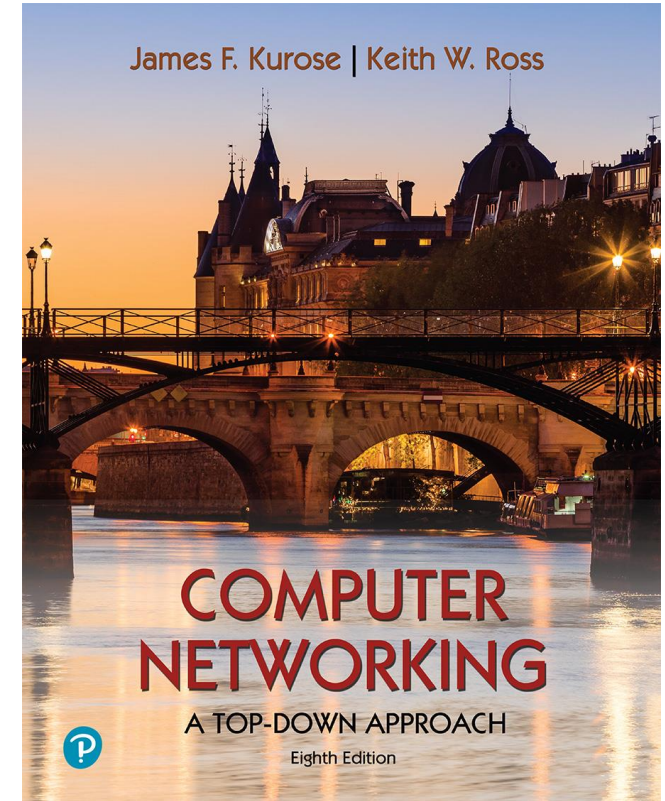# Chapter 3
Transport Layer

Yaxiong Xie

Department of Computer Science and Engineering
University at Buffalo, SUNY

Adapted from the slides of the book's authors

Computer Networking: A Top-Down Approach

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
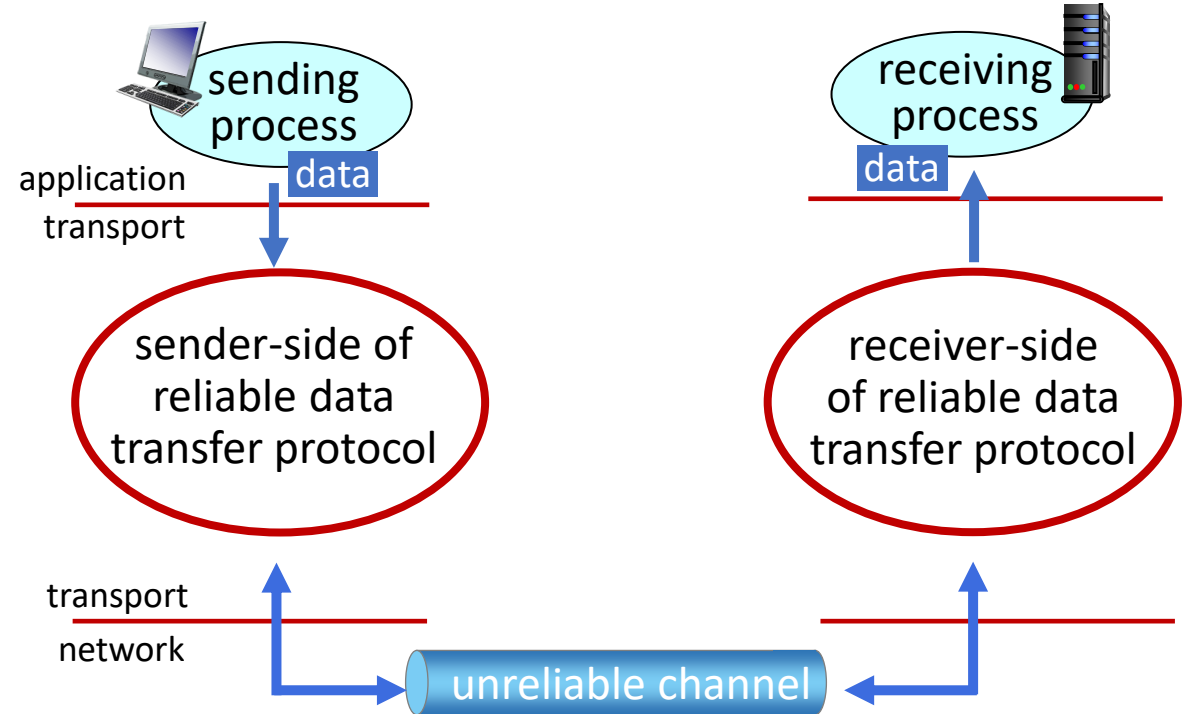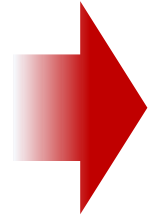- Evolution of transport-layer functionality

Principles of reliable data transfer

reliable service *abstraction*
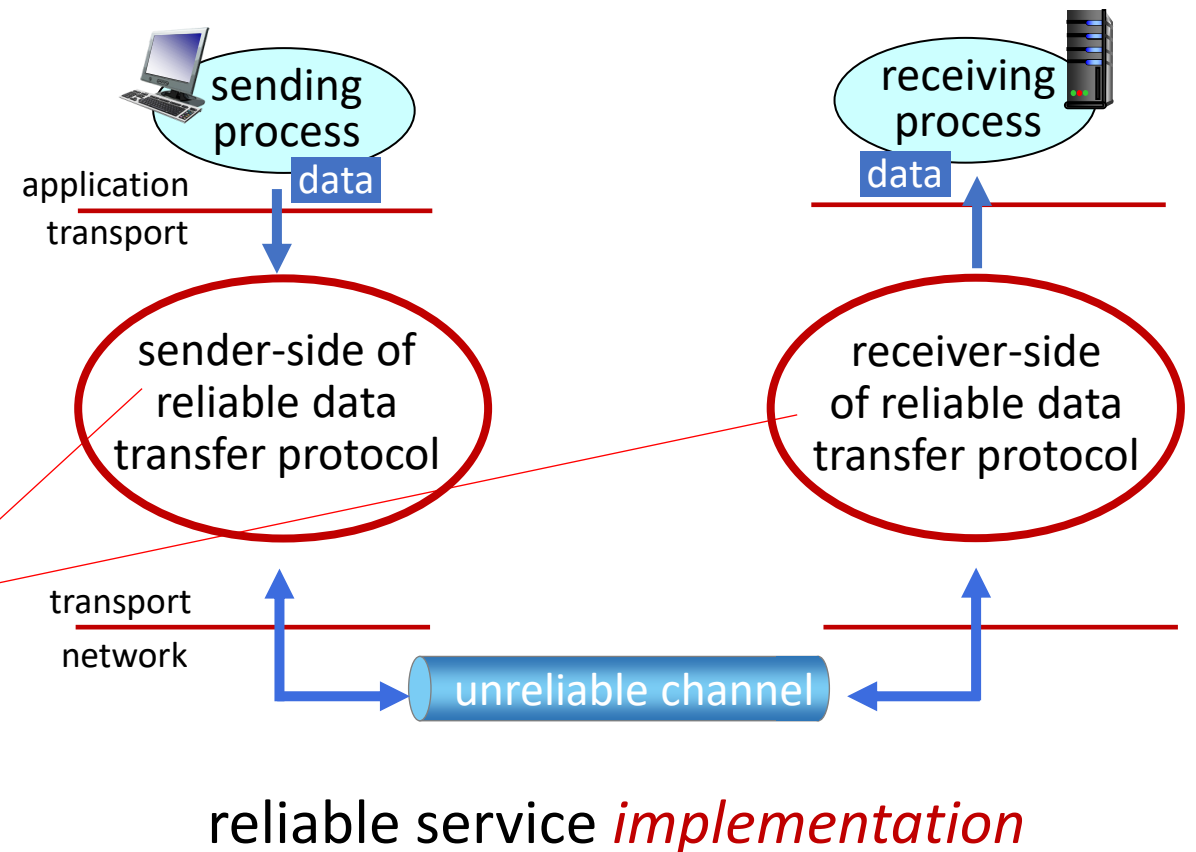
Principles of reliable data transfer

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

sending process

application
transport

data

sender-side of reliable data transfer protocol

transport
network

reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces

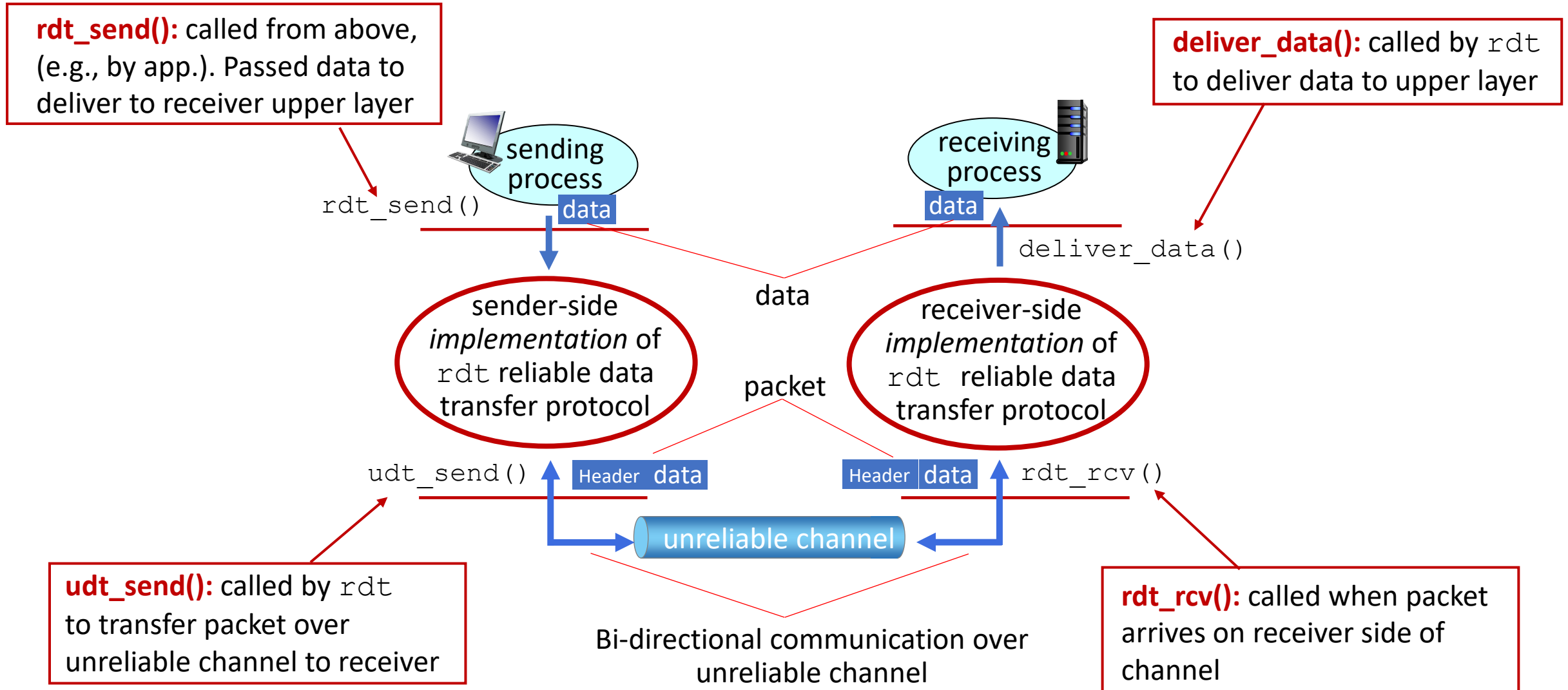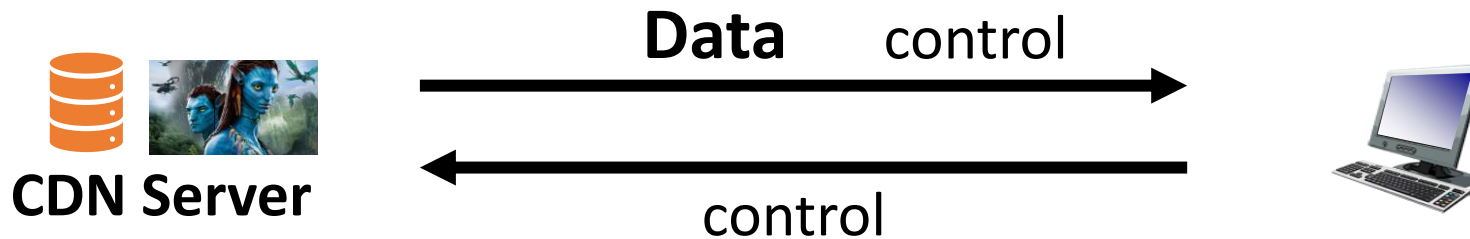**rdt_send():** called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by `rdt` to deliver data to upper layer

