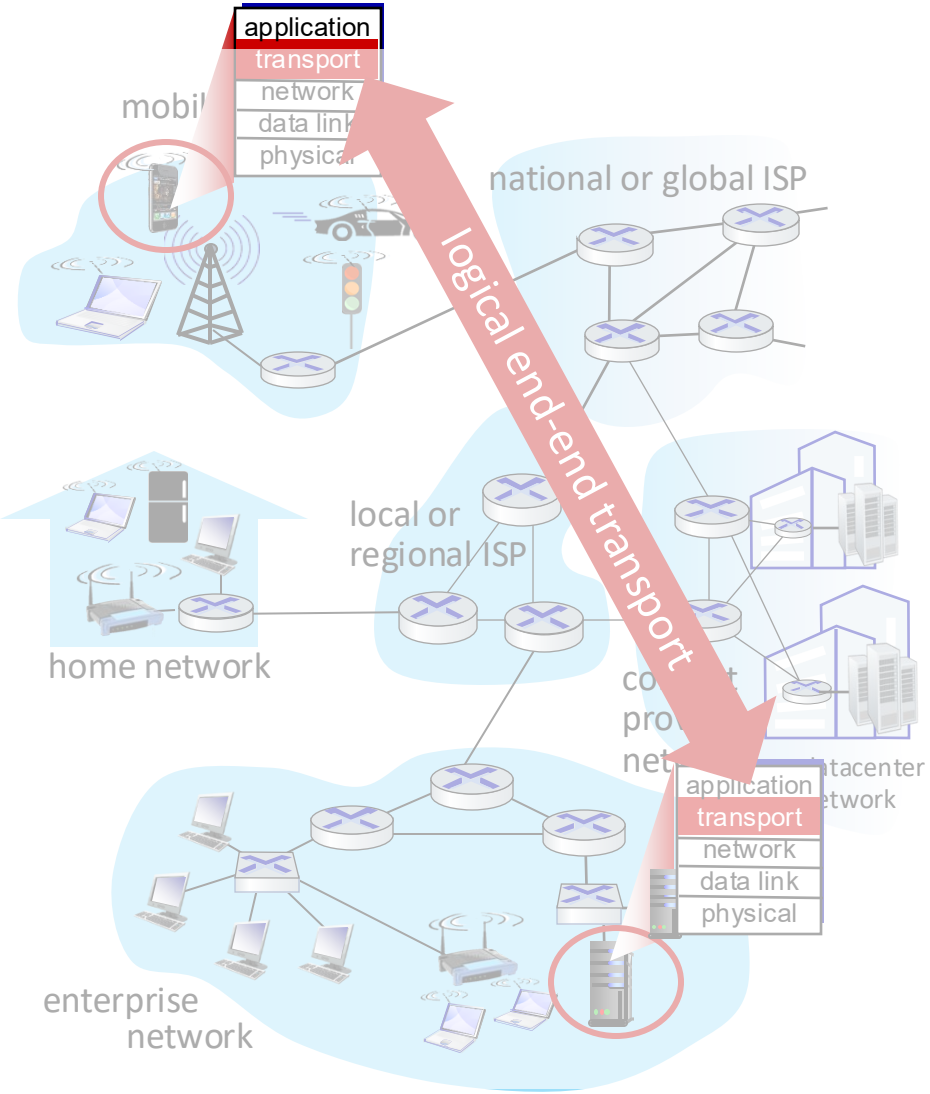
Dr. Fei Cheng & Dr. Gordon Boateng

Lecture 4 – Transport Layer (1)

- **Roadmap**
  1. Overview: Transport-layer services
  2. Multiplexing and demultiplexing
  3. Connectionless transport: UDP
  4. Principles of reliable data transfer

Transport services and protocols

- Provide logical communication between app processes running on different hosts
- Transport protocols run in end systems
  - Send side: breaks app msg into segments, passes to network layer
  - Rcv side: reassembles segments into messages, passes to app layer
- Transport-layer protocols for Internet:
  - TCP and UDP



Transport vs. Network layer

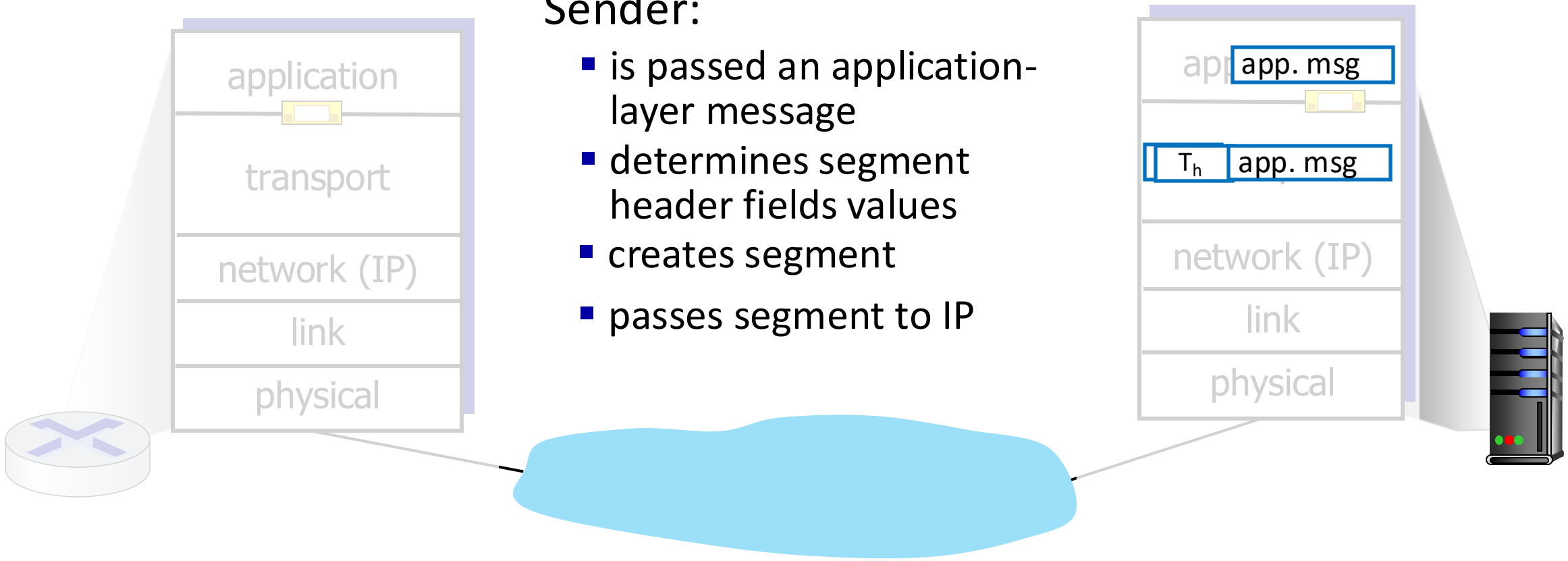
- **Transport layer: logical communication between processes**
  - Relies on, enhances, network layer services
- **Network layer: logical communication between hosts**

Household analogy:

10 kids in Ann's house sending letters to 10 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

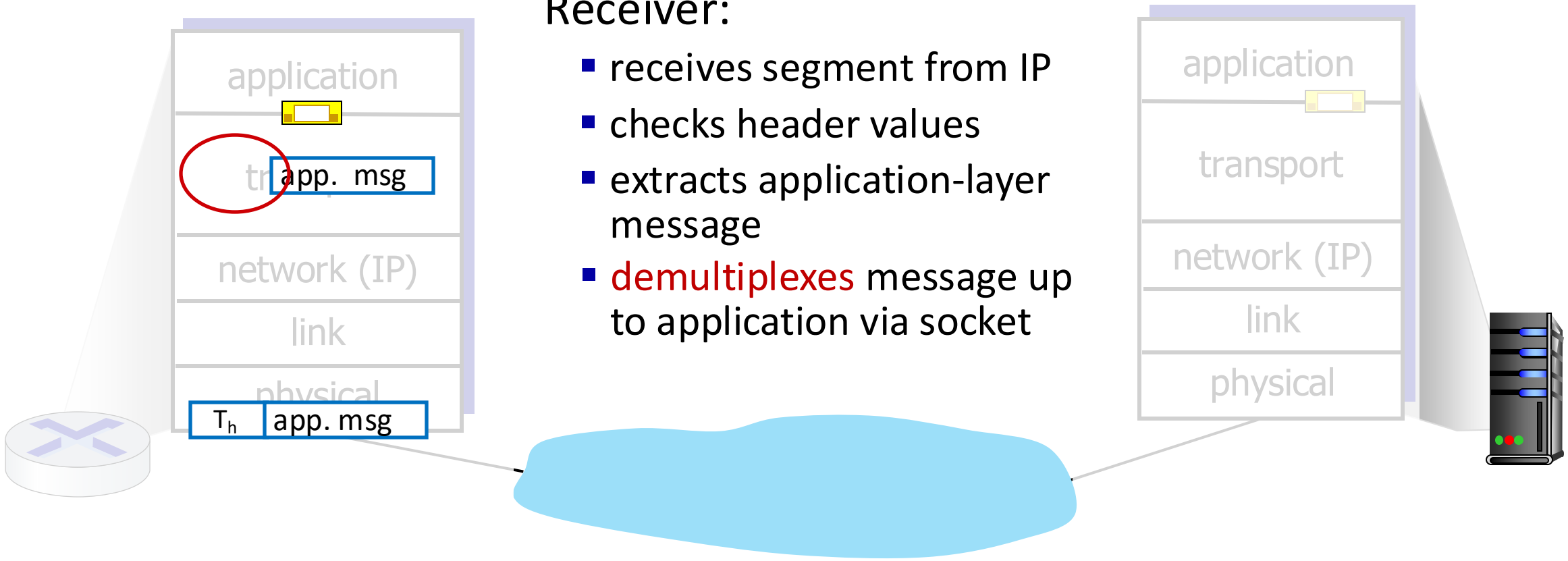
Transport Layer Actions



Sender:

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

Transport Layer Actions

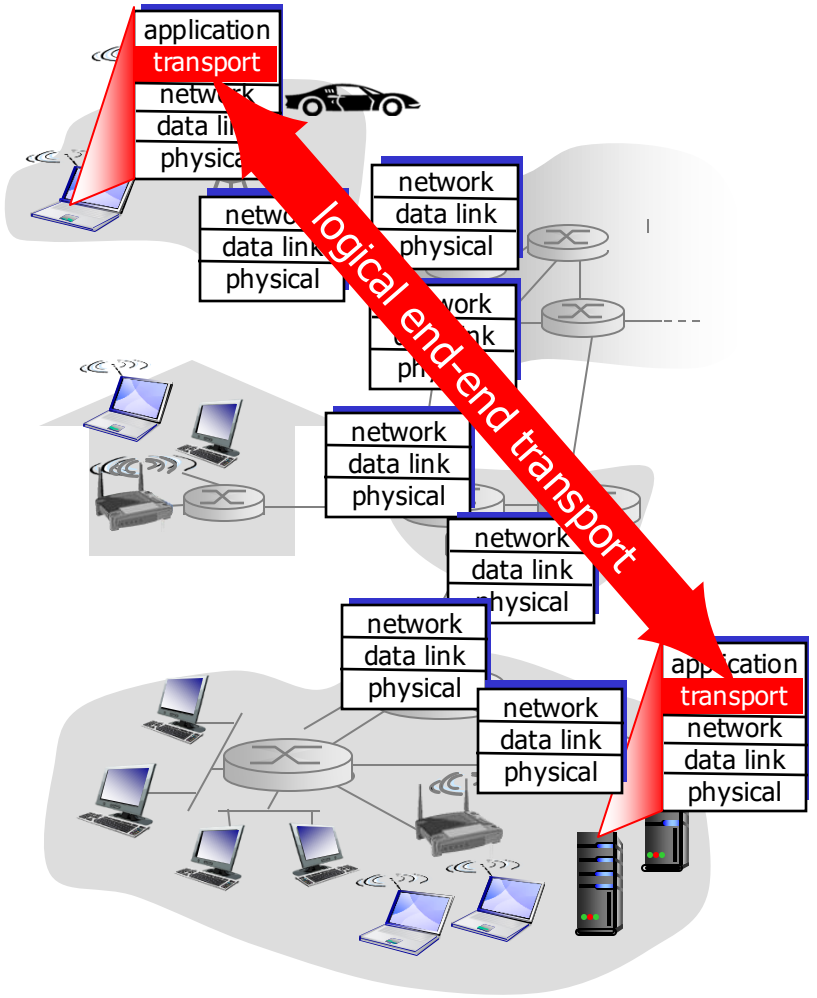


Receiver:

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

Two Principal Internet transport protocols

- **Unreliable, unordered delivery: UDP**
  - No-frills extension of "best-effort" network layer (internet protocol)
- **Reliable, in-order delivery: TCP**
  - Congestion control
  - Flow control
  - Connection setup
- **Services not available:**
  - Delay guarantees
  - Bandwidth guarantees



Feature	UDP	UDP-Lite	TCP	Multipath TCP	SCTP	DCCP	RUDP[a]
Packet header size	8 bytes	8 bytes	20–60 bytes	50–90 bytes	12 bytes[b]	12 or 16 bytes	14+ bytes
Typical data-packet overhead	8 bytes	8 bytes	20 bytes	?? bytes	44–48+ bytes[c]	12 or 16 bytes	14 bytes
Transport-layer packet entity	Datagram	Datagram	Segment	Segment	Datagram	Datagram	Datagram
Connection-oriented	No	No	Yes	Yes	Yes	Yes	Yes
Reliable transport	No	No	Yes	Yes	Yes	No	Yes
Unreliable transport	Yes	Yes	No	No	Yes	Yes	Yes
Preserve message boundary	Yes	Yes	No	No	Yes	Yes	Yes
Delivery	Unordered	Unordered	Ordered	Ordered	Ordered / Unordered	Unordered	Unordered
Data checksum	Optional	Yes	Yes	Yes	Yes	Yes	Optional
Checksum size	16 bits	16 bits	16 bits	16 bits	32 bits	16 bits	16 bits
Partial checksum	No	Yes	No	No	No	Yes	No
Path MTU	No	No	Yes	Yes	Yes	Yes	?
Flow control	No	No	Yes	Yes	Yes	No	Yes
Congestion control	No	No	Yes	Yes	Yes	Yes	?
Explicit Congestion Notification	No	No	Yes	Yes	Yes	Yes	?
Multiple streams	No	No	No	No	Yes	No	No
Multi-homing	No	No	No	Yes	Yes	No	No
Bundling / Nagle	No	No	Yes	Yes	Yes	No	?

More transport layer protocols

Lecture 4 – Transport Layer (1)

- **Roadmap**
  1. Overview: Transport-layer services
  2. Multiplexing and demultiplexing
  3. Connectionless transport: UDP
  4. Principles of reliable data transfer

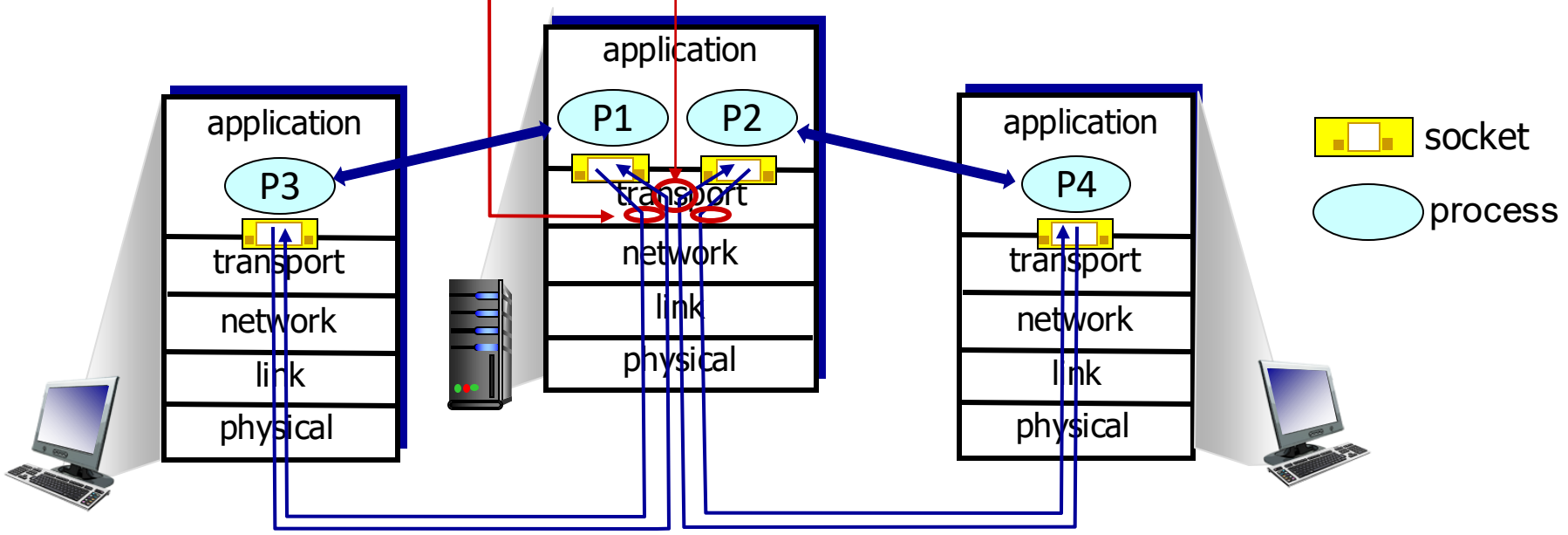
Multiplexing/demultiplexing

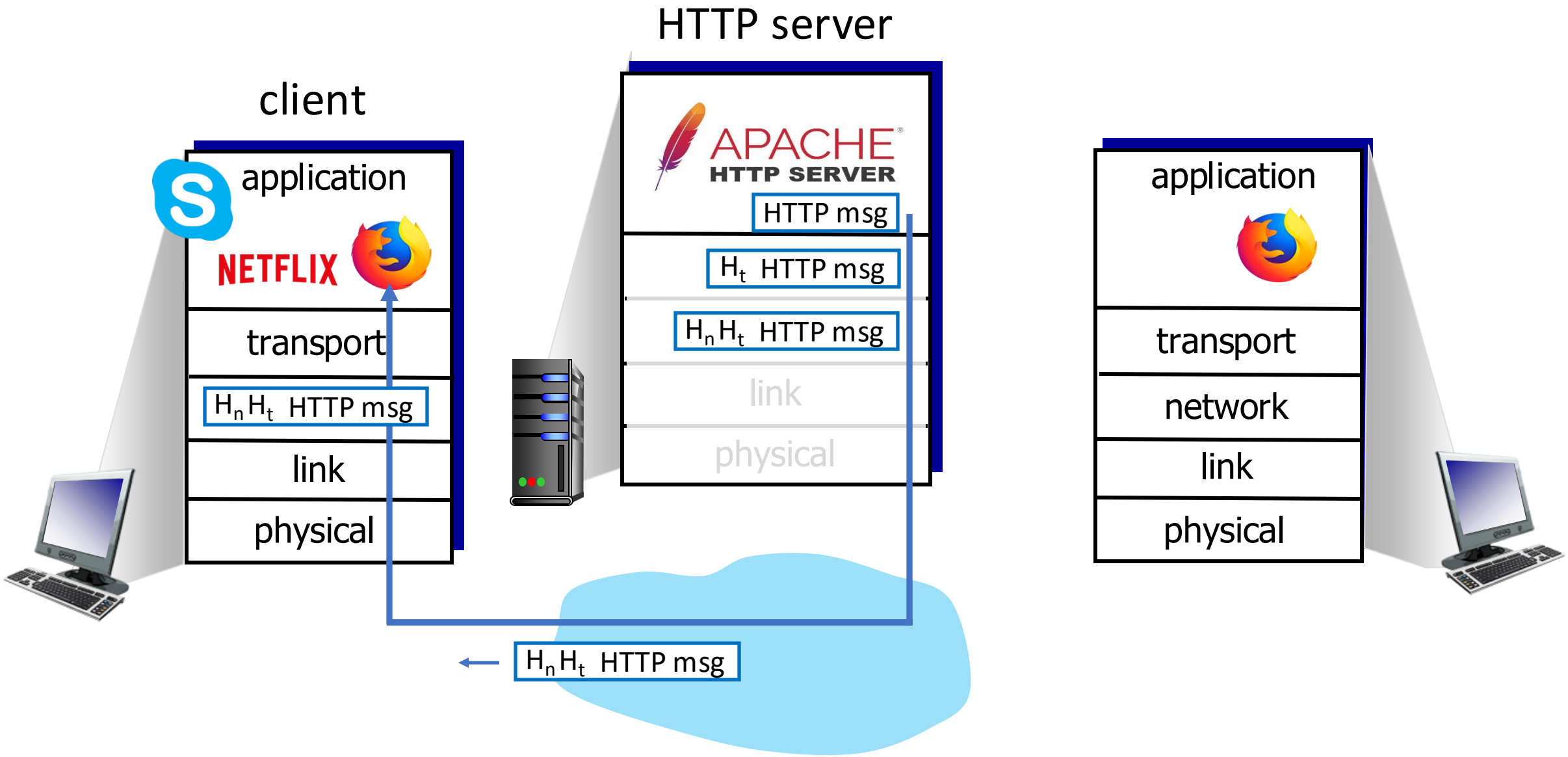
multiplexing as sender:

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

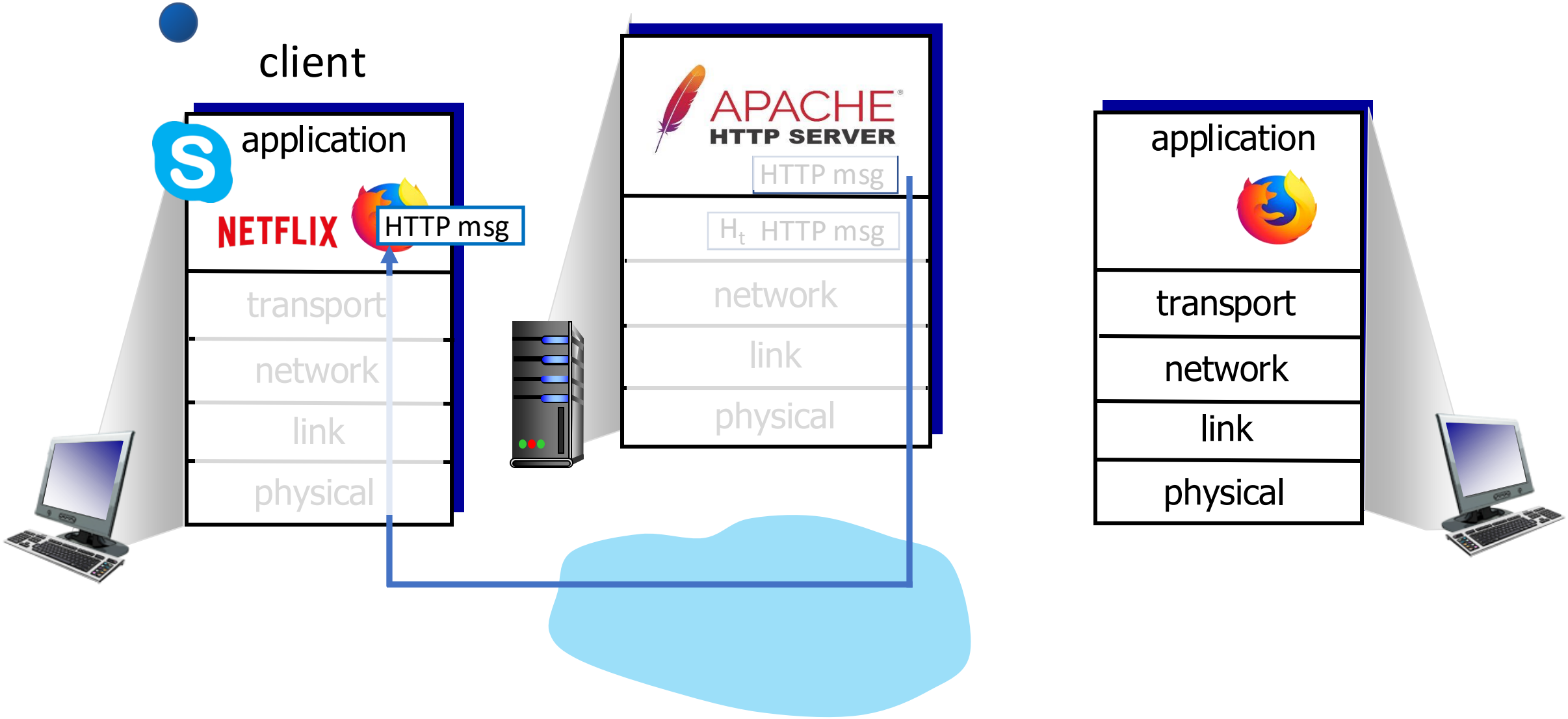
demultiplexing as receiver:

