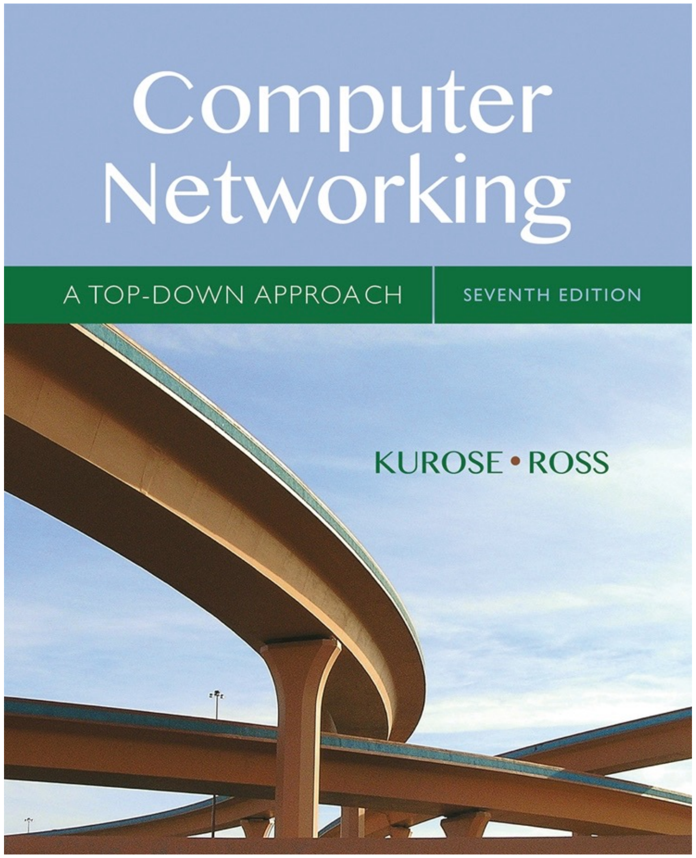
# Chapter 3
# Transport Layer




*Computer Networking: A Top Down Approach*

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

### A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

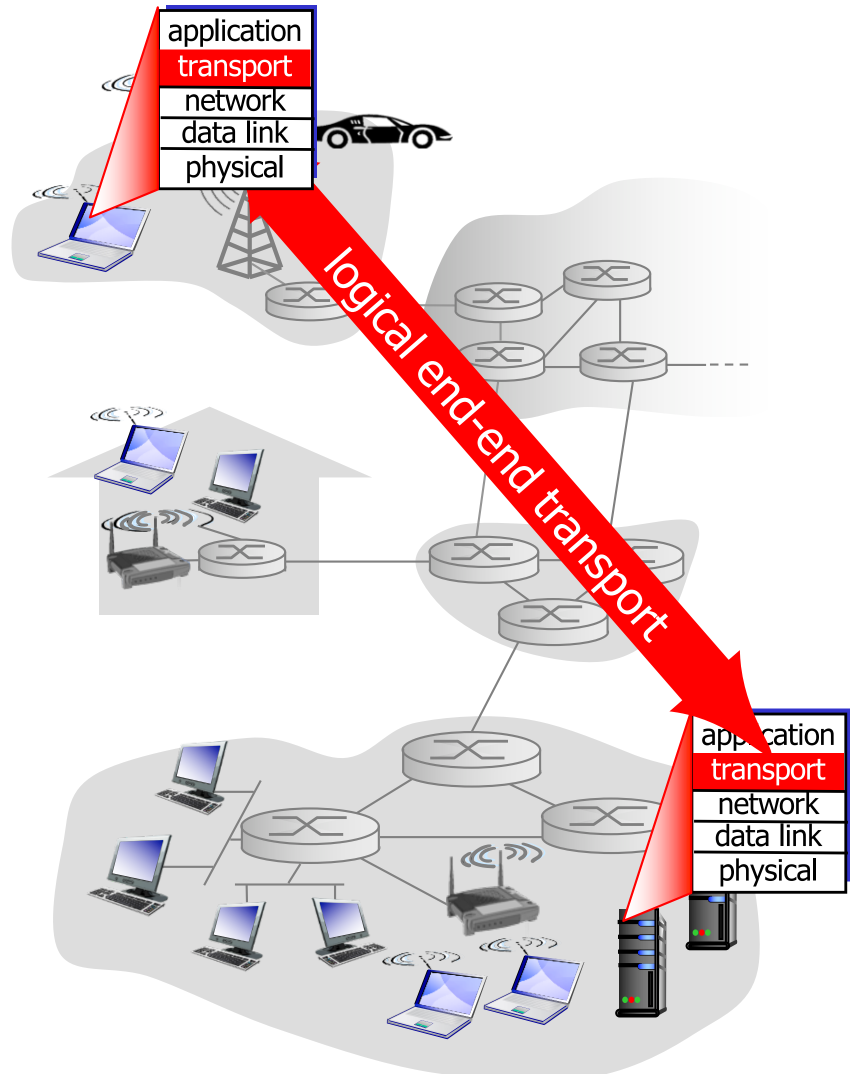
# Chapter 3: Transport Layer

## 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

# 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

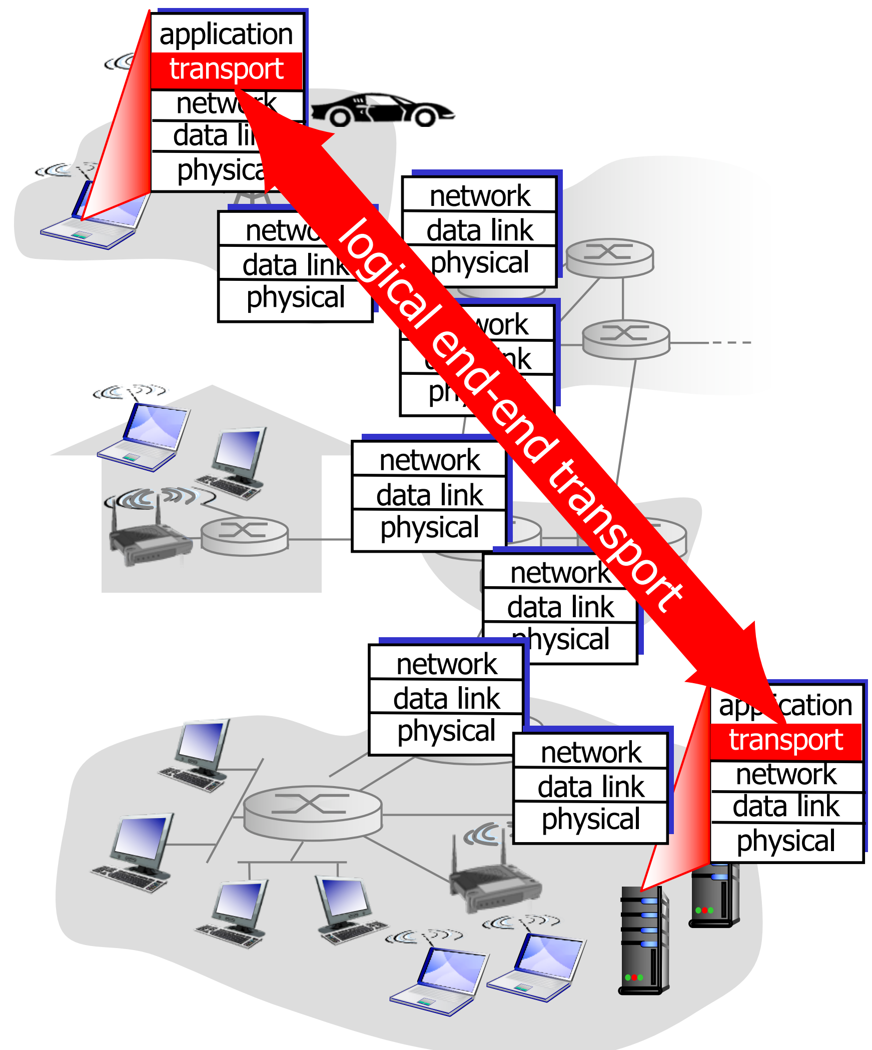
*household analogy:*

*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-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery (UDP)
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



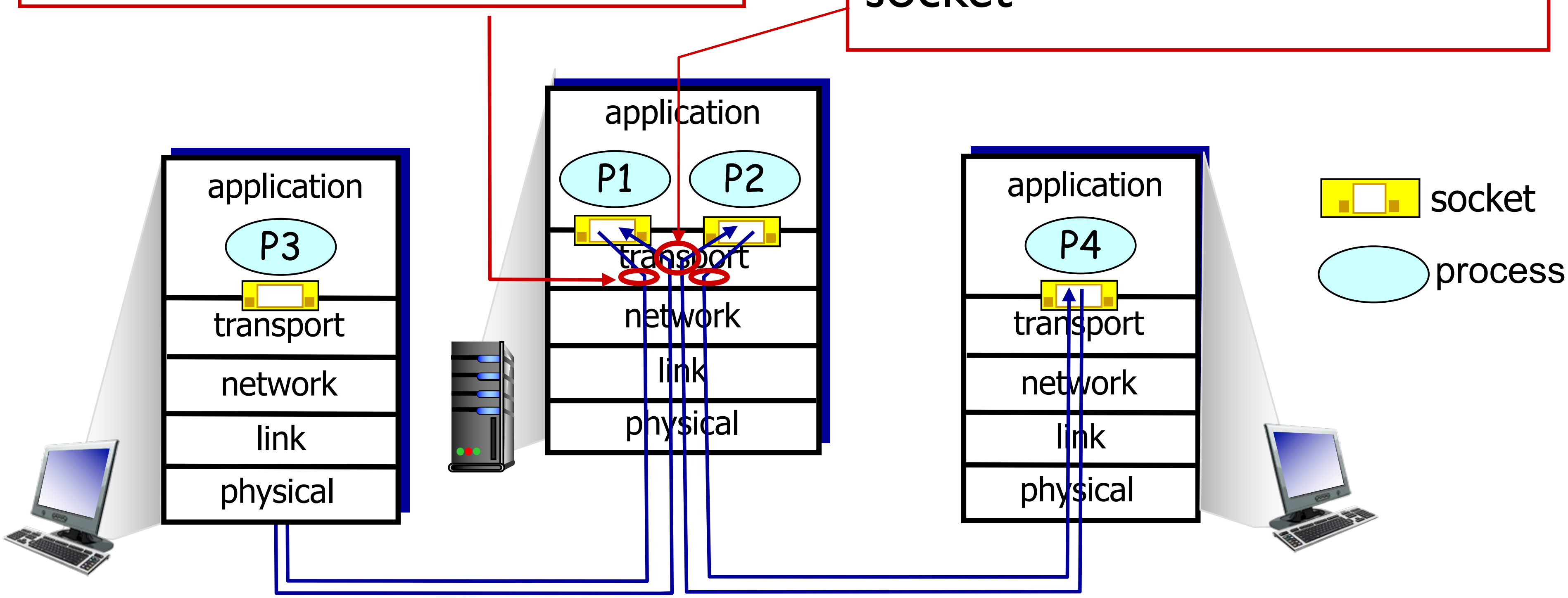
# Multiplexing/demultiplexing

*multiplexing at sender:*

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

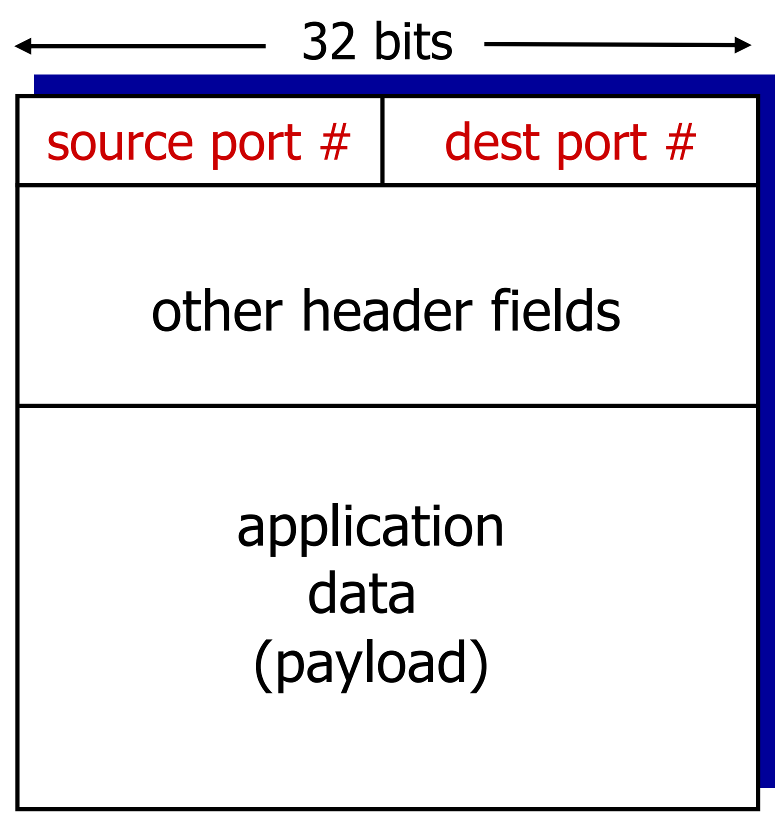
*demultiplexing at receiver:*

use header info to deliver received segments to correct socket



# How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket




TCP/UDP segment format

# Connectionless demultiplexing

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

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

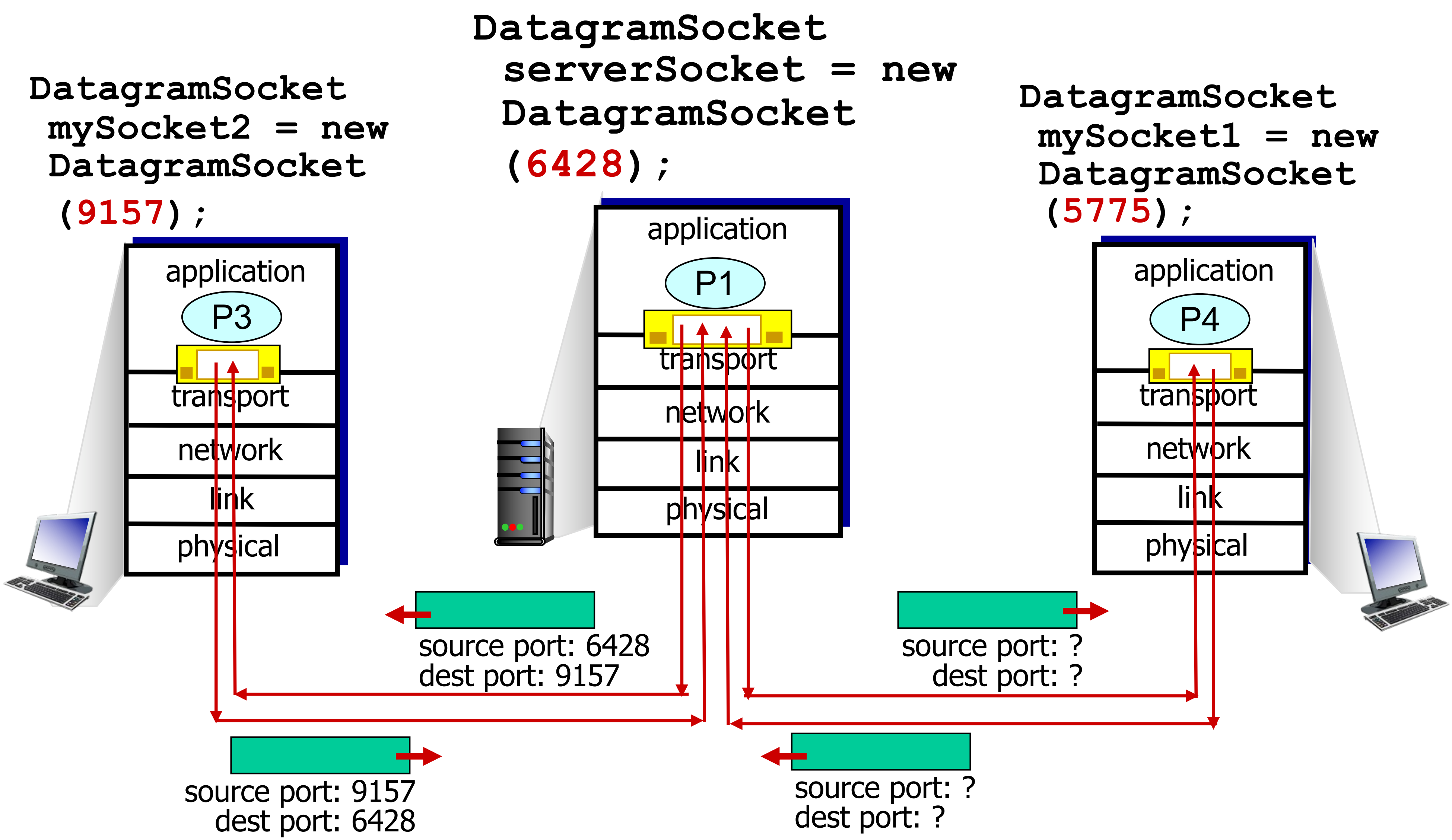
- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