`rdt_send()`

sending process

data

receiving process

data

`deliver_data()`

data

sender-side *implementation* of `rdt` reliable data transfer protocol

receiver-side *implementation* of `rdt` reliable data transfer protocol

packet

`udt_send()`

Header data

Header data

`rdt_rcv()`

unreliable channel

udt_send(): called by `rdt` to transfer packet over unreliable channel to receiver

Bi-directional communication over unreliable channel

rdt_rcv(): called when packet arrives on receiver side of channel

Reliable data transfer: getting started
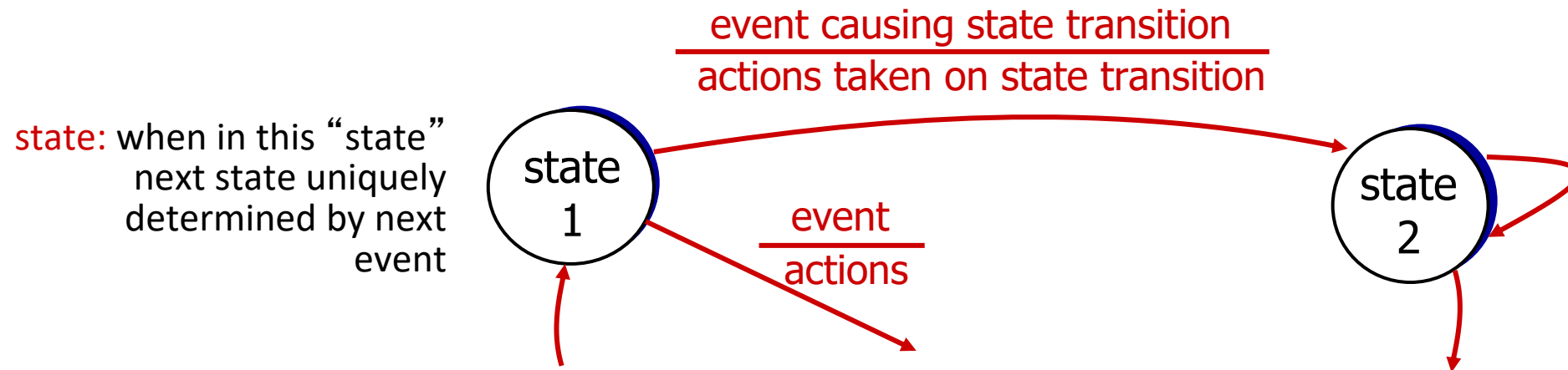
We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)

- consider only unidirectional data transfer
  - but control info will flow in both directions!

Data control

control

CDN Server

Reliable data transfer: Protocol States

- use finite state machines (FSM) to specify sender, receiver

event causing state transition
actions taken on state transition

state: when in this "state"
next state uniquely
determined by next
event

state 1

event
actions

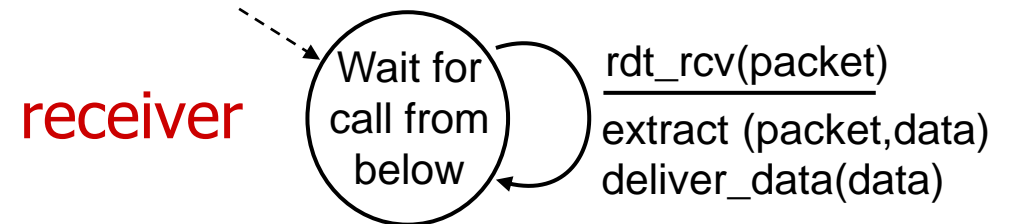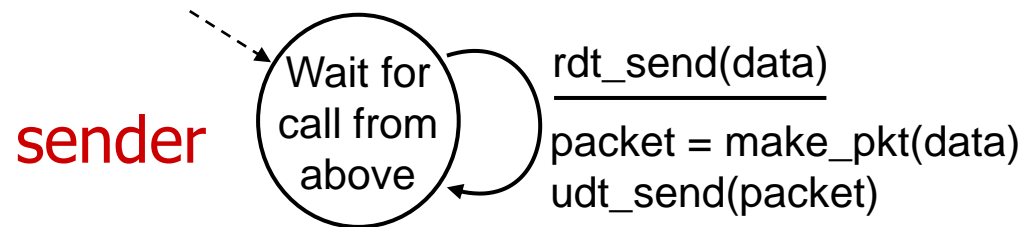state 2

Channel model: Reliable Channel

reliable service *abstraction*

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

rdt1.0: reliable transfer over a reliable channel

- *separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender

Wait for call from above

rdt_send(data)
_____
packet = make_pkt(data)
udt_send(packet)

receiver

Wait for call from below

rdt_rcv(packet)
_____
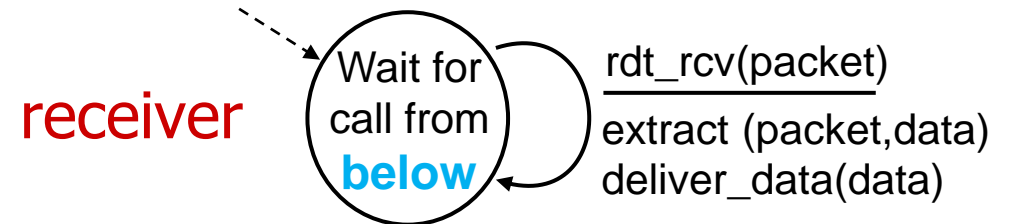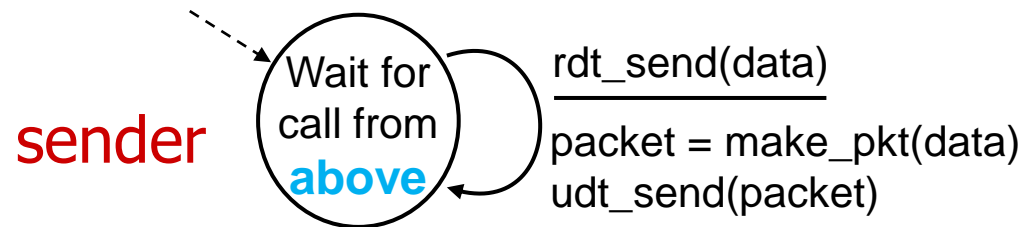extract (packet,data)
deliver_data(data)
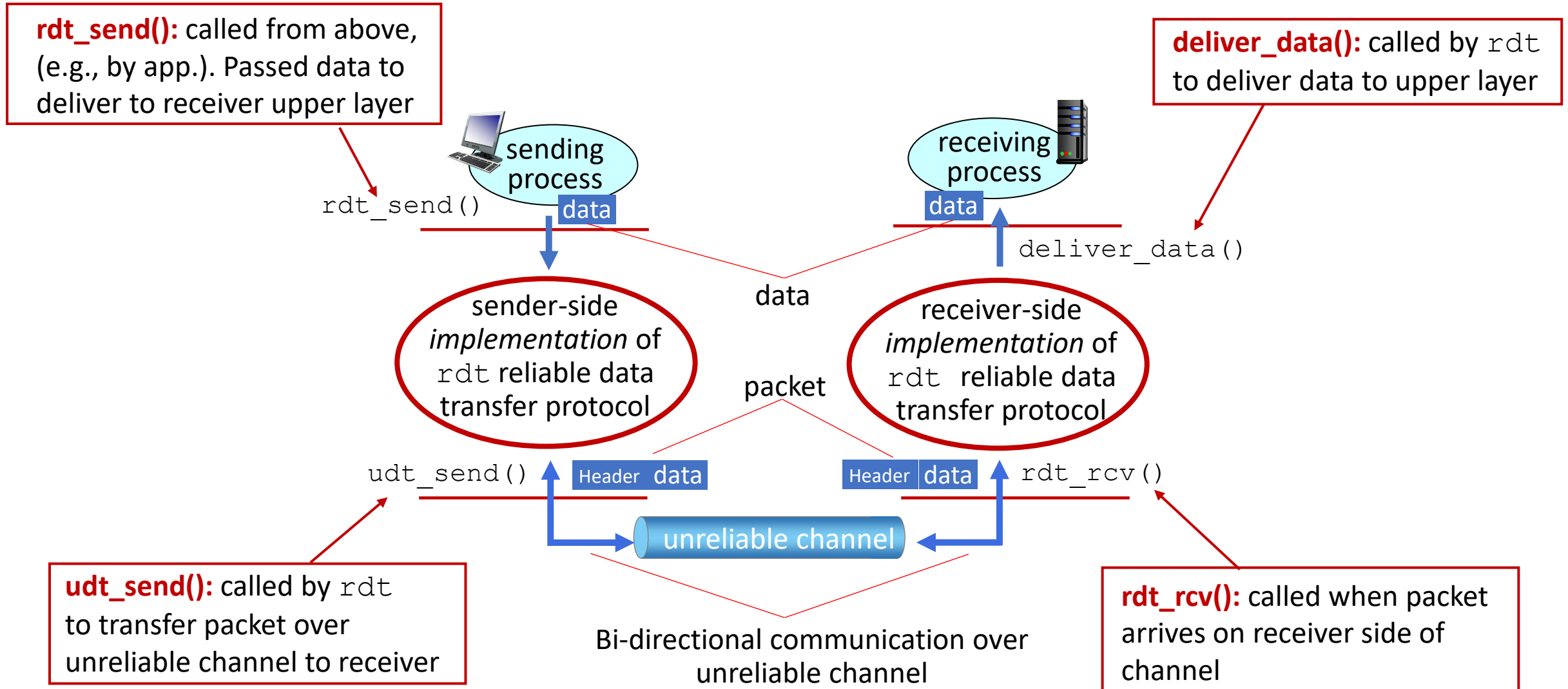
rdt1.0: reliable transfer over a reliable channel

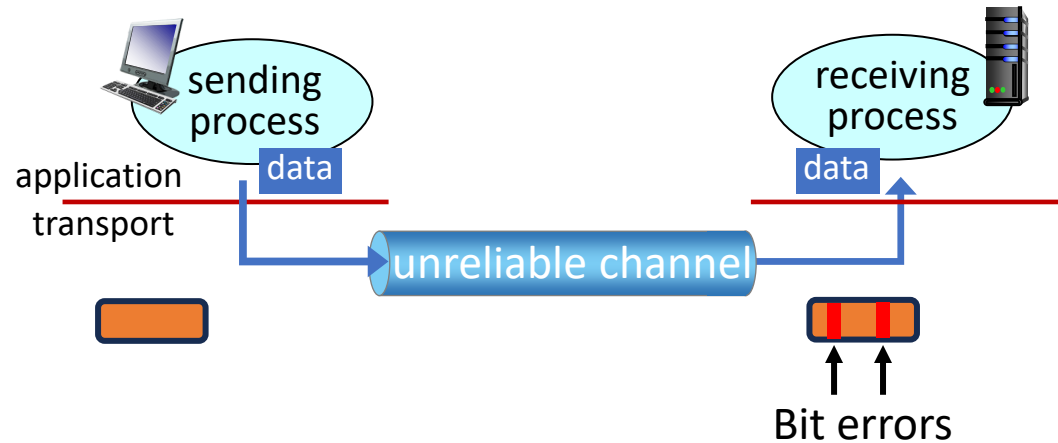- *separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

sender

Wait for call from **above**

rdt_send(data)
_____
packet = make_pkt(data)
udt_send(packet)

receiver

Wait for call from **below**

rdt_rcv(packet)
_____
extract (packet,data)
deliver_data(data)

Reliable data transfer protocol (rdt): interfaces

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by `rdt` to deliver data to upper layer

`rdt_send()`

sending process

data

receiving process

data

`deliver_data()`

sender-side *implementation* of `rdt` reliable data transfer protocol

data

receiver-side *implementation* of `rdt` reliable data transfer protocol

packet

`udt_send()`

Header data

Header data

`rdt_rcv()`

unreliable channel

udt_send(): called by `rdt` to transfer packet over unreliable channel to receiver