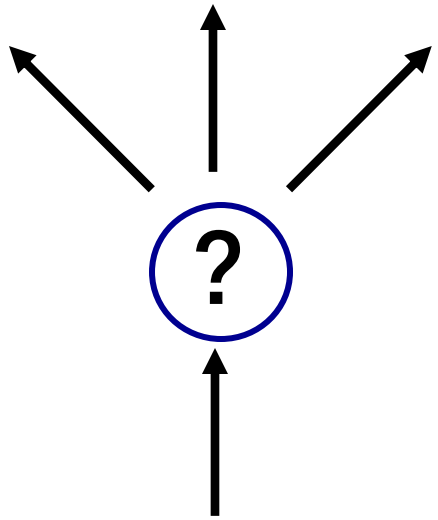
use header info to deliver received segments to correct socket



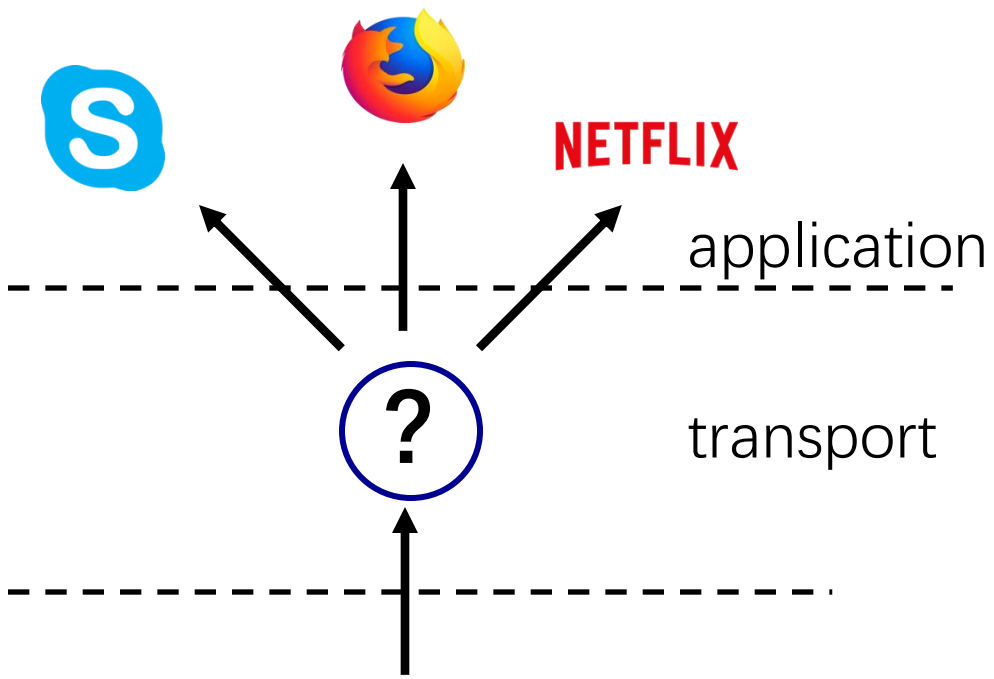

client
application
NETFLIX
transport
$H_n H_t$ HTTP msg
link
physical
HTTP server
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
$H_n H_t$ HTTP msg
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg

Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?



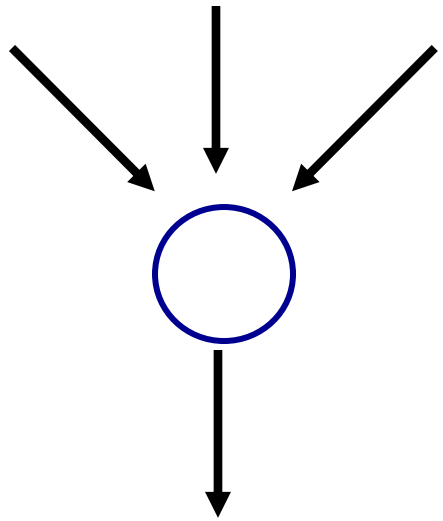


de-multiplexing

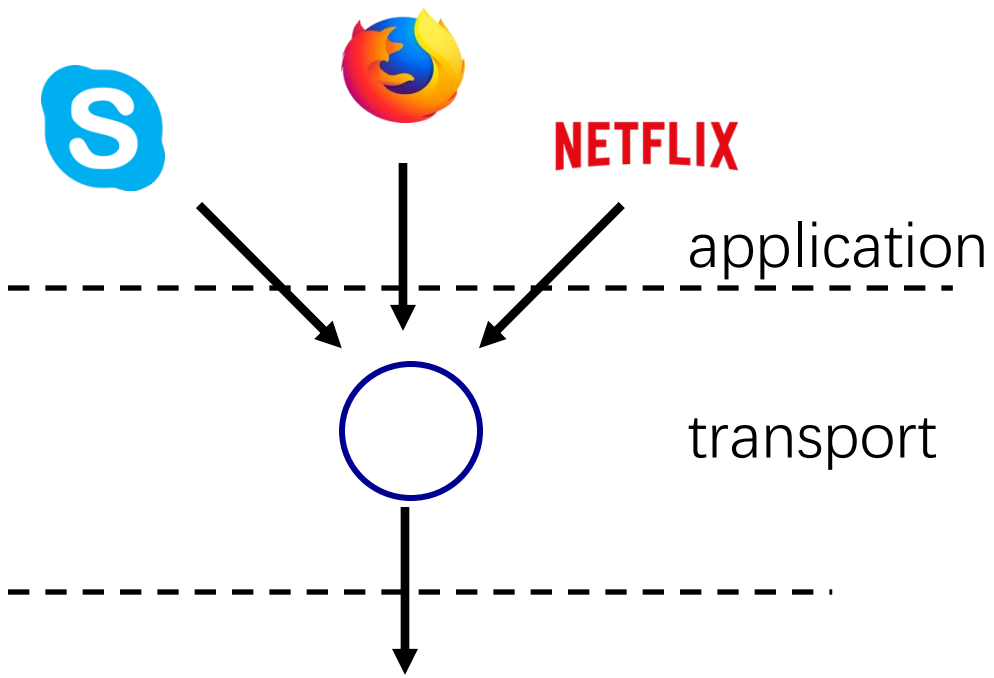


de-multiplexing

Demultiplexing



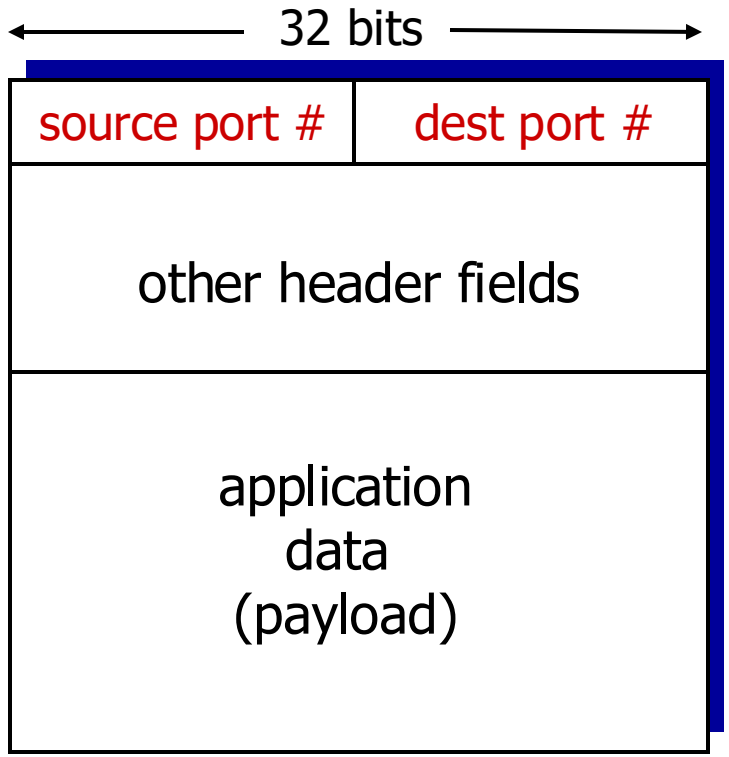
multiplexing


NETFLIX
application
transport
multiplexing


Multiplexing

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 suitable socket**



TCP/UDP segment format

Connectionless demultiplexing (UDP)

- **Created socket has host-local port number:**

```
Socket = socket(AF_INET, SOCK_DGRAM)
Socket.bind(('', 12345))
```

- **When creating datagram to send into UDP socket, must specify:**
  - **Destination IP address**
  - **Destination port number**

```
clientSocket.sendto(msg,(server_name, server_port))
```

- **When host receives UDP segment:**
  - Checks destination port # in segment
  - Directs UDP segment to socket with that port #

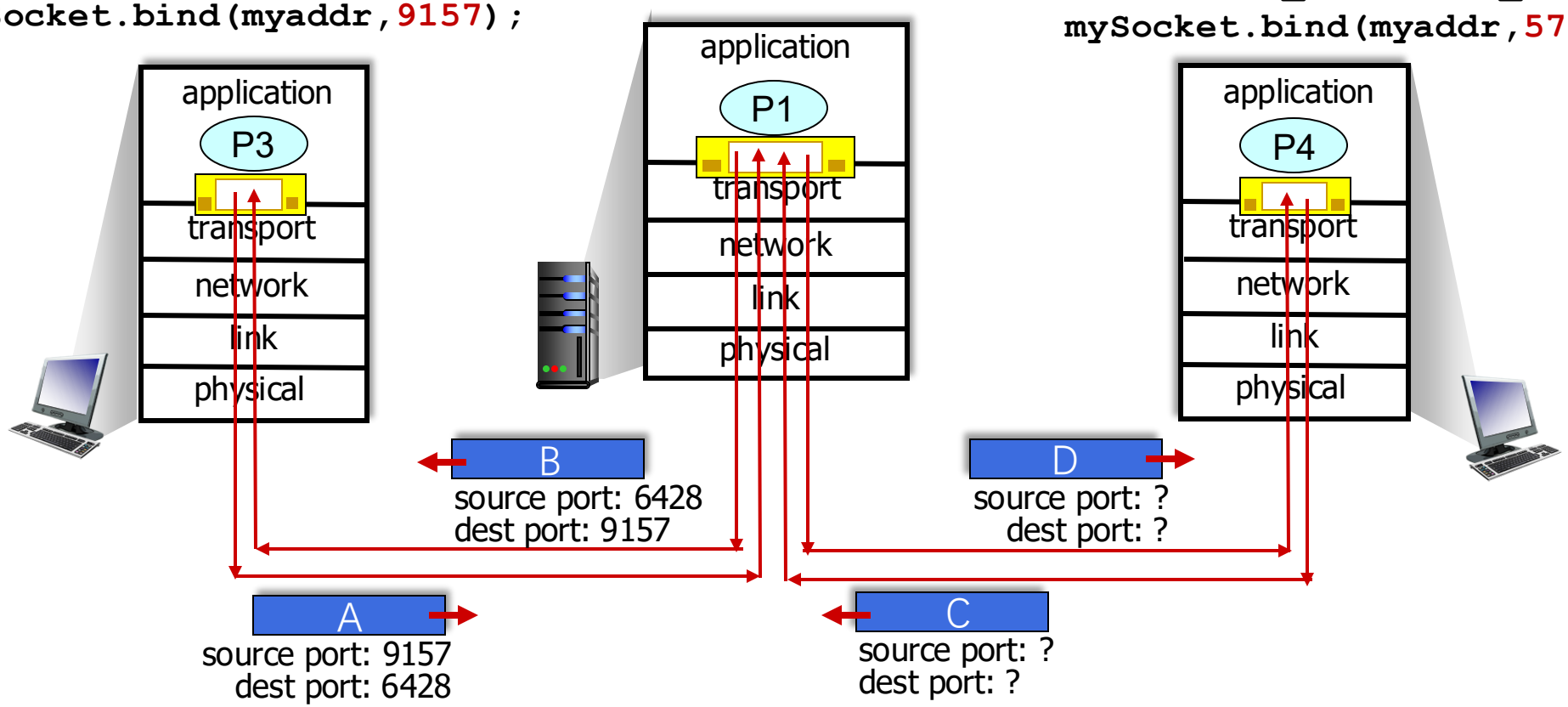
IP datagrams with same dest. port #, but different source IP addresses and/or source port numbers will be directed to same socket at dest

Connectionless demultiplexing: an example

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

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

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

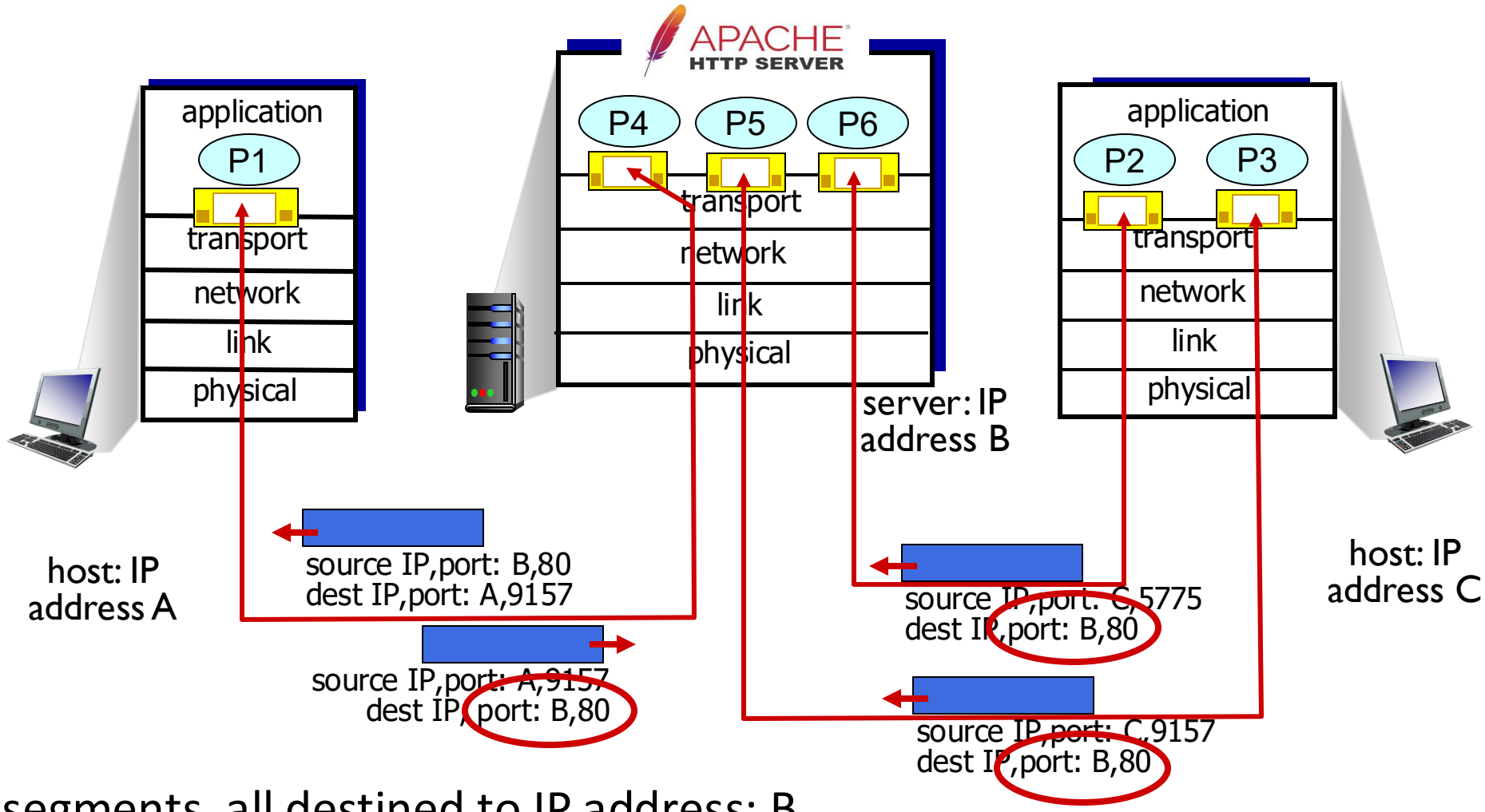


Connection-oriented demux

- **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**
  - non-persistent HTTP will have different socket for each request

Connection-oriented demultiplexing: example



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

Lecture 4 – Transport Layer (1)

- **Roadmap**
  1. Overview: Transport-layer services
  2. Multiplexing and demultiplexing
  3. Connectionless transport: UDP
  4. Principles of reliable data transfer

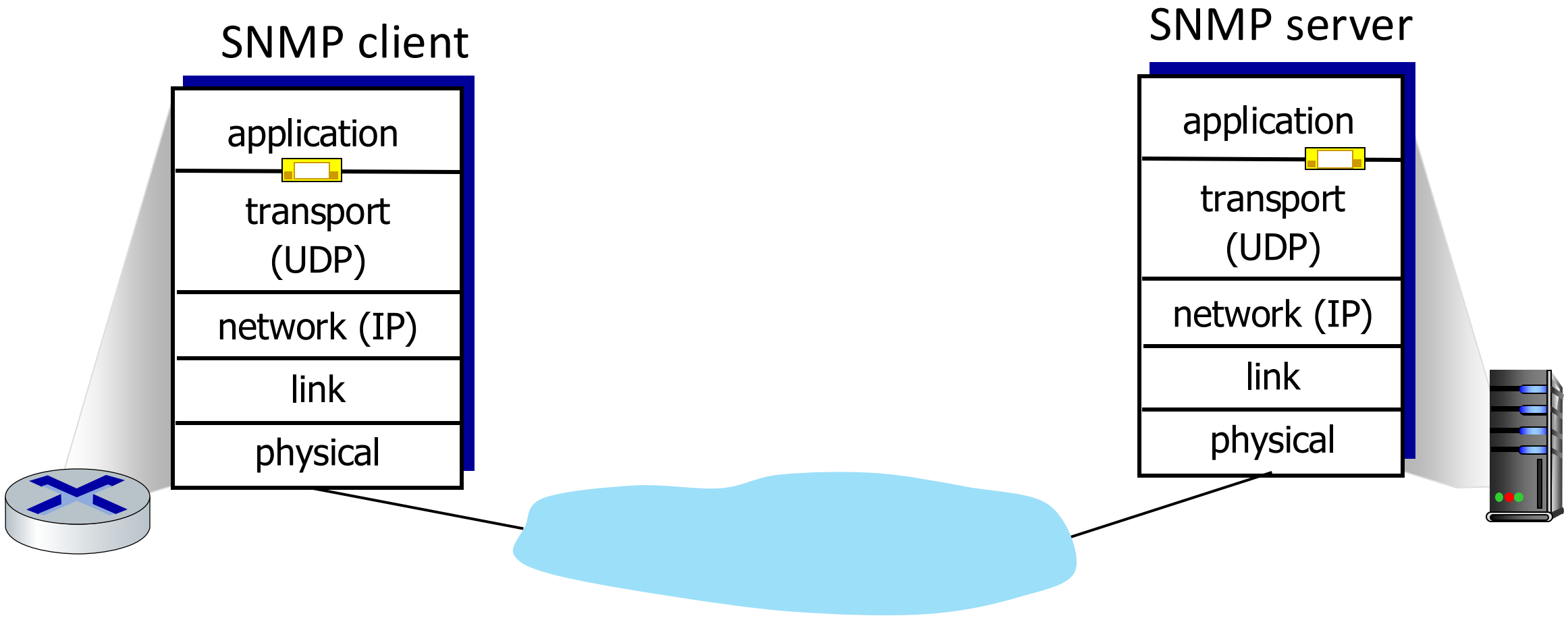
UDP: User Datagram Protocol [RFC 768]

Feature:

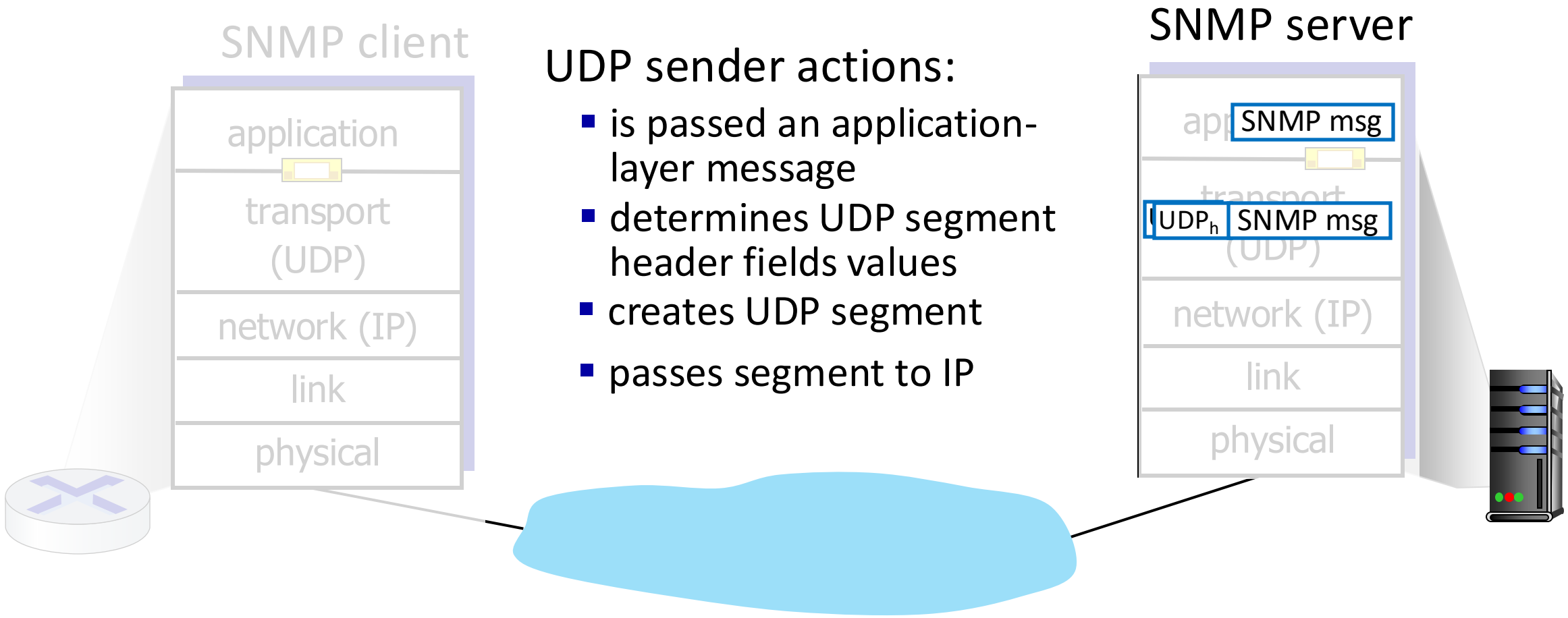
- **Simple and straightforward**
- **Best effort**
- **Lost**
- **Connectionless**
  - No handshaking
  - Each UDP segment handled independently of others
    - Out-of-order to APP

- **UDP use:**
  - Streaming multimedia apps
  - DNS
  - HTTP/3
- **Reliable transfer over UDP:**
  - Add reliability at application layer
  - Application-specific error recovery!