---

- 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

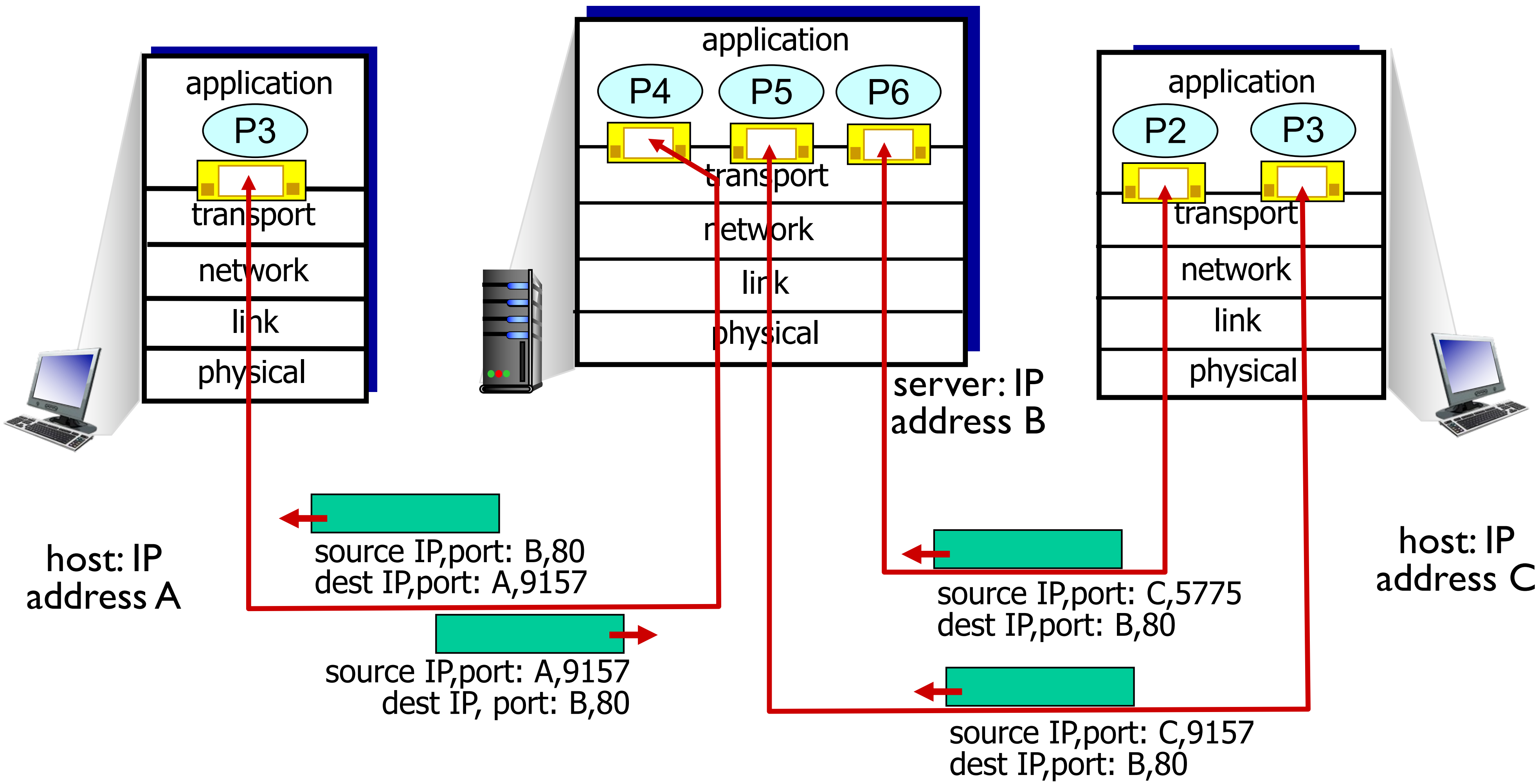
# Connectionless demux: example



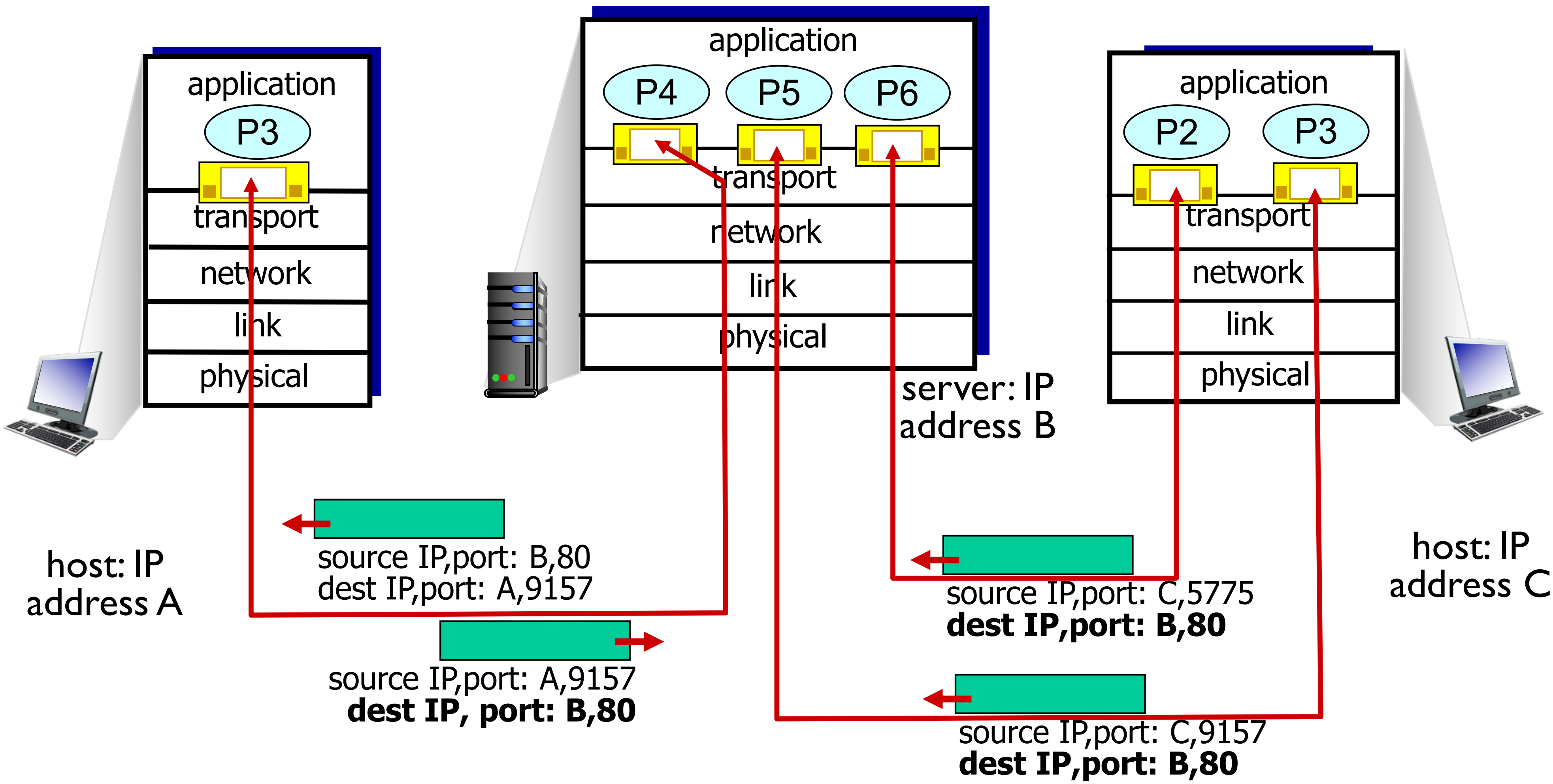
# Connection-oriented demux

- 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 demux: example

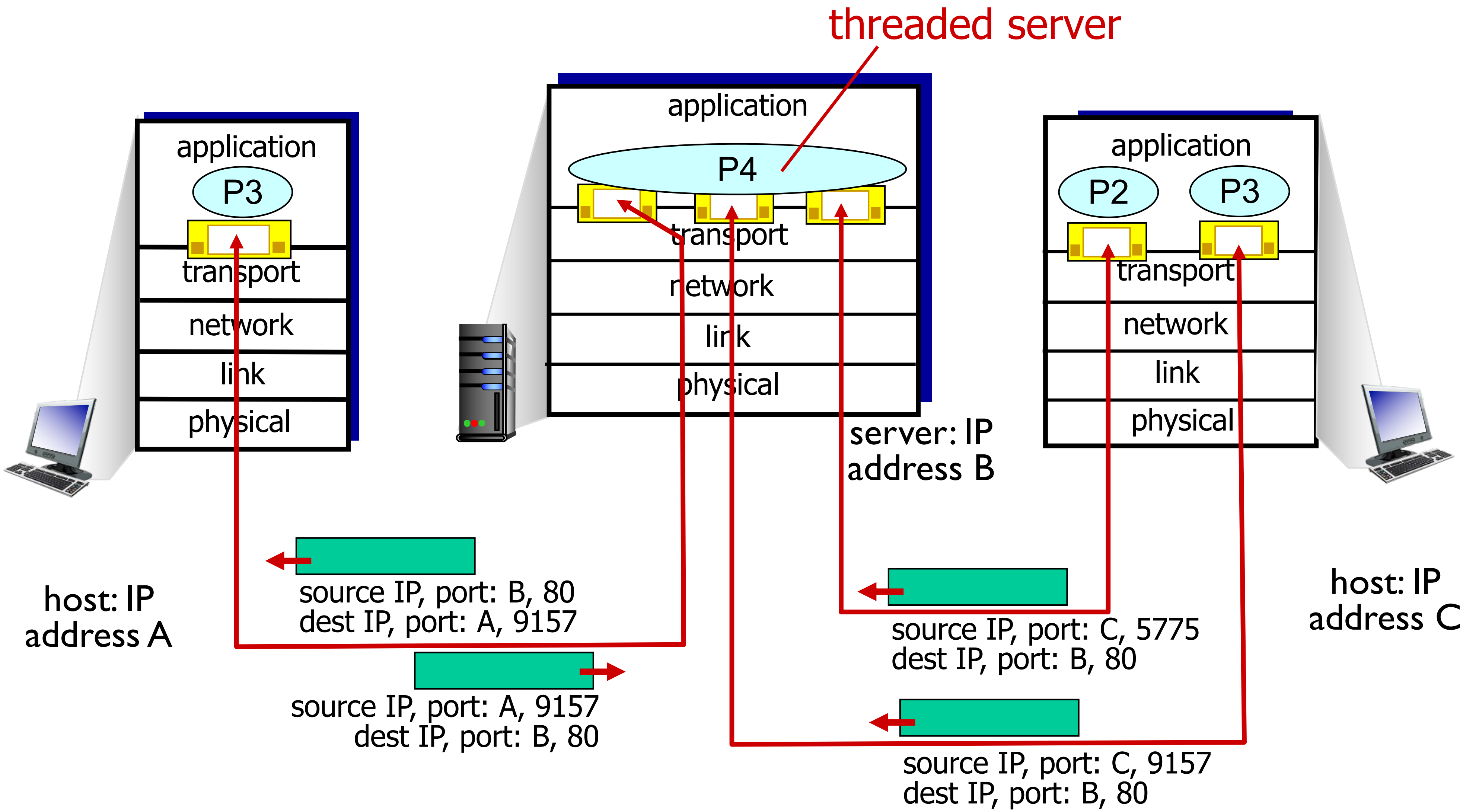


# Connection-oriented demux: example




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

# Connection-oriented demux: example

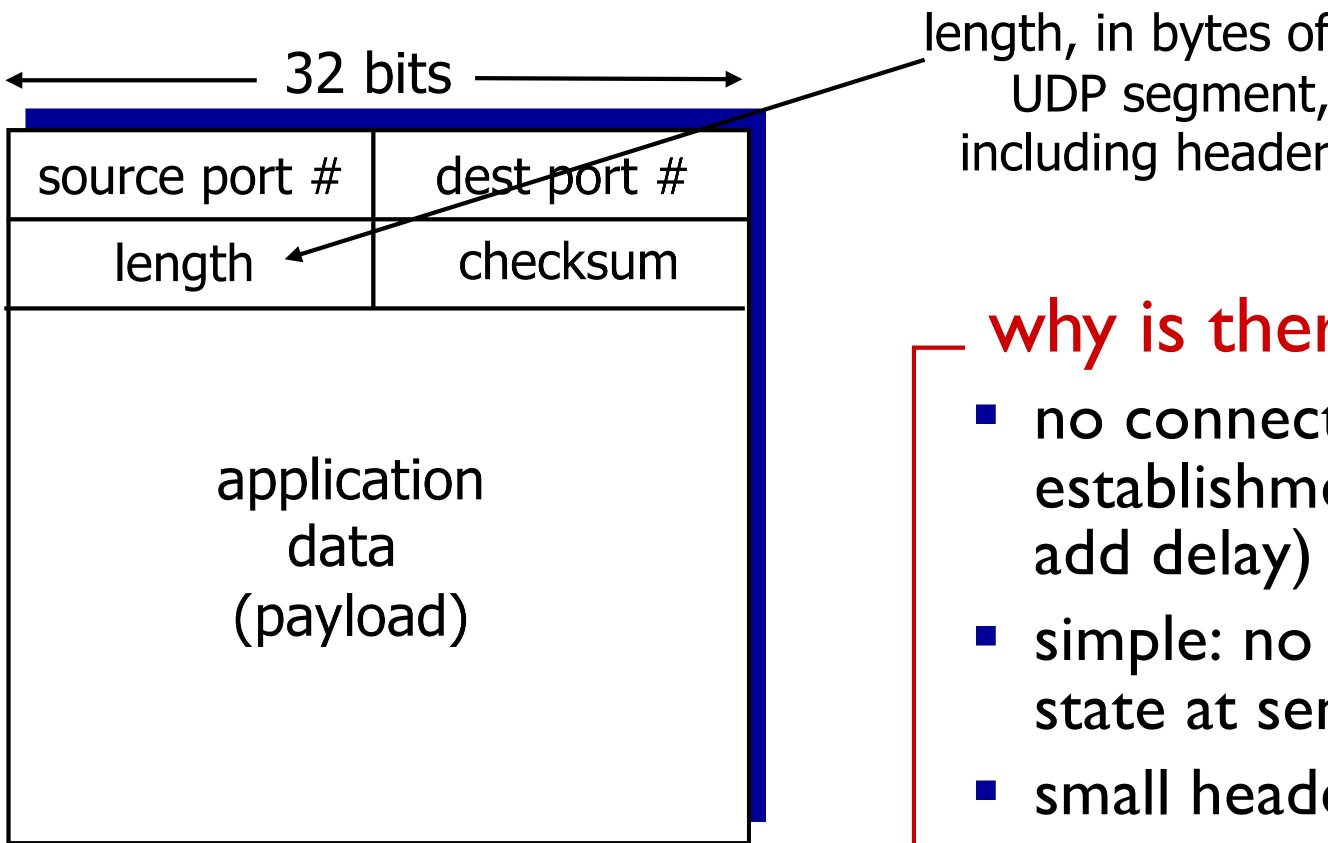


# UDP: User Datagram Protocol [RFC 768]

- "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

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- How to achieve reliable transfer over UDP?
  - add reliability at application layer
  - application-specific error recovery!

# UDP: segment header




UDP segment format

## 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

# UDP checksum

*Goal:* detect "errors" (e.g., flipped bits) 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:**

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

# Internet checksum: example

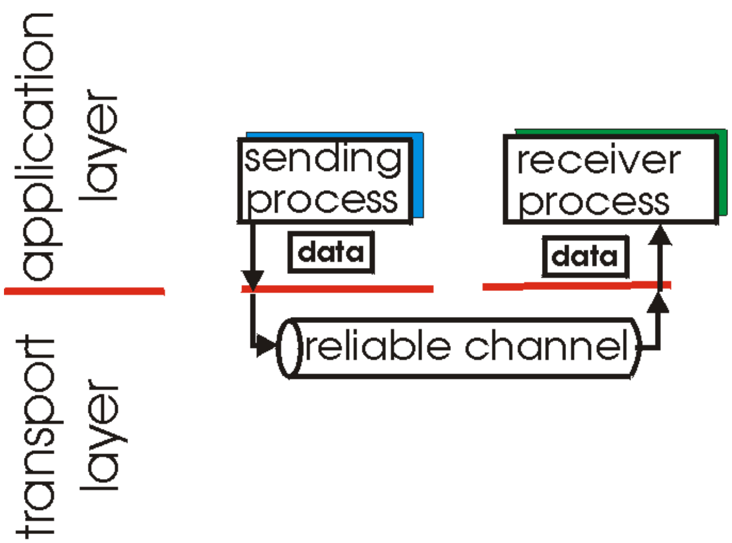
example: add two 16-bit integers

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 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 |
| 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 carryout from the most significant bit needs to be added to the result