Bi-directional communication over unreliable channel

rdt_rcv(): called when packet arrives on receiver side of channel

Channel model: channel with bit errors

- **underlying channel may flip bits in packet**
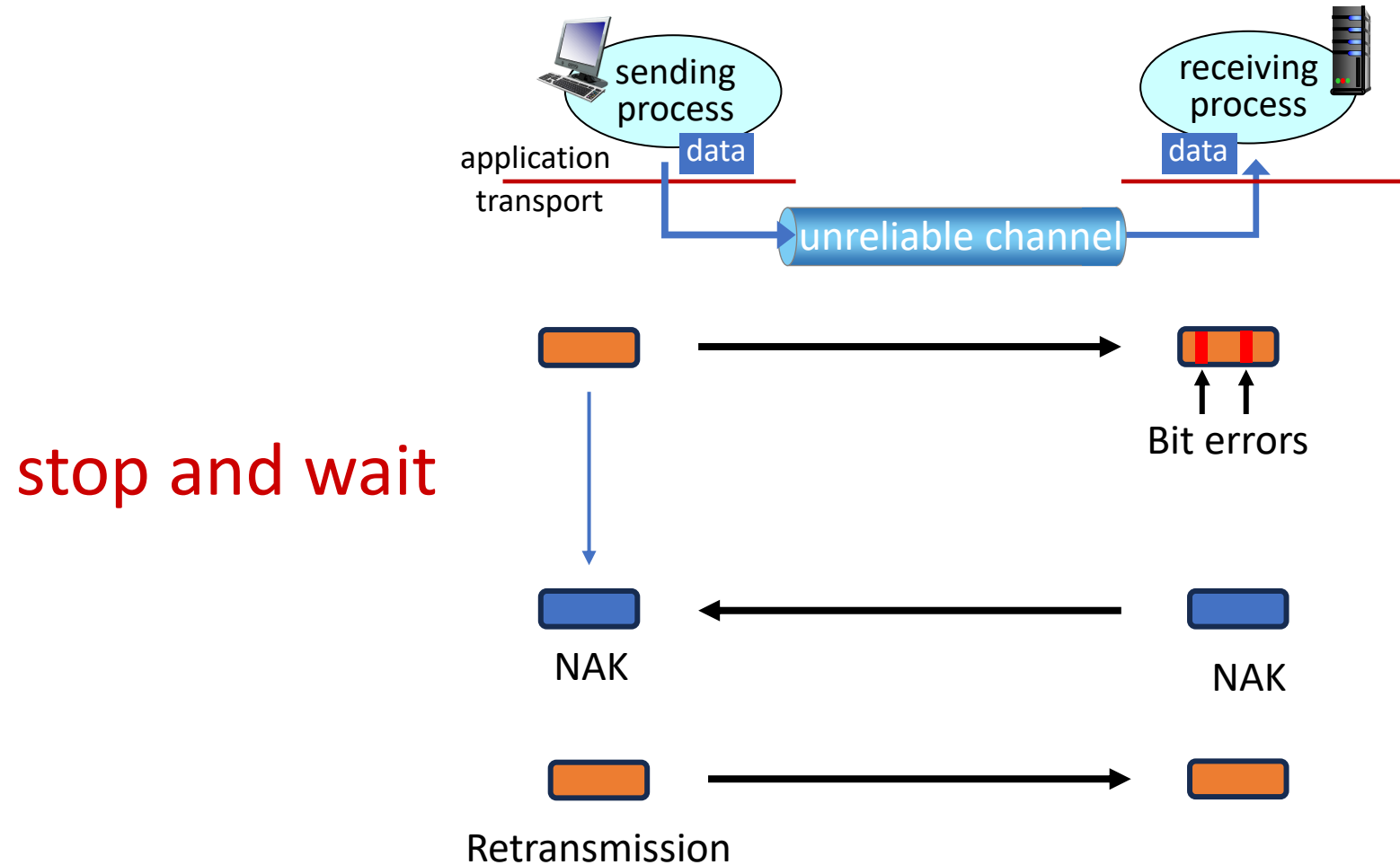 - checksum (e.g., Internet checksum) to detect bit errors

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
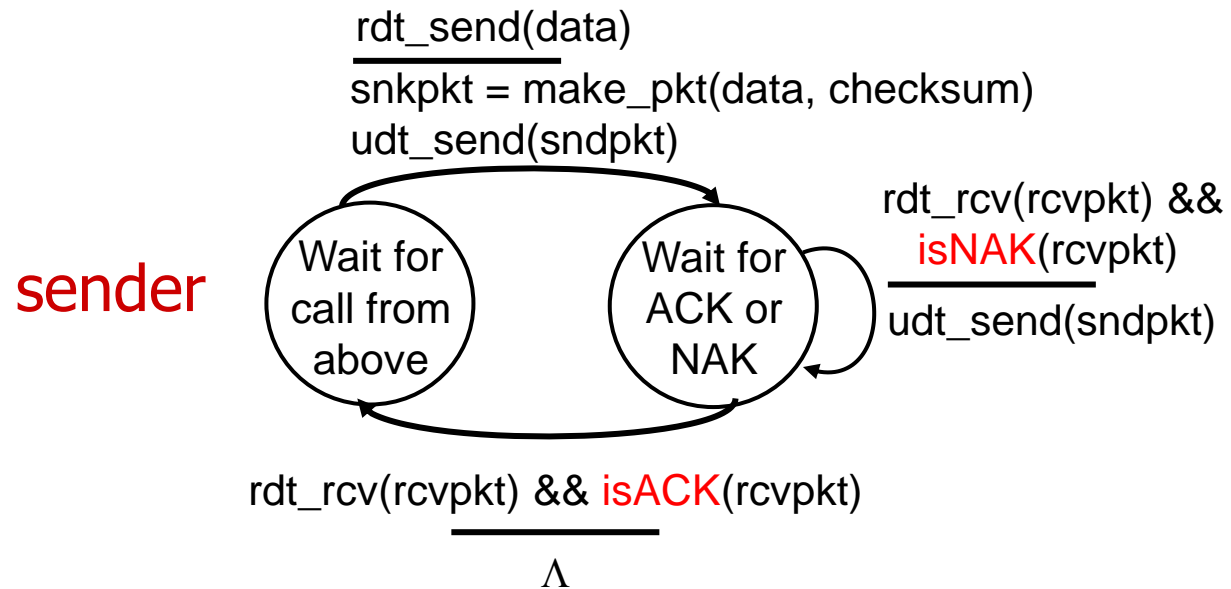 - sender *retransmits* pkt on receipt of NAK

---

stop and wait
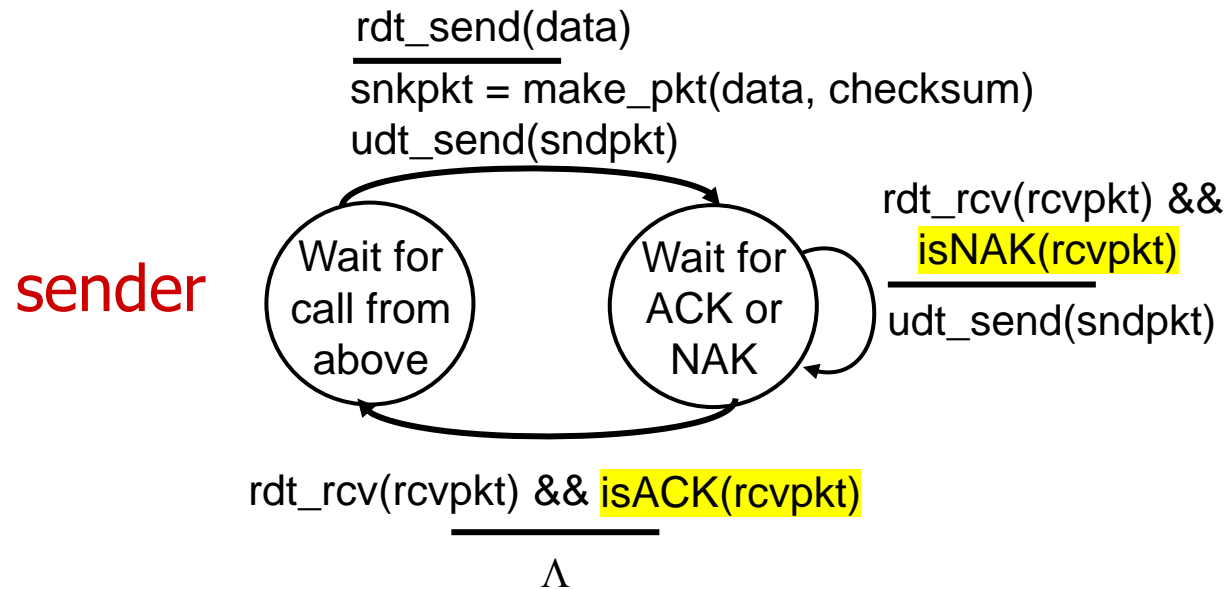sender sends one packet, then waits for receiver response