UDP: Transport Layer Actions



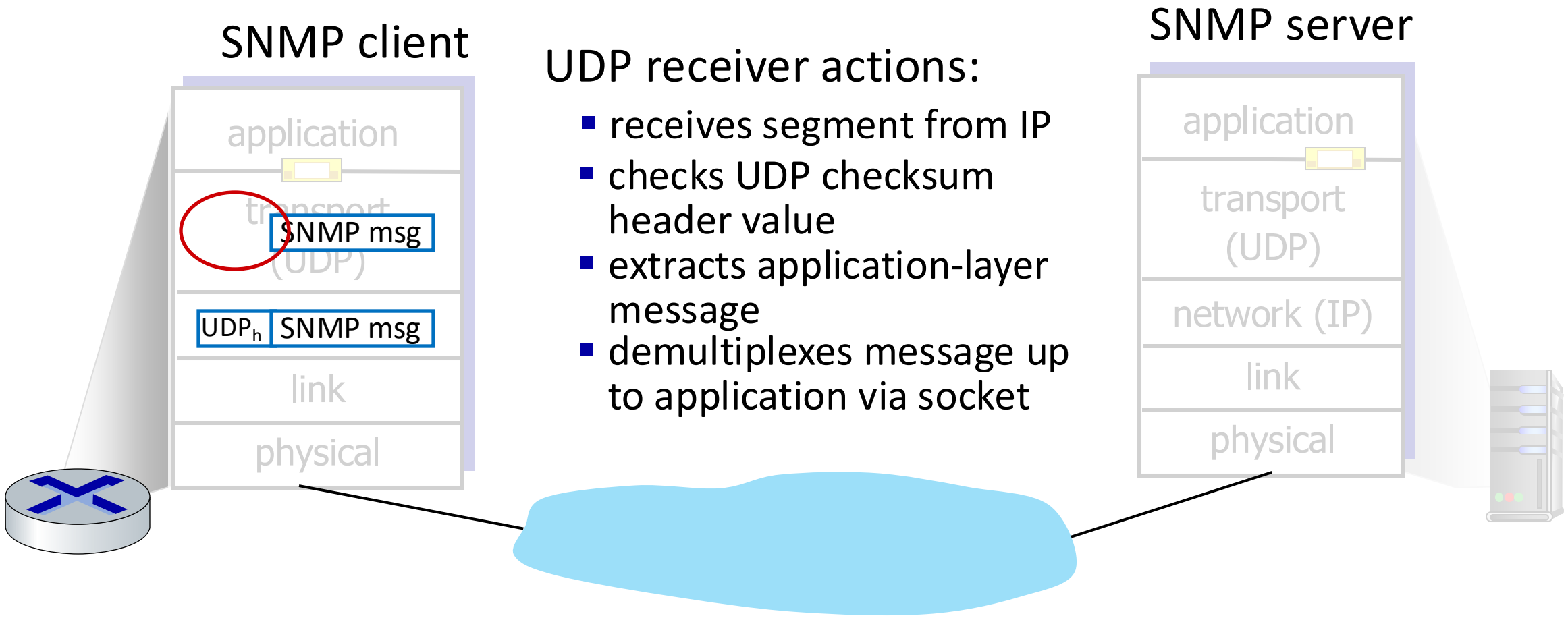
UDP: Transport Layer Actions



UDP sender actions:

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

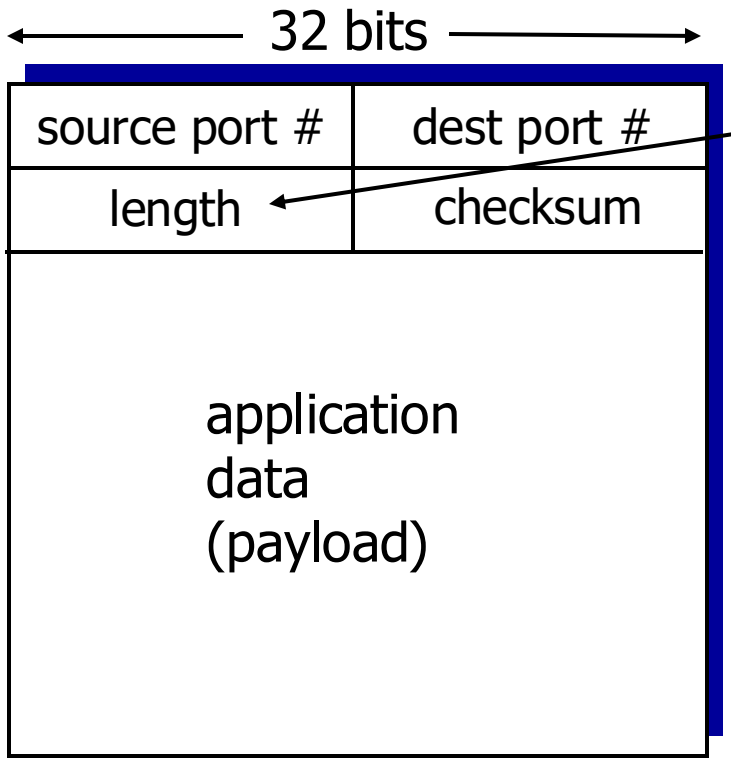
UDP: Transport Layer Actions



UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

UDP: segment header



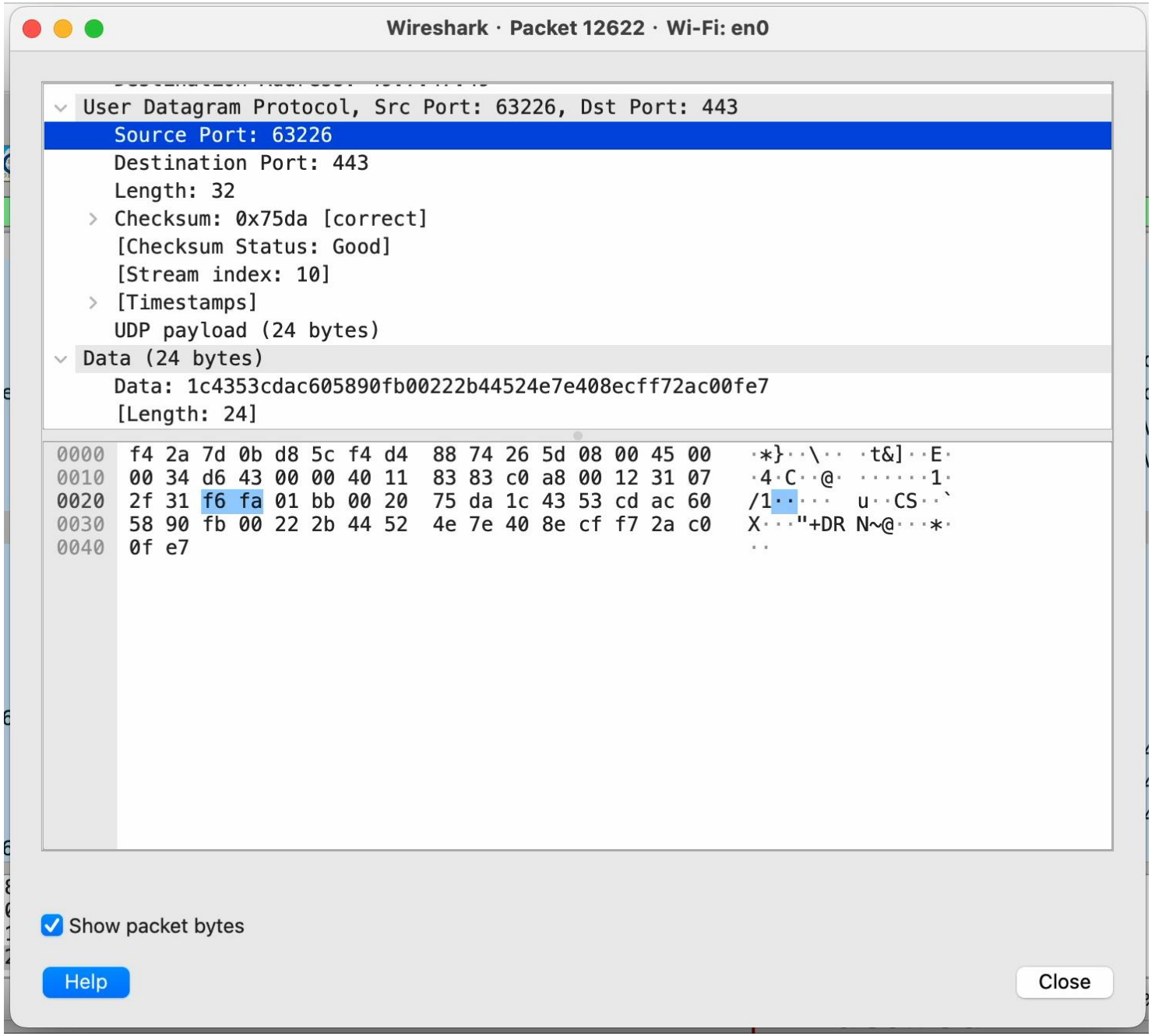
UDP segment format

length, in bytes of UDP segment, including header

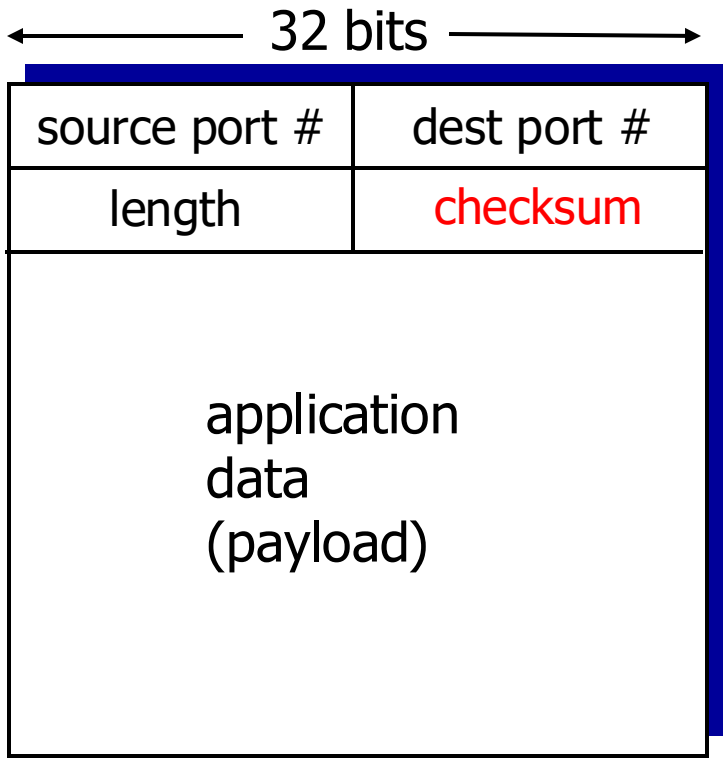
why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state at sender, receiver
- Small header size
- No congestion control: UDP can blast away as fast as desired

Real UDP datagram



UDP: segment header



UDP segment format

UDP checksum

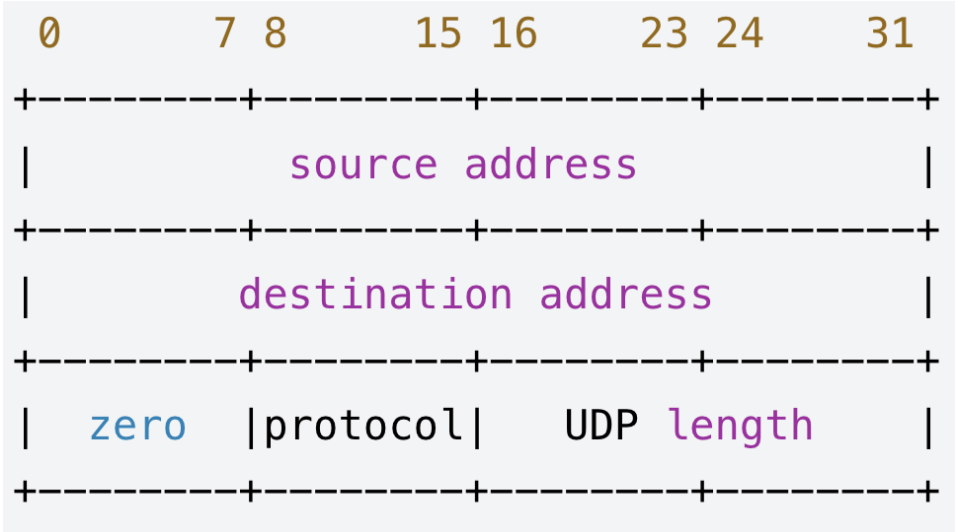
- Goal: detect “errors” in transmitted segment

Sender:

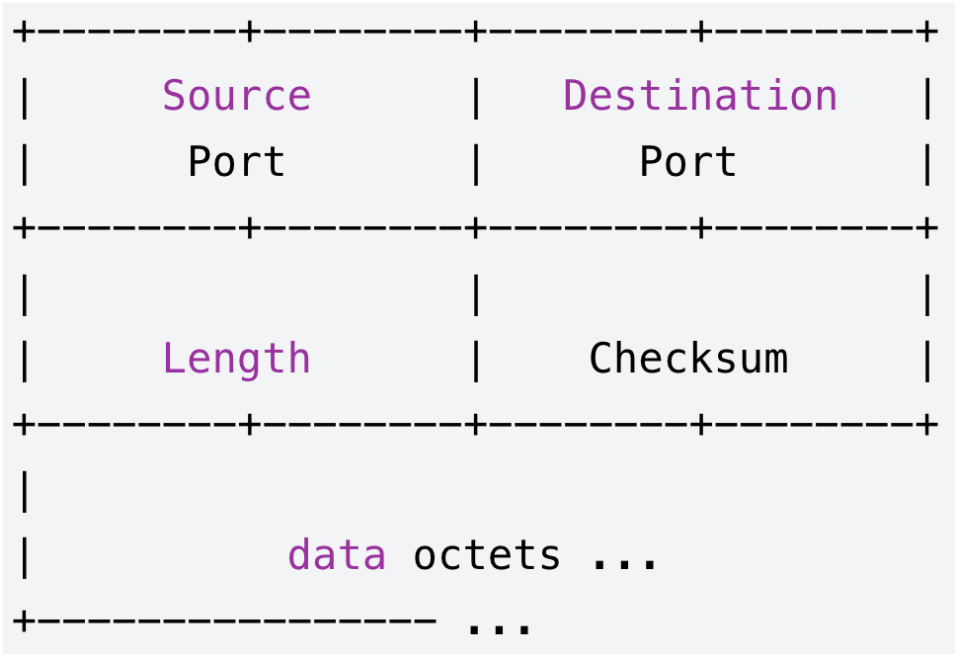
- Treat segment contents, including header fields, as sequence of 16-bit integers
- Checksum: addition (one’s complement sum) of **pseudo header, UDP header** and **UDP data**
- Sender puts checksum value into UDP checksum field

Receiver

- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless?



Pseudo-header



UDP Header + Data

Internet 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
-------------------------------------------------
```

Internet 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
-------------------------------------------------
                                                1
```

Internet 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
-------------------------------------------------
                                          1  1
```

Internet 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
-------------------------------------------------
                                       0  1  1
```

Internet 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
-------------------------------------------------
                                    1  0  1  1
```

Internet 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
-------------------------------------------------
                                    1  1  0  1  1
```

Internet 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
-------------------------------------------------
                                1  1  1  0  1  1
```

Internet 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
-------------------------------------------------
                                0  1  1  1  0  1  1
```

Internet 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
-------------------------------------------------
                                1  0  1  1  1  0  1  1
```

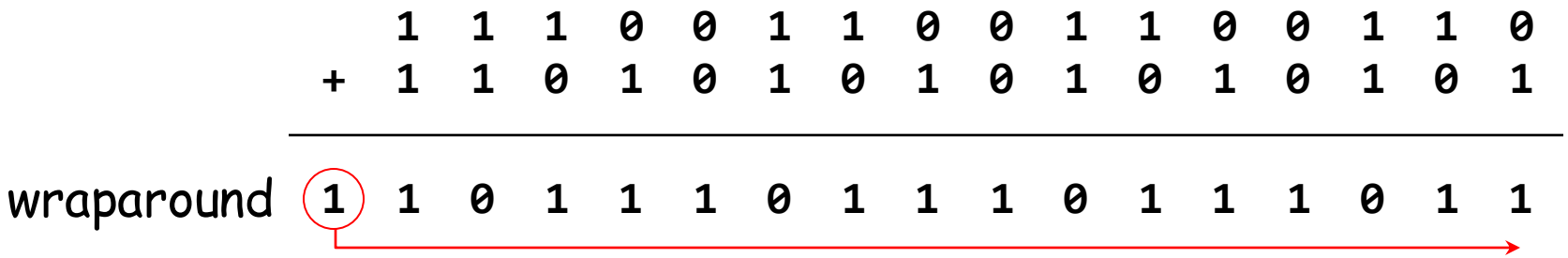
Internet checksum: example

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
-------------------------------------------------
   1  1  0  1  1  1  0  1  1  1  0  1  1  1  0  1  1
```

Internet checksum: example

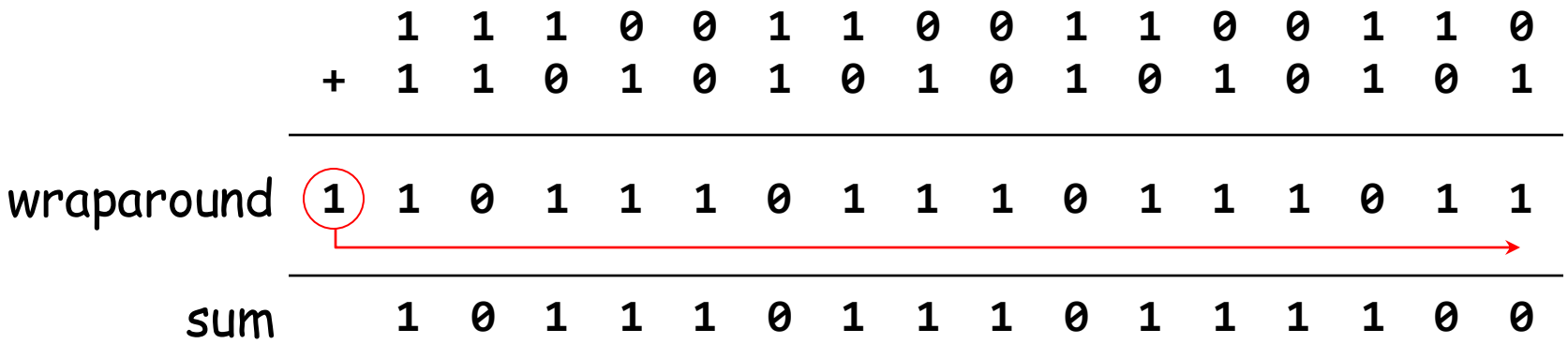
Example: add two 16-bit integers



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

Internet checksum: example

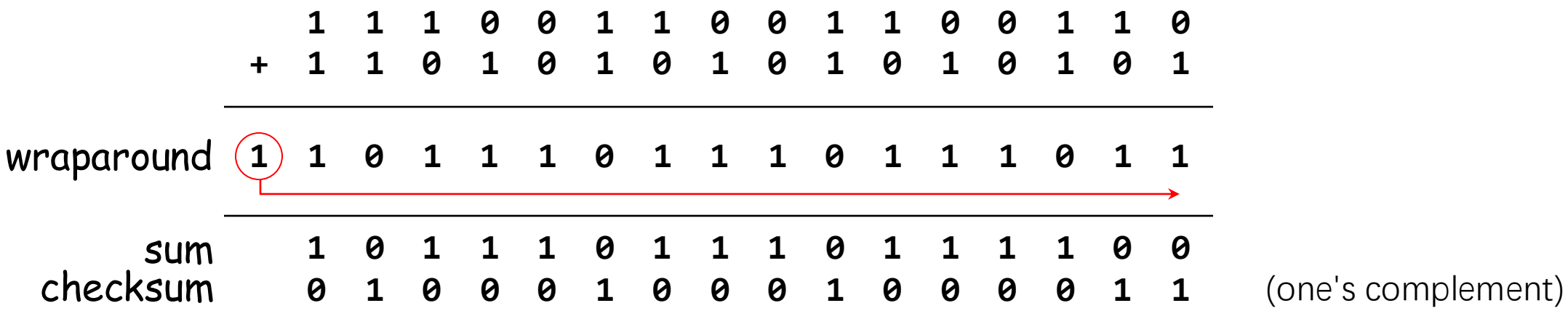
Example: add two 16-bit integers



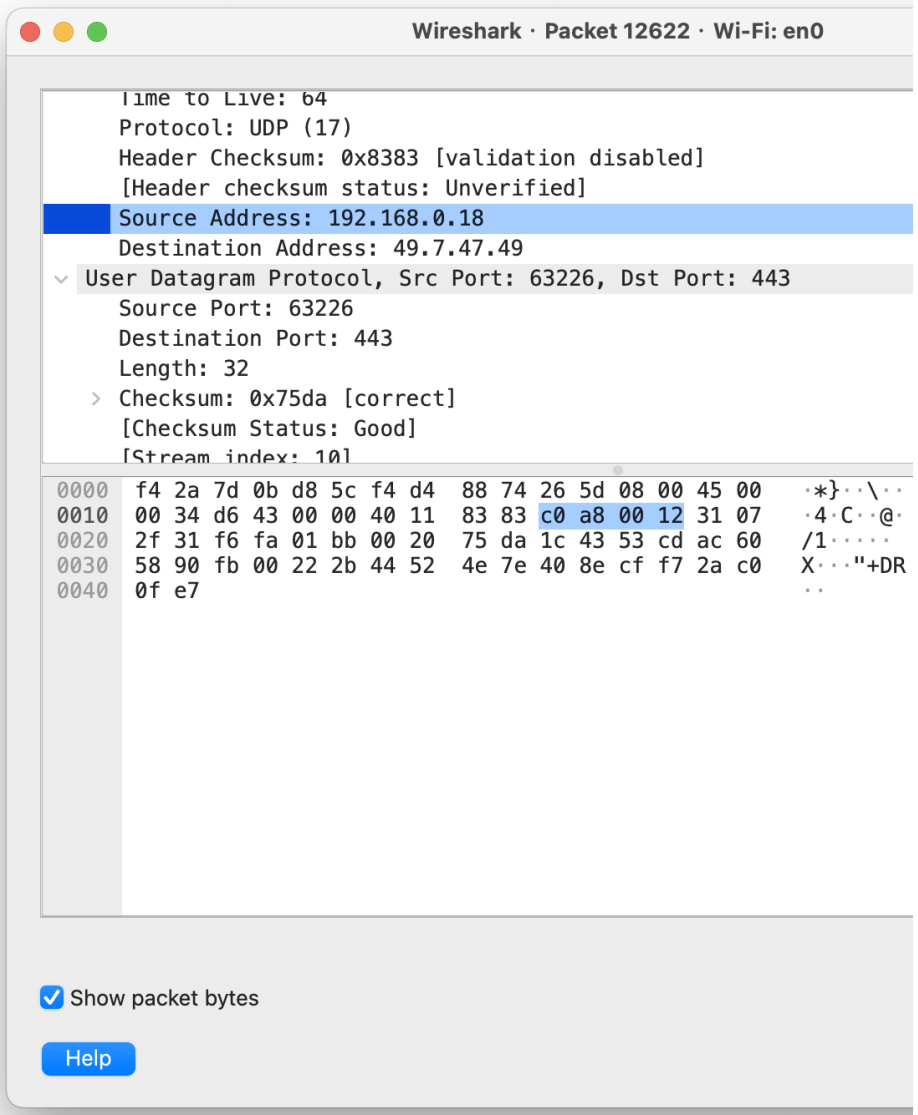
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Internet checksum: example