# Principles of reliable data transfer

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

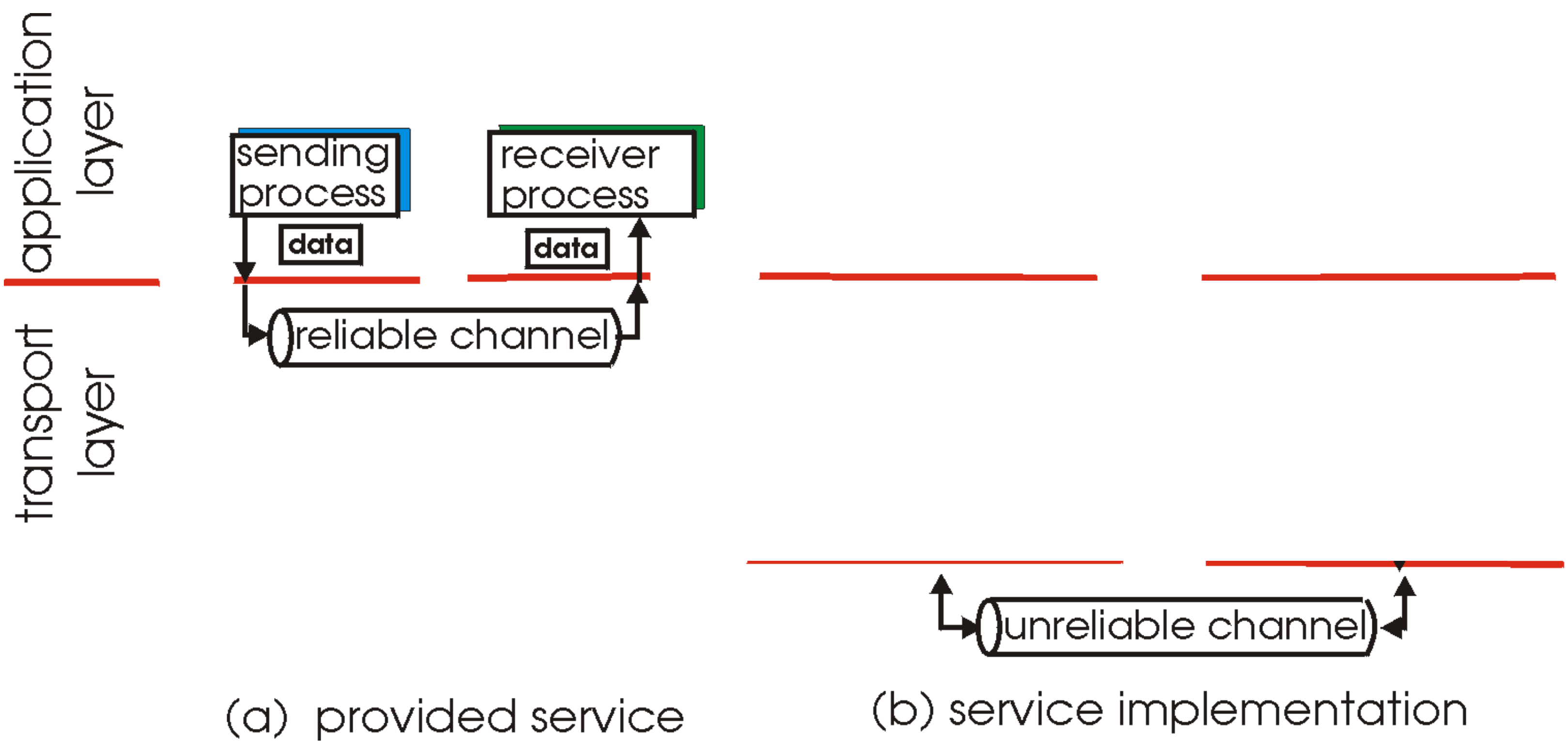



(a) provided service

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

# Principles of reliable data transfer

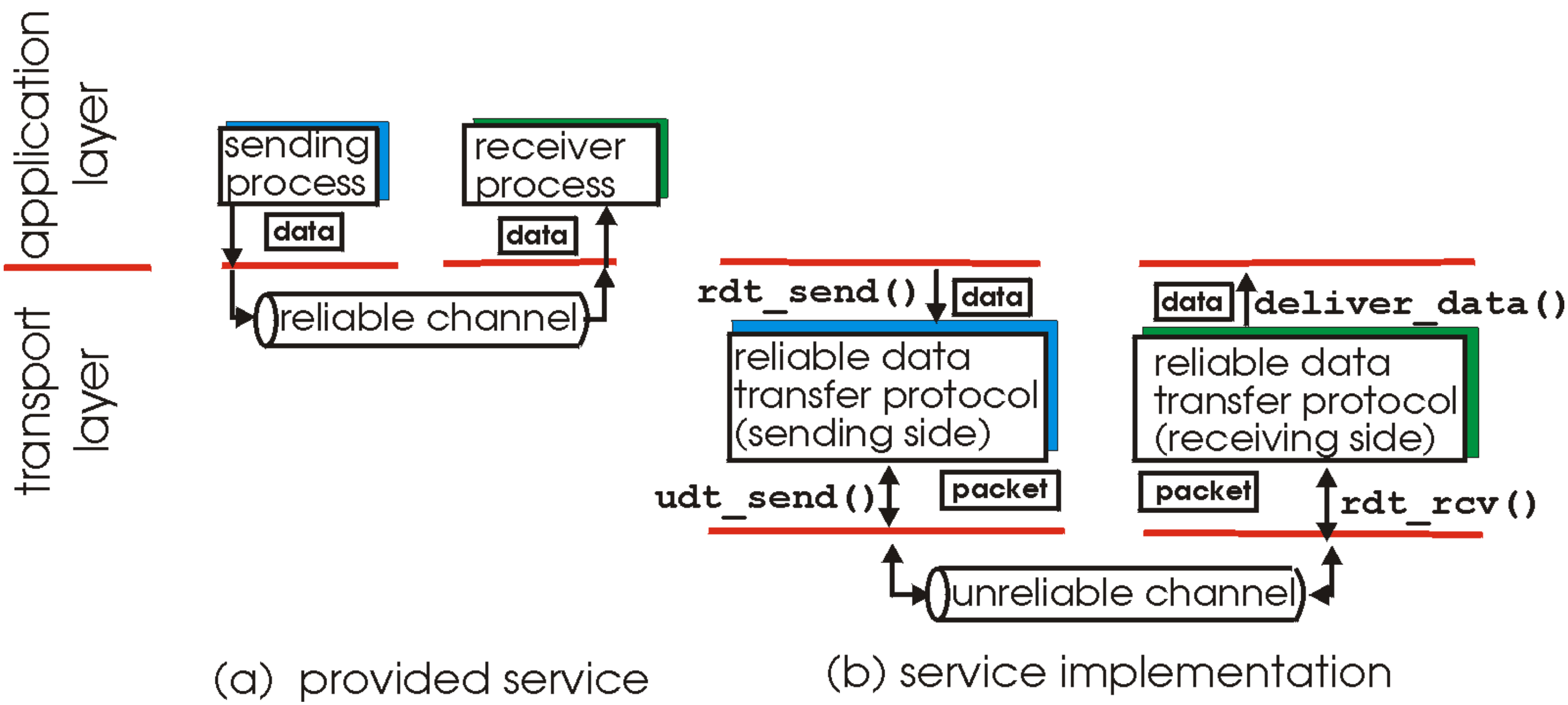
- important in application, transport, link layers
  - top-10 list of important networking topics!




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

# Principles of reliable data transfer

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

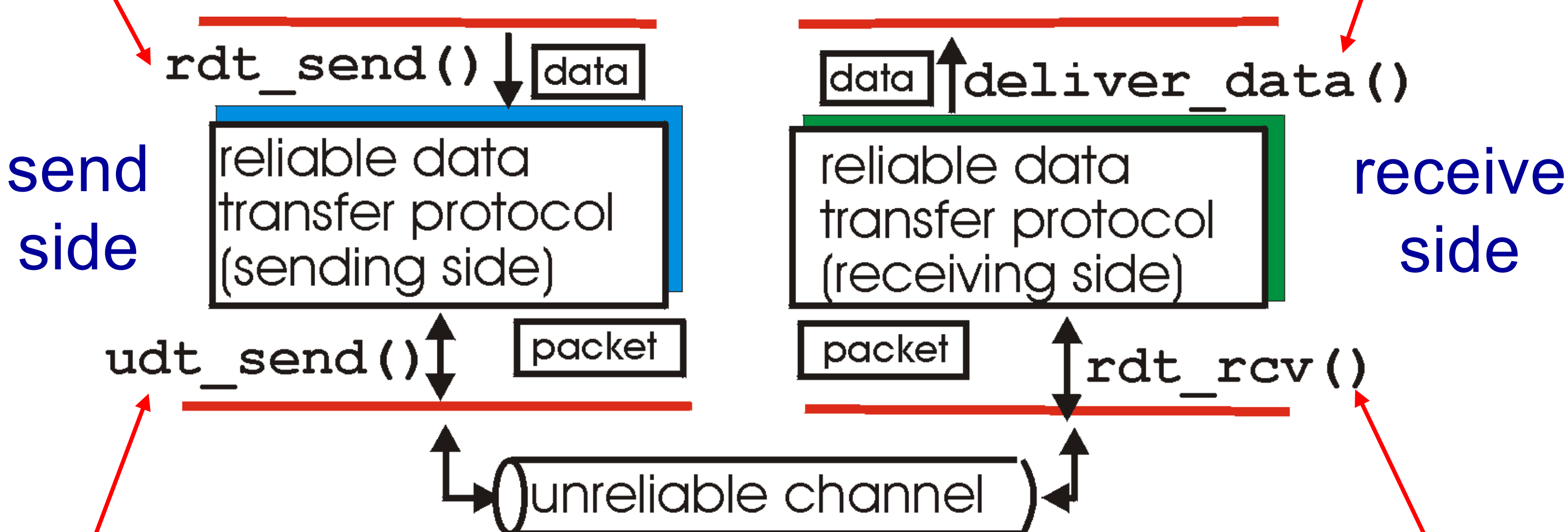



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Reliable data transfer: getting started

**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



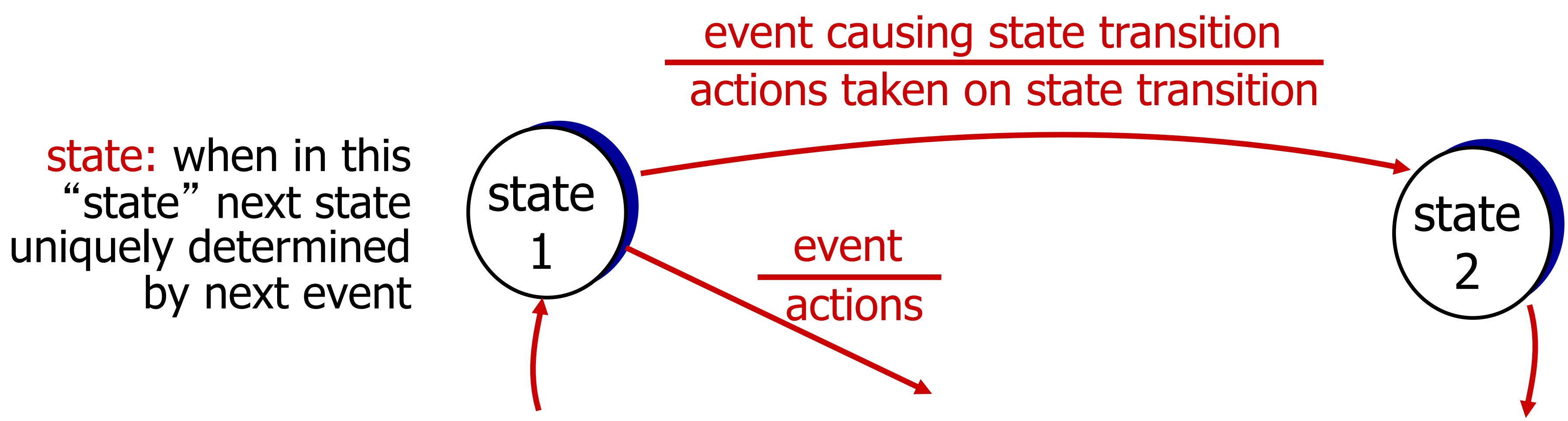

**udt_send():** called by rdt, to transfer packet over unreliable channel to receiver

**rdt_rcv():** called when packet arrives on rcv-side of channel

# Reliable data transfer: getting started

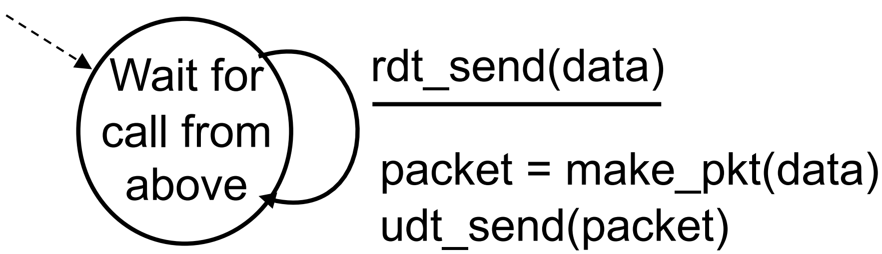
We will:

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

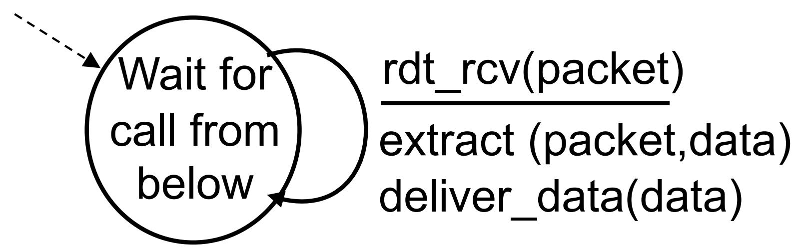


# rdt1.0: reliable transfer over a reliable channel

- 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




sender




receiver

# rdt2.0: channel with bit errors

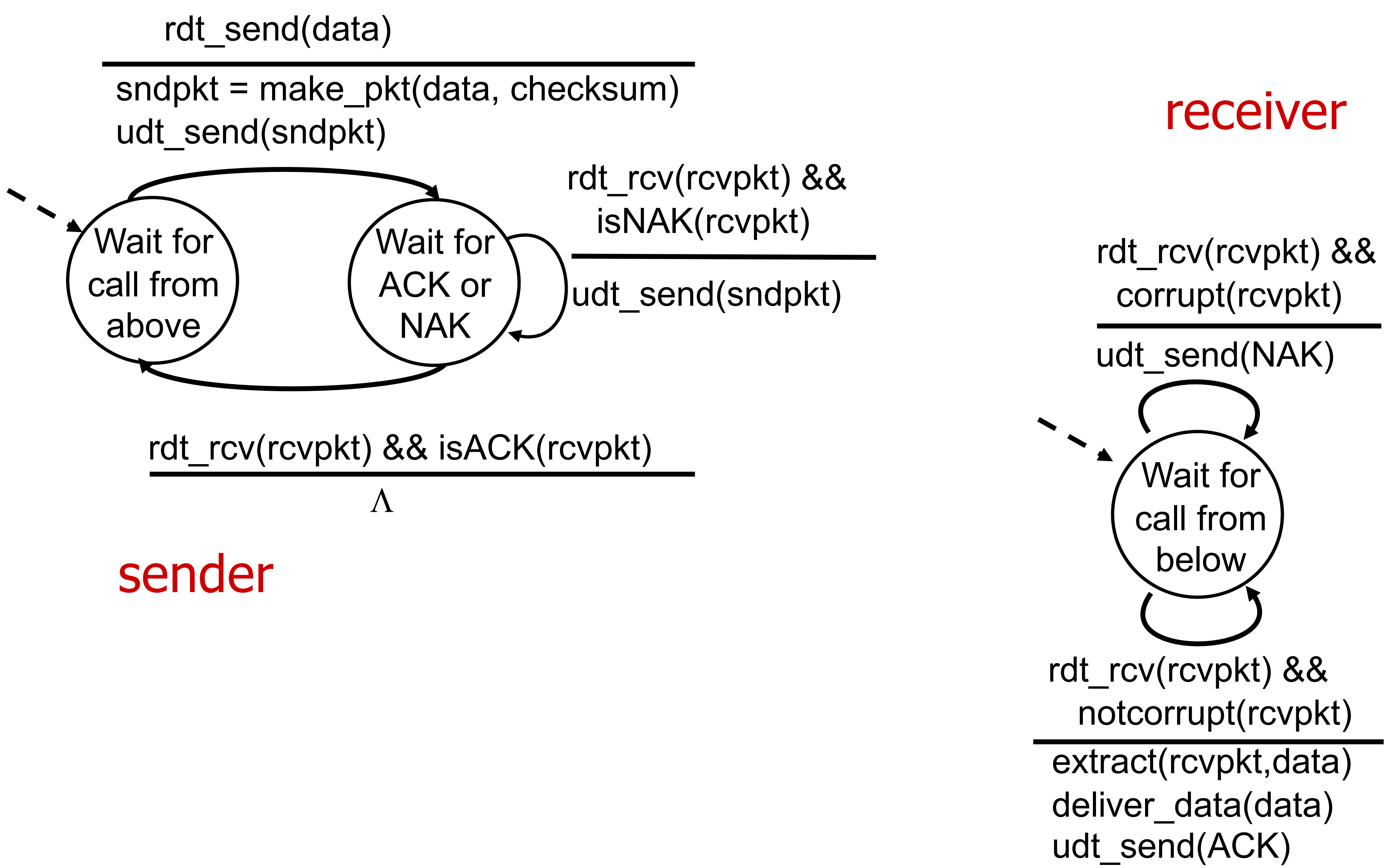
- underlying channel may flip bits in packet
  - **checksum** to detect bit errors
- *the* question: how to recover from errors:

*How do humans recover from "errors" during conversation?*

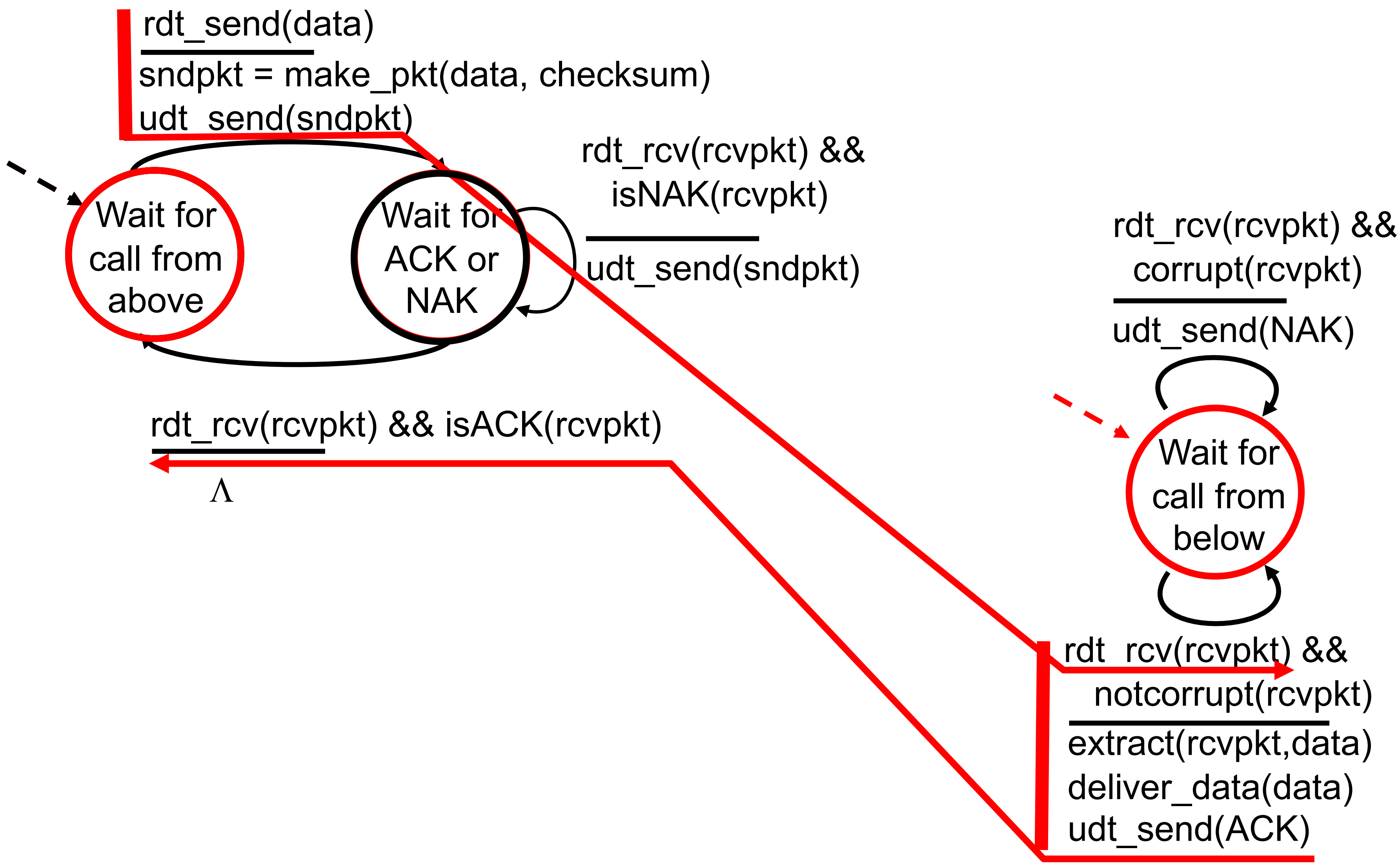
# rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - **checksum** to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - feedback: control msgs (ACK, NAK) from receiver to sender