---

rdt2.0: channel with bit errors

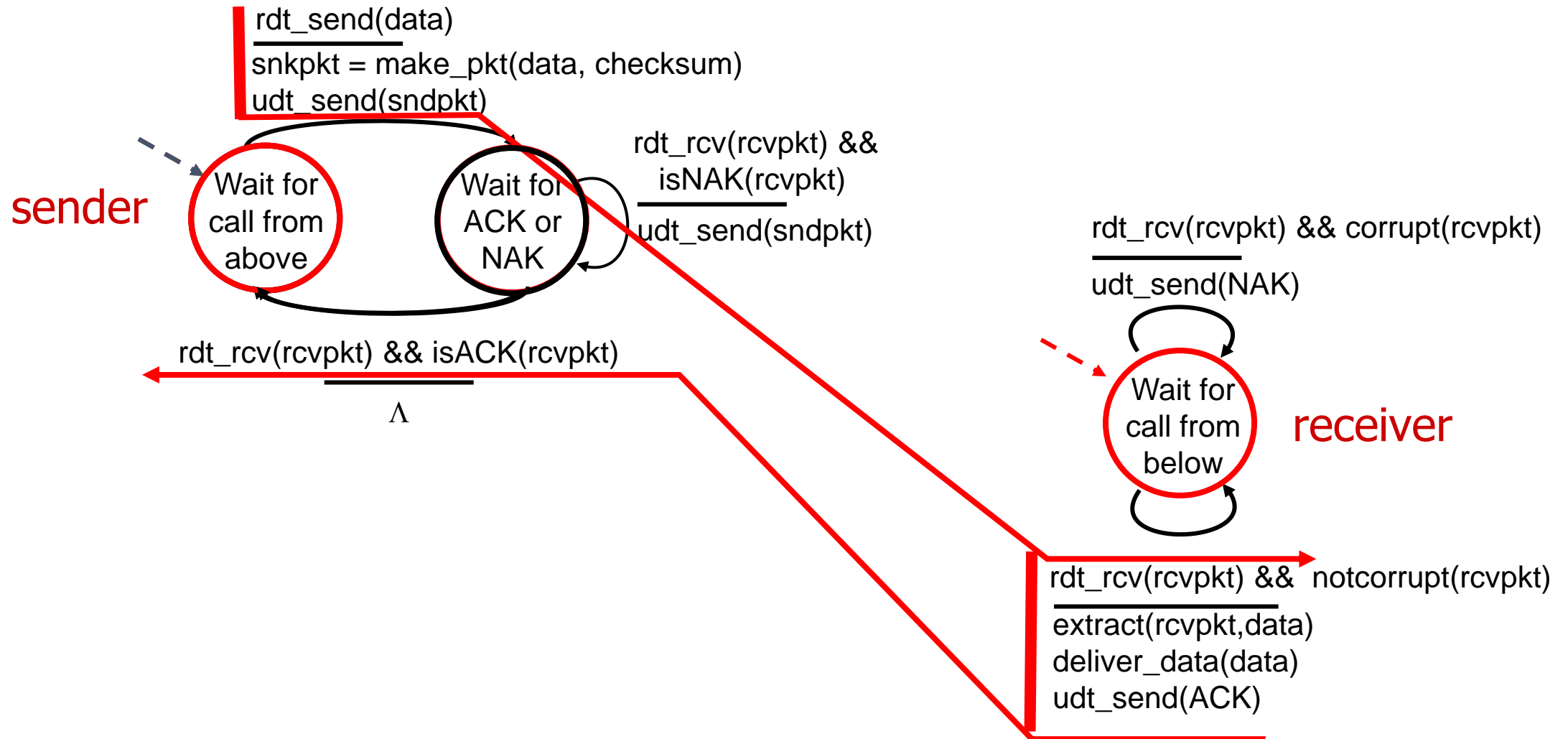stop and wait

Bit errors

NAK   NAK

Retransmission

rdt2.0: FSM specifications

sender

rdt2.0: FSM specification

sender

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
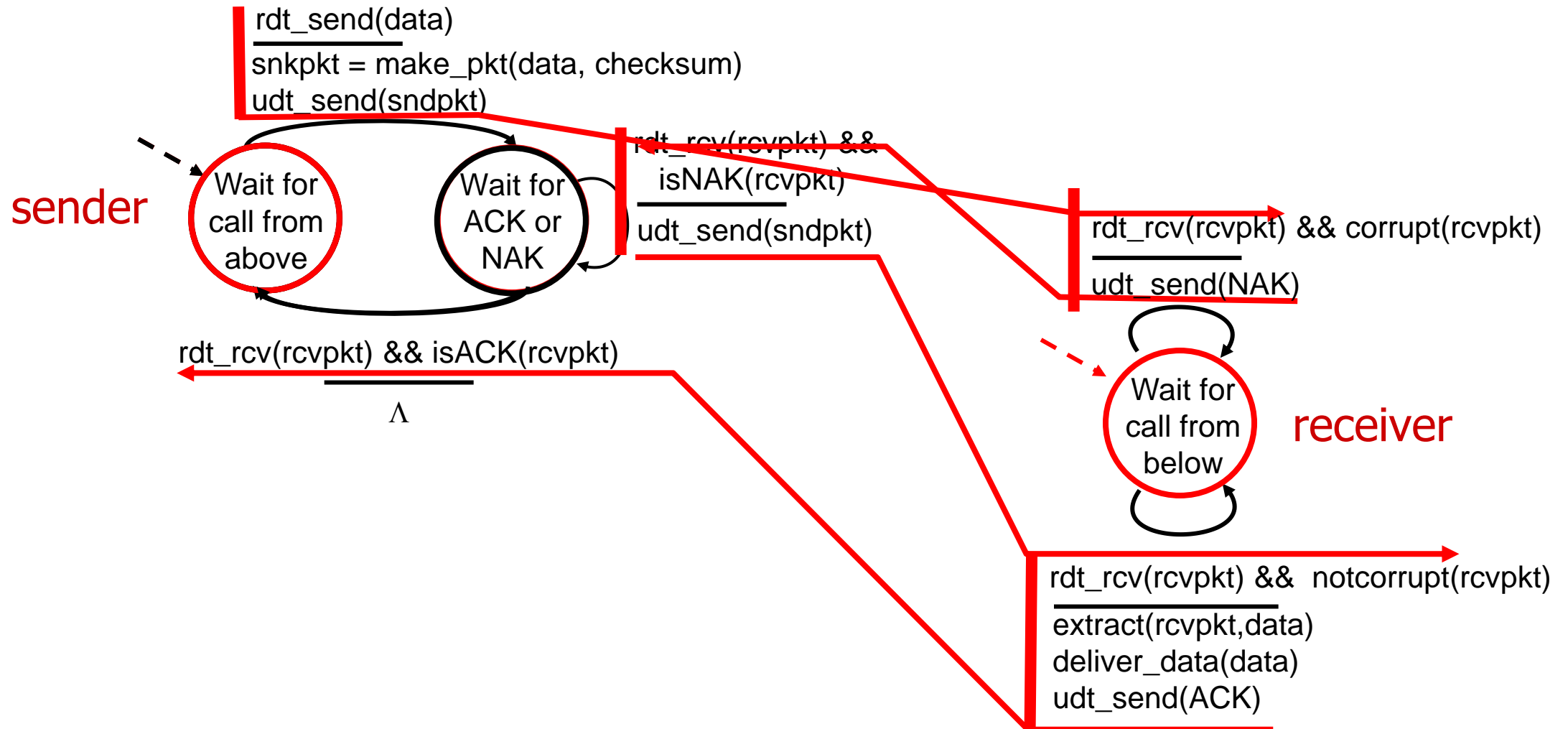udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender
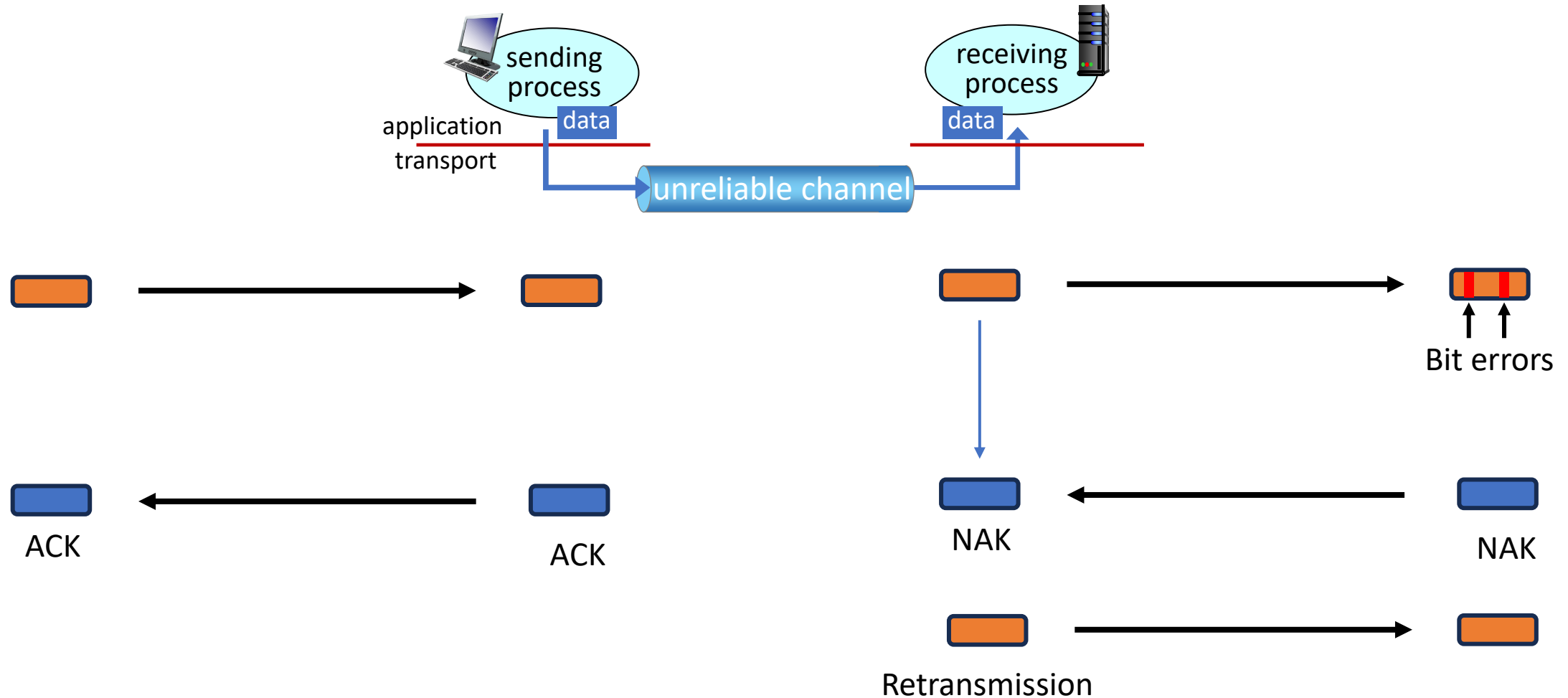- that's why we need a protocol!

rdt2.0: operation with no errors

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

sender

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
_____
udt_send(NAK)

Wait for call from below

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: corrupted packet scenario

sender

rdt_send(data)
———
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
———
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
———
Λ

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
———
udt_send(NAK)

receiver

Wait for call from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
———
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

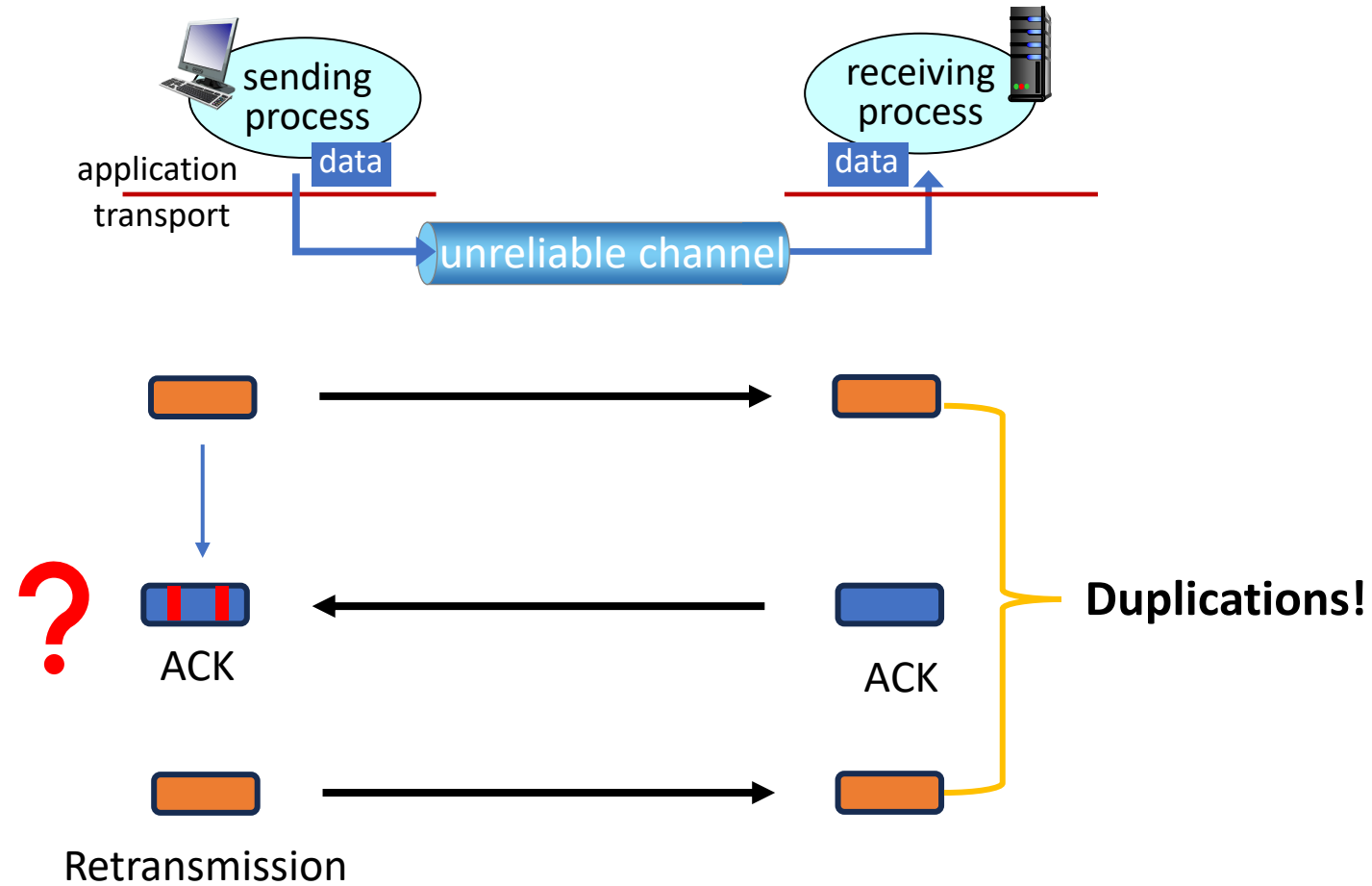# rdt2.0: no errors VS. corrupted packets

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!