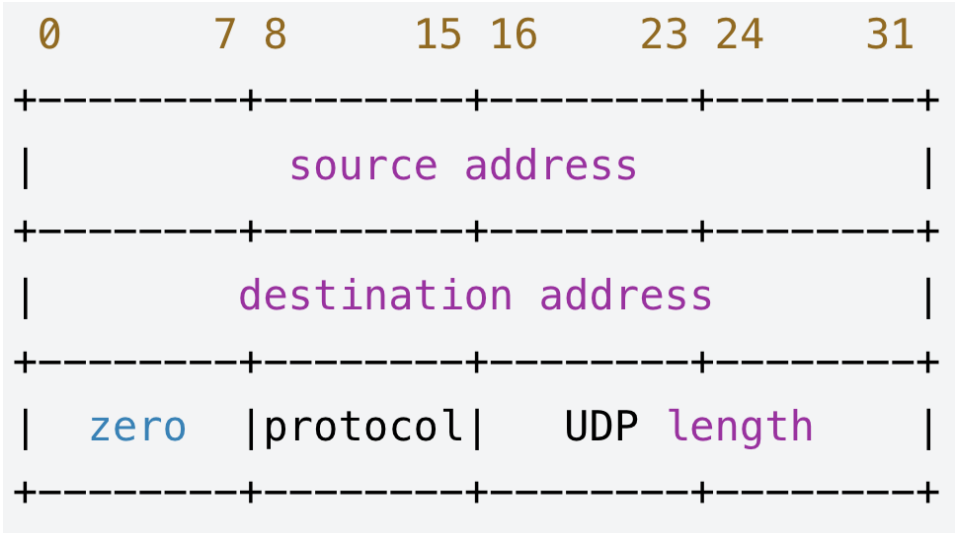
Example: add two 16-bit integers



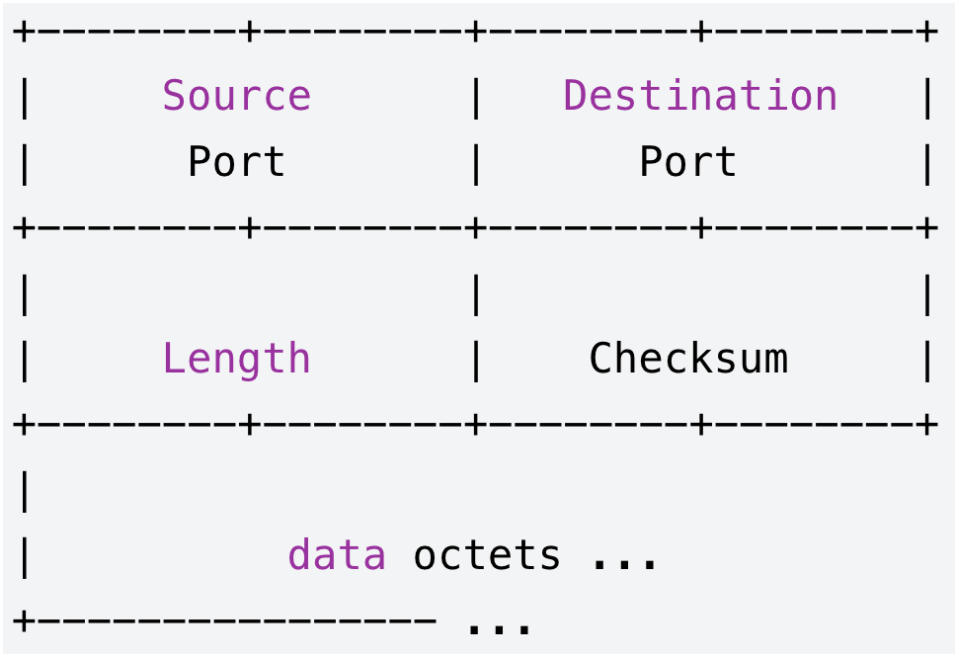
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result




```python
data = 'c0a80012' \
       '31072f31' \
       '00110020' \
       'f6fa01bb' \
       '0020' \
       '1c4353cdac605890fb00222b44524e7e408ecff72ac00fe7'

checksum = 0

while data != '':
    checksum += int(data[:4], 16)
    if checksum > 65565:
        checksum &= 0xFFFF
        checksum += 1
    data = data[4:]
    if len(data) == 2:
        data += '00'

checksum = 65535 - checksum

print('%x' % checksum)
```



Pseudo-header



UDP Header + Data

Lecture 4 – Transport Layer (1)

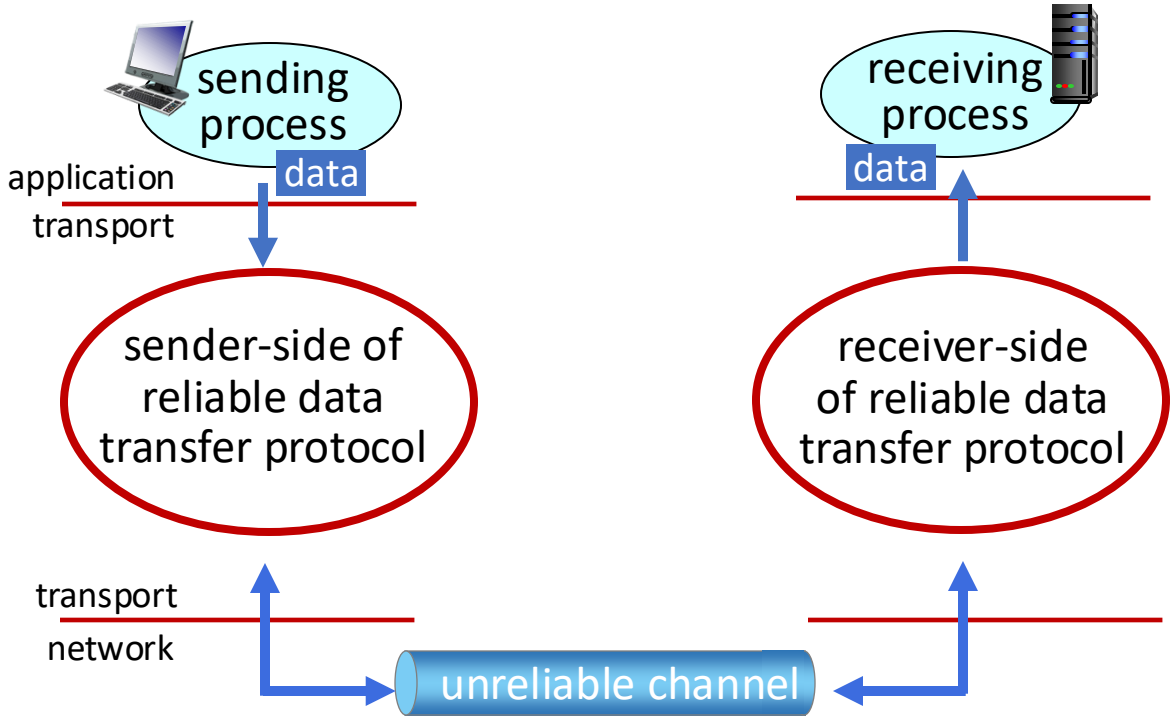
- **Roadmap**
  1. Overview: Transport-layer services
  2. Multiplexing and demultiplexing
  3. Connectionless transport: UDP
  4. Principles of reliable data transfer

Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

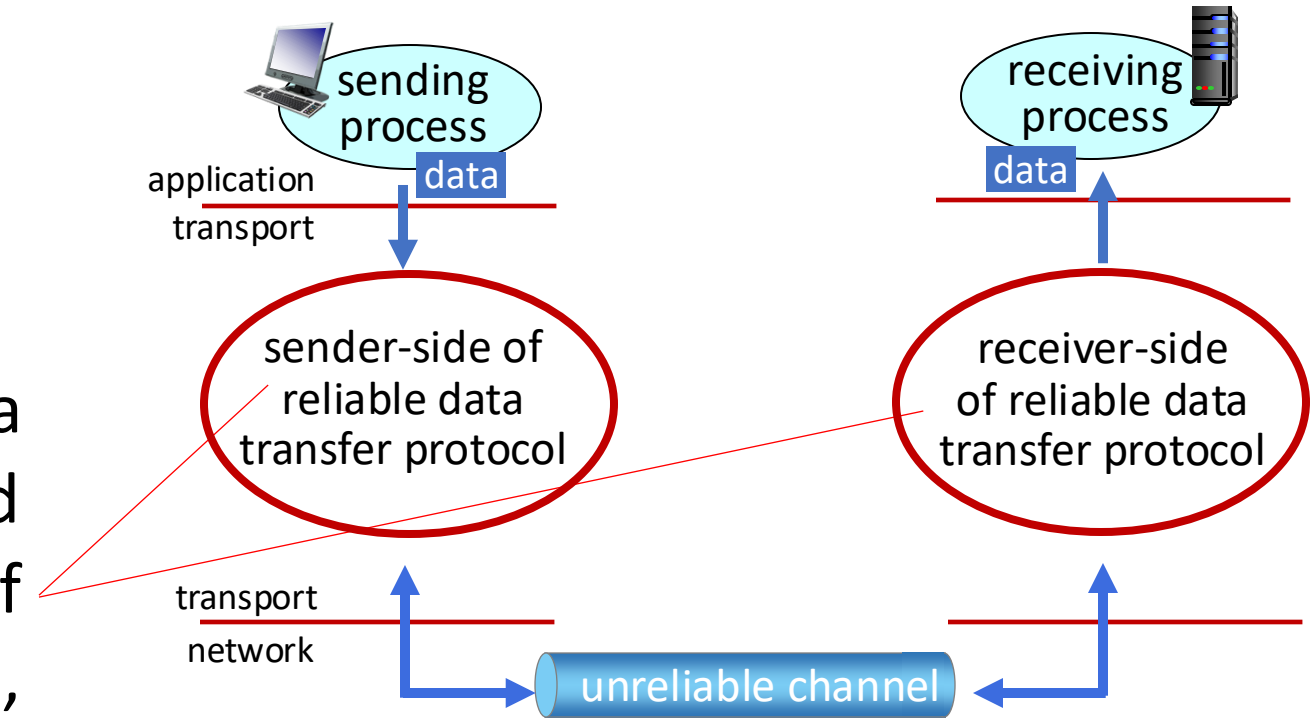
reliable service *abstraction*



reliable service *implementation*

Principles of reliable data transfer

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

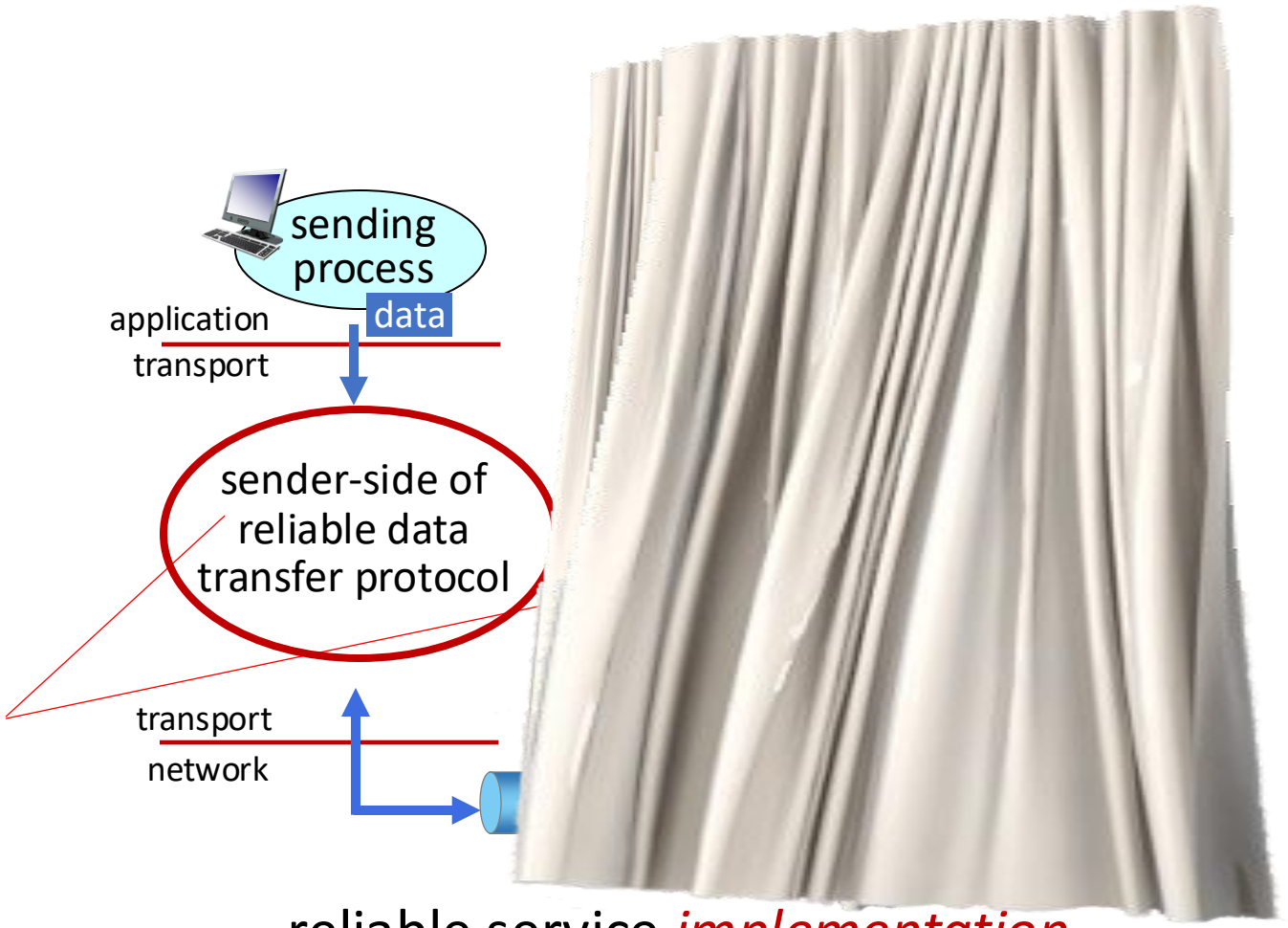


reliable service *implementation*

Principles of reliable data transfer

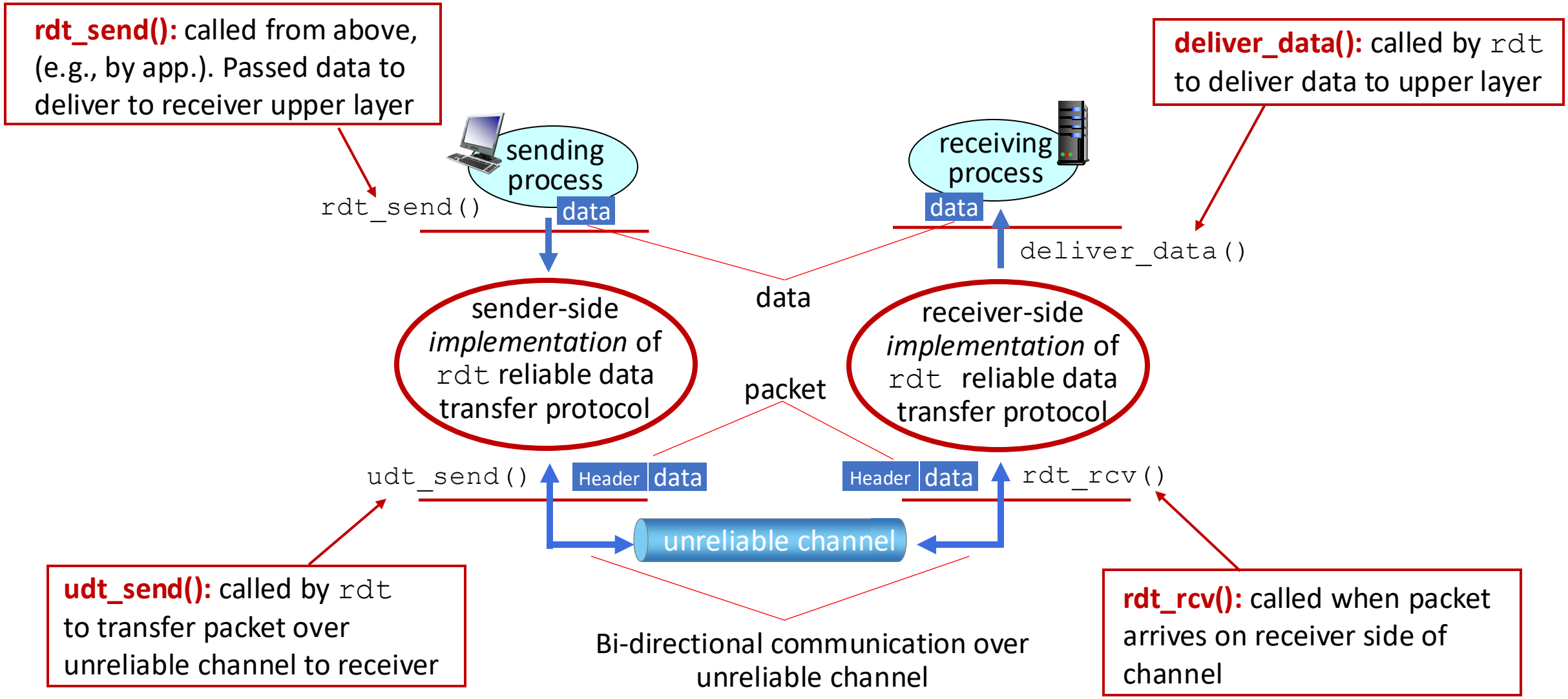
Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message



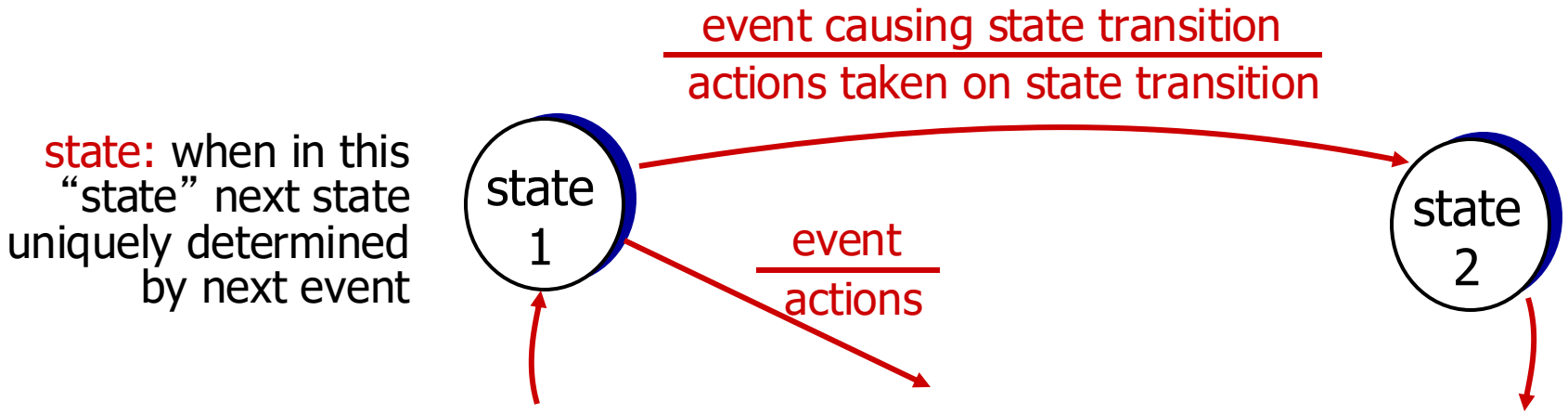
reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces



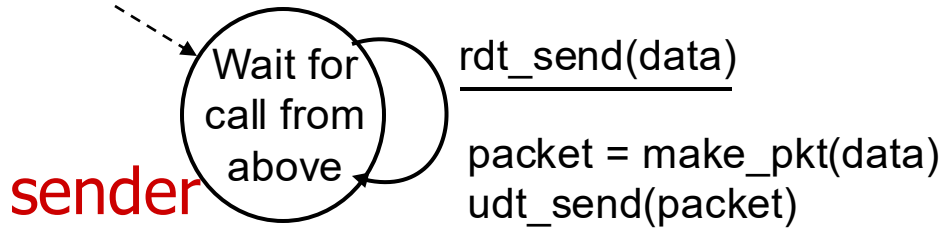
Reliable data transfer: getting started

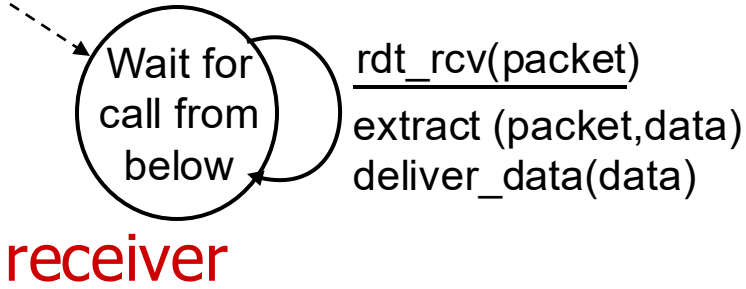
- **Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)**
- **Consider only unidirectional data transfer**
  - But control info will flow on both directions!
- **Use finite state machines (FSM) to specify sender, receiver**



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





rdt2.0: channel with bit errors

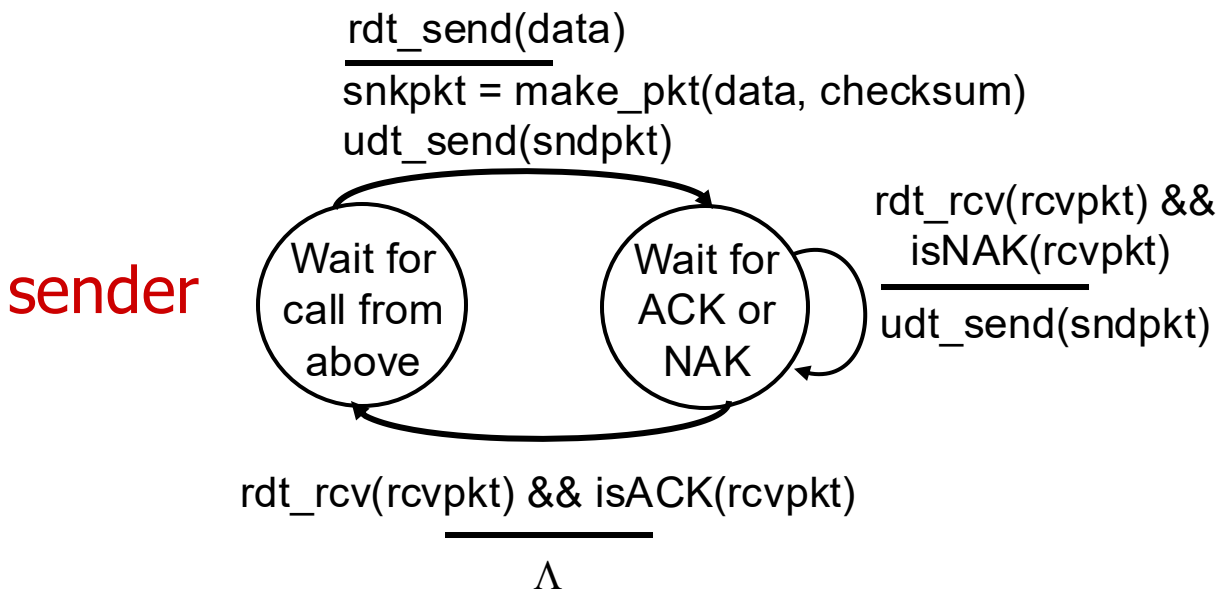
- **Underlying channel may flip bits in packet**
  - Checksum to detect bit errors
- **Question : how to recover from errors:**

How do humans recover from "errors" during conversation?