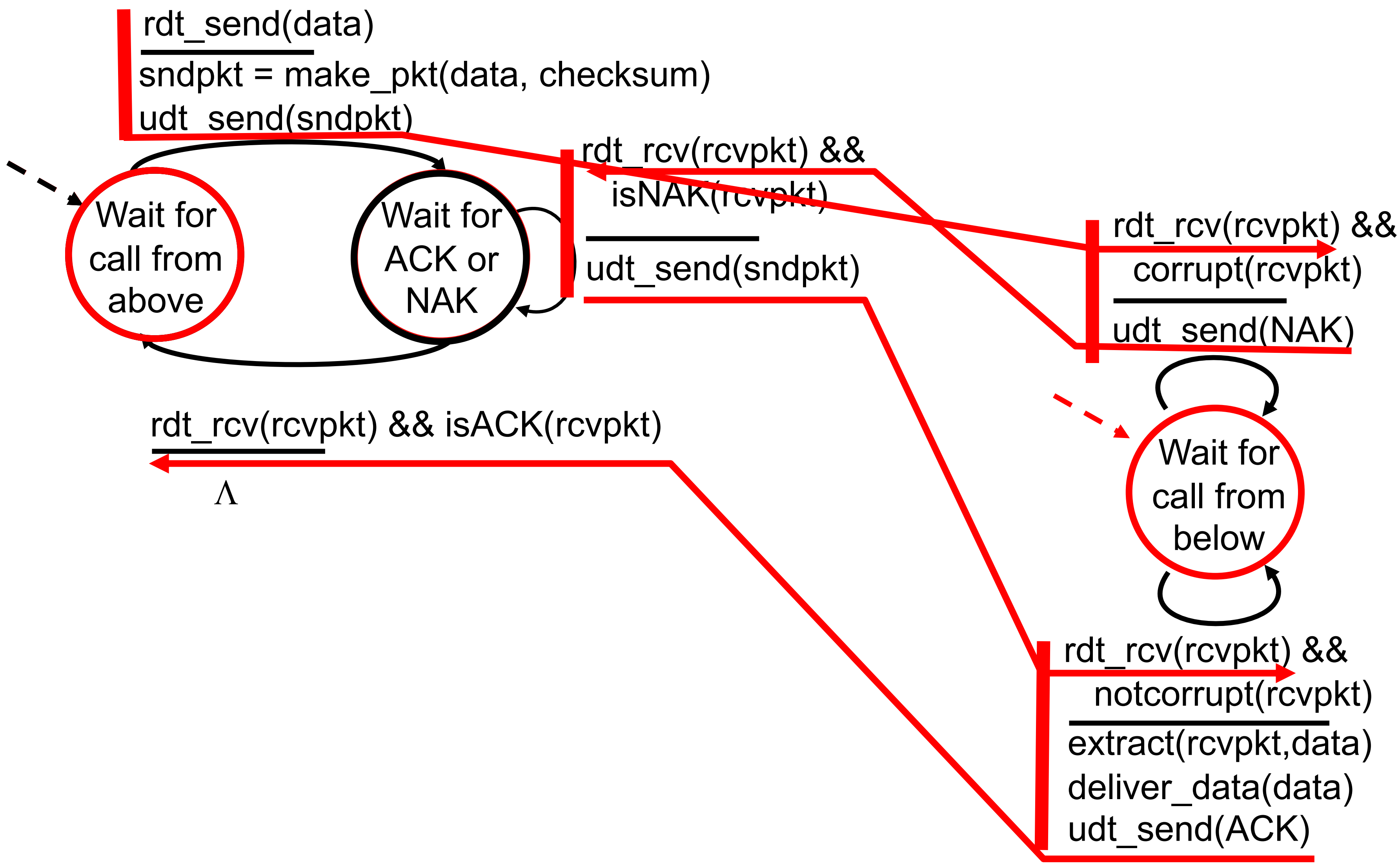
# rdt2.0: FSM specification



# 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

stop and wait

sender sends one packet, then waits for receiver response

# rdt2.1: handles garbled ACK/NAKs

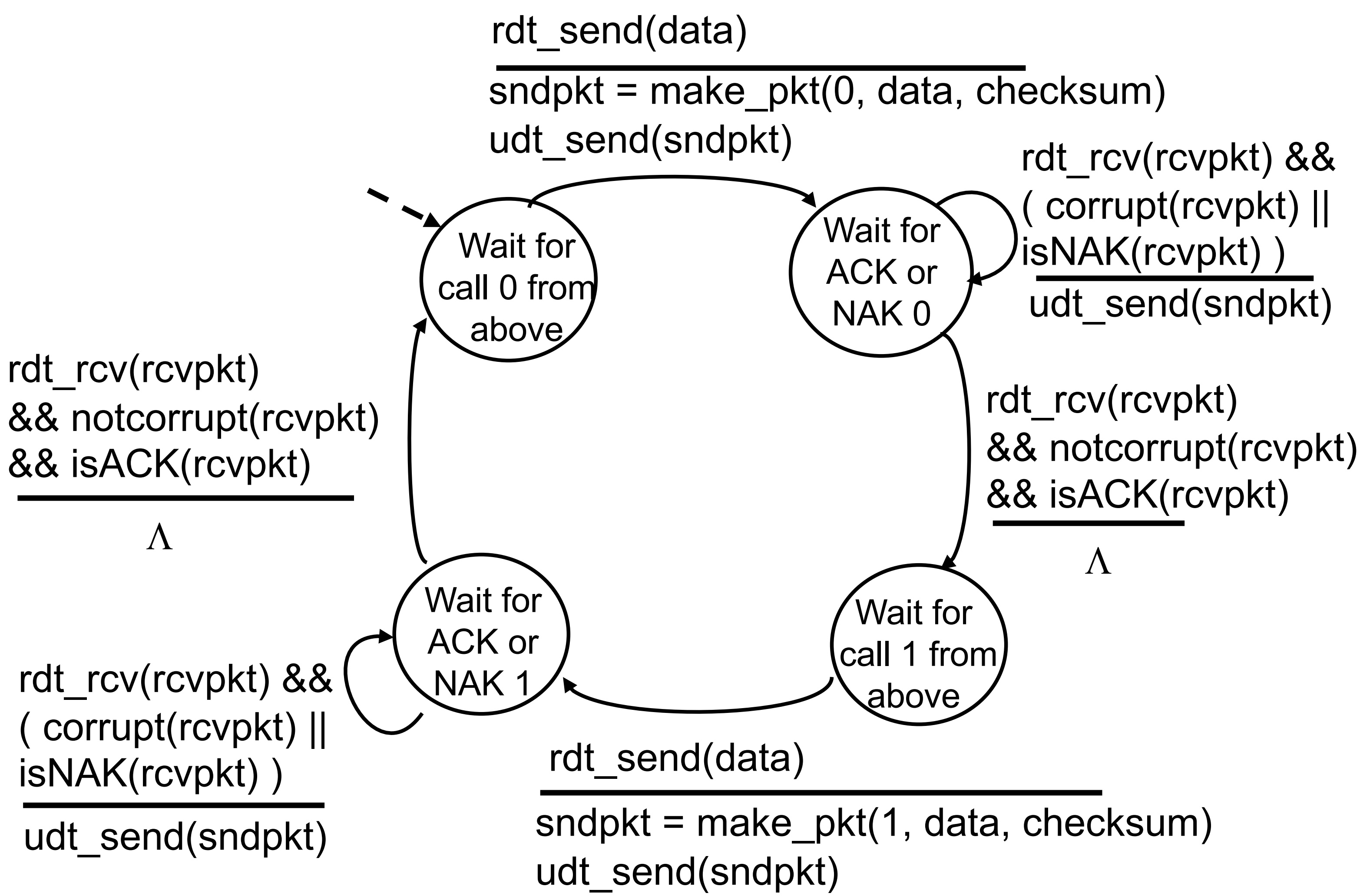
<u>sender:</u>

- 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

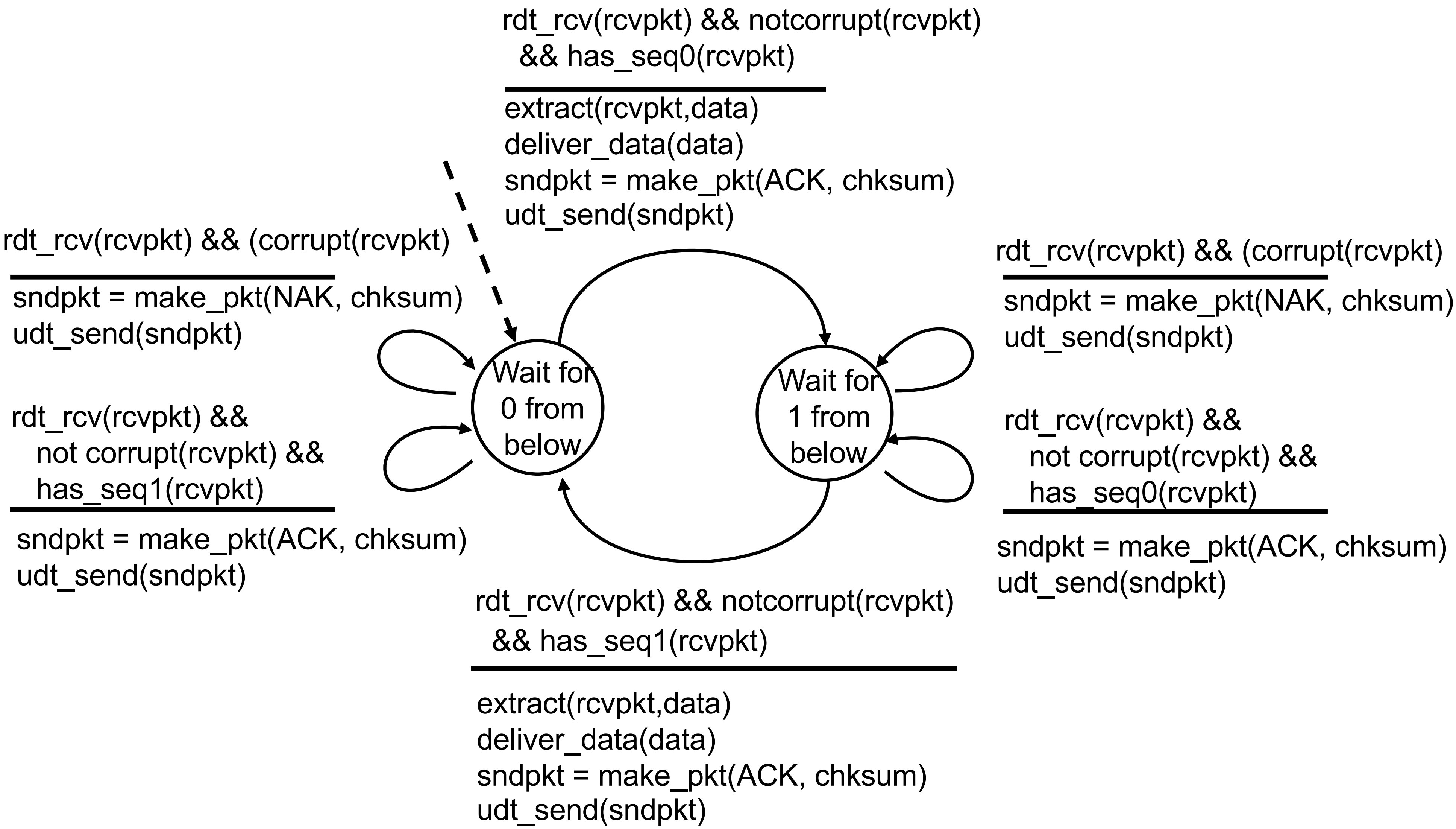
<u>receiver:</u>

- 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, handles garbled ACK/NAKs



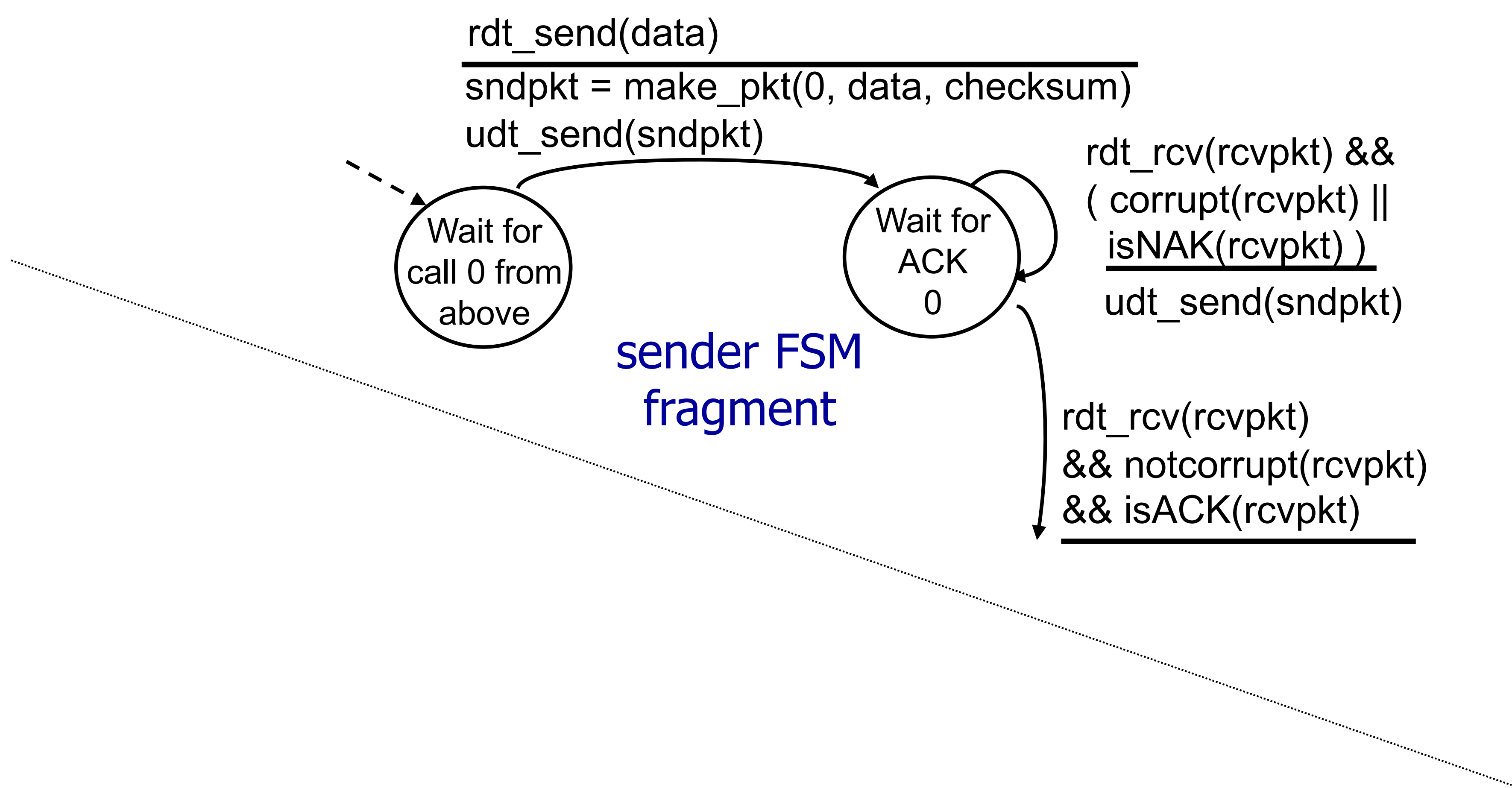
# rdt2.1: receiver, handles garbled ACK/NAKs