- can't just retransmit: possible duplicate

rdt2.0: corrupted ACK

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
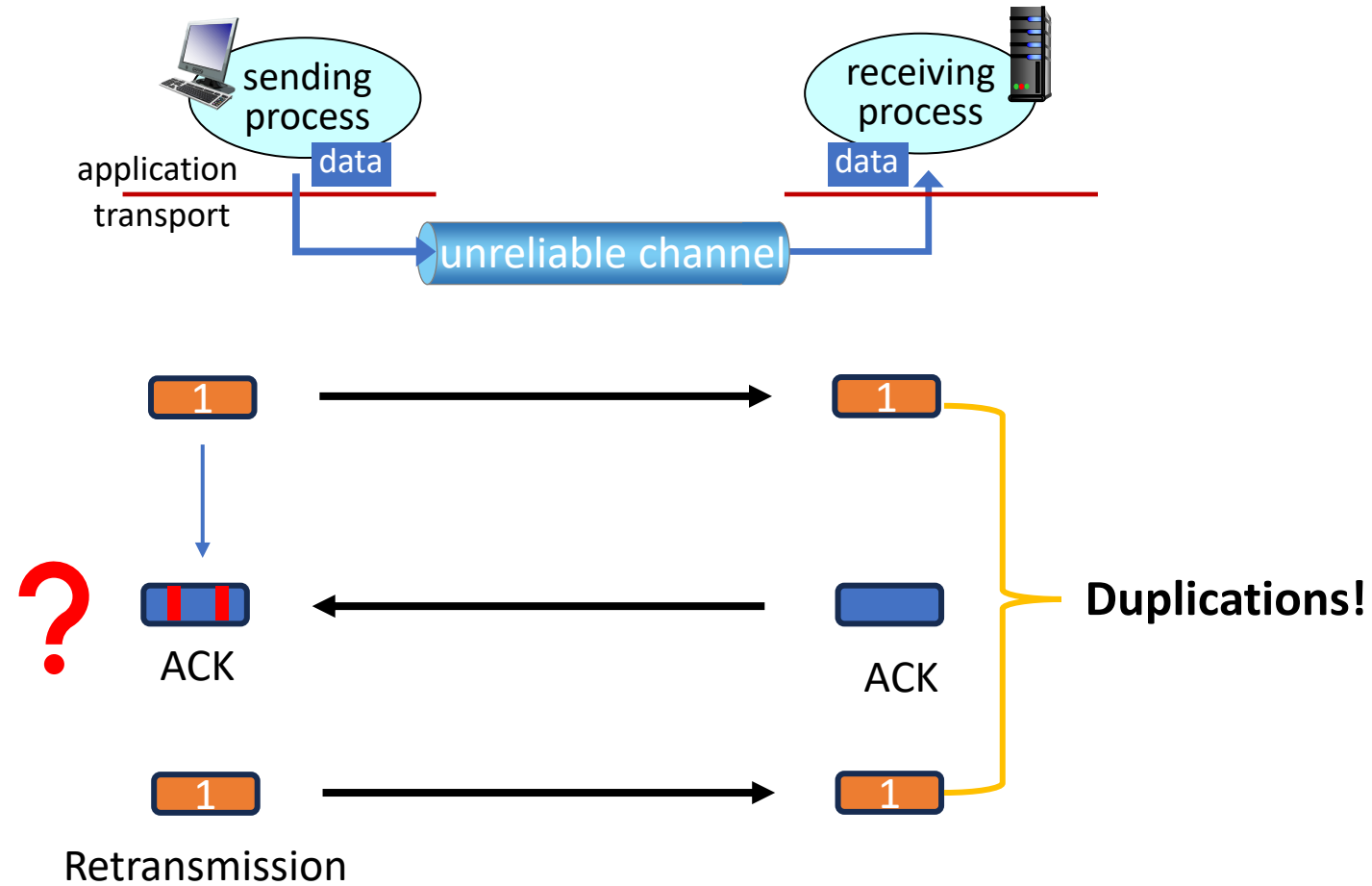- can't just retransmit: possible duplicate

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

stop and wait

sender sends one packet, then waits for receiver response
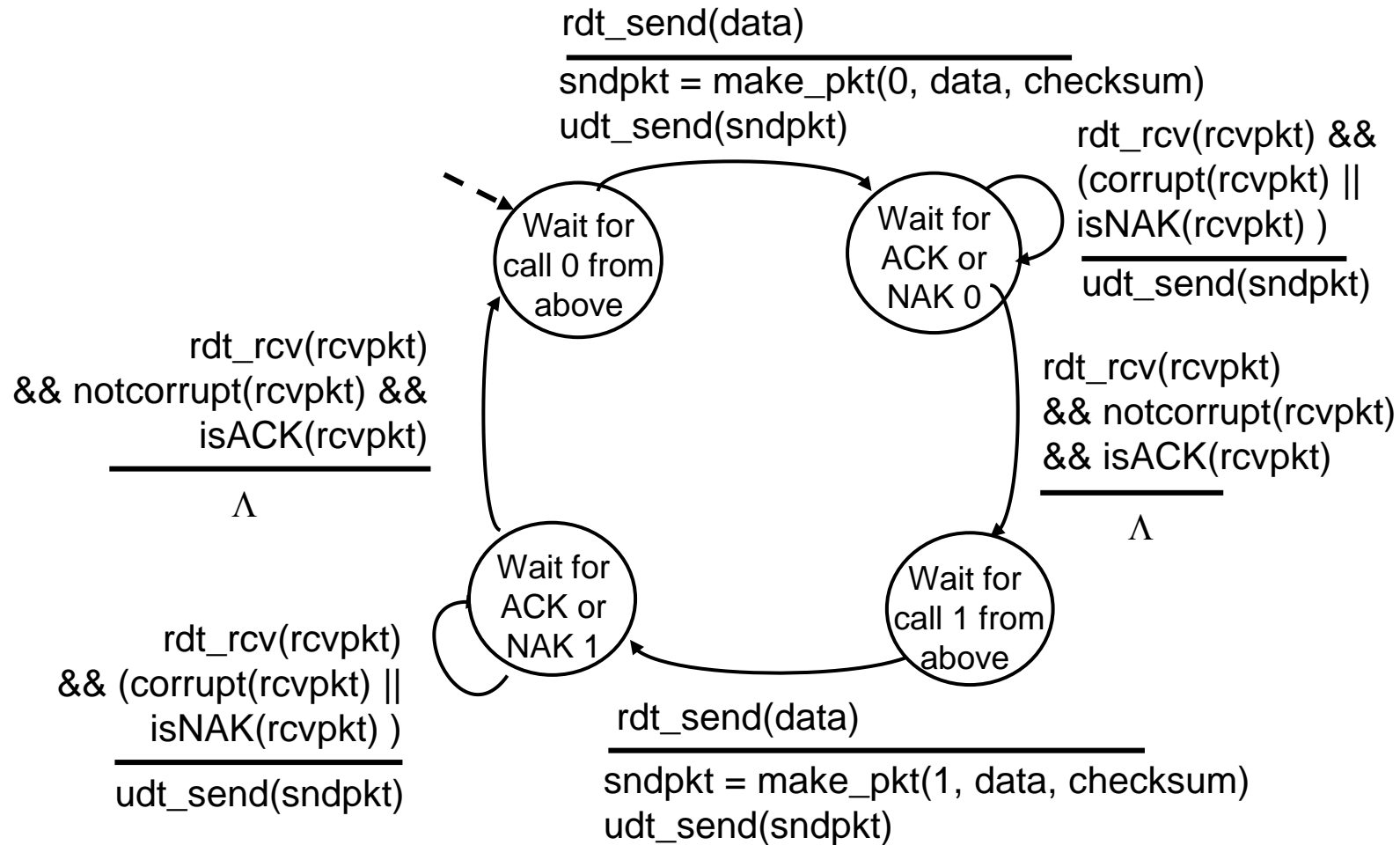
rdt2.0: corrupted ACK

rdt2.1: summary

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
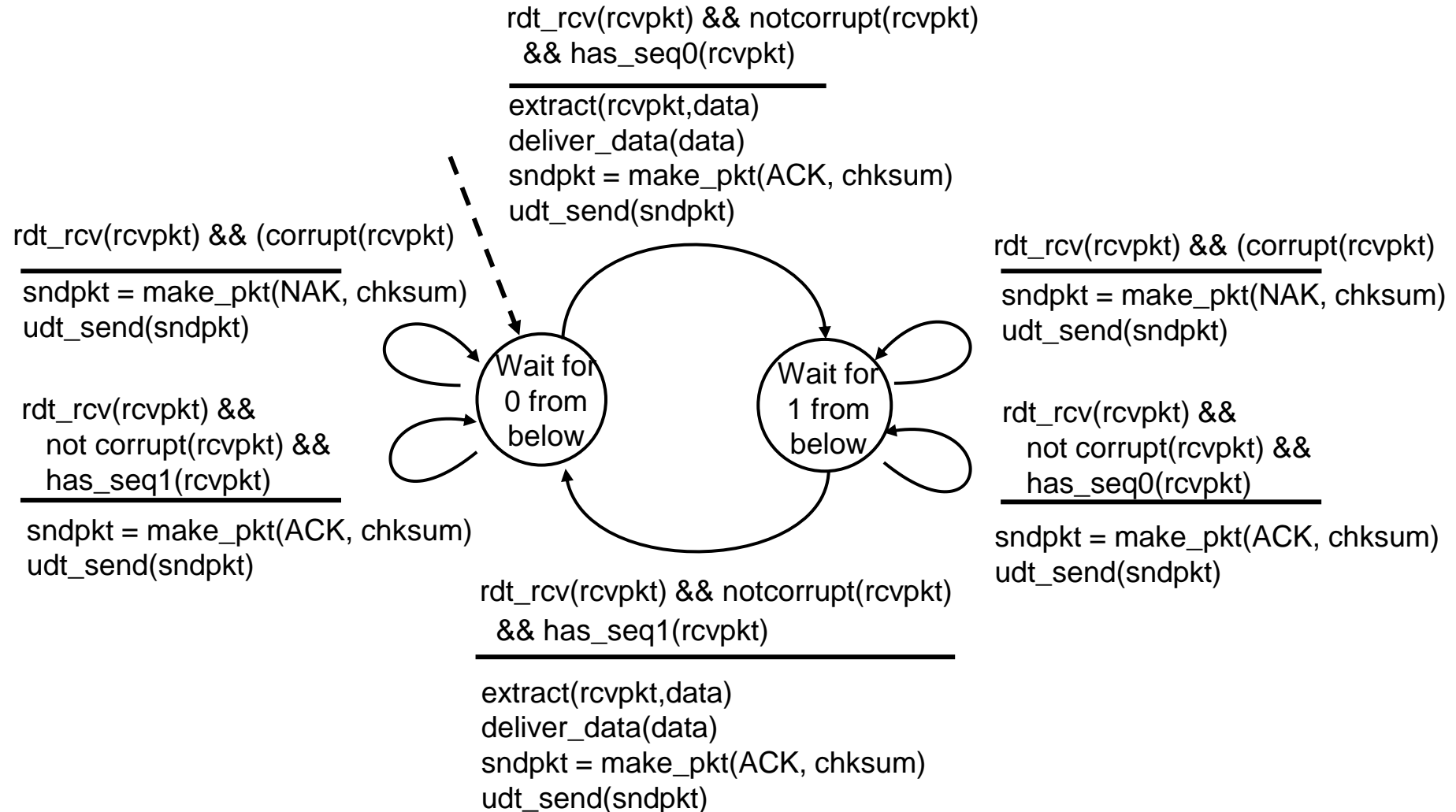
receiver:

- 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

rdt2.1: sender, handling garbled ACK/NAKs

rdt2.1: receiver, handling garbled ACK/NAKs

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt) &&
  has_seq1(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

(Wait for 0 from below)   (Wait for 1 from below)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
_____
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
  not corrupt(rcvpkt) &&
  has_seq0(rcvpkt)
_____
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
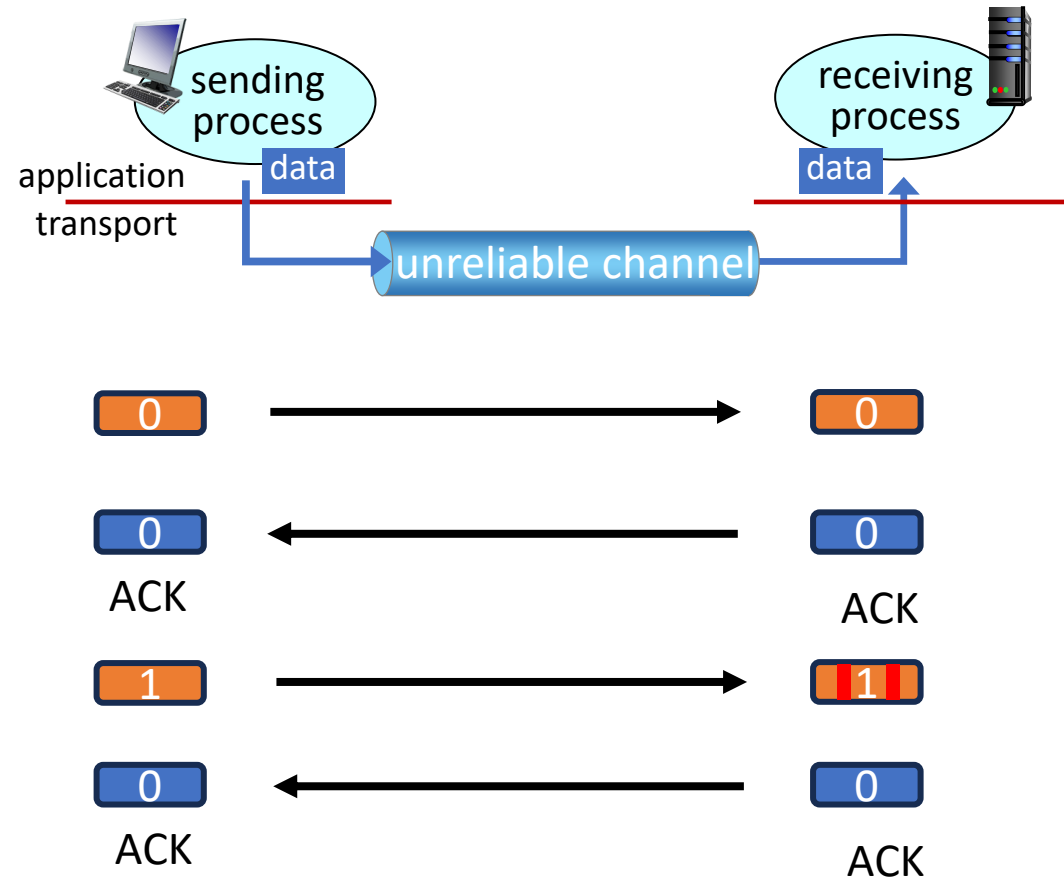udt_send(sndpkt)