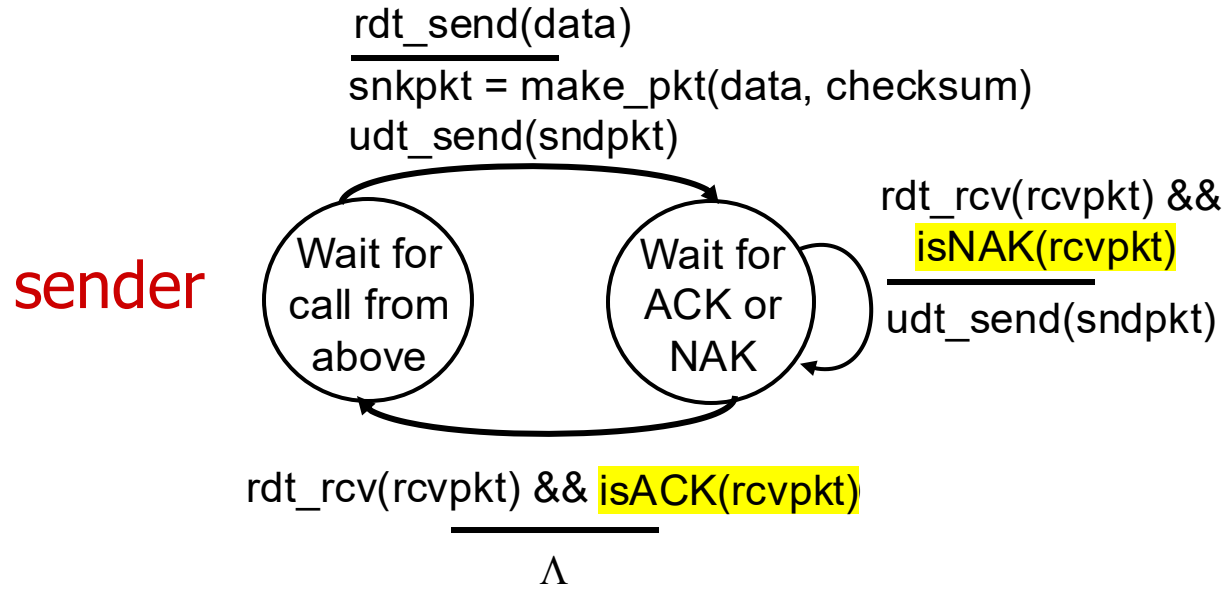
rdt2.0: channel with bit errors

- **Underlying channel may flip bits in packet**
  - Checksum to detect bit errors
- **Question : how to recover from errors:**
  - *Acknowledgements (ACKs):* receiver tells sender that pkt received OK
  - *Negative acknowledgements (NAKs):* receiver 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

rdt2.0: FSM specifications



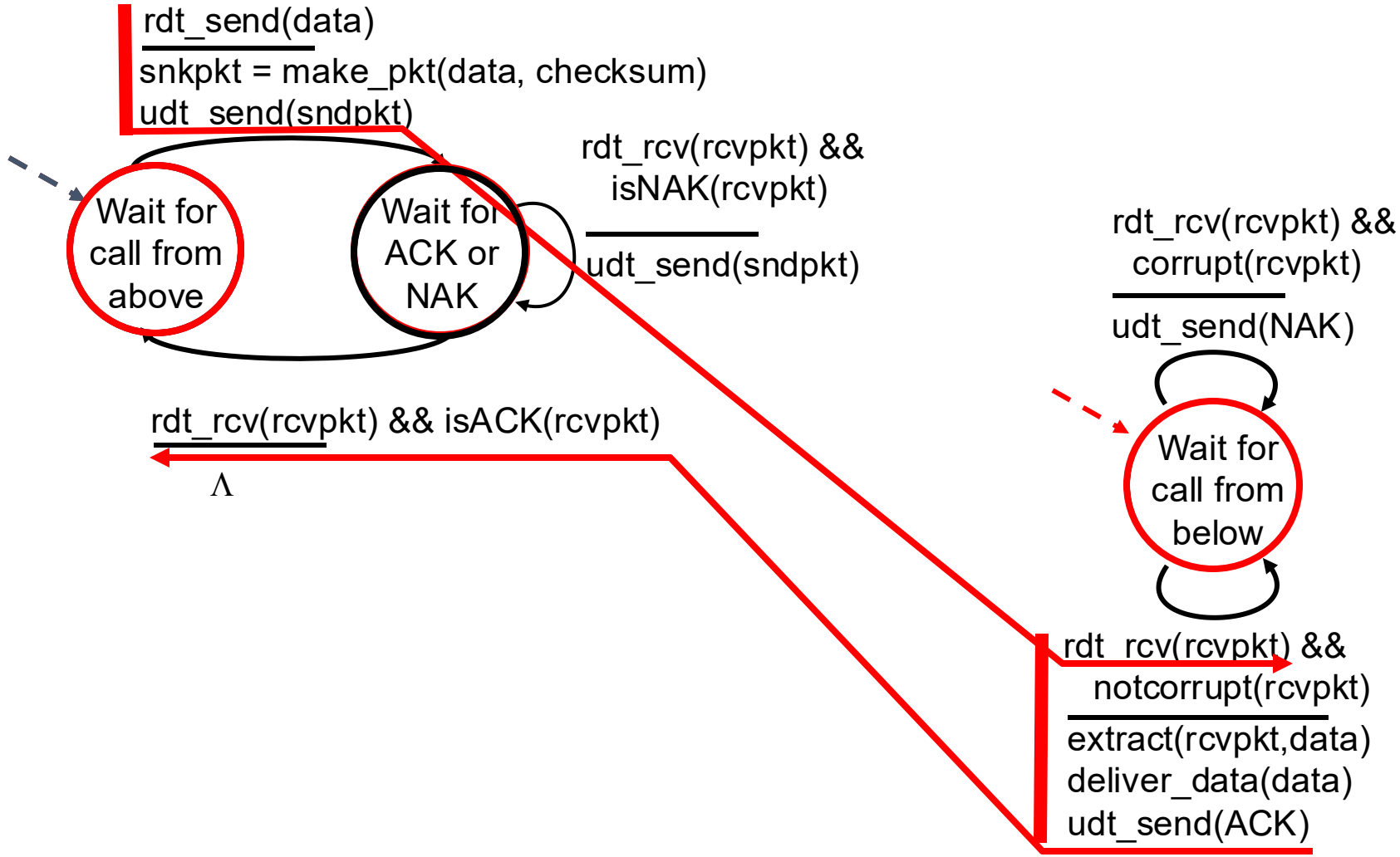
rdt2.0: FSM specification



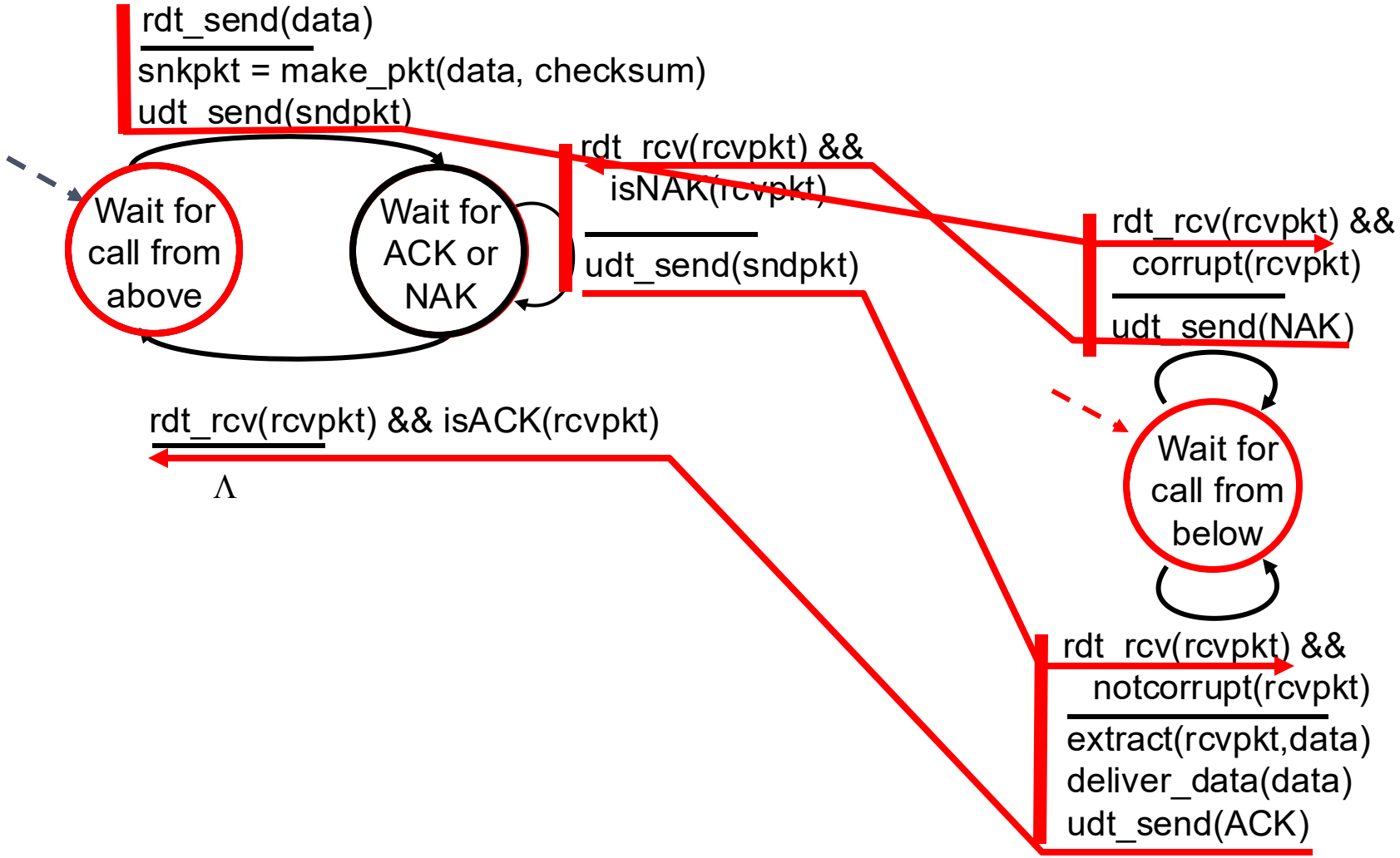
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

rdt2.0: operation with no errors



rdt2.0: Error scenario



rdt2.0 has a fatal flaw!

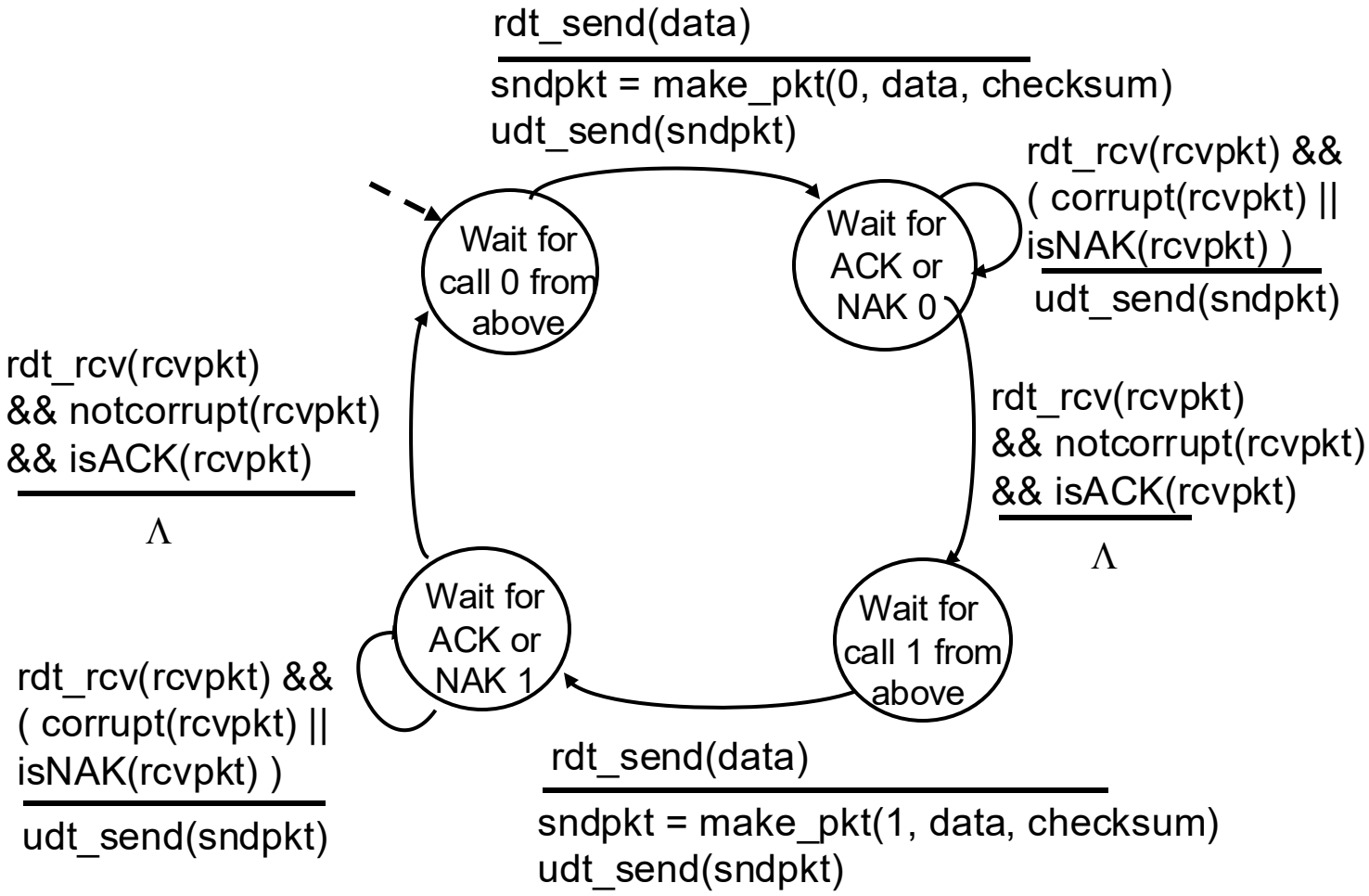
What happens if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver!
- Can‘t just retransmit -> possible duplicate

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



rdt2.1: receiver, handles garbled ACK/NAKs

rdt_rcv(rcvpkt) && (

sndpkt = make_pkt
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpk
has_seq1(rcvpkt)

sndpkt = make_pk
udt_send(sndpkt)

t_rcv(rcvpkt) && (corrupt(rcvpkt)

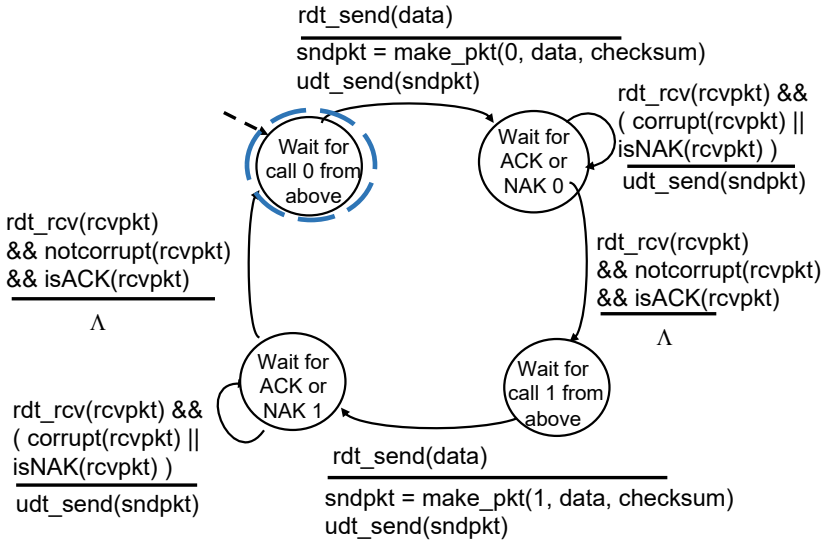
ndpkt = make_pkt(NAK, chksum)
dt_send(sndpkt)

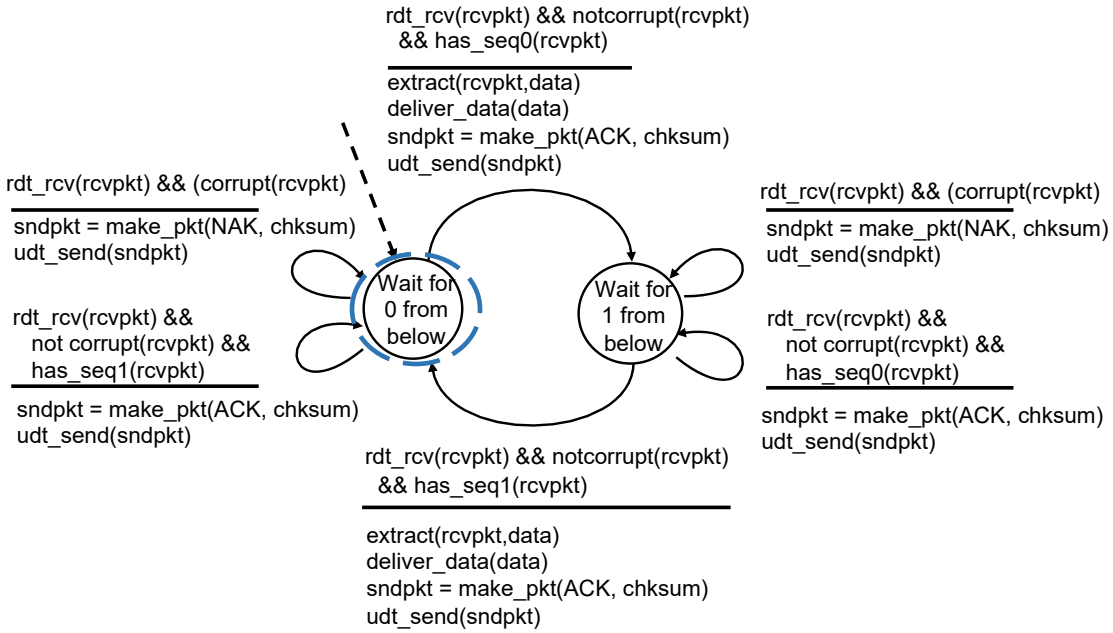
dt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

ndpkt = make_pkt(ACK, chksum)
dt_send(sndpkt)

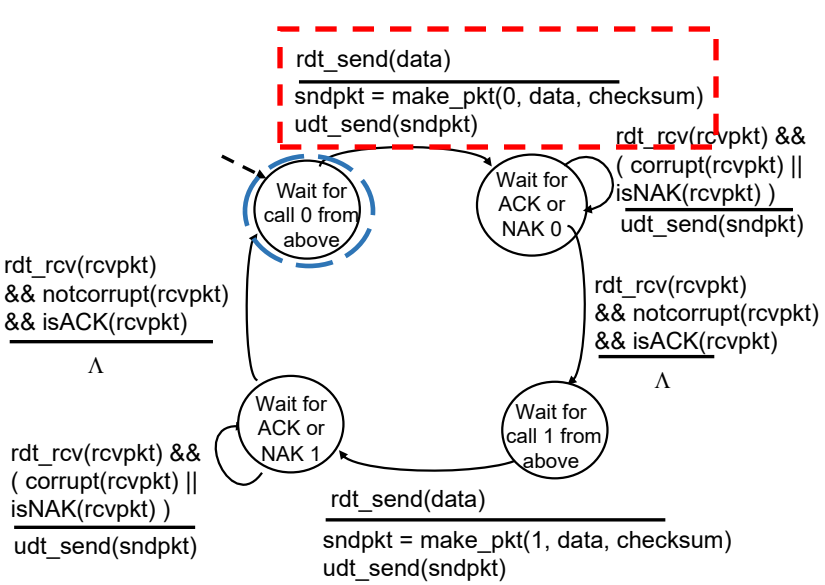
udt_send(sndpkt)