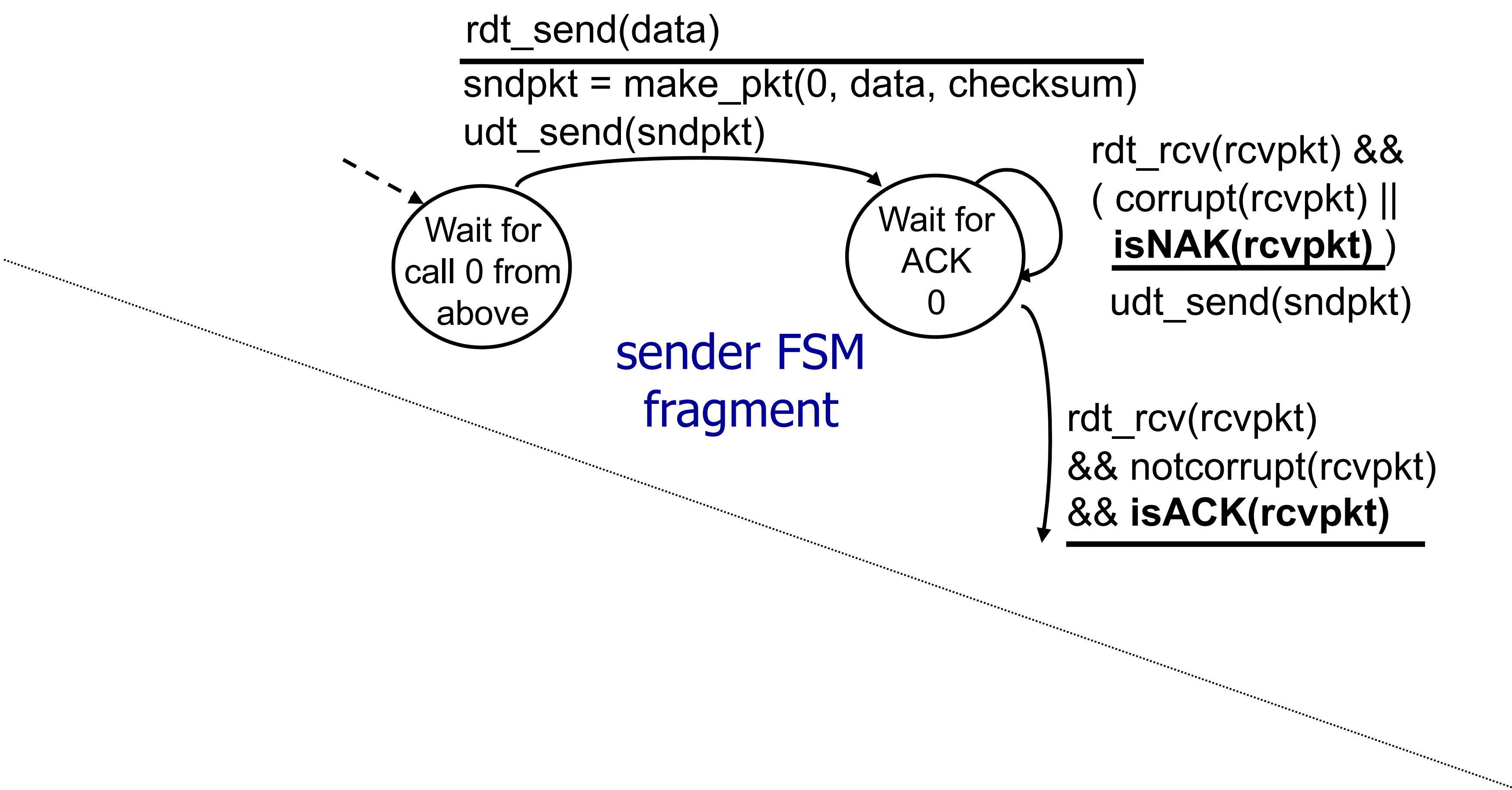
# rdt2.2: a NAK-free protocol

- 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*

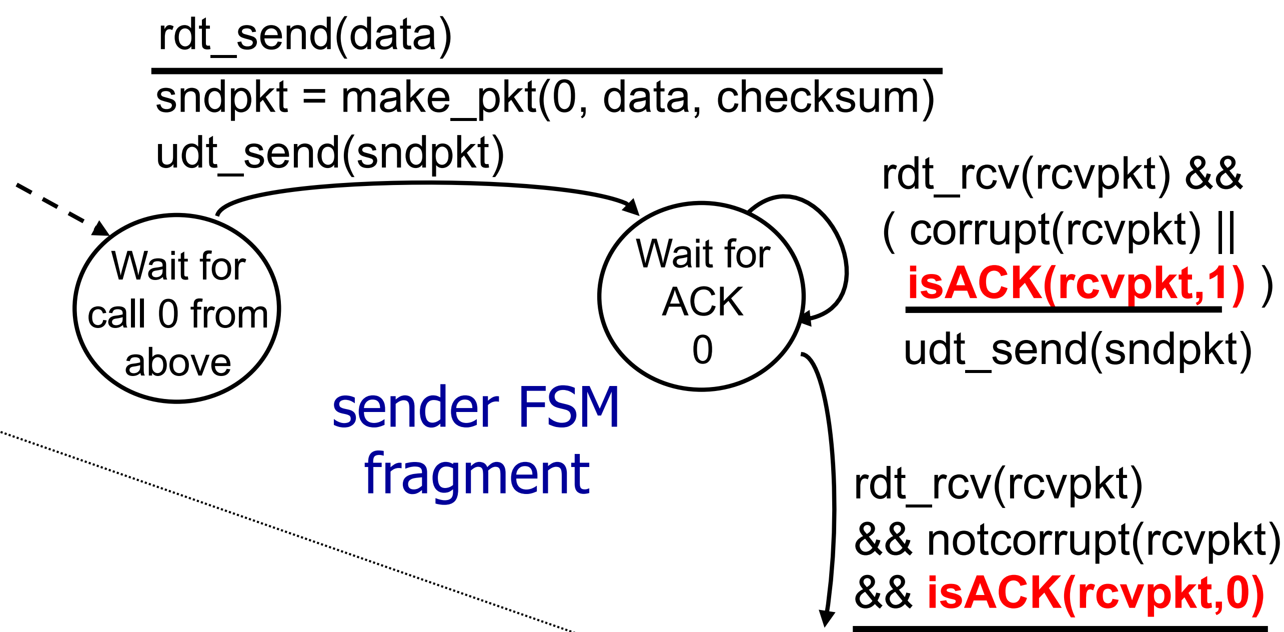
# rdt2.2: sender, receiver fragments



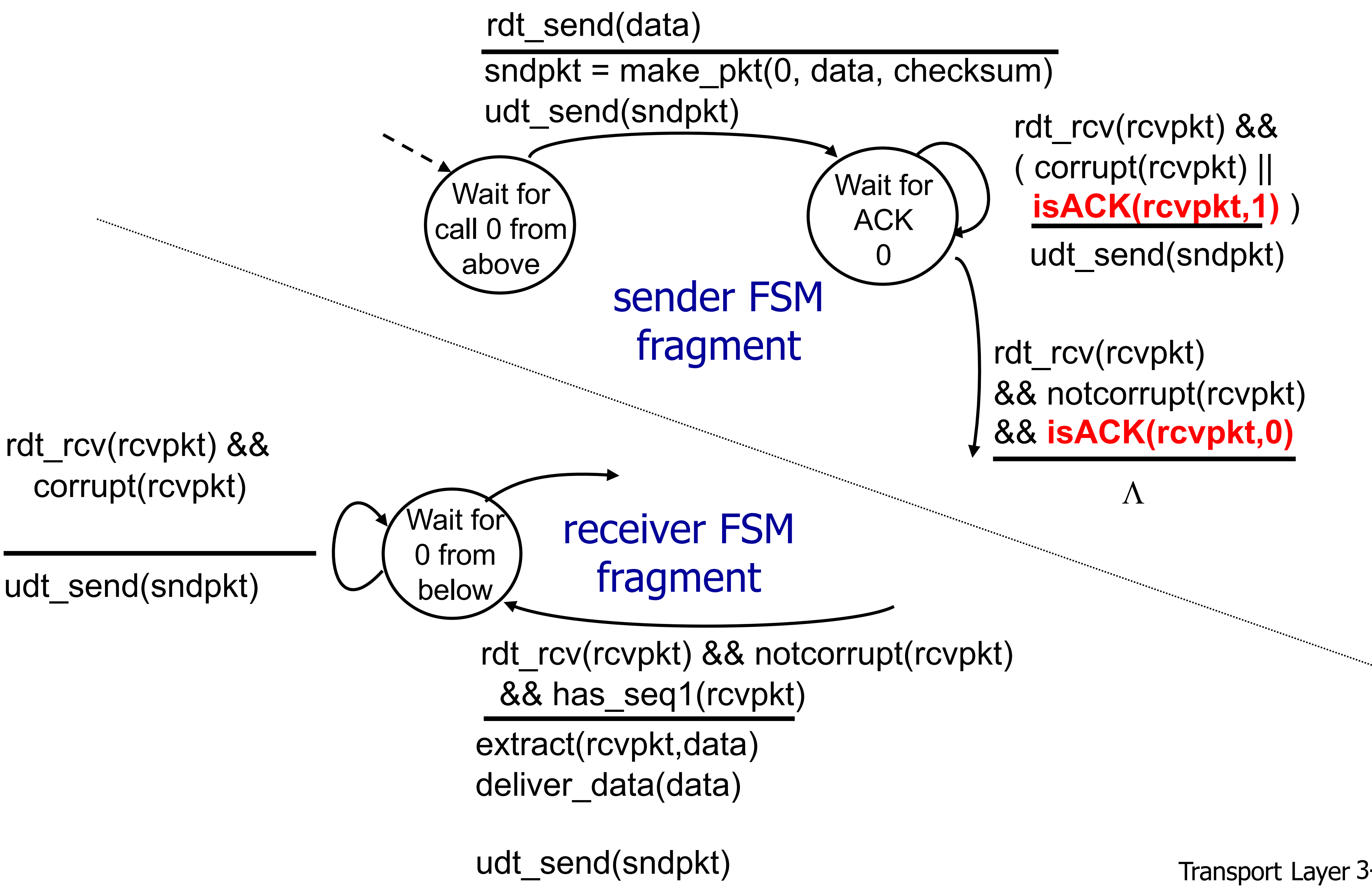
# rdt2.2: sender, receiver fragments



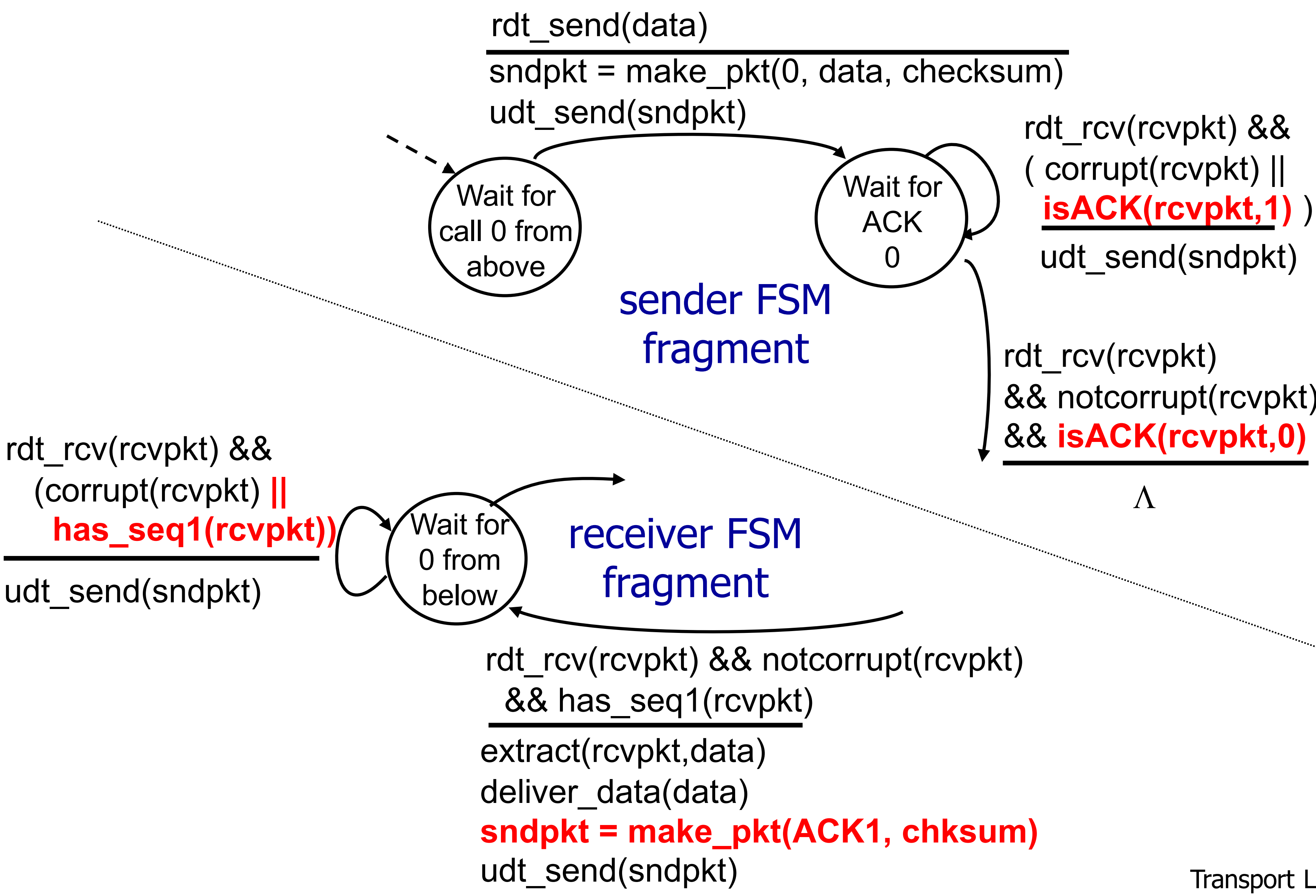
# rdt2.2: sender, receiver fragments



# rdt2.2: sender, receiver fragments

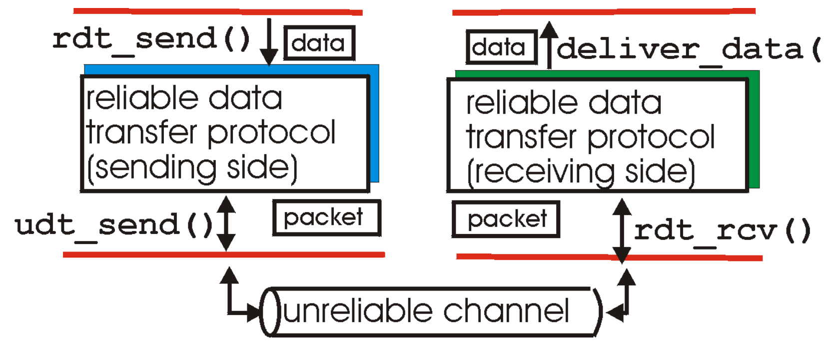


# rdt2.2: sender, receiver fragments



# Reliable data transfer - review

- V1.0, no bit errors, no loss of packets
- V2.0, channel with bit errors
  - checksum
  - feedback: control msgs (ACK,NAK)
- V2.1, garbled ACK/NAKs
  - two sequence # (0,1) added to pkt
  - sender must check if received ACK/NAK corrupted
  - receiver must check if received packet is duplicate
- V2.2, NAK-free protocol
  - receiver sends ACK for last pkt received OK
  - duplicate ACK at sender results in same action as NAK: *retransmit current pkt*



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

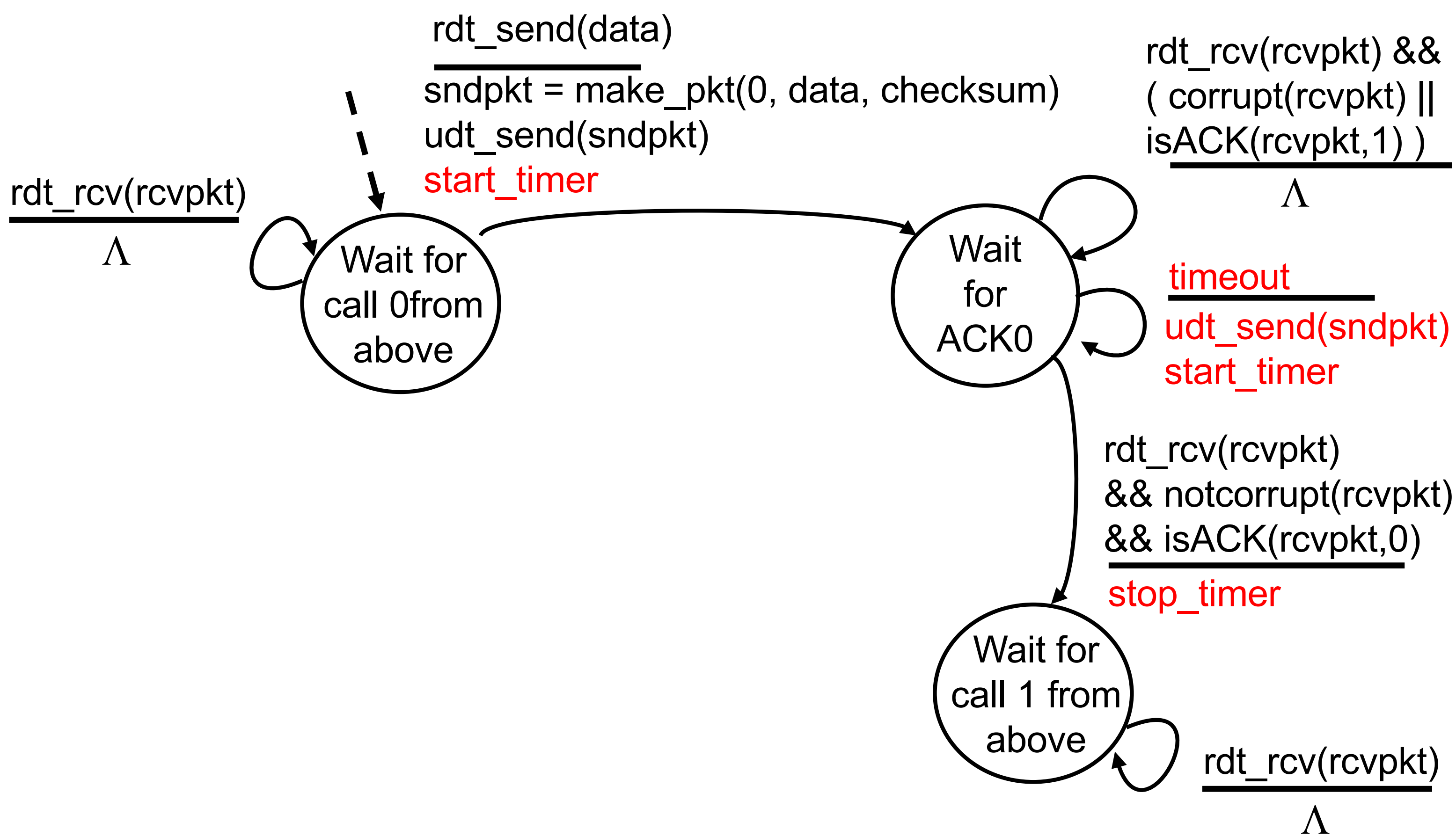
new assumption: underlying channel can also lose packets (data, ACKs)

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

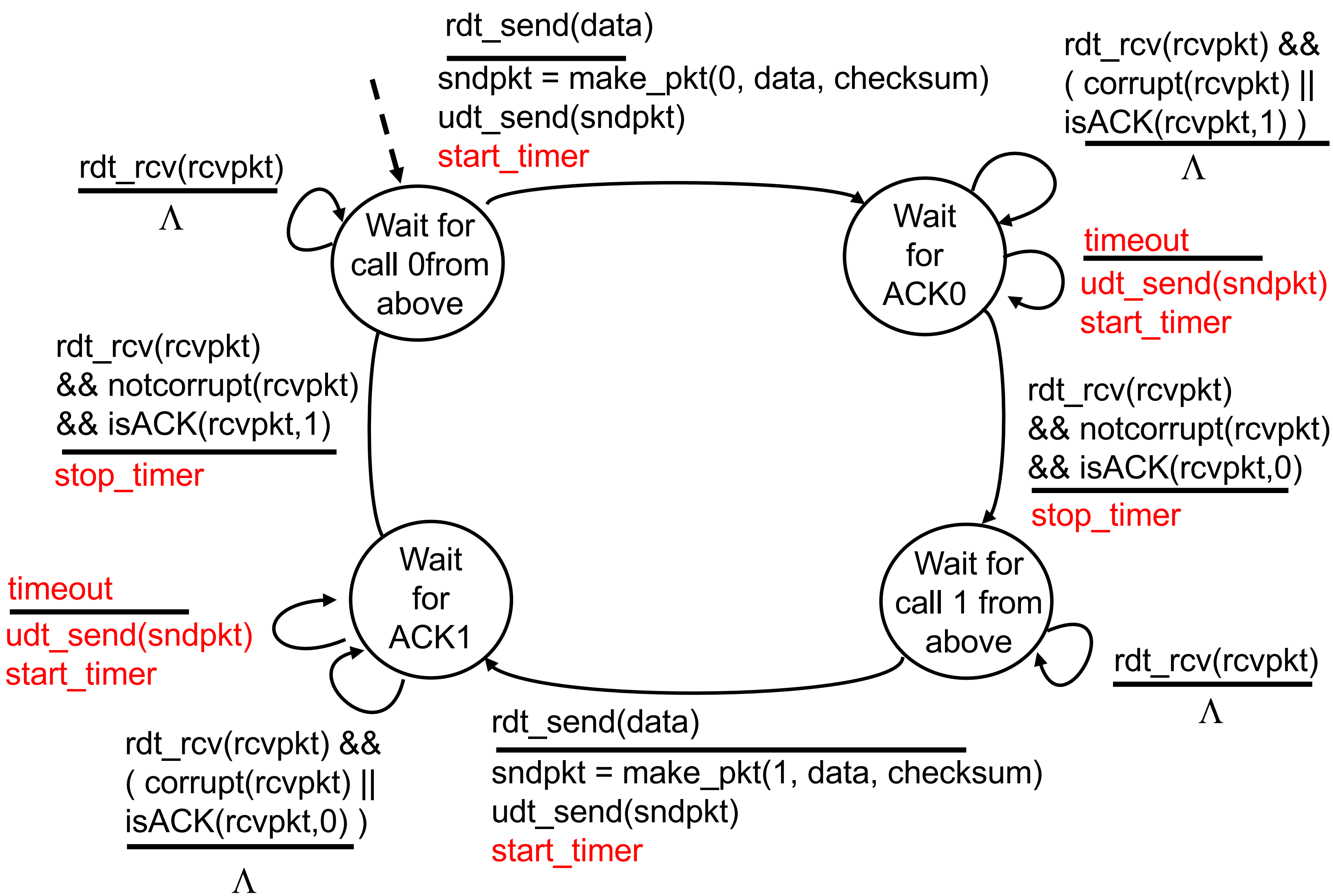
approach: sender waits "reasonable" amount of time for ACK