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only

- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed

- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

As we will see, TCP uses this approach to be NAK-free

rdt2.0: NAK-free

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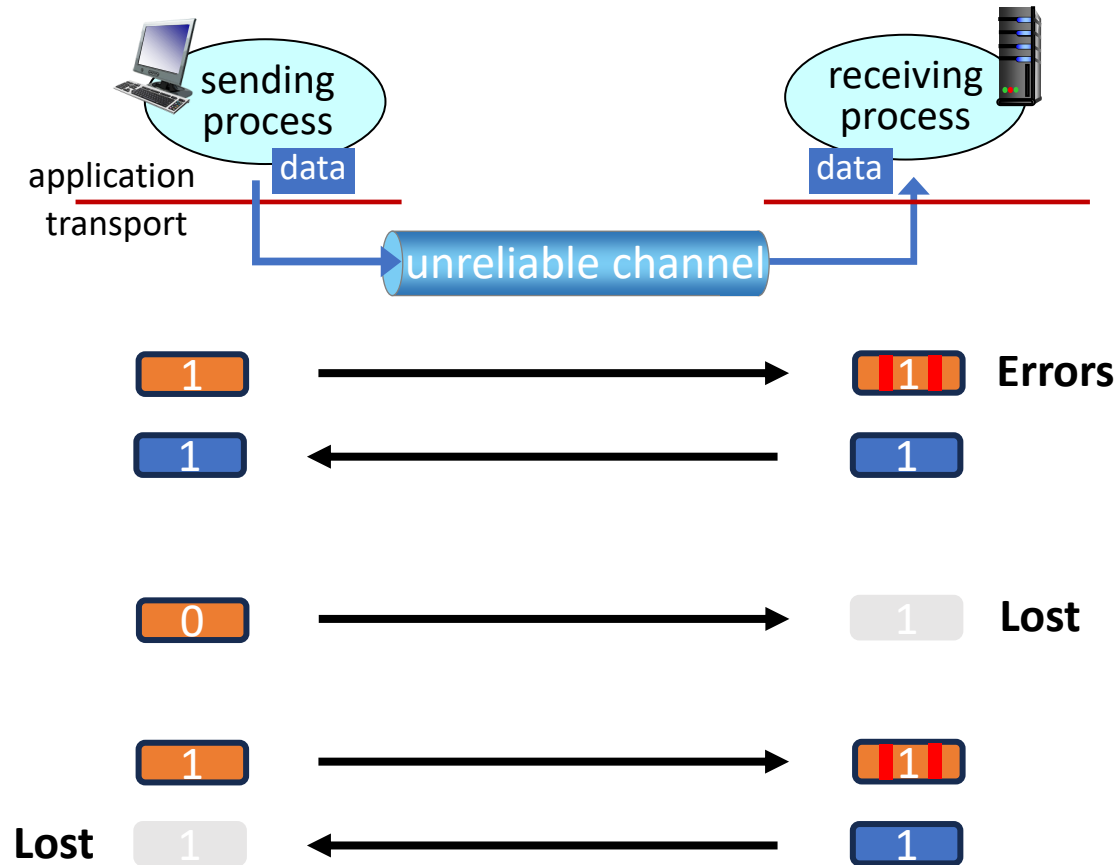
rdt3.0: channels with errors *and* loss

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

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?
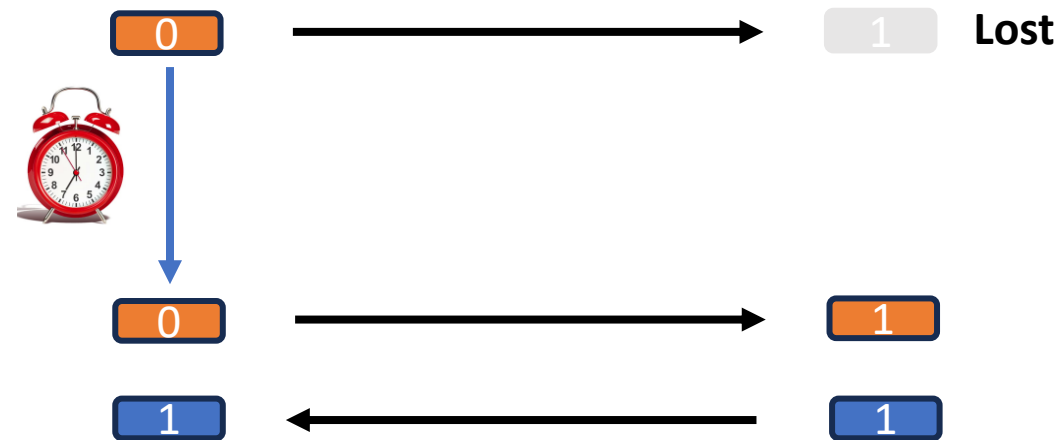
rdt3.0: channels with errors *and* loss

Approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time

- if pkt (or ACK) just delayed (not lost):

    - retransmission will be duplicate, but seq #s already handles this!

    - receiver must specify seq # of packet being ACKed

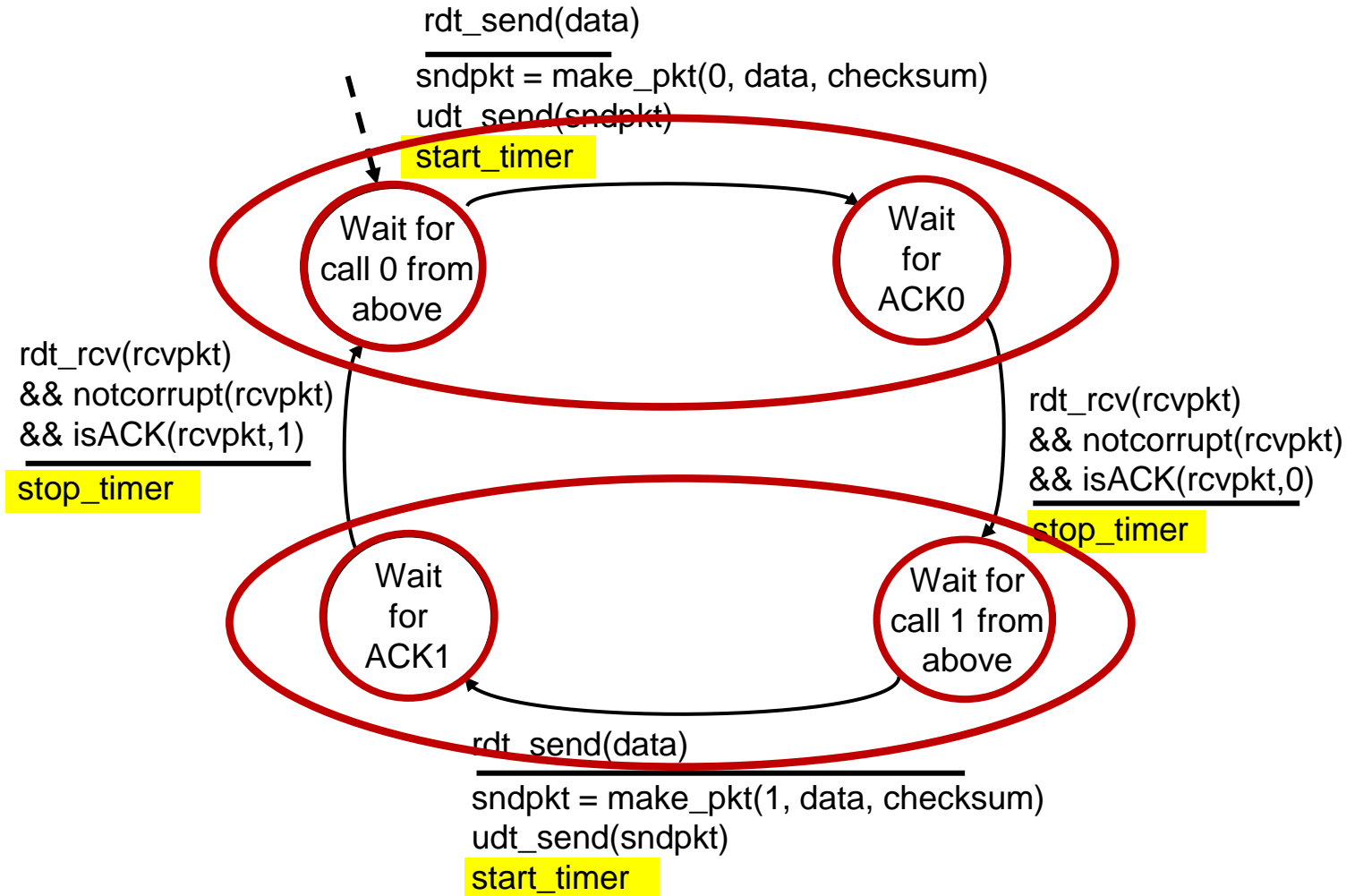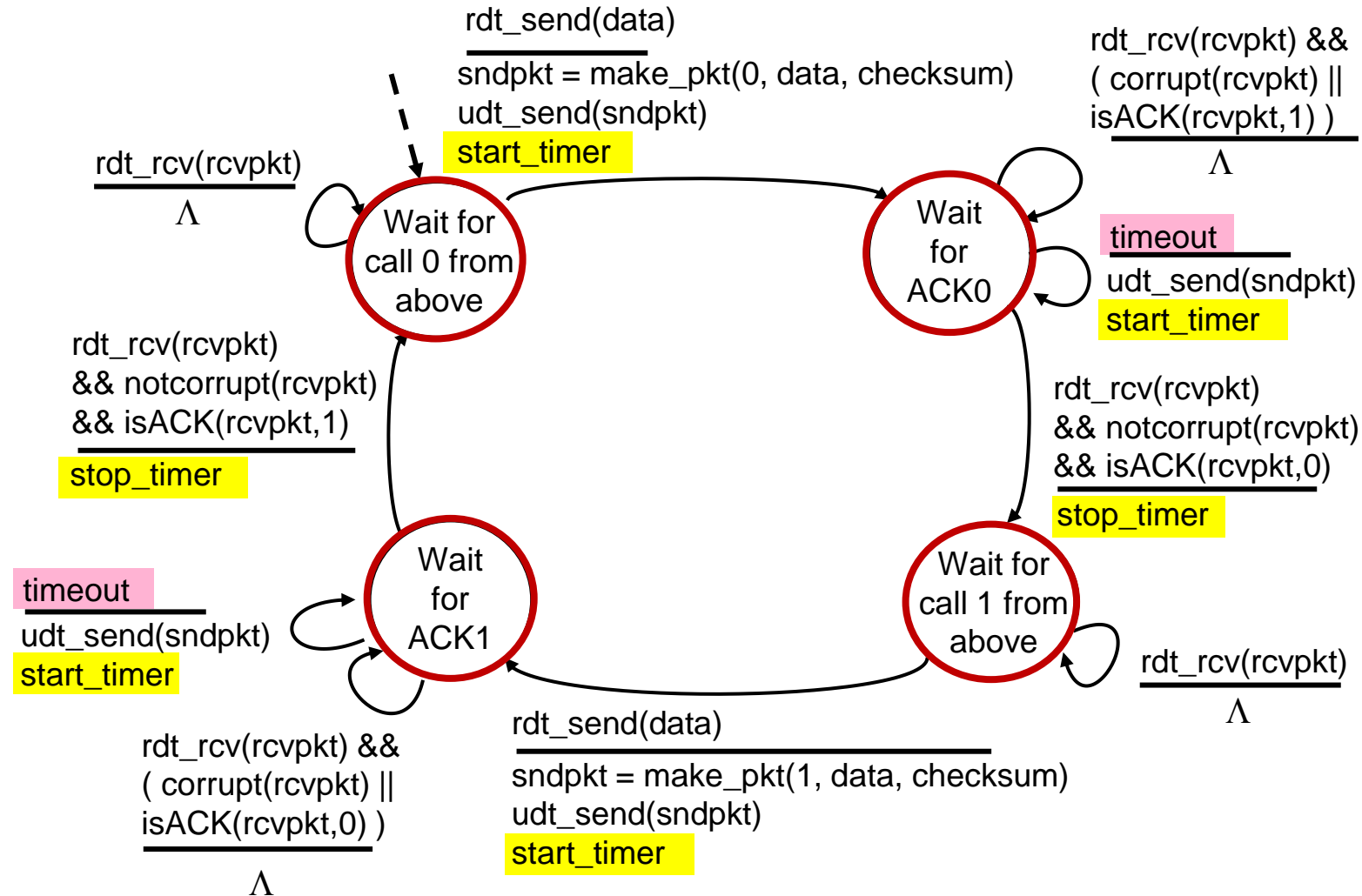- use countdown timer to interrupt after "reasonable" amount of time