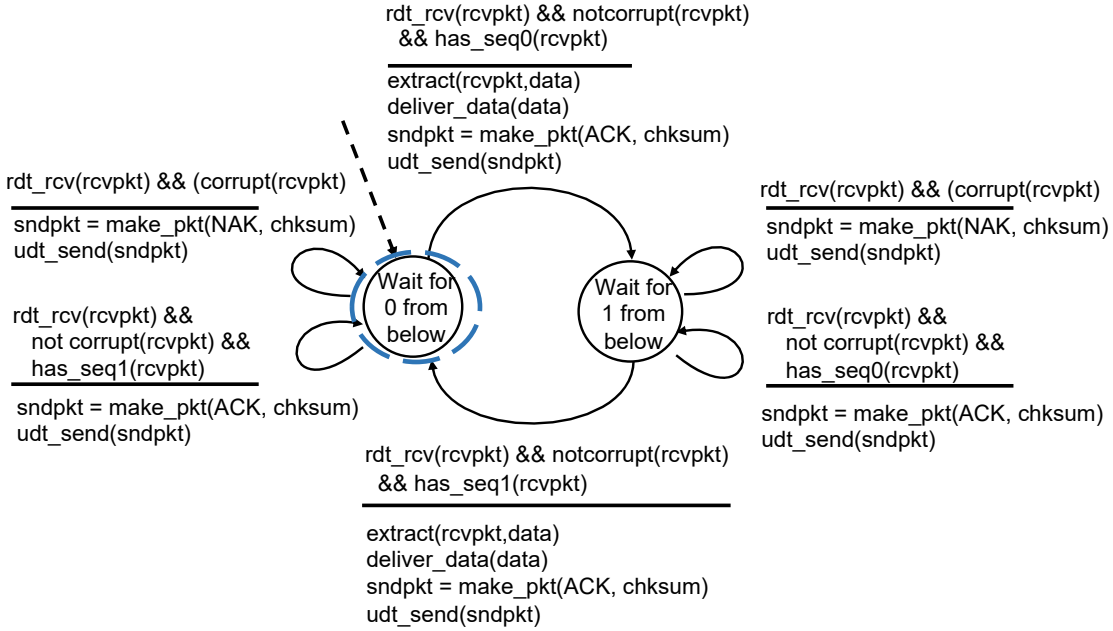
rdt2.1: sender vs receiver: no error



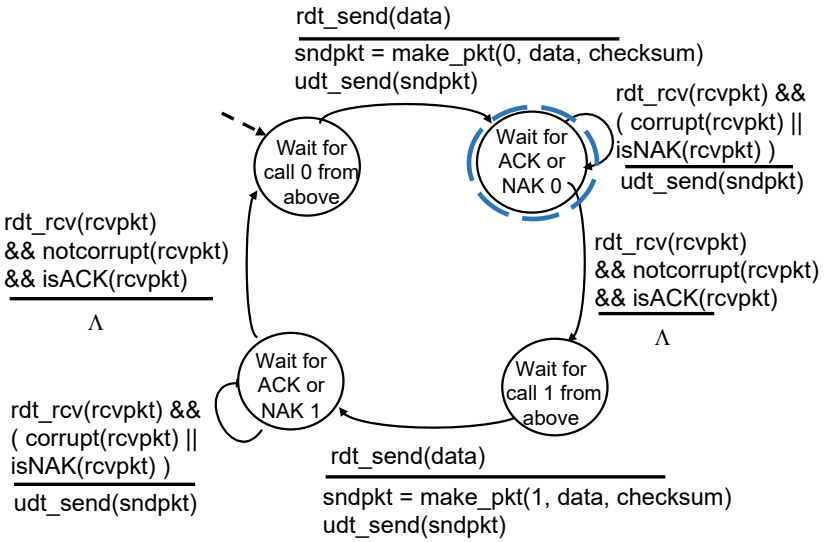


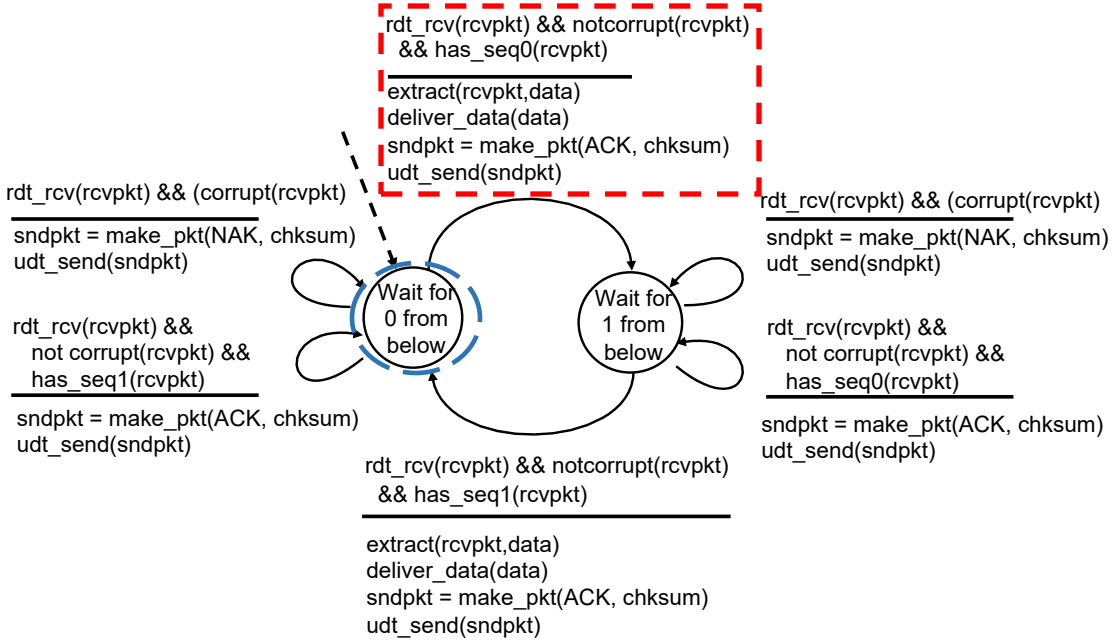
rdt2.1: sender vs receiver: no error



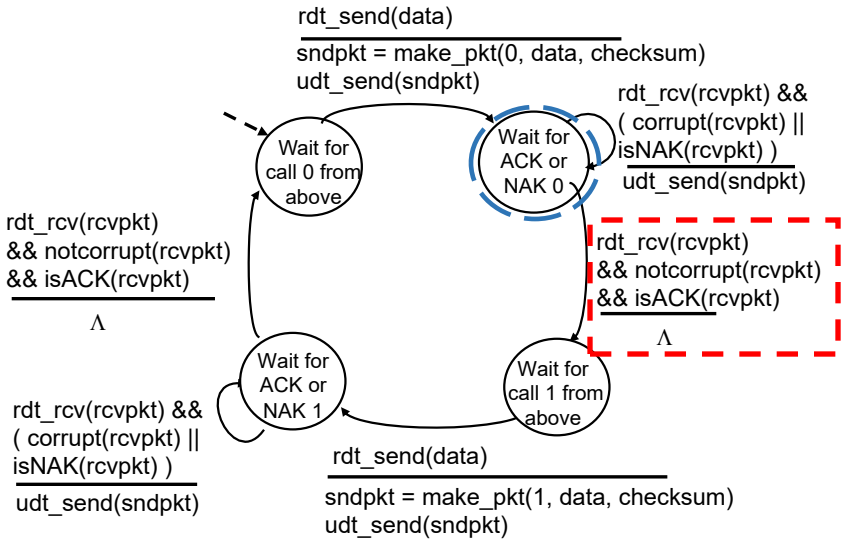


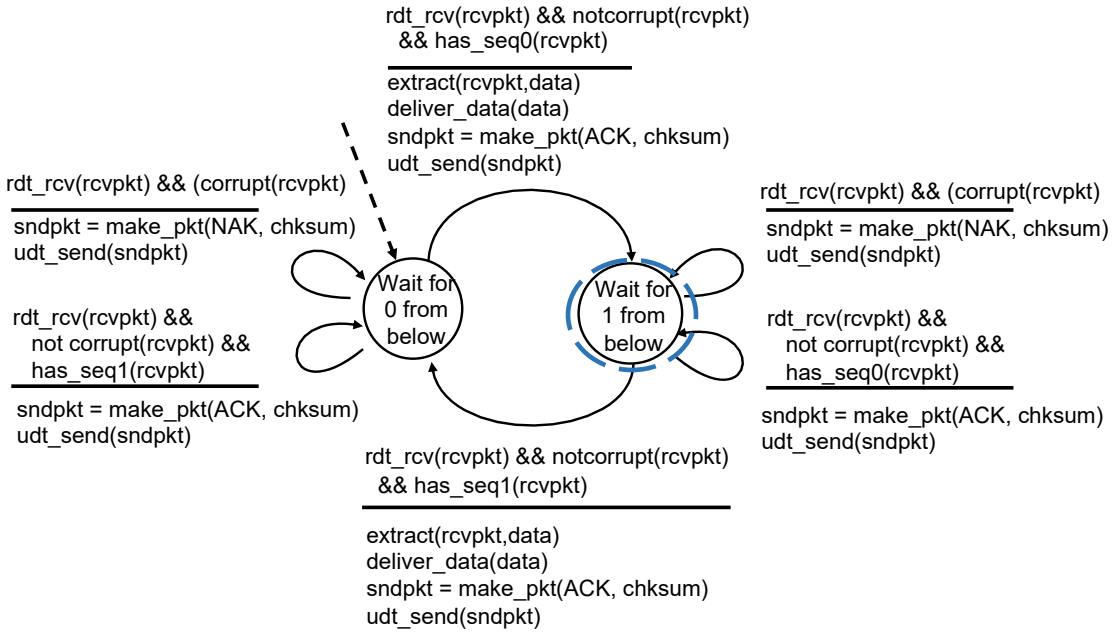
rdt2.1: sender vs receiver: no error



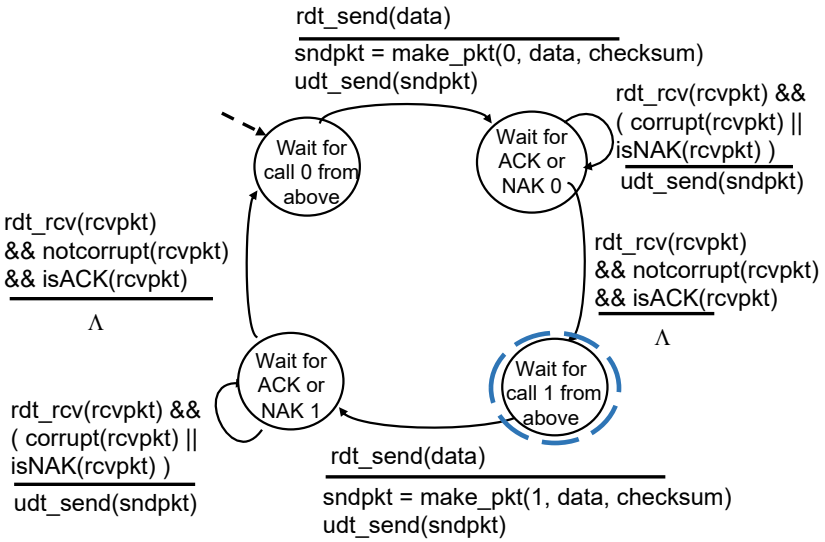


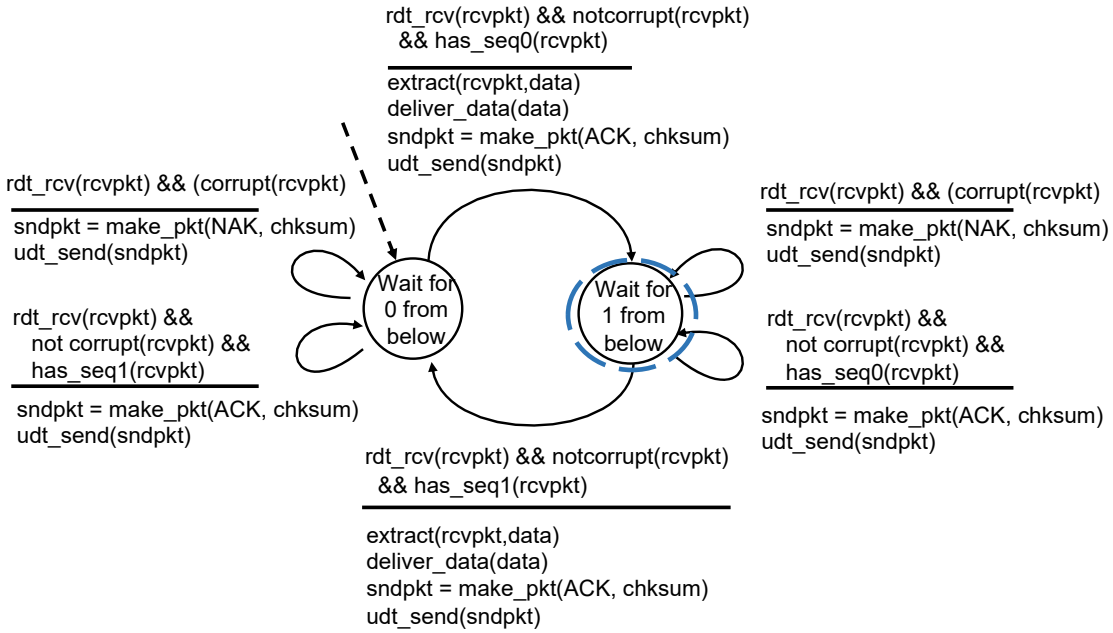
rdt2.1: sender vs receiver: no error



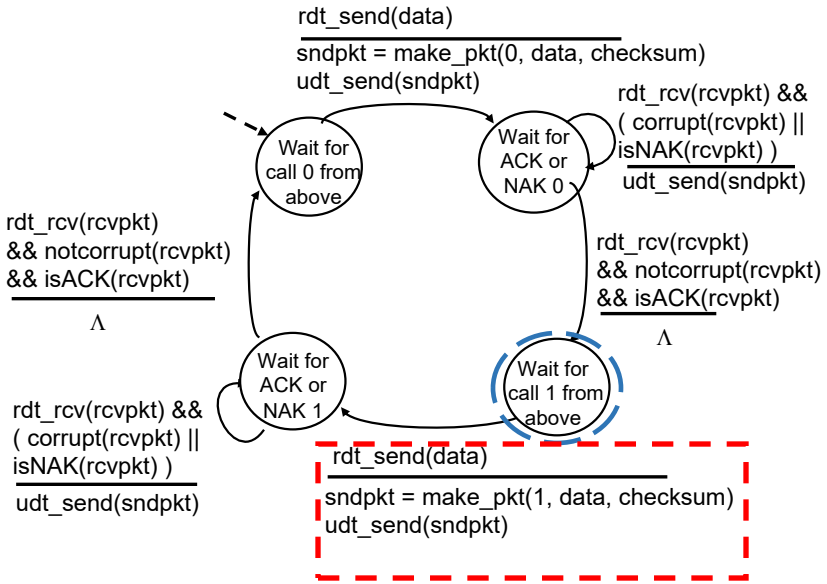


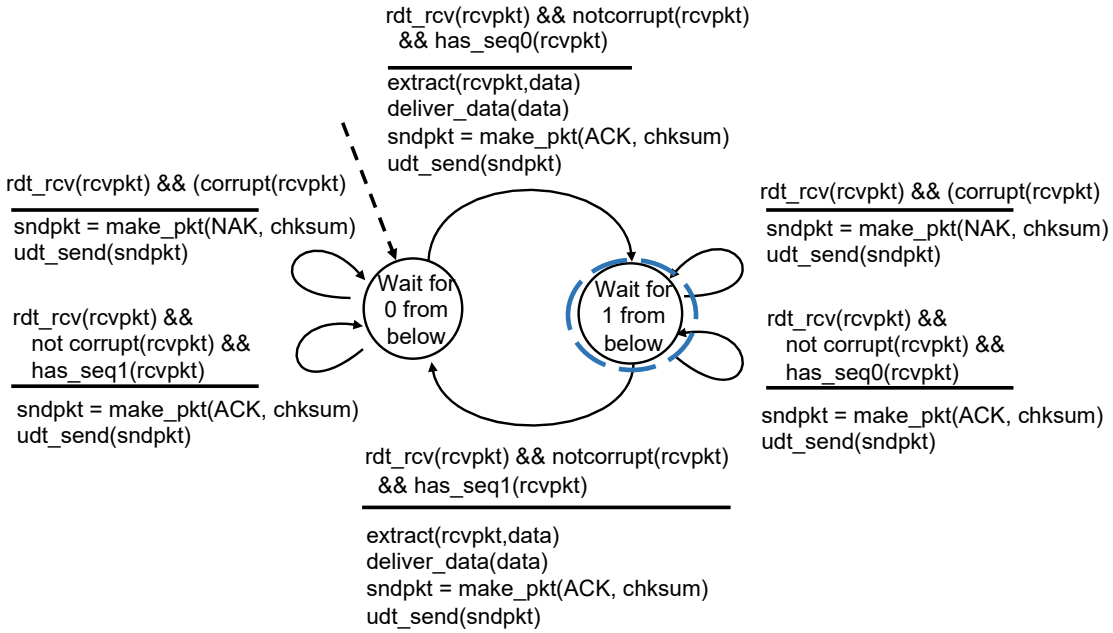
rdt2.1: sender vs receiver: no error



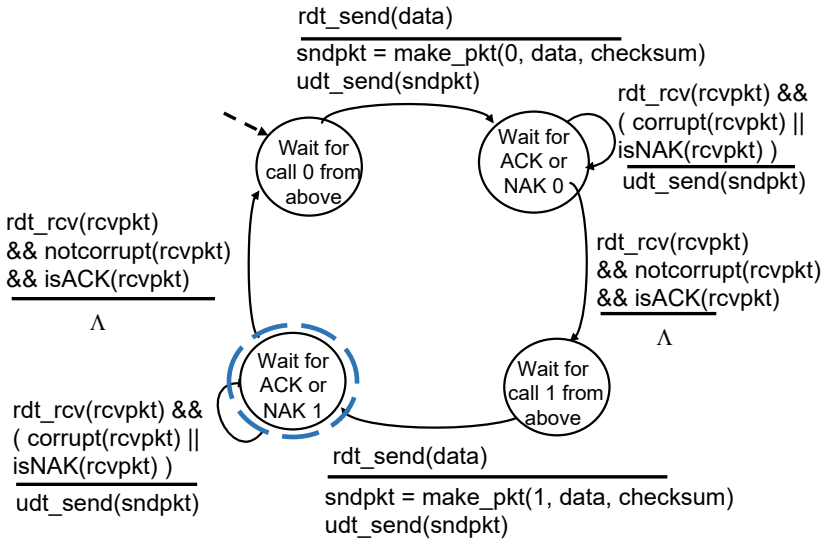


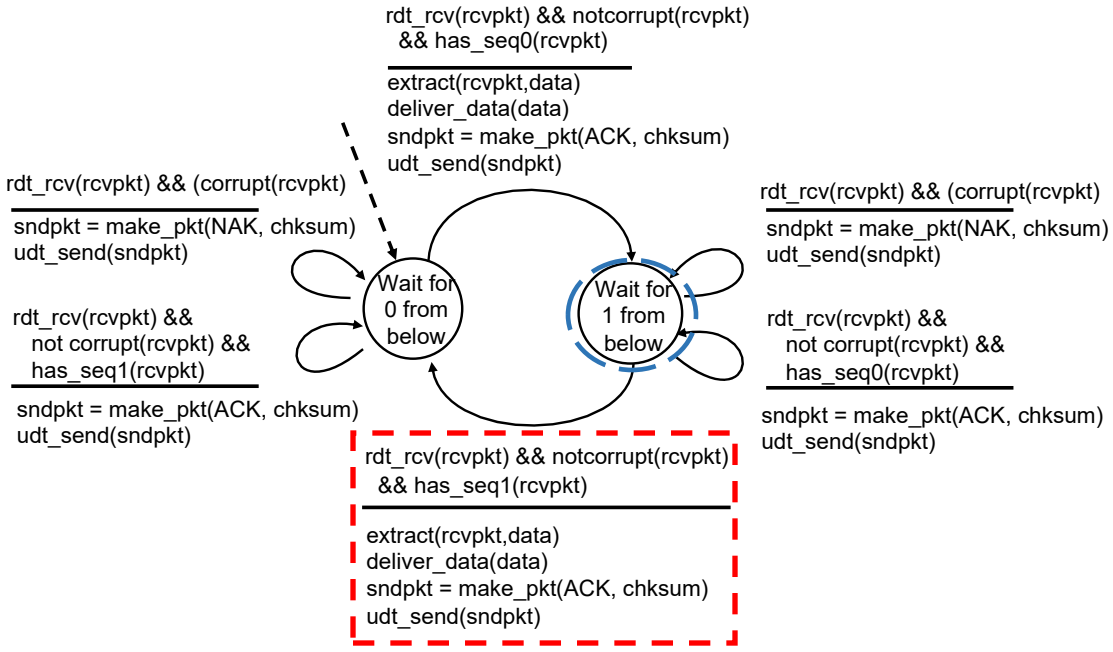
rdt2.1: sender vs receiver: no error



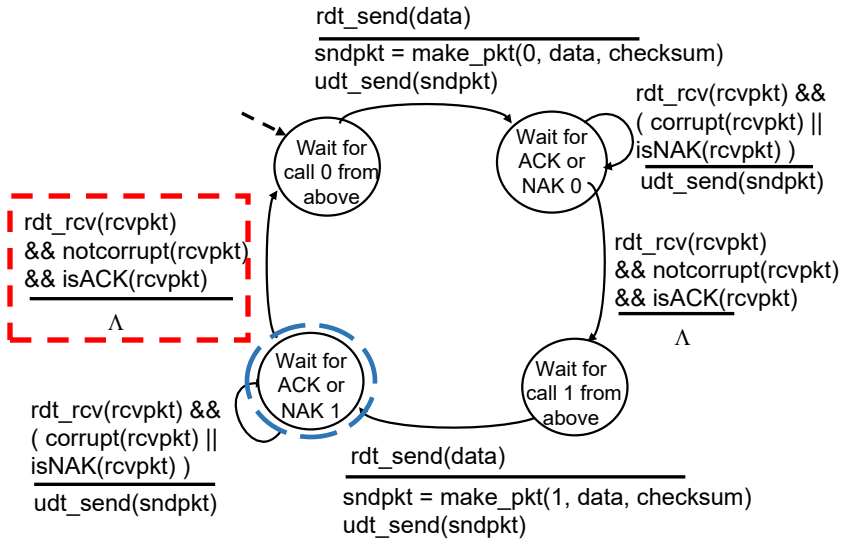


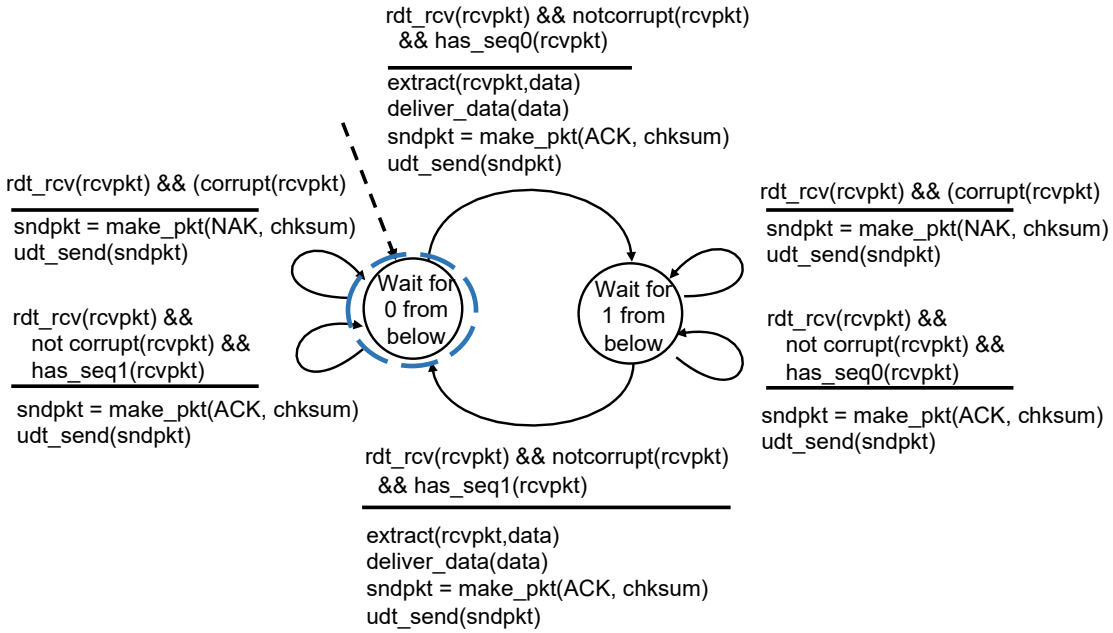
rdt2.1: sender vs receiver: no error



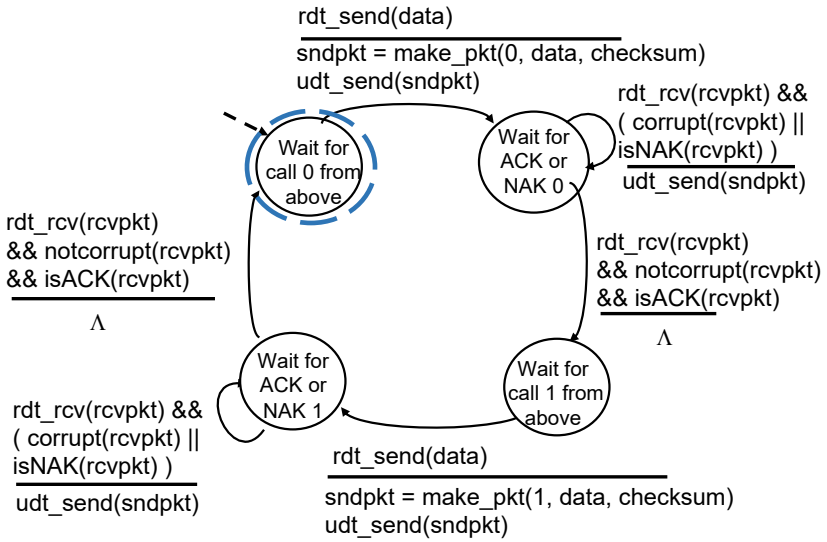


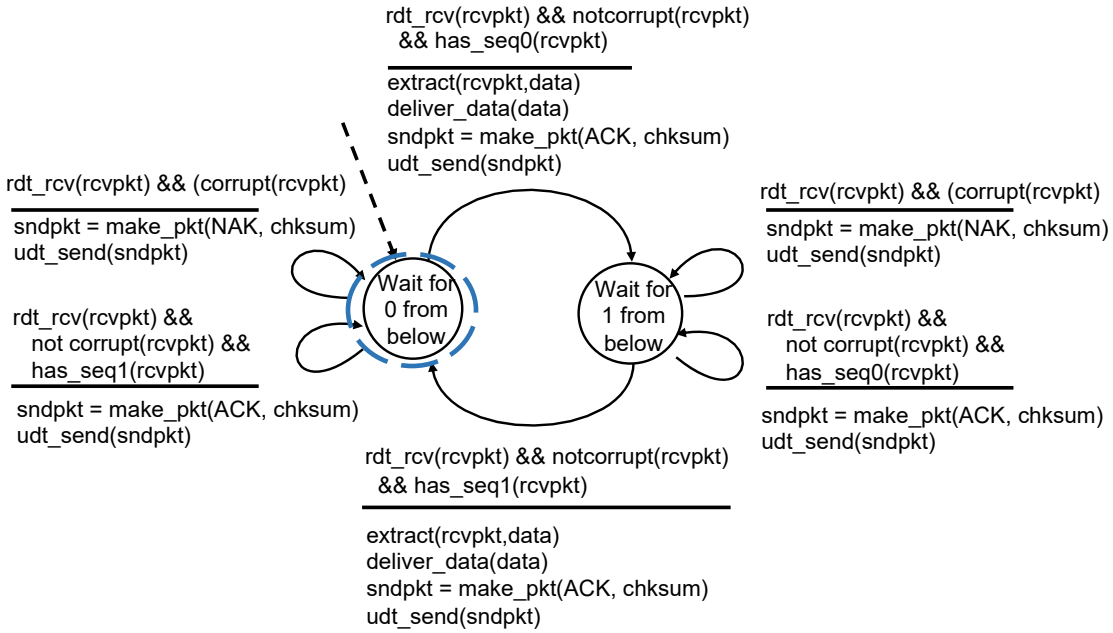
rdt2.1: sender vs receiver: no error



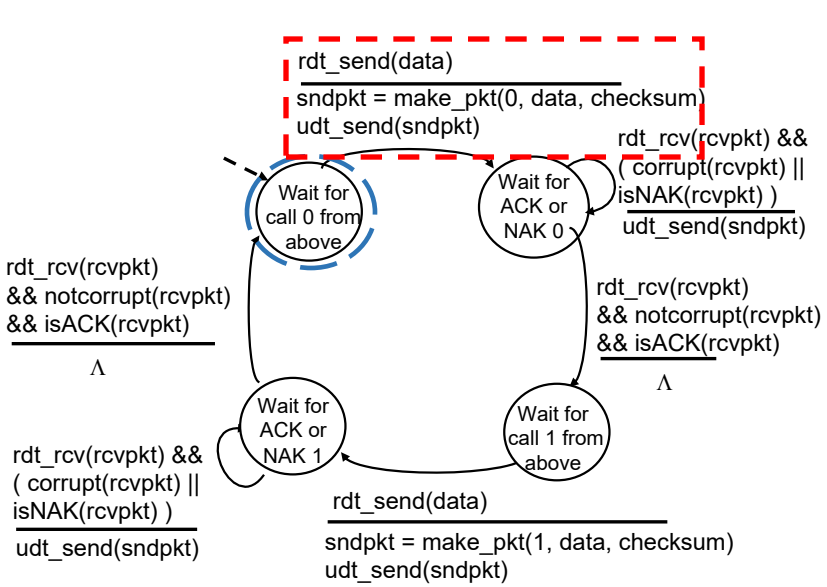


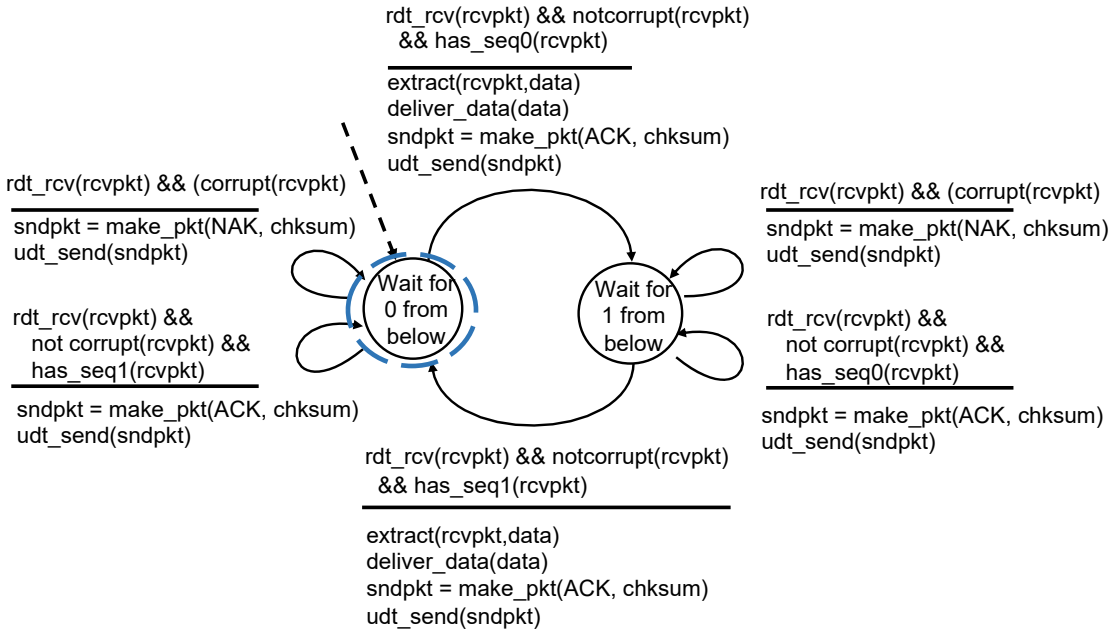
rdt2.1: sender vs receiver: no error