- 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

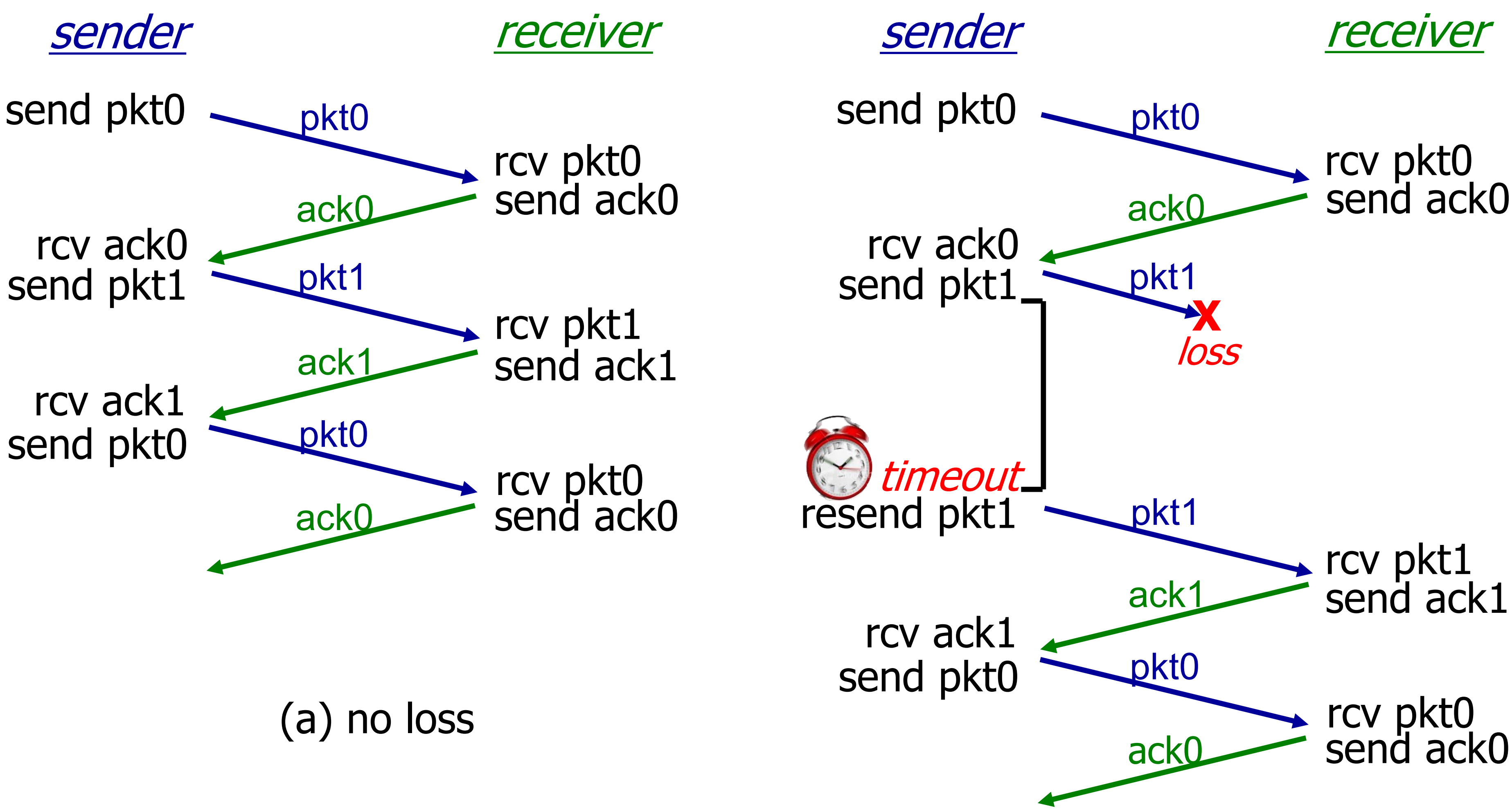
# rdt3.0 sender



# rdt3.0 sender



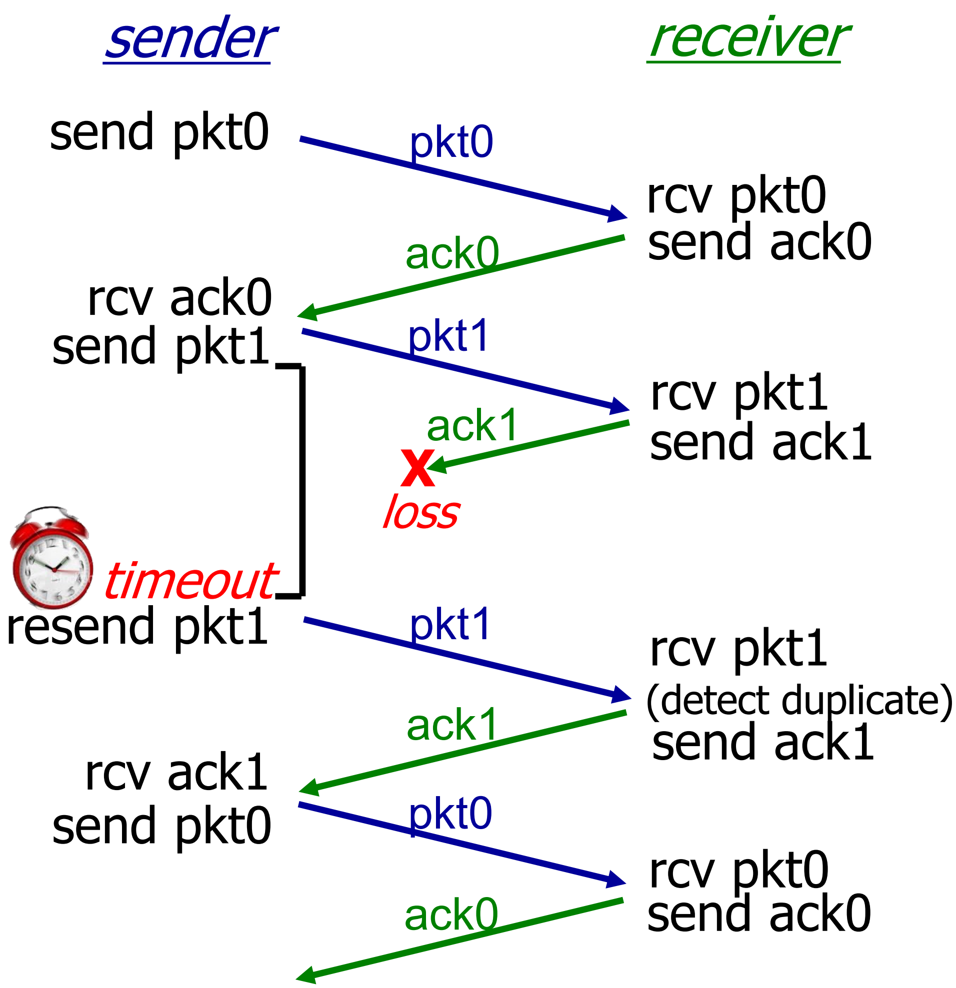
# rdt3.0 in action



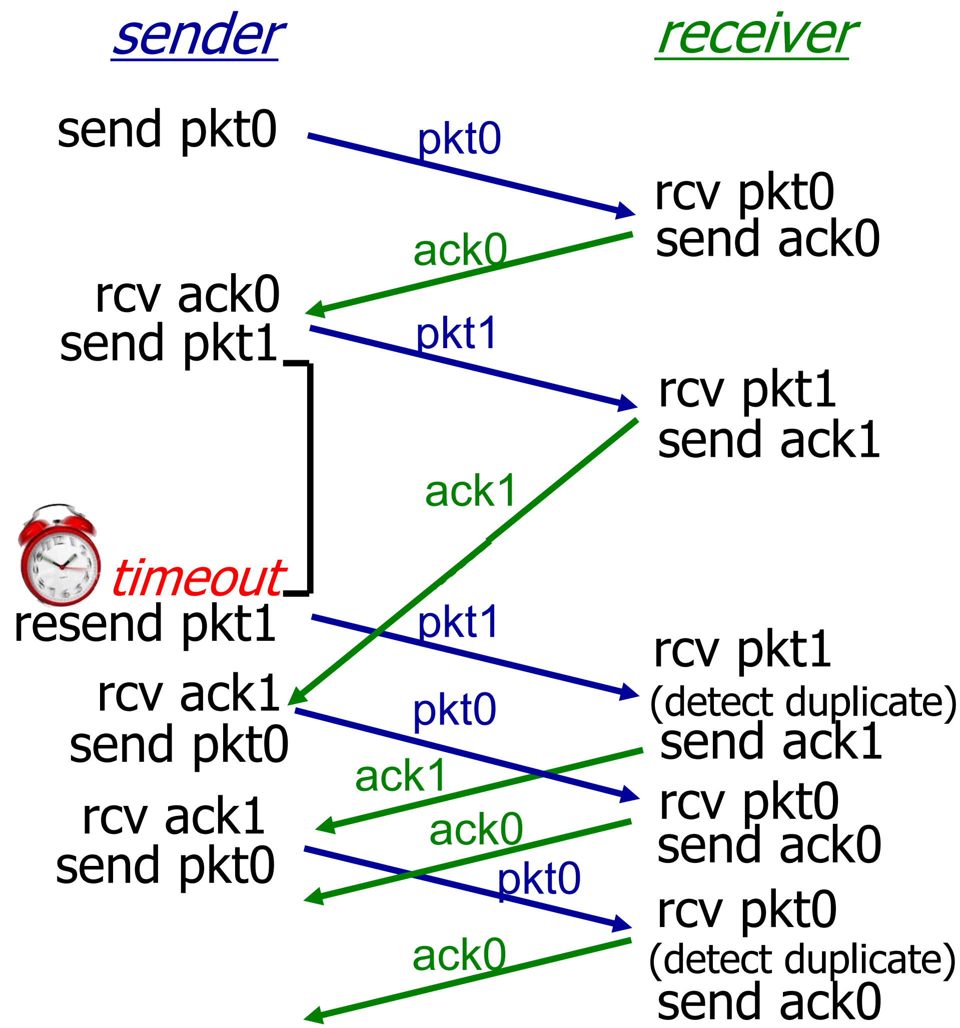

(a) no loss

(b) packet loss

# rdt3.0 in action




(c) ACK loss

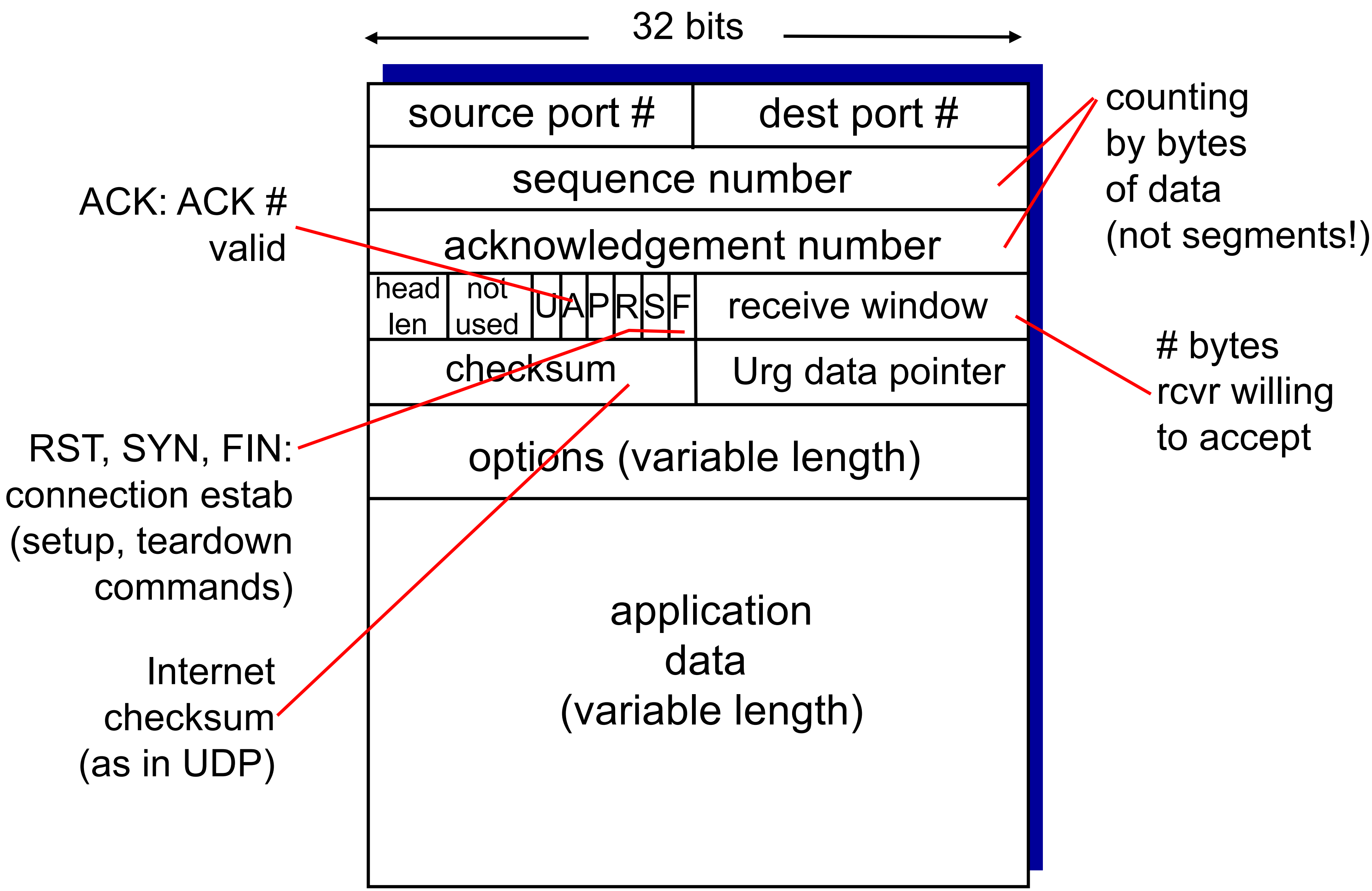



(d) premature timeout/ delayed ACK

# TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

# TCP segment structure



# TCP seq. numbers, ACKs

**sequence numbers:**

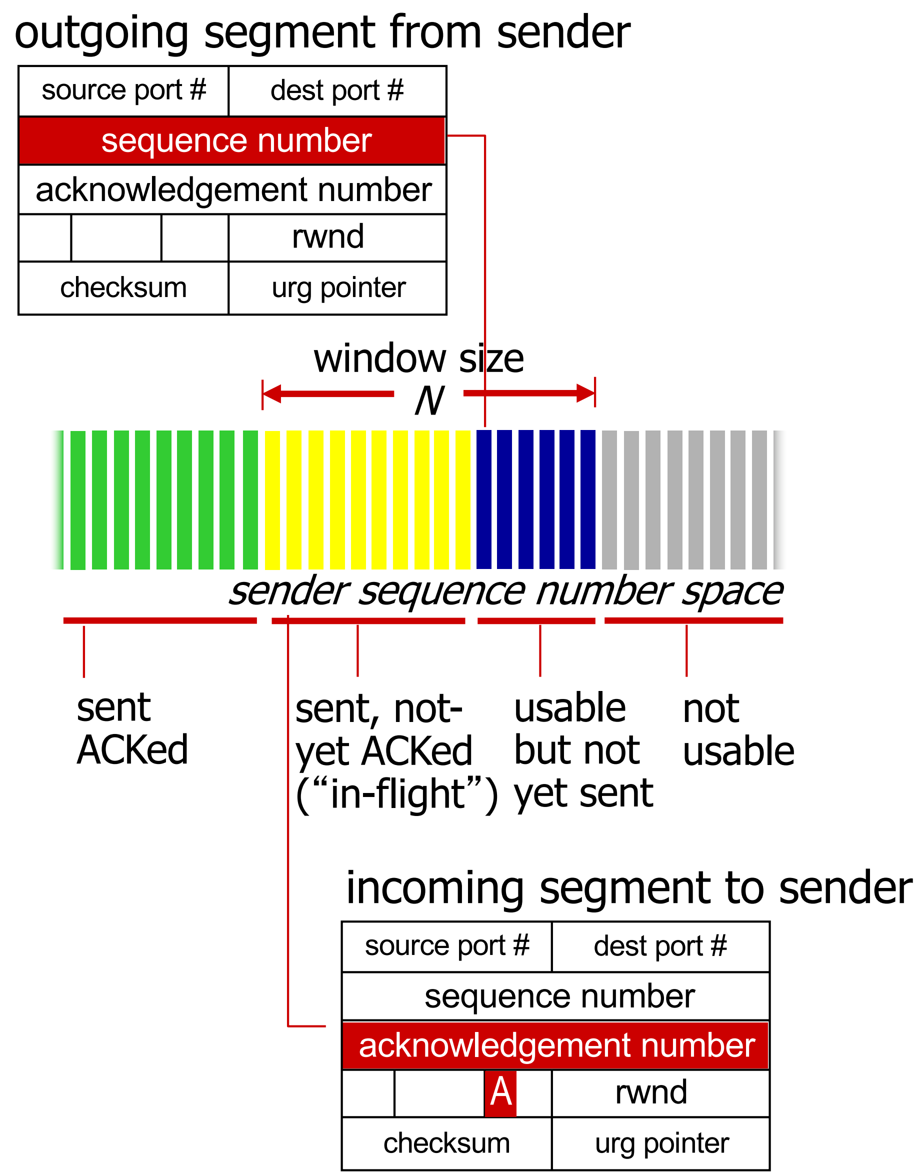
- byte stream "number" of first byte in segment's data

**acknowledgements:**

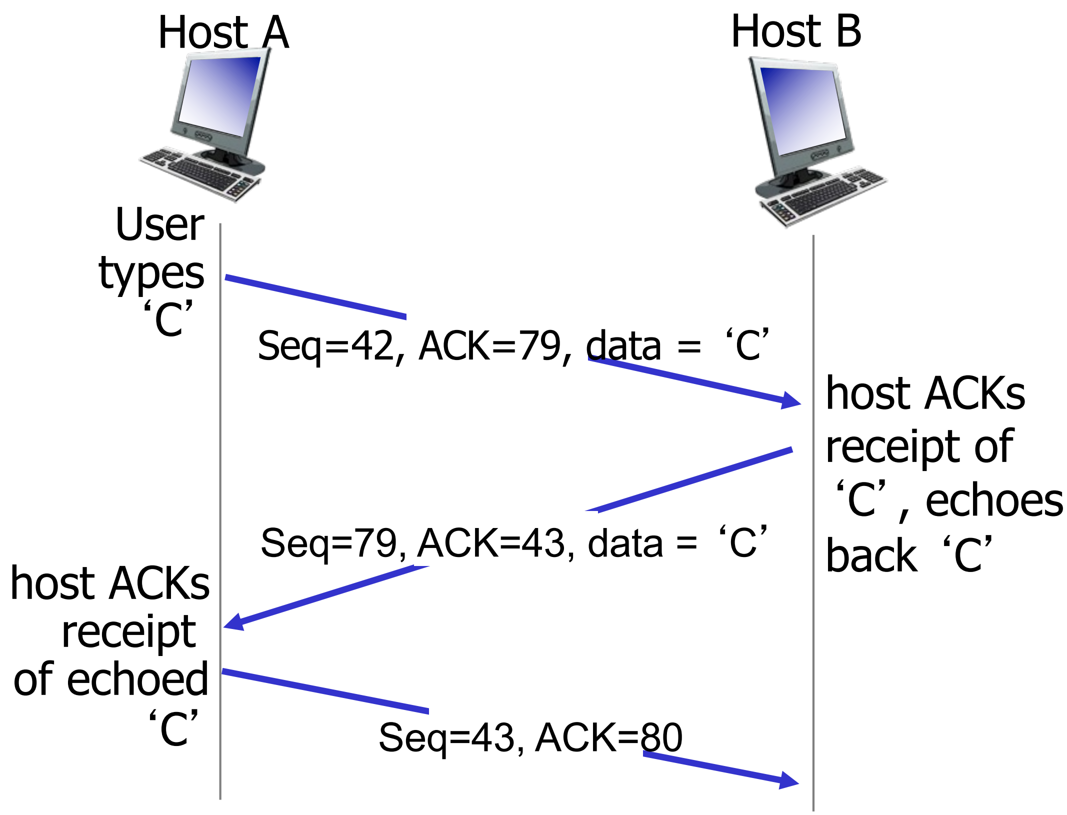
- seq # of next byte **expected** from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