timeout

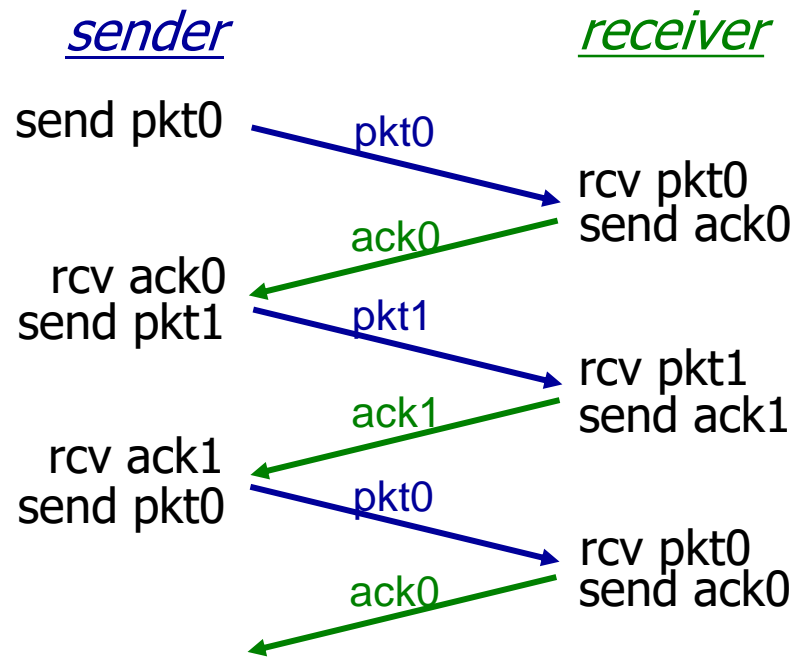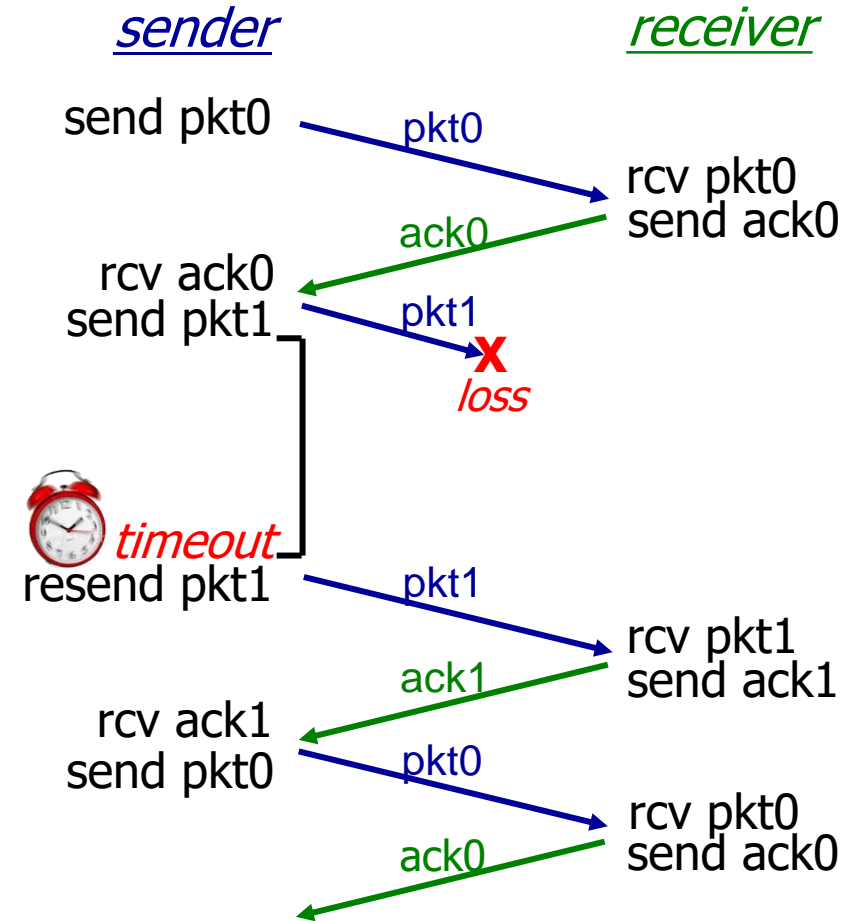# rdt3.0: channels with errors *and* loss
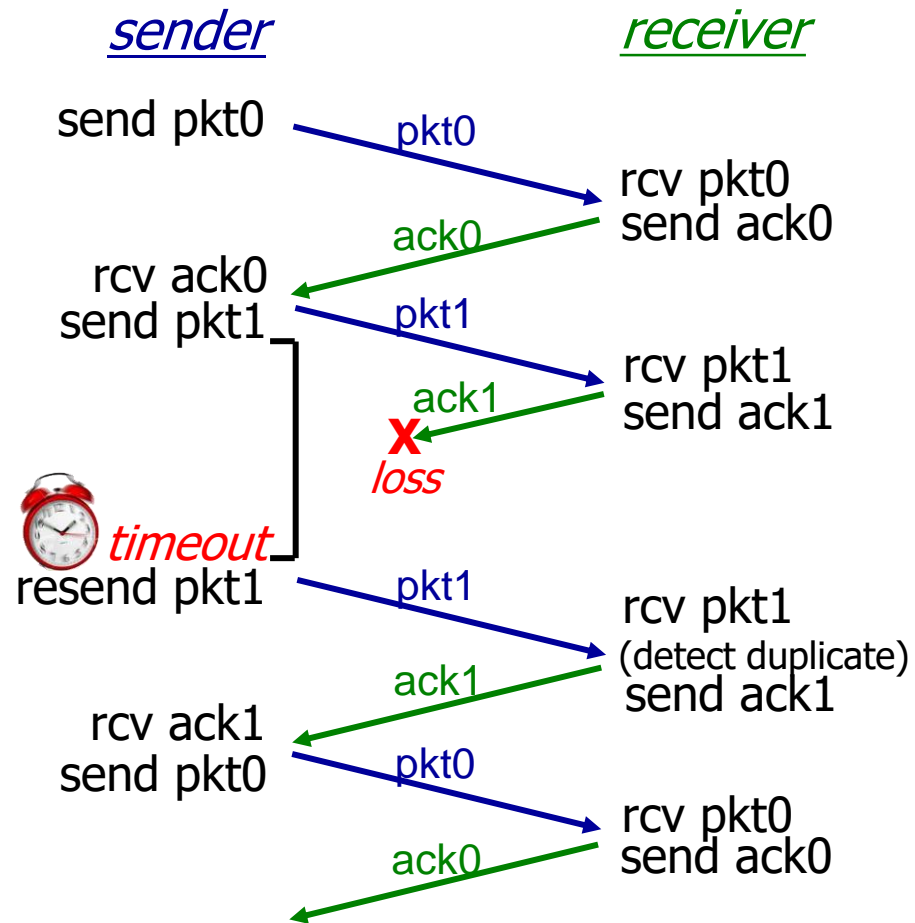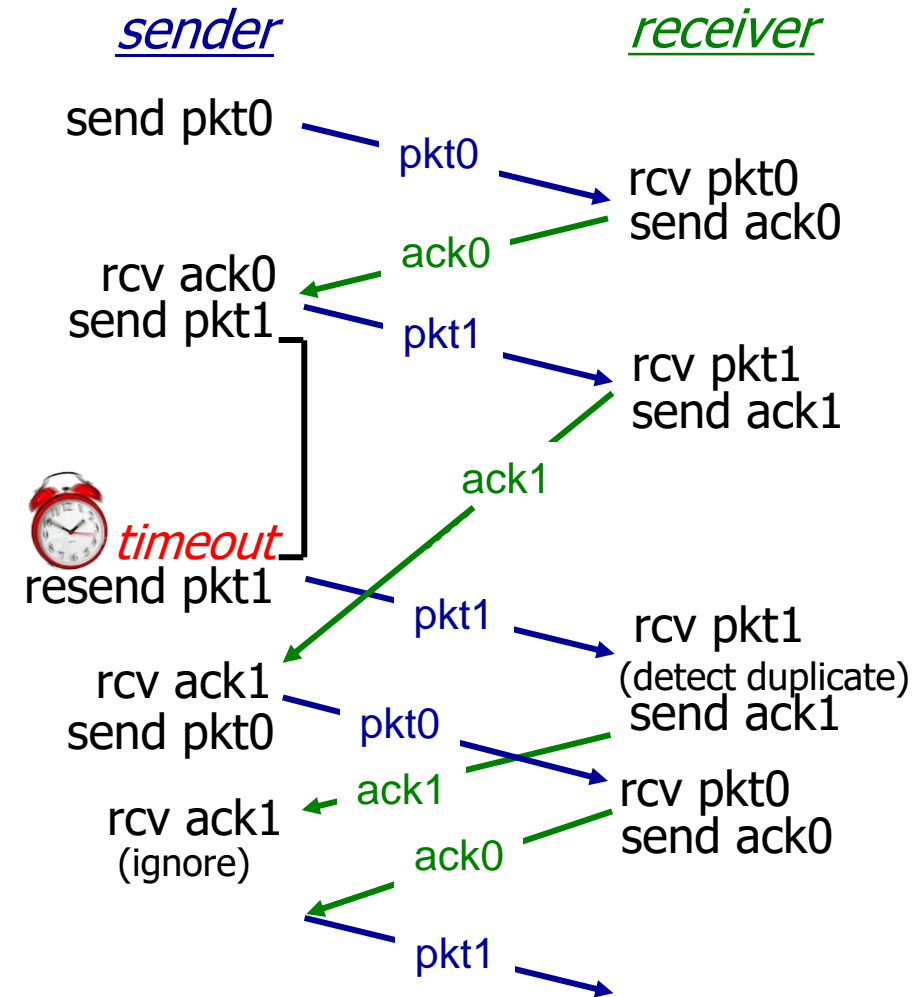
rdt3.0 sender

rdt3.0 sender

rdt3.0 in action

sender   receiver

send pkt0 —— pkt0 ——→
                              rcv pkt0
                              send ack0
rcv ack0 ←—— ack0 ——
send pkt1 —— pkt1 ——→
                              rcv pkt1
                              send ack1
rcv ack1 ←—— ack1 ——
send pkt0 —— pkt0 ——→
                              rcv pkt0
                              send ack0
         ←—— ack0 ——

(a) no loss

sender   receiver

send pkt0 —— pkt0 ——→
                              rcv pkt0
                              send ack0
rcv ack0 ←—— ack0 ——
send pkt1 —— pkt1 ——→ ✗
                              loss
timeout
resend pkt1 —— pkt1 ——→
                              rcv pkt1
                              send ack1
rcv ack1 ←—— ack1 ——
send pkt0 —— pkt0 ——→
                              rcv pkt0
                              send ack0
         ←—— ack0 ——