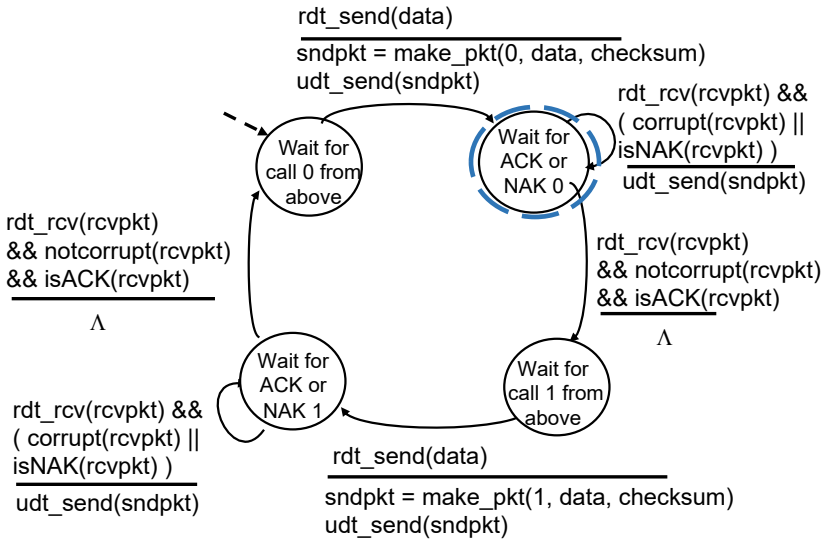


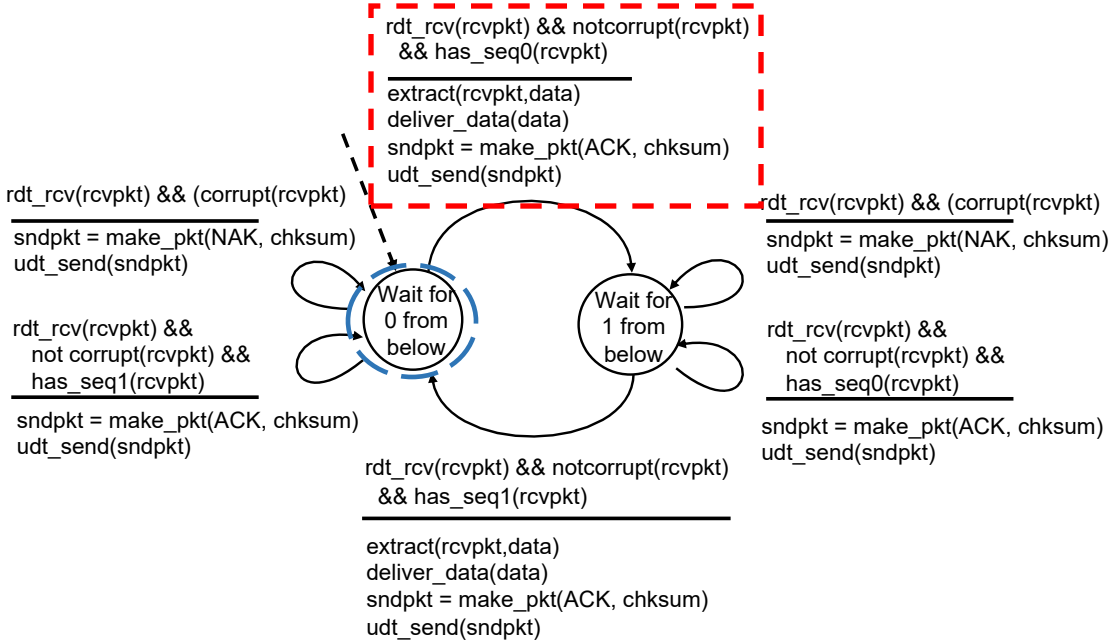
rdt2.1: sender vs receiver: ACK corrupt



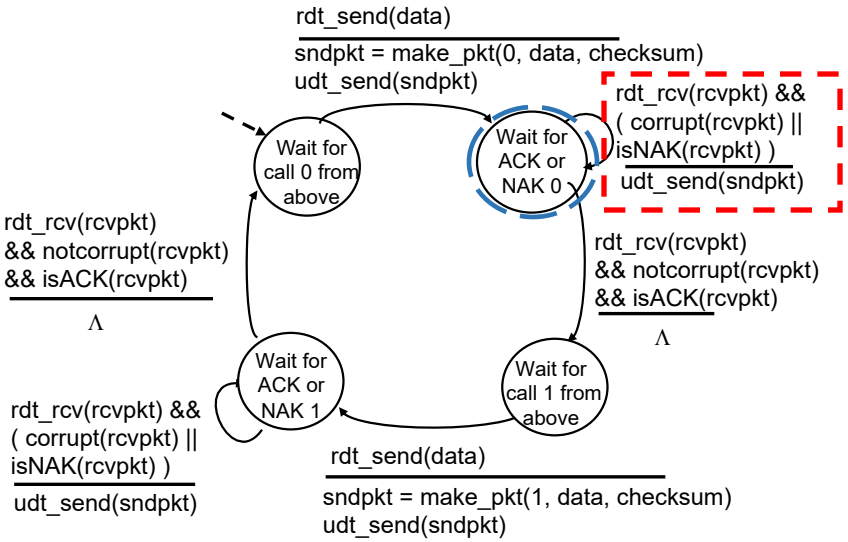


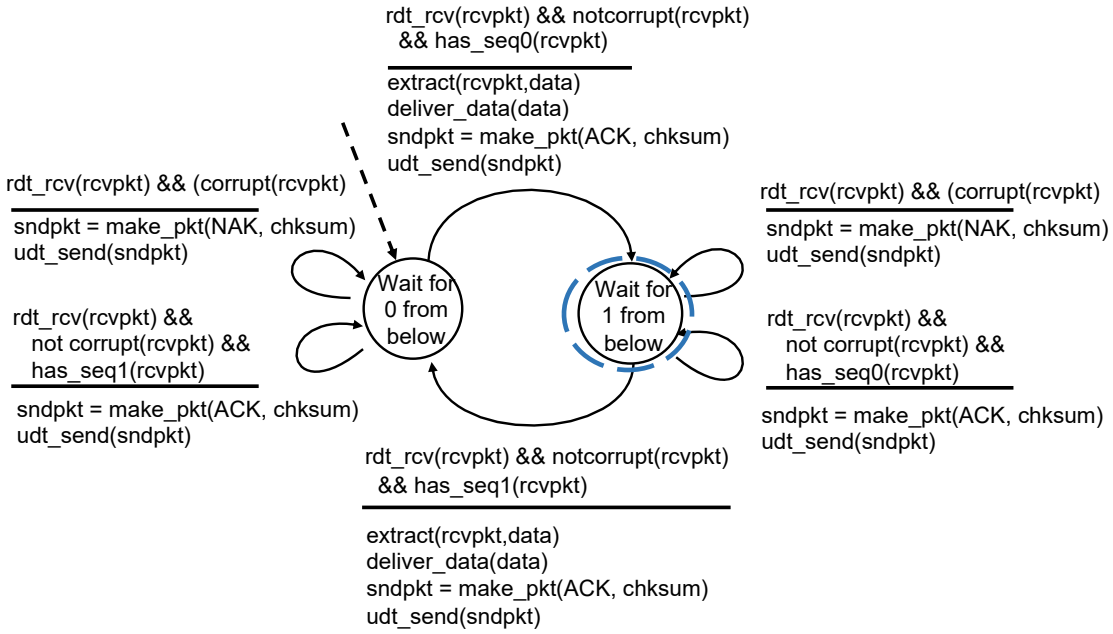
rdt2.1: sender vs receiver: ACK corrupt



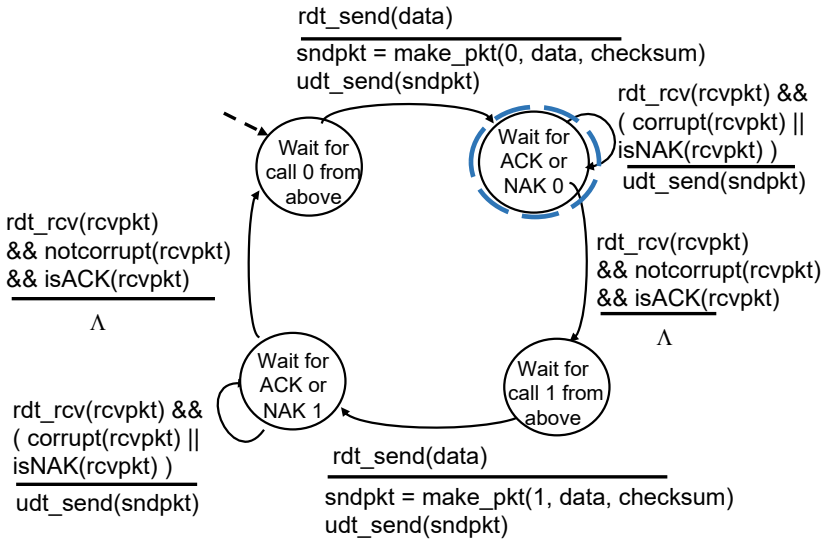


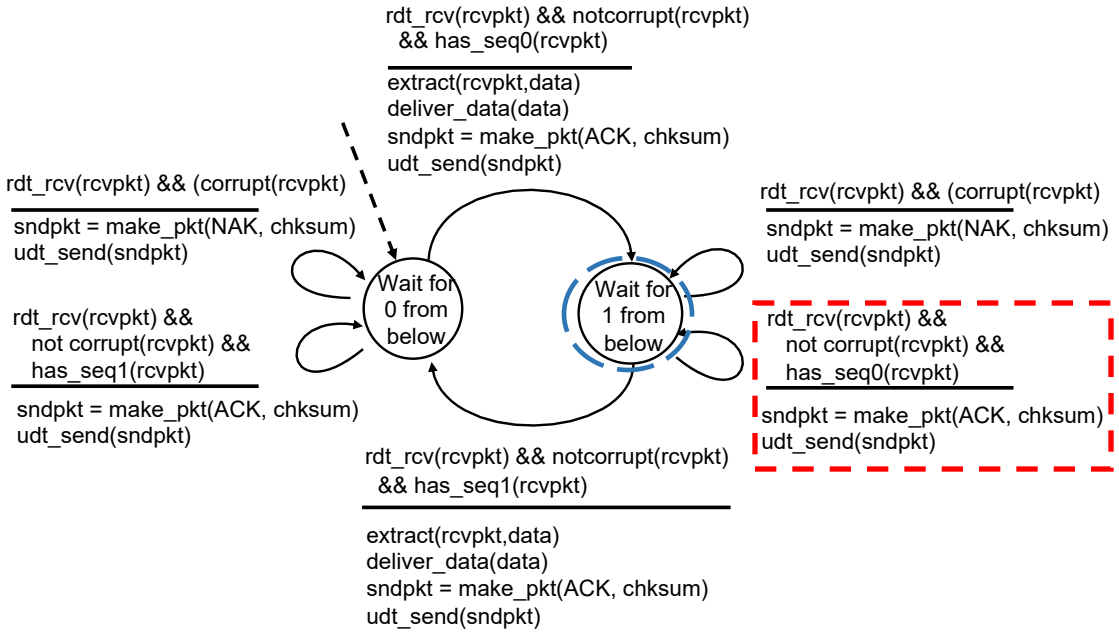
rdt2.1: sender vs receiver: ACK corrupt



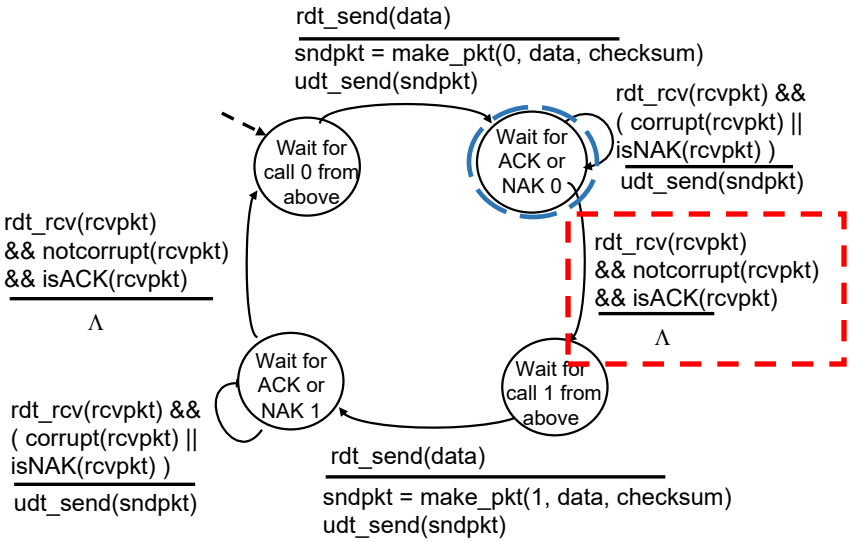


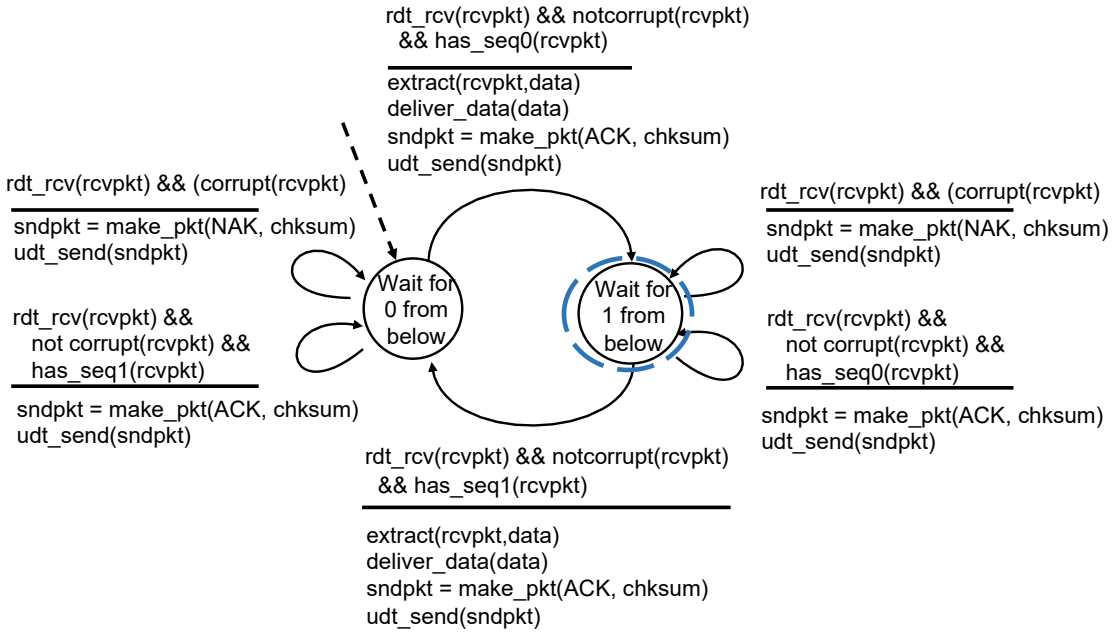
rdt2.1: sender vs receiver: ACK corrupt



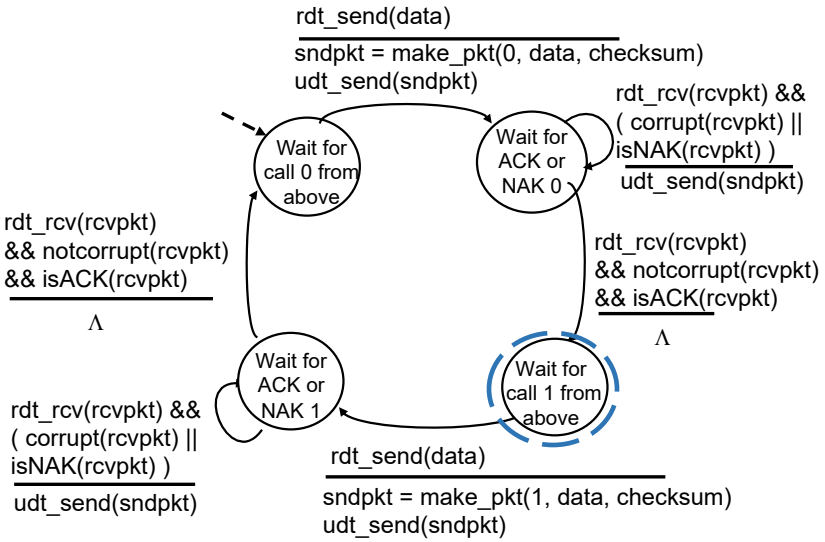


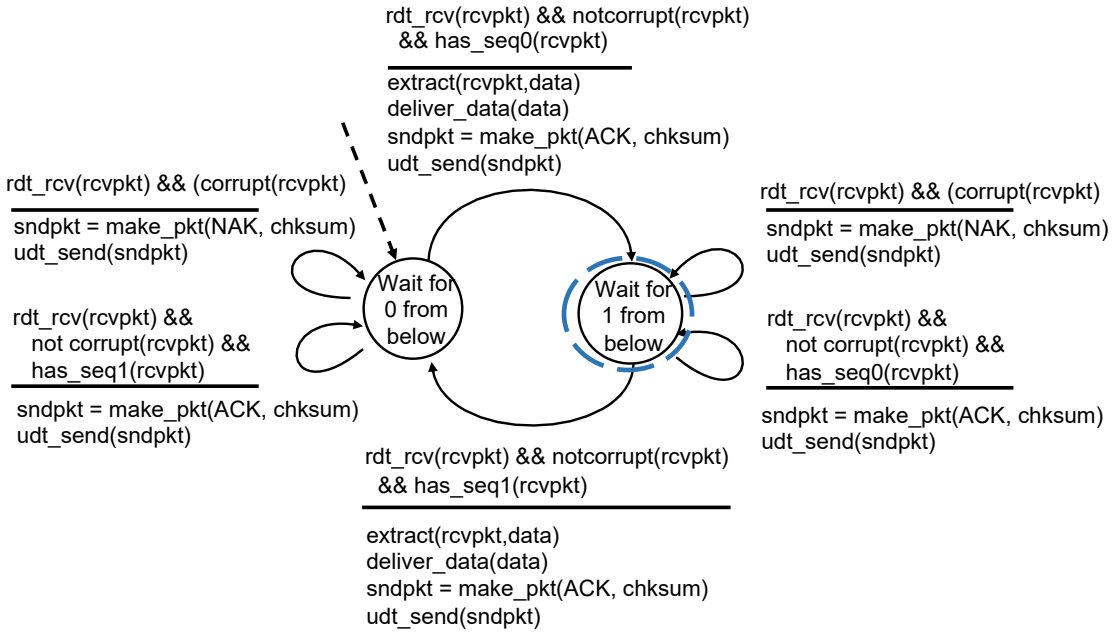
rdt2.1: sender vs receiver: ACK corrupt





rdt2.1: sender vs receiver: ACK corrupt





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

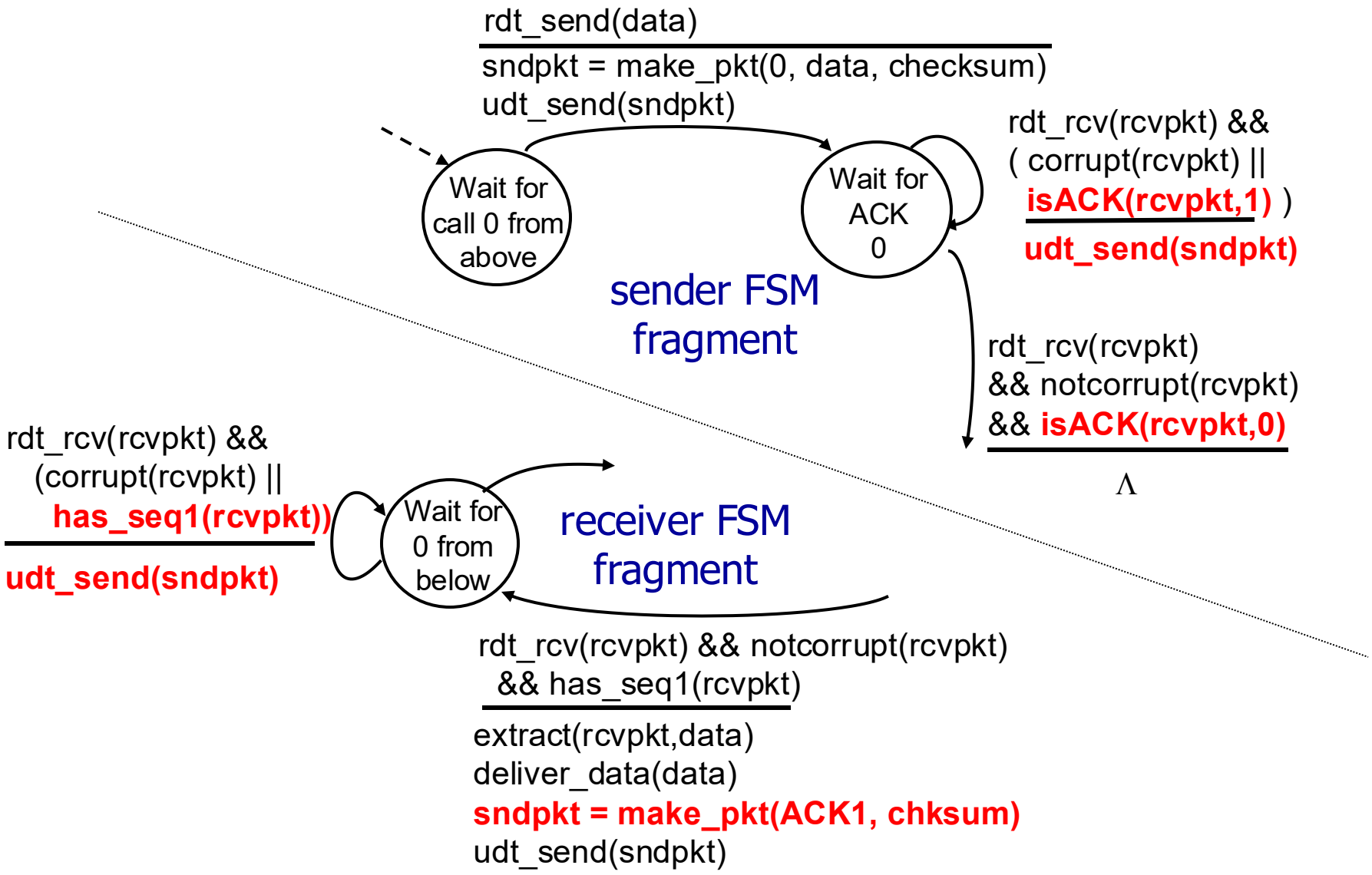
Receiver:

- 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**

rdt2.2: Sender, receiver fragments



rdt3.0: Channels with errors *and* loss

New assumption:

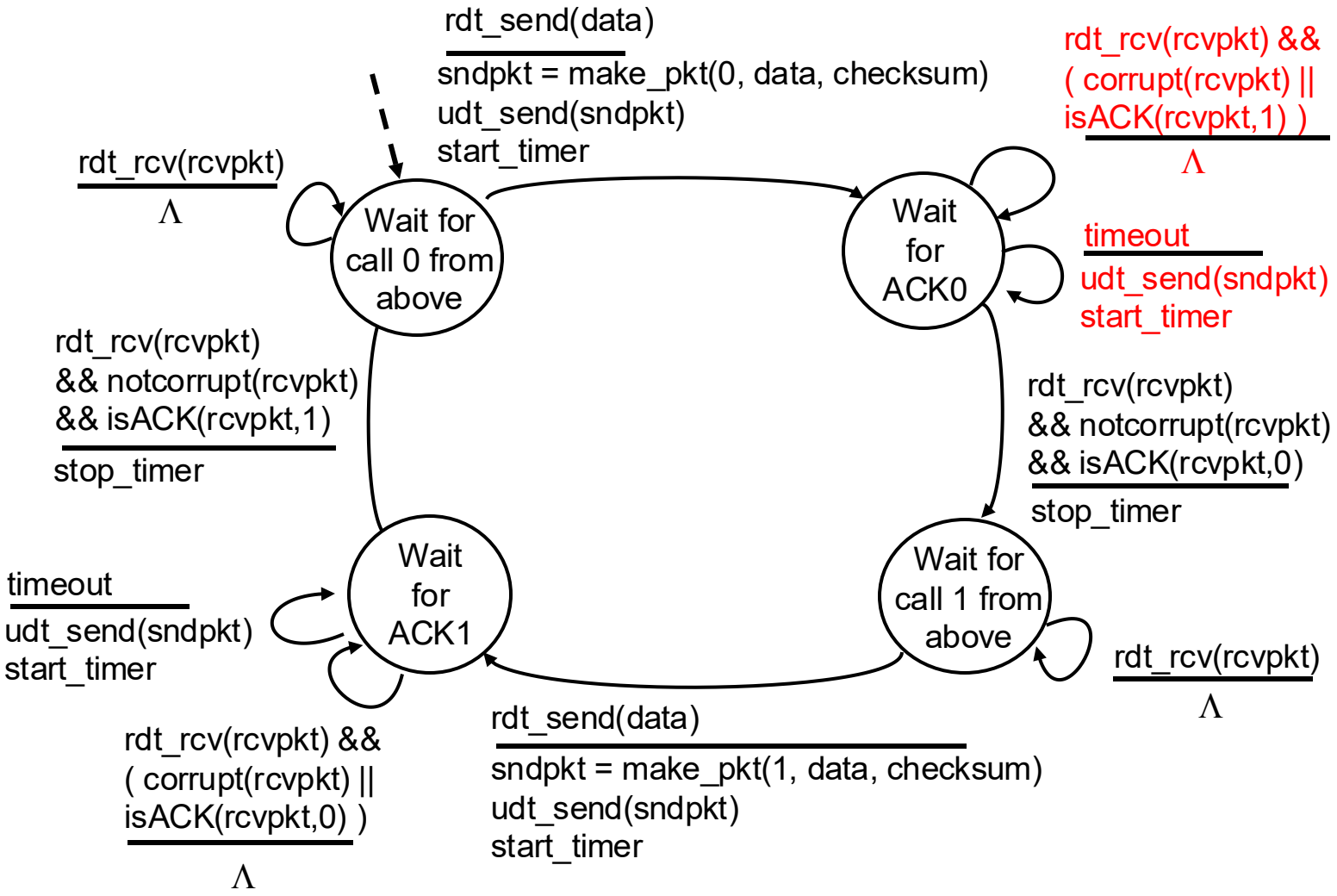
- **Underlying channel can also lose packets (data, ACKs)**