# TCP seq. numbers, ACKs




simple telnet scenario

# TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short?*
  - premature timeout, unnecessary retransmissions
- *too long?*
  - slow reaction to segment loss

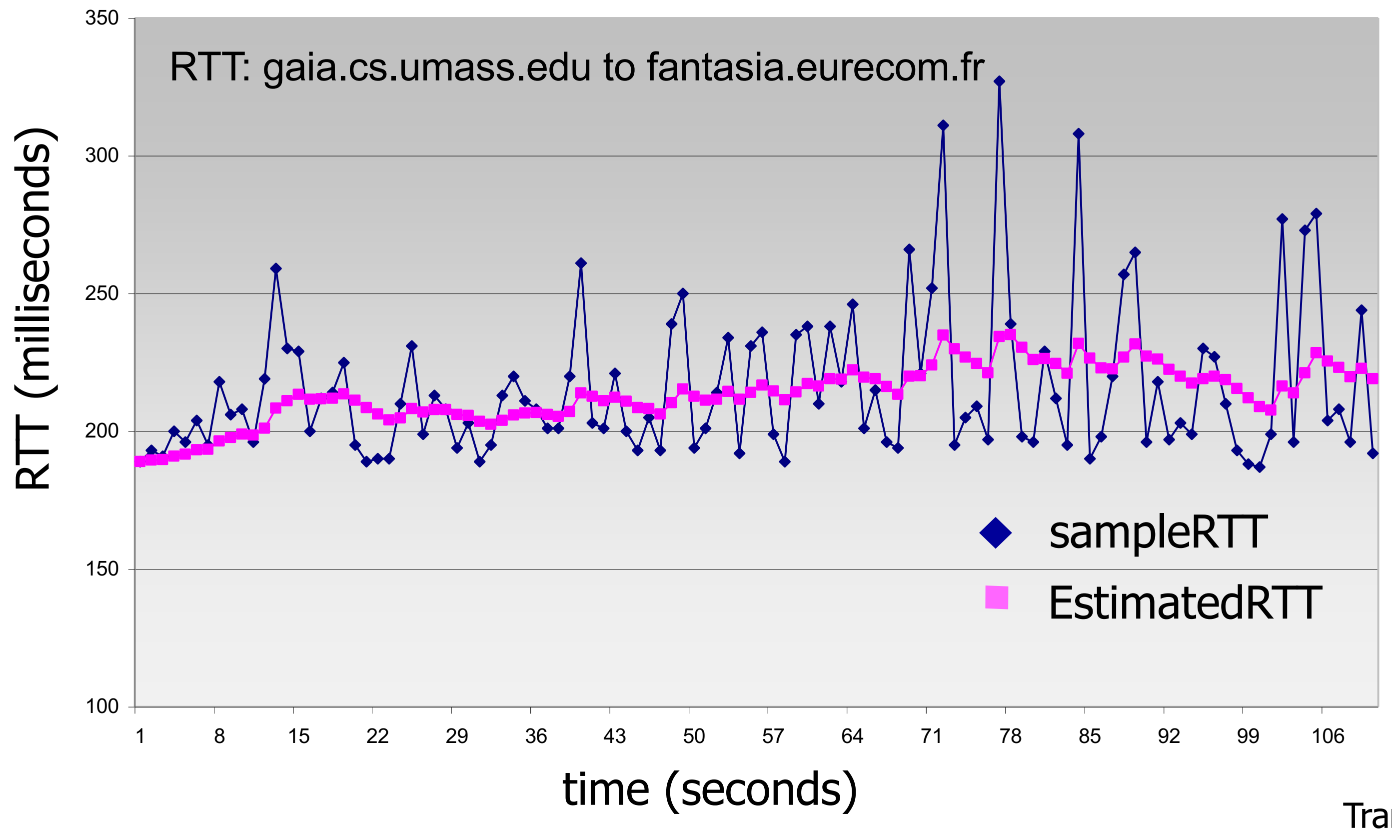
Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

# TCP round trip time, timeout

**EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT**

- exponential weighted moving average
- typical value: $\alpha = 0.125$



# TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT -> `** larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
                  β*|SampleRTT-EstimatedRTT|

                 (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

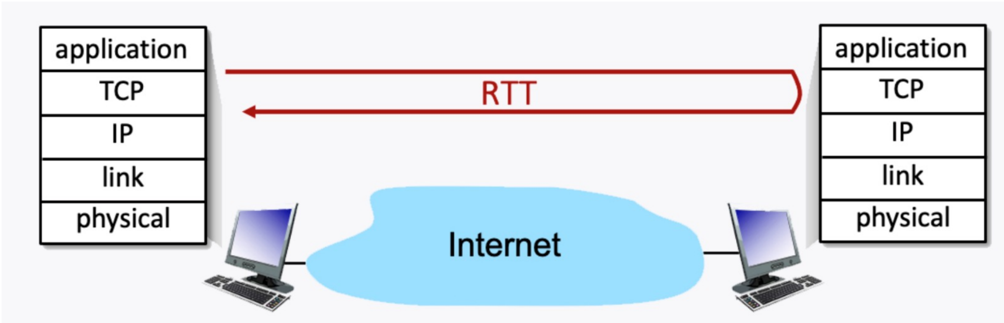
estimated RTT

"safety margin"

# Example

```
estimatedRTT = (1-alpha)*estimatedRTT + alpha*sampleRTT
DevRTT = (1-beta)*DevRTT + beta * |estimatedRTT - sampleRTT|
TCP timeout = estimatedRTT + (4*DevRTT)
```

- TCP's current estimatedRTT = 390 msec and DevRTT = 23 msec.
- Next three measured values of the RTT: 270 msec, 380 msec, and 270 msec.
- $\alpha$ = 0.125, and $\beta$ = 0.25




Compute TCP's new value of (1) DevRTT, (2) estimatedRTT, and (3) TCP timeout after each of three measured RTT values is obtained.

For RTT 1

estimatedRTT = (1-alpha) * estimatedRTT + alpha * sampleRTT

= 0.875 * current estimatedRTT + 0.125 * sampleRTT

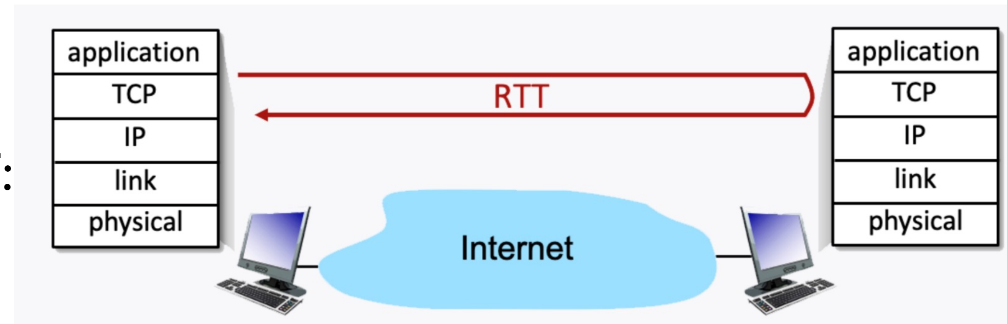
= 0.875 * 390 + 0.125 * 270

= 375 msec

# Example

```
estimatedRTT = (1-alpha)*estimatedRTT + alpha*sampleRTT
DevRTT = (1-beta)*DevRTT + beta * |estimatedRTT - sampleRTT|
TCP timeout = estimatedRTT + (4*DevRTT)
```

- TCP's current estimatedRTT = 390 msec and DevRTT = 23 msec.
- Next three measured values of the RTT: 270 msec, 380 msec, and 270 msec.
- $\alpha = 0.125$, and $\beta = 0.25$




Compute TCP's new value of (1) DevRTT, (2) estimatedRTT, and (3) TCP timeout after each of three measured RTT values is obtained.

For RTT 1

DevRTT = (1-beta) * DevRTT + beta * |estimatedRTT - sampleRTT|

= 0.75 * DevRTT + 0.25 * |estimatedRTT - sampleRTT|

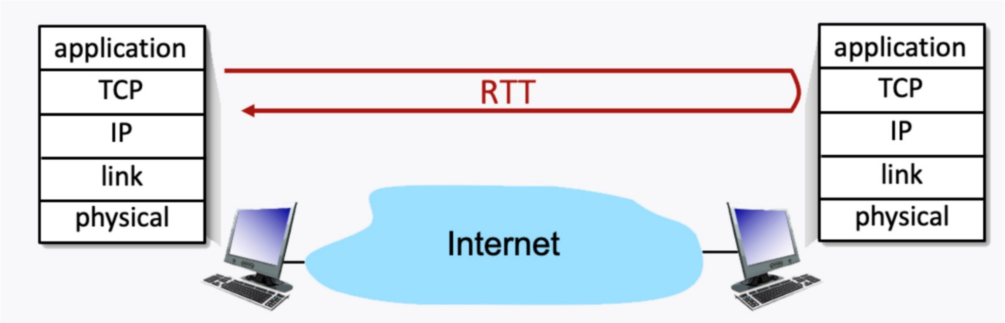
= 0.75 * 23 + 0.25 * |390 - 270|

= 47.25 msec

# Example

```
estimatedRTT = (1-alpha)*estimatedRTT + alpha*sampleRTT
DevRTT = (1-beta)*DevRTT + beta * |estimatedRTT - sampleRTT|
TCP timeout = estimatedRTT + (4*DevRTT)
```

- TCP's current estimatedRTT = 390 msec and DevRTT = 23 msec.
- Next three measured values of the RTT: 270 msec, 380 msec, and 270 msec.
- $\alpha = 0.125$, and $\beta = 0.25$




Compute TCP's new value of (1) DevRTT, (2) estimatedRTT, and (3) TCP timeout after each of three measured RTT values is obtained.

For RTT 1

TCP timeout = estimatedRTT + (4 * DevRTT)

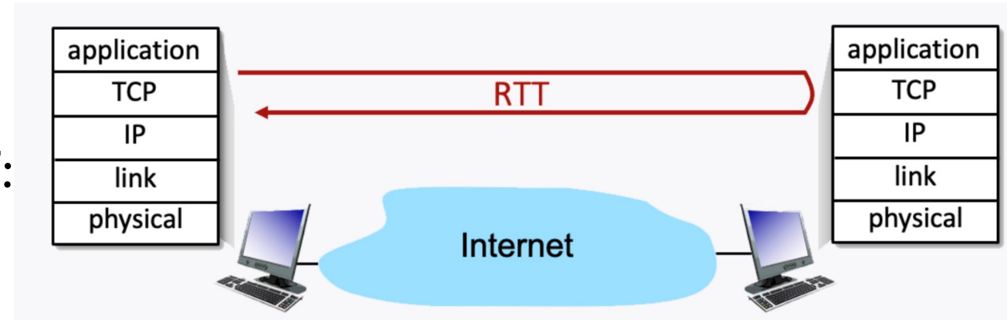
= 375 + 4 * 47.25

= 564 msec

# Example

```
estimatedRTT = (1-alpha)*estimatedRTT + alpha*sampleRTT
DevRTT = (1-beta)*DevRTT + beta * |estimatedRTT - sampleRTT|
TCP timeout = estimatedRTT + (4*DevRTT)
```

- TCP's current estimatedRTT = 390 msec and DevRTT = 23 msec.
- Next three measured values of the RTT: 270 msec, 380 msec, and 270 msec.
- $\alpha = 0.125$, and $\beta = 0.25$




Compute TCP's new value of (1) DevRTT, (2) estimatedRTT, and (3) TCP timeout after each of three measured RTT values is obtained.

For RTT 2

estimatedRTT = (1-alpha) * estimatedRTT + alpha * sampleRTT

= 0.875 * current estimatedRTT + 0.125 * sampleRTT

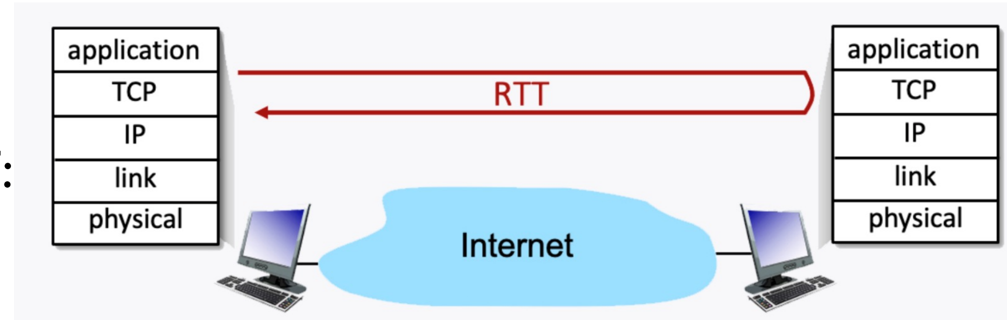
= 0.875 * 375 + 0.125 * 380

= 375.63 msec

# Example

```
estimatedRTT = (1-alpha)*estimatedRTT + alpha*sampleRTT
DevRTT = (1-beta)*DevRTT + beta * |estimatedRTT - sampleRTT|
TCP timeout = estimatedRTT + (4*DevRTT)
```

- TCP's current estimatedRTT = 390 msec and DevRTT = 23 msec.
- Next three measured values of the RTT: 270 msec, 380 msec, and 270 msec.
- $\alpha = 0.125$, and $\beta = 0.25$




Compute TCP's new value of (1) DevRTT, (2) estimatedRTT, and (3) TCP timeout after each of three measured RTT values is obtained.

For RTT 2

DevRTT = (1-beta) * DevRTT + beta * |estimatedRTT - sampleRTT|

= 0.75 * DevRTT + 0.25 * |estimatedRTT - sampleRTT|

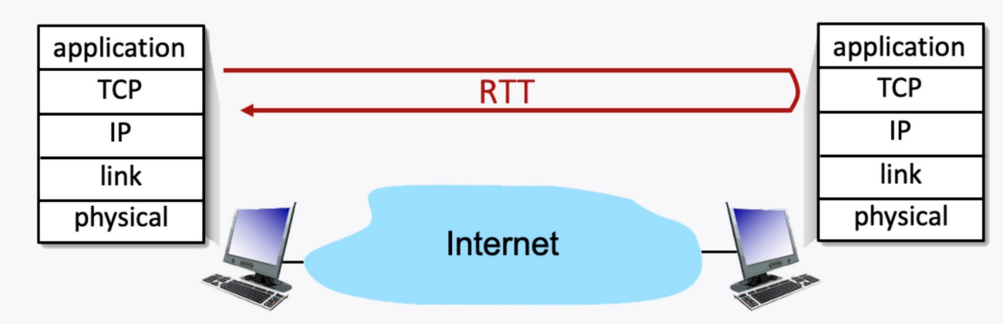
= 0.75 * 47.25 + 0.25 * |375 - 380|

= 36.69 msec

# Example

```
estimatedRTT = (1-alpha)*estimatedRTT + alpha*sampleRTT
DevRTT = (1-beta)*DevRTT + beta * |estimatedRTT - sampleRTT|
TCP timeout = estimatedRTT + (4*DevRTT)
```

- TCP's current estimatedRTT = 390 msec and DevRTT = 23 msec.
- Next three measured values of the RTT: 270 msec, 380 msec, and 270 msec.
- $\alpha = 0.125$, and $\beta = 0.25$




Compute TCP's new value of (1) DevRTT, (2) estimatedRTT, and (3) TCP timeout after each of three measured RTT values is obtained.