(b) packet loss

rdt3.0 in action

(c) ACK loss

(d) premature timeout/ delayed ACK

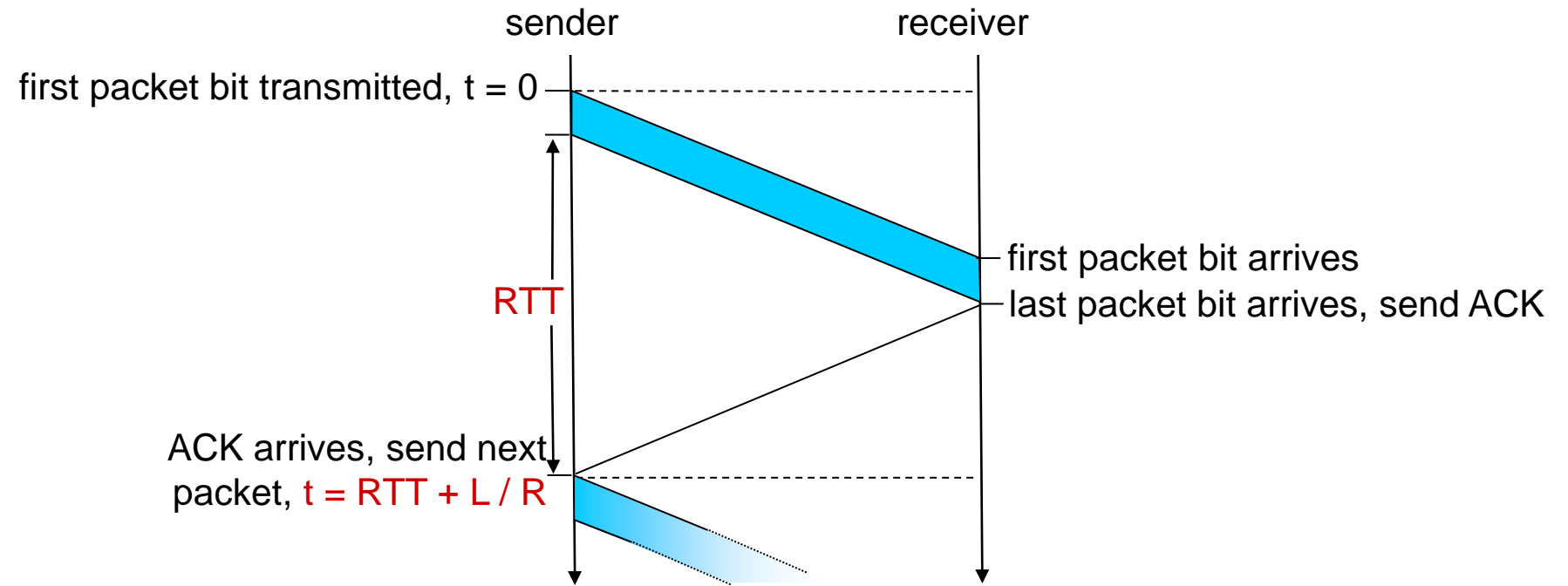# rdt3.0: Efficiency
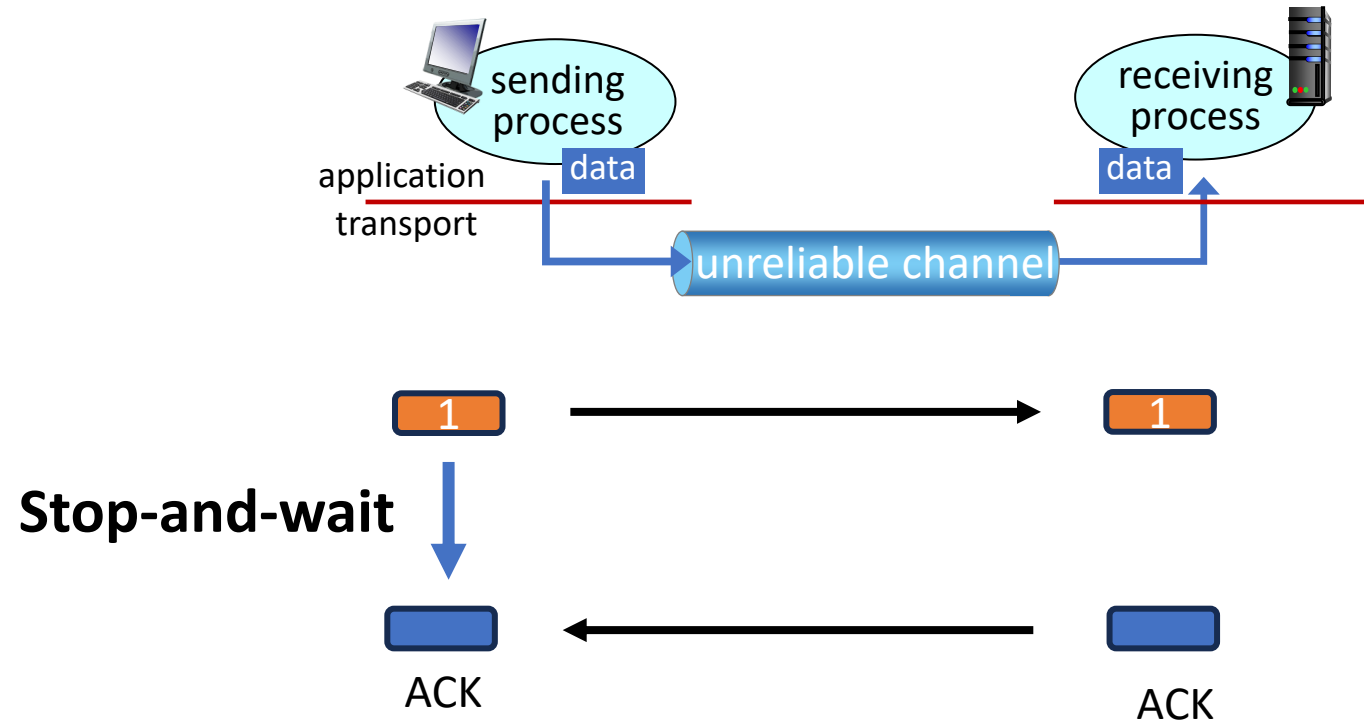
Stop-and-wait

rdt3.0: Efficiency

- *U* $_{sender}$: *utilization* – fraction of time sender busy sending

- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
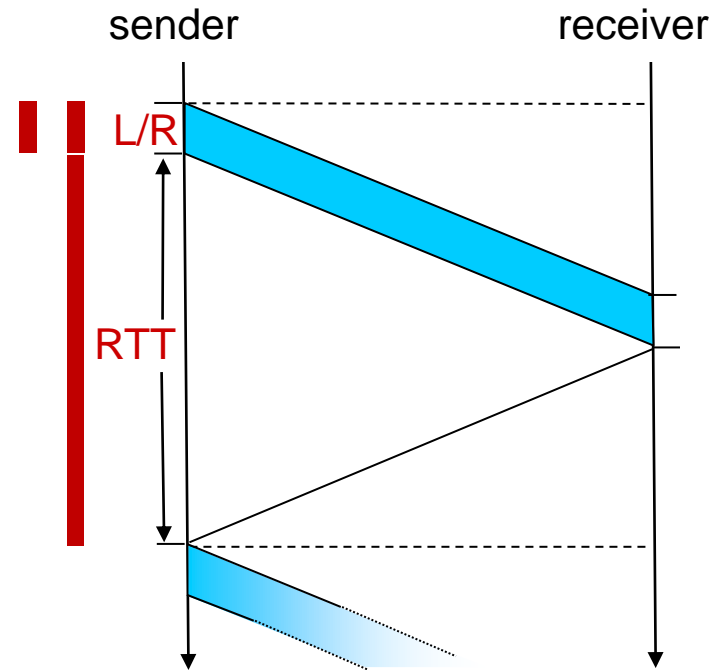
  - time to transmit packet into channel:

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

rdt3.0: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0

RTT

first packet bit arrives

last packet bit arrives, send ACK

ACK arrives, send next packet, t = RTT + L / R

rdt3.0: stop-and-wait operation

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

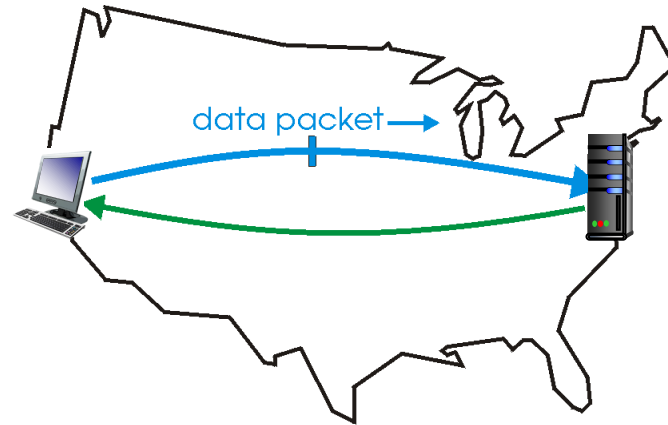$$= \frac{.008}{30.008}$$

$$= 0.00027$$

- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)
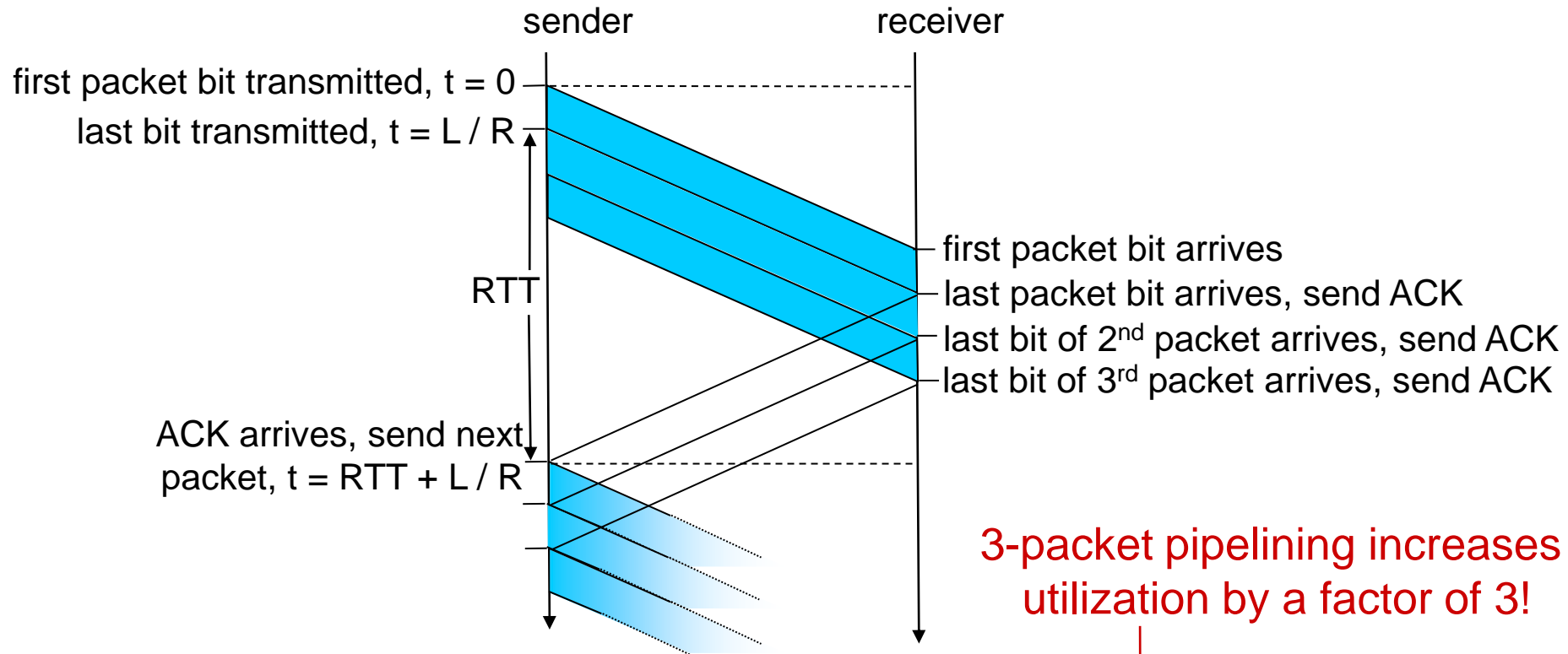
rdt3.0: pipelined protocols operation

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

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

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

Pipelining: increased utilization

sender   receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R

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

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