  - Checksum, seq. #, ACKs, retransmissions will be of help … but not enough

Approach

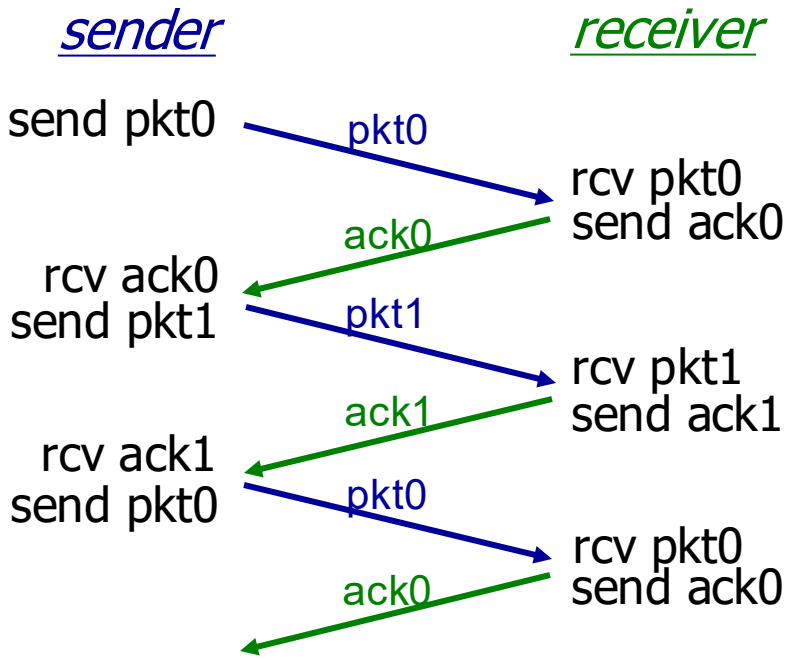
Sender waits "reasonable" amount of time for ACK

- **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**

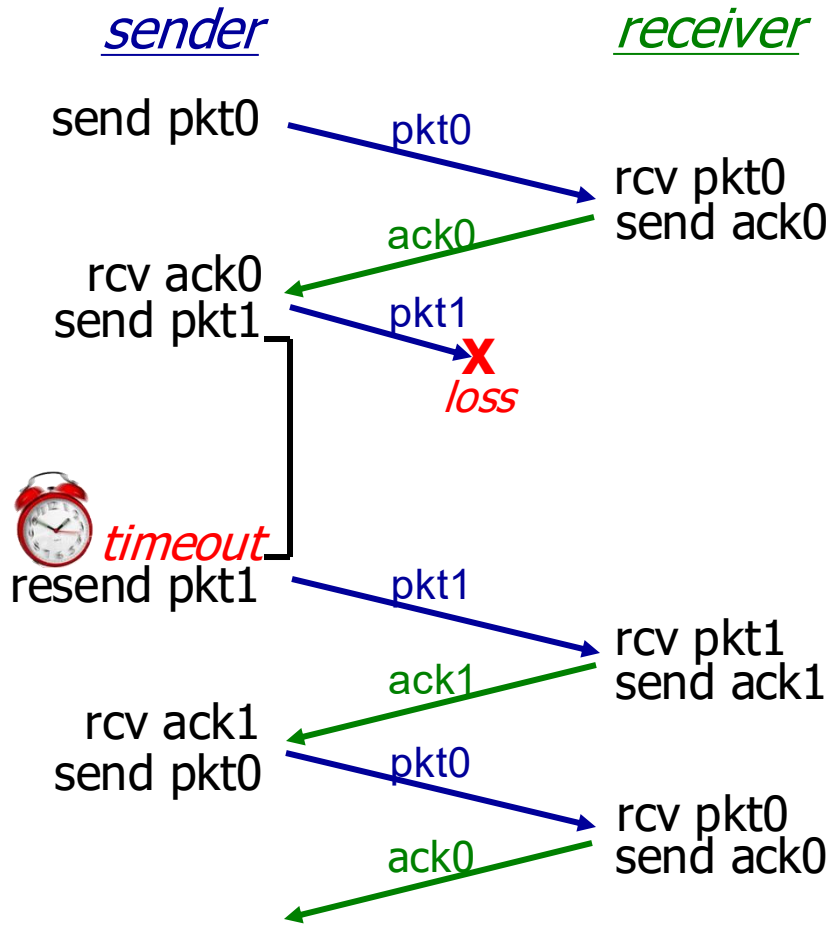
rdt3.0 sender



rdt3.0 in action

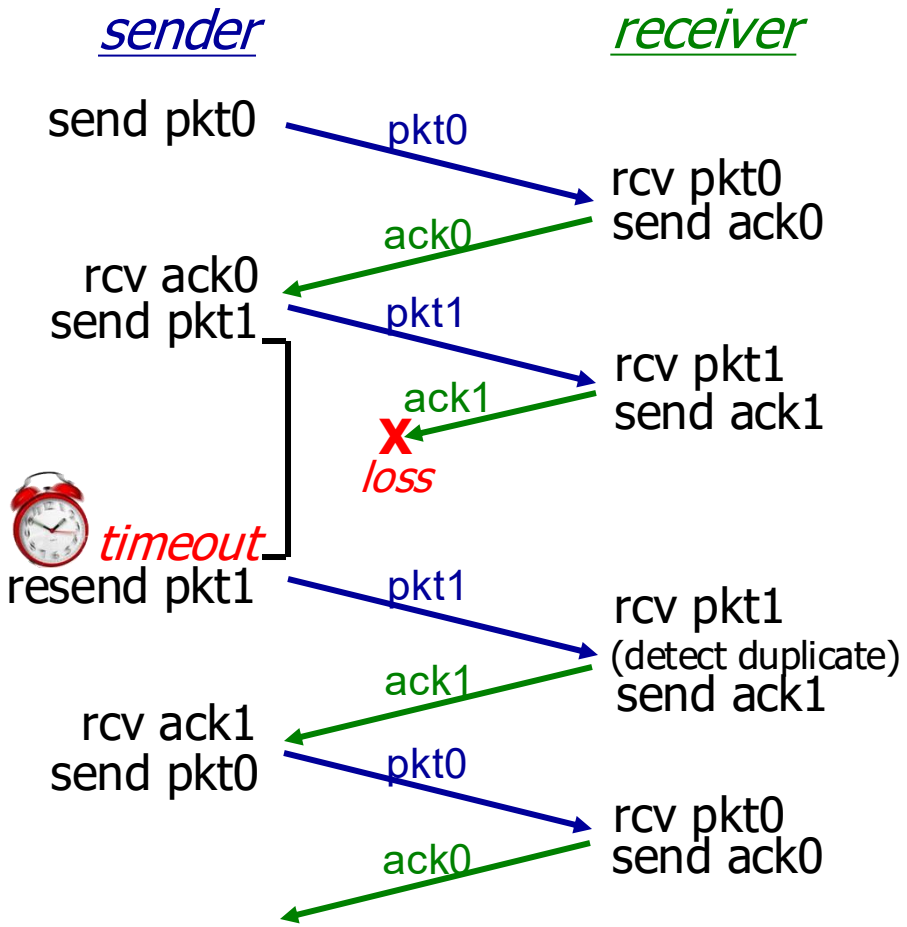


(a) no loss

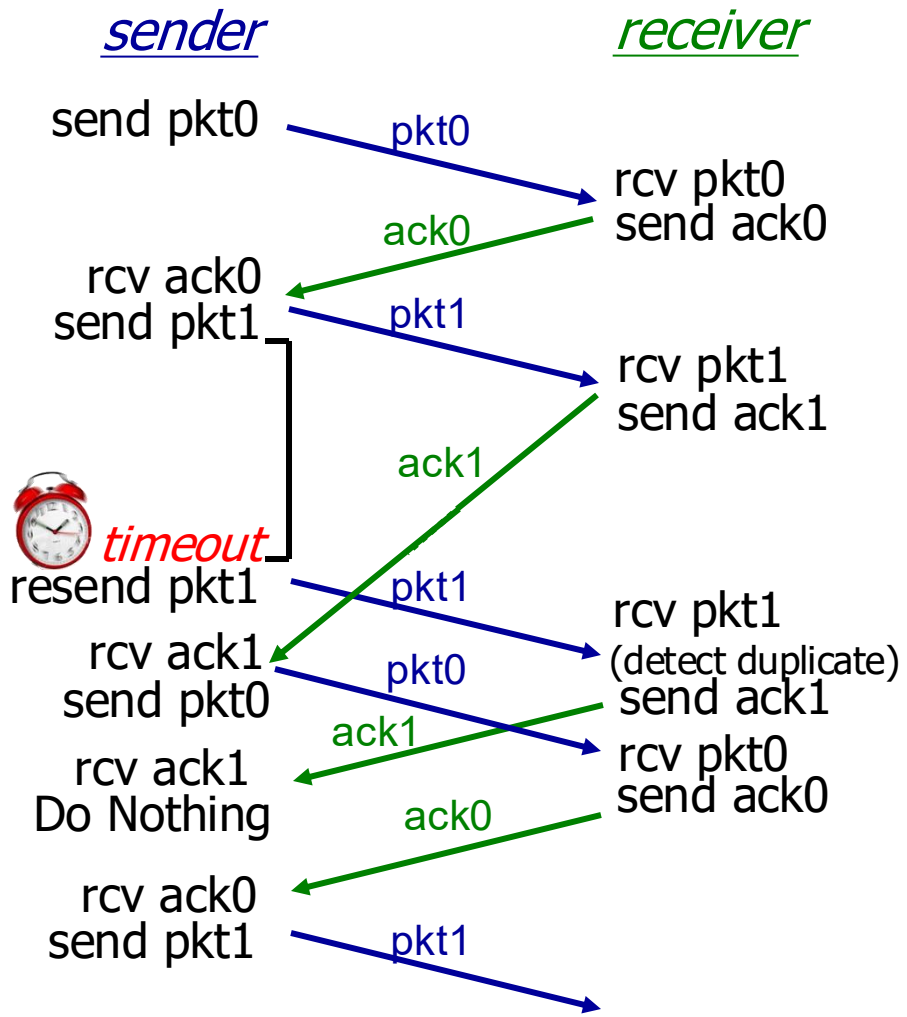


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

- **rdt3.0 is correct, but performance stinks**
- **e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:**

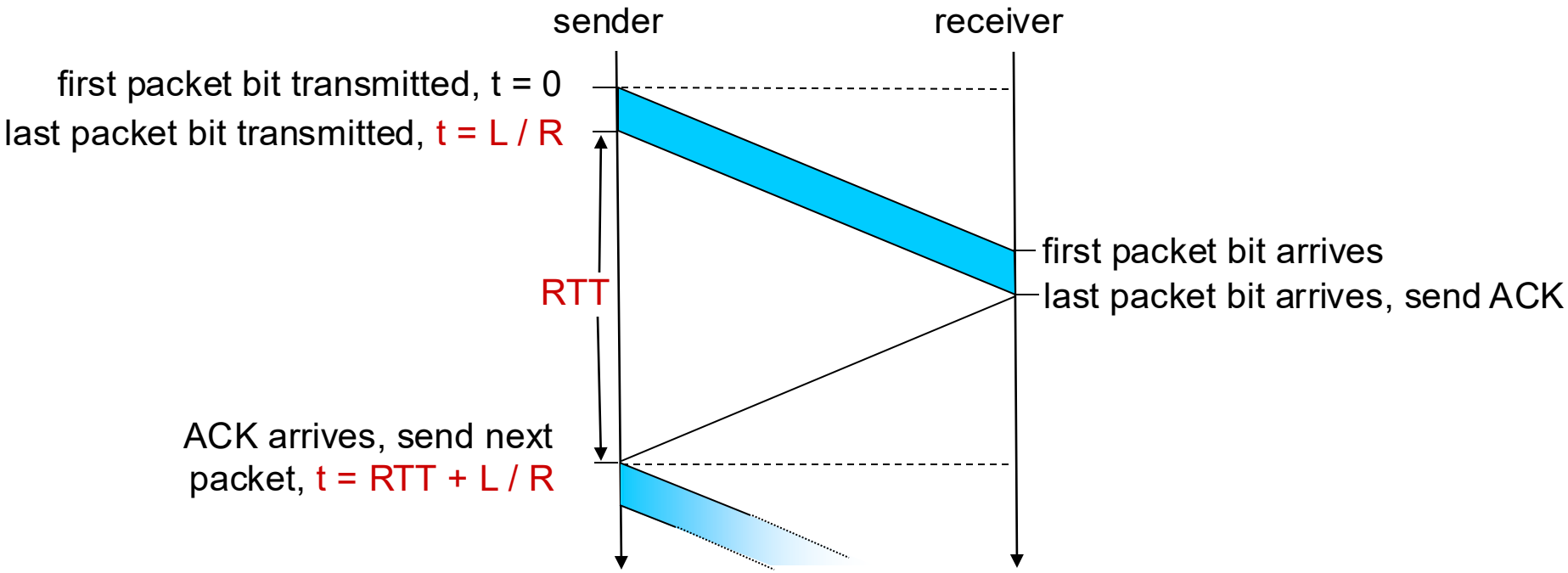
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U_{sender}: utilization – fraction of time sender busy sending

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

  - if RTT=30ms, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
- **Network protocol limits use of physical resources!**

rdt3.0: stop-and-wait operation



$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Question

- How to increase utilization?