For RTT 2

TCP timeout = 522.38

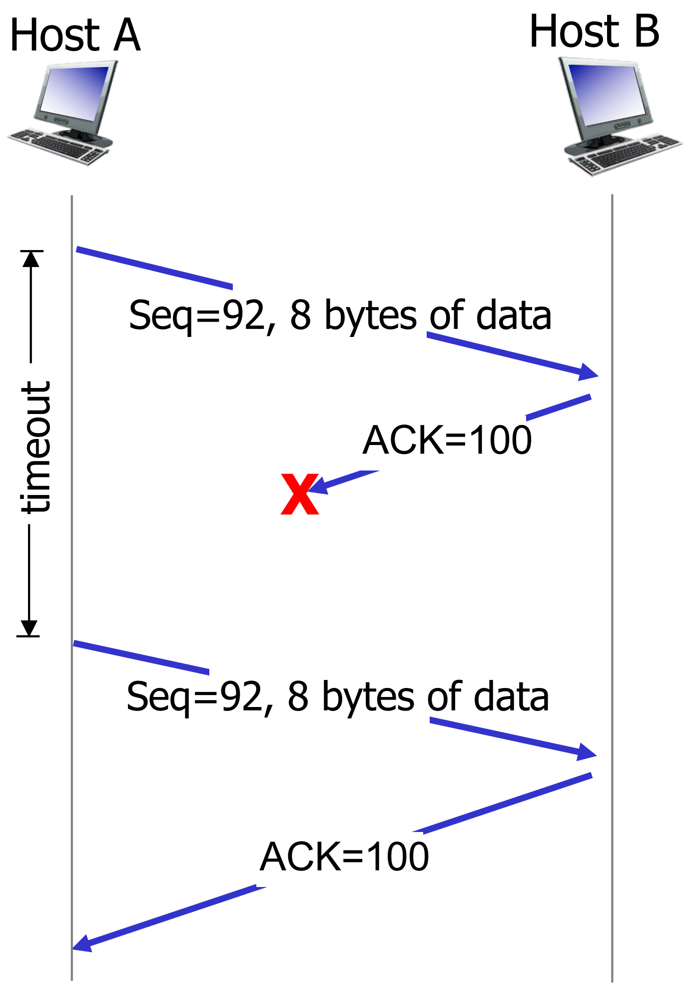
For RTT 3

estimatedRTT = 362.42

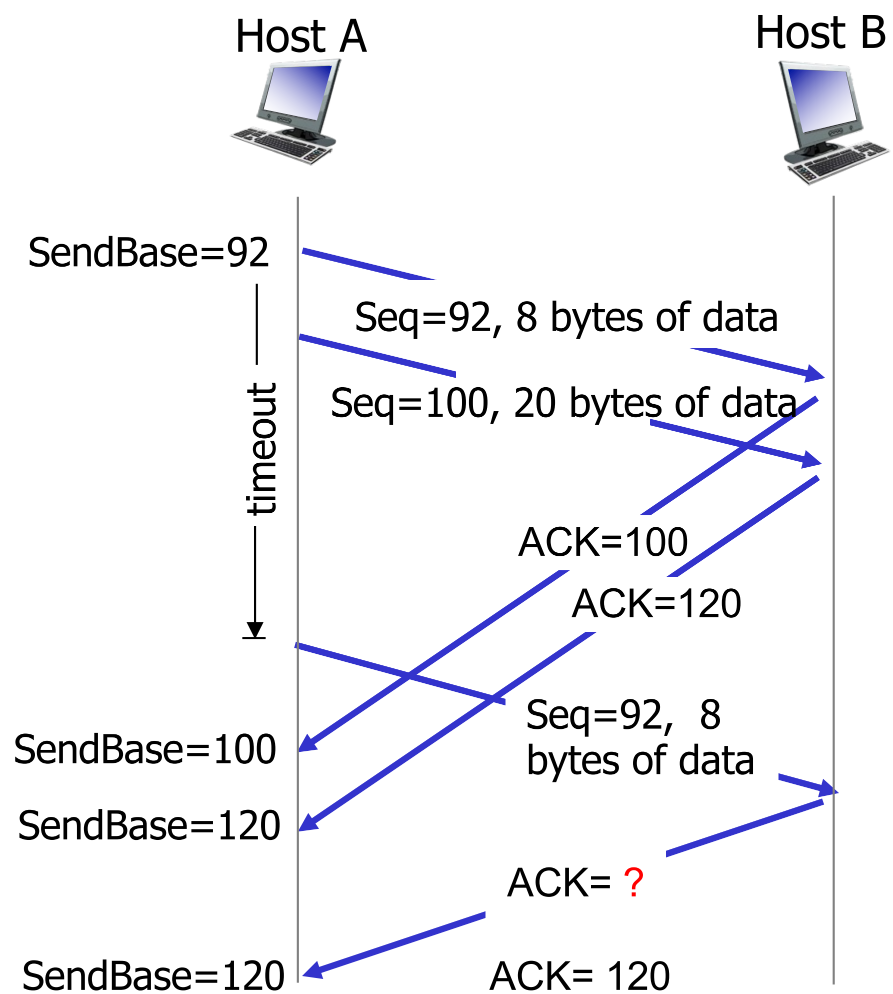
DevRTT =53.92

TCP timeout = 578.11

# TCP: retransmission scenarios

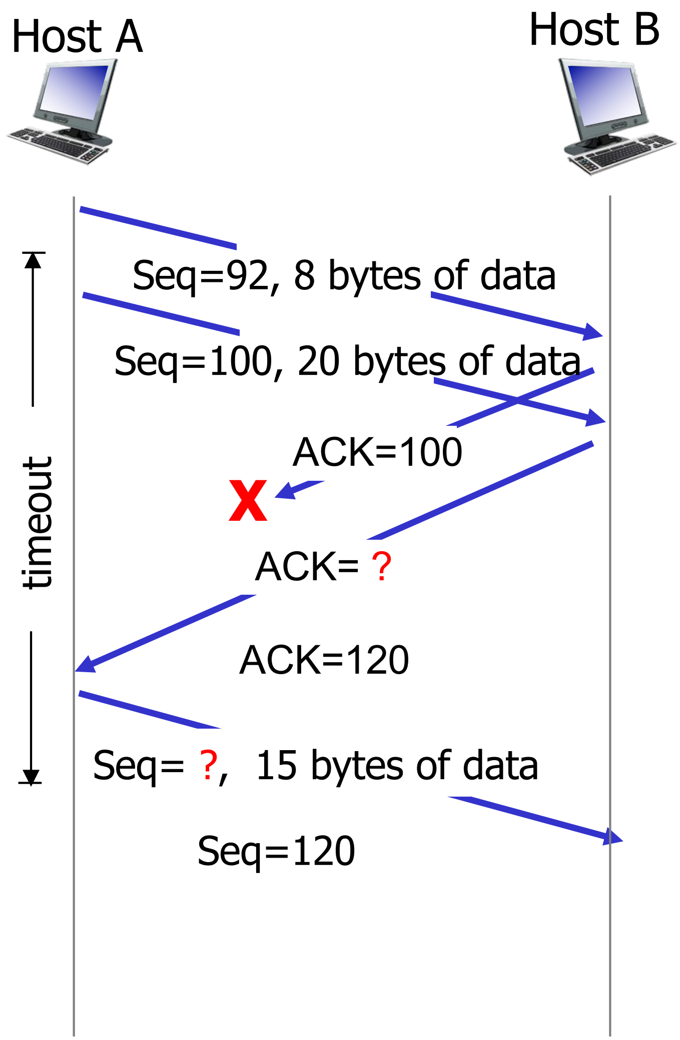



lost ACK scenario




premature timeout

# TCP: retransmission scenarios




cumulative ACK

# TCP fast retransmit

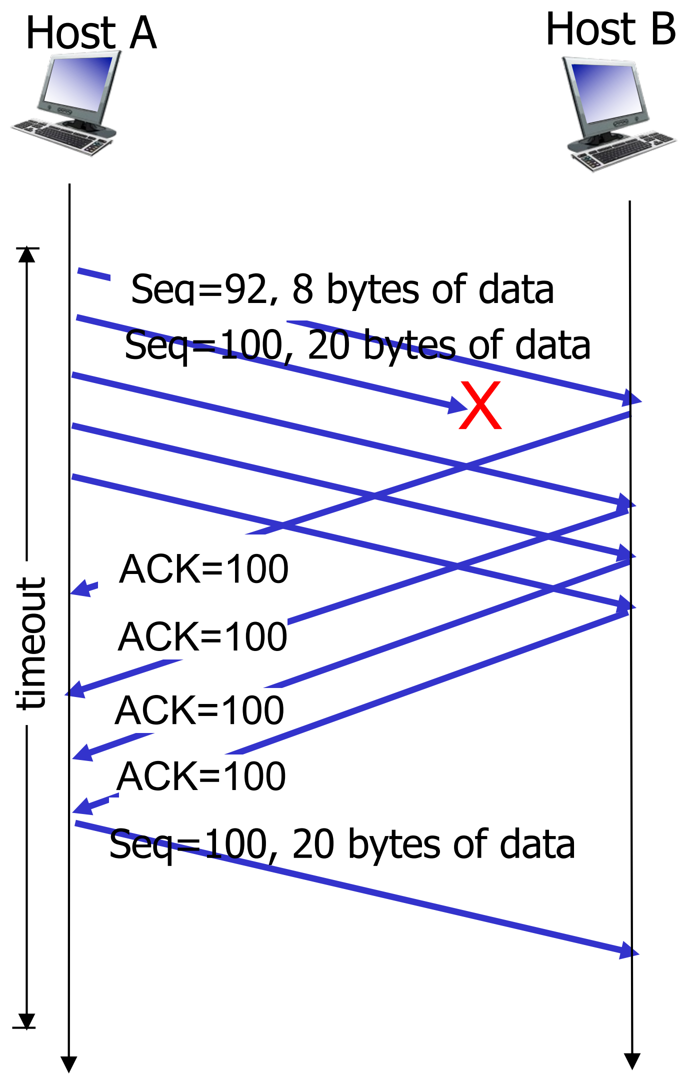
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

*TCP fast retransmit*

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

# TCP fast retransmit

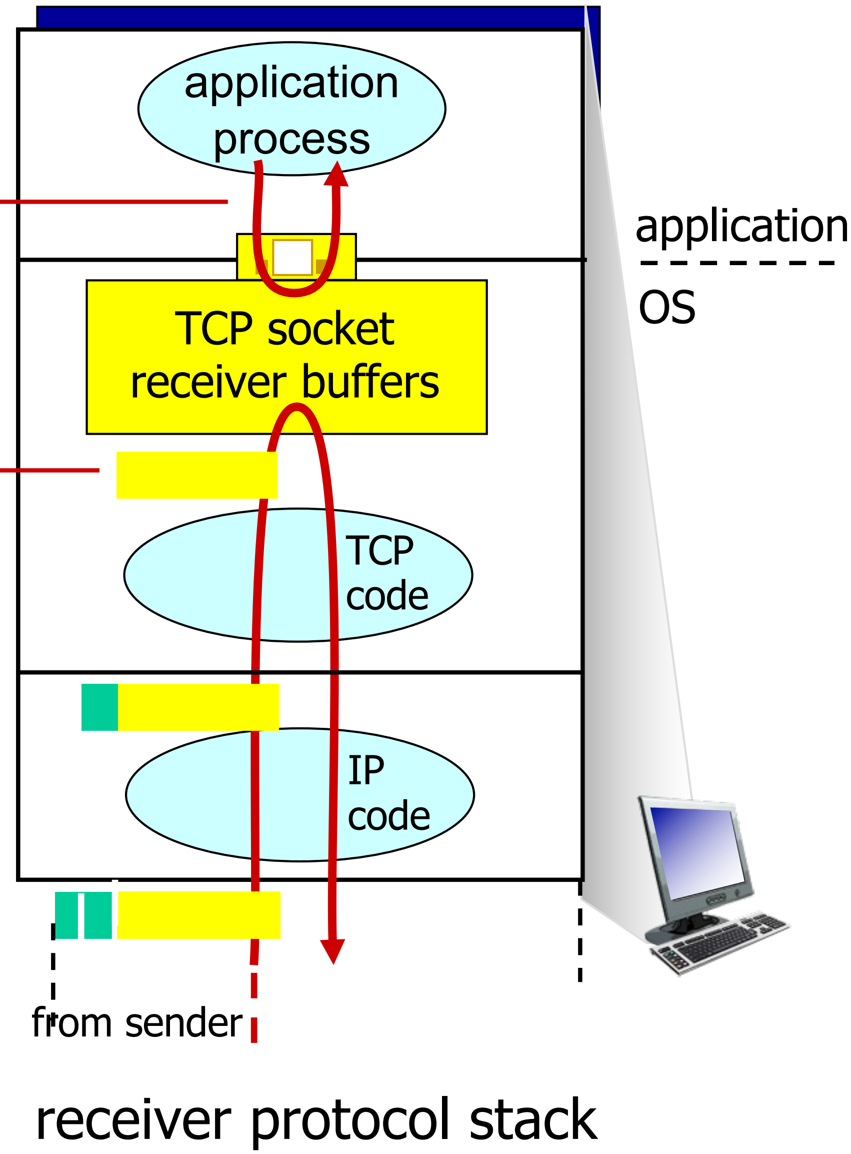



Fast retransmit after sender receipt of **triple** duplicate ACKs

# TCP flow control

application may remove data from TCP socket buffers ....

... slower than TCP receiver is delivering (sender is sending)

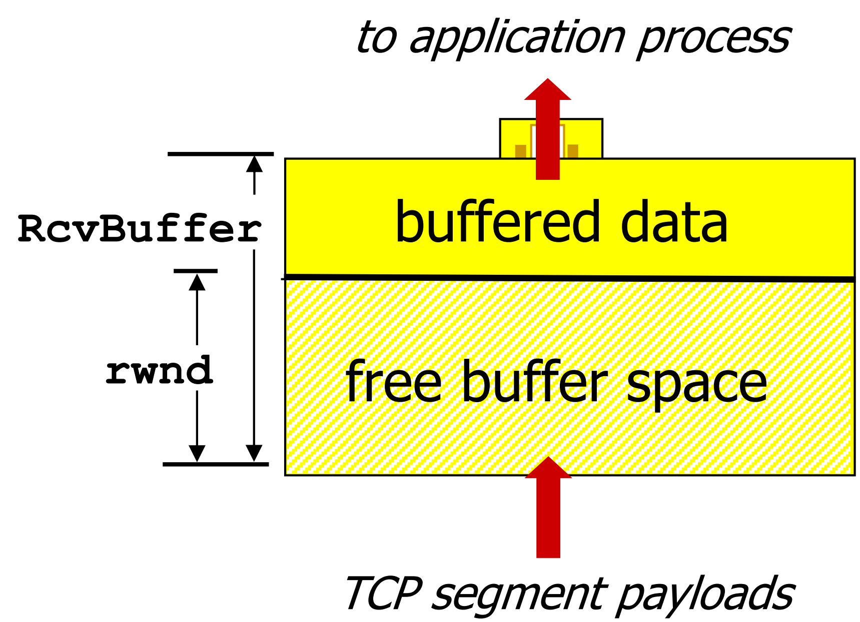



*flow control*

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

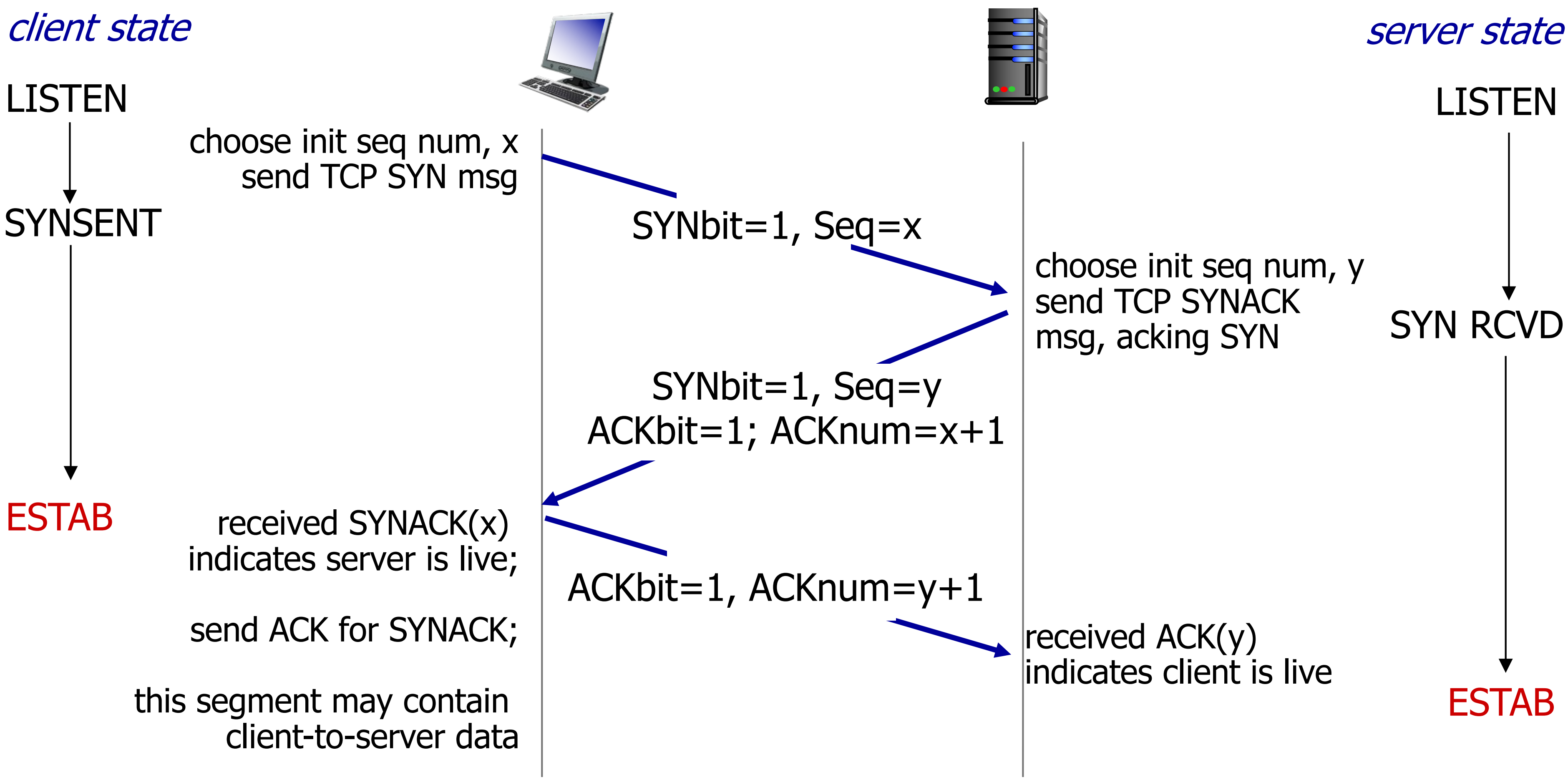
# TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow




*receiver-side buffering*

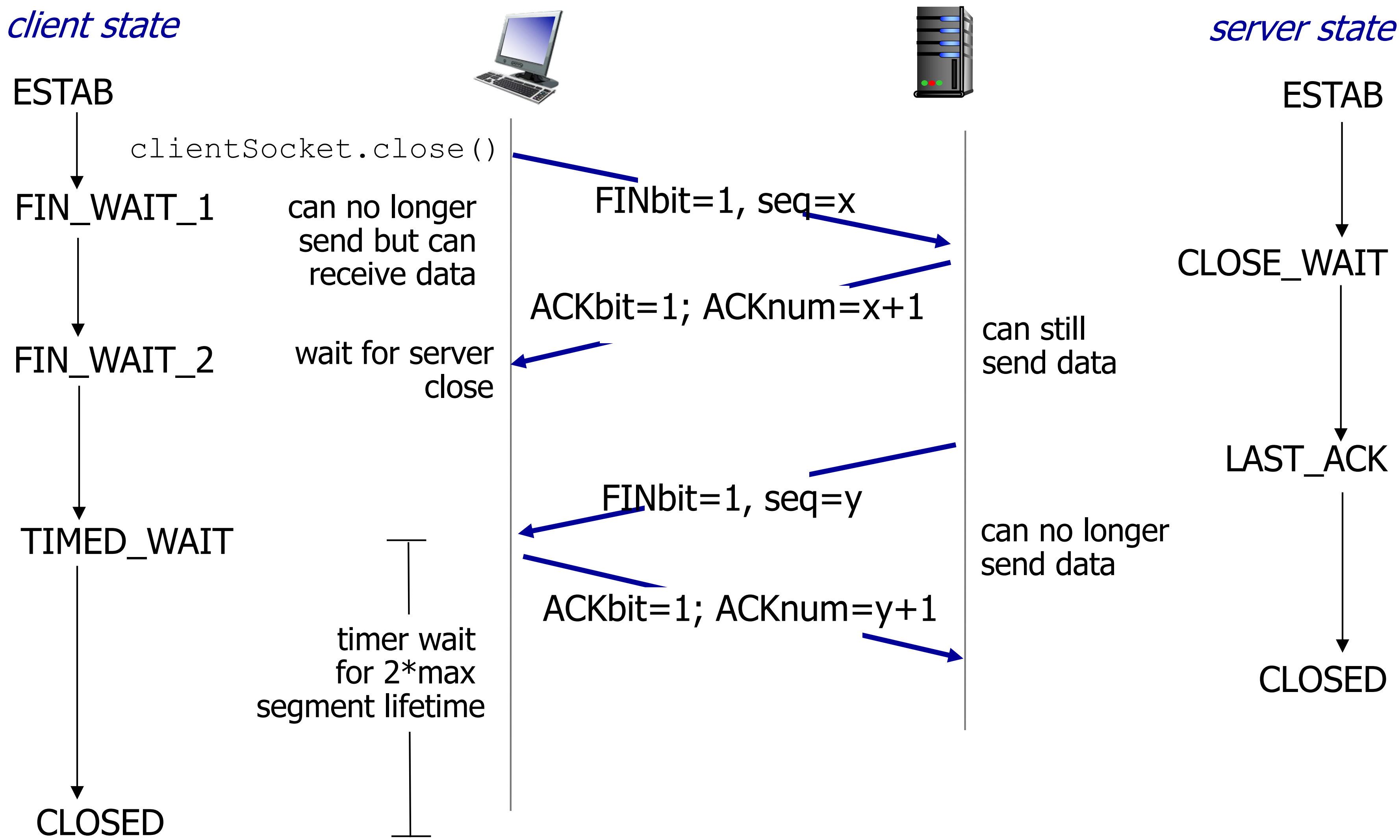
# TCP 3-way handshake



# TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

# TCP: closing a connection



# Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
- instantiation, implementation on the Internet
  - UDP
  - TCP

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network layer chapters:
  - data plane
  - control plane