Computer Networks (CS F303)

BITS Pilani
Pilani Campus

Virendra Singh Shekhawat
Department of Computer Science and Information Systems

innovate achieve lead

BITS Pilani

Pilani Campus

Second Semester 2020-2021

Module-3 <Transport layer>

Lecture: 10-13

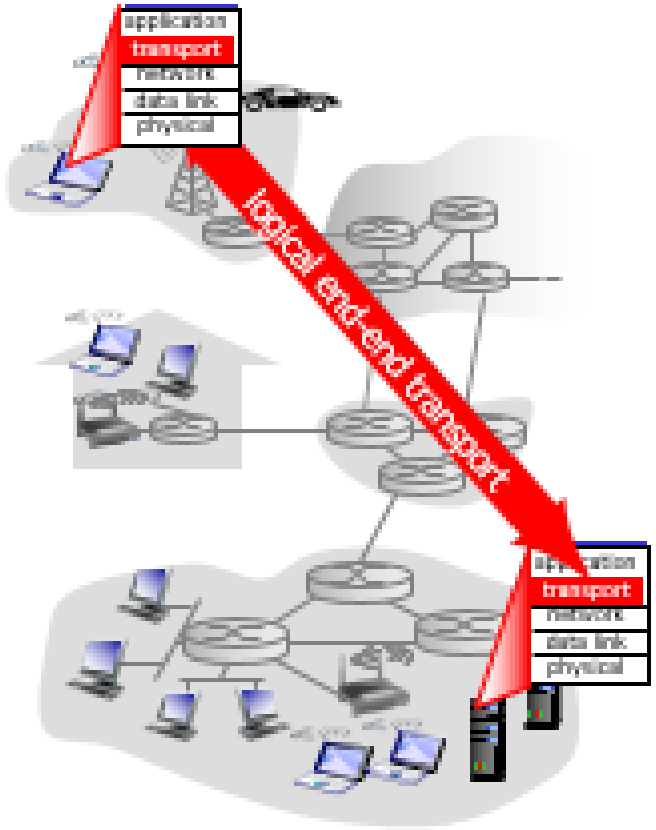
Topics

- Transport Layer
  - Transport Layer Services
    - Multiplexing/Demultiplexing
      - Connectionless and Connection Oriented
        - » TCP and UDP
    - Reliable data transfer (Protocol design)
    - Flow control
    - Congestion control

Transport Layer Services and Protocols

- Provides logical communication between app processes
  - Apps processes sends msgs to each other using the logical communication

- Extend **host-to-host** delivery to **process-to-process** delivery



TP Layer vs. Network Layer

- Network layer: logical communication between hosts
- TP Layer: logical communication between processes
- TP layer services are constrained by the service model of underlying network-layer protocol
- But certain services can be offered by the TP layer even when the network layer doesn't offer
  - e.g., Reliable data transfer

Transport Layer Services

- Reliable in-order delivery (TCP)
  - Congestion control
  - Flow control
  - Connection setup

- Unreliable, unordered delivery (UDP)
  - Extension of “best-effort” IP

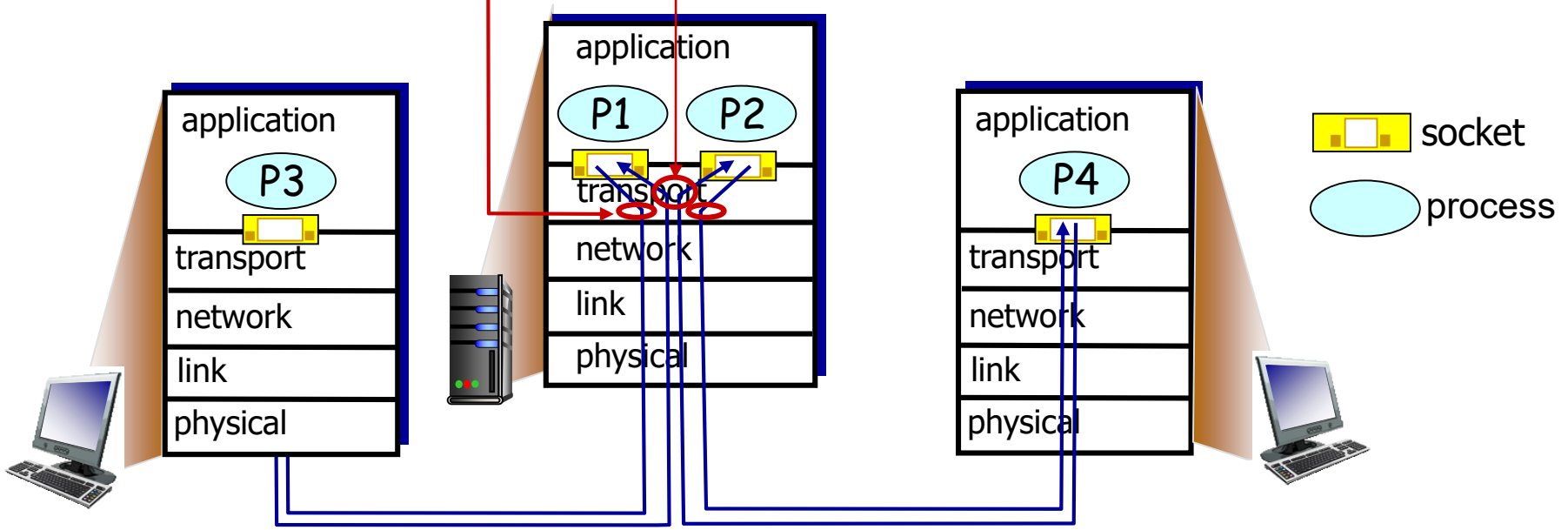
Process-to-Process Delivery Service

Multiplexing at sendening time:

handle data from multiple sockets, add transport **header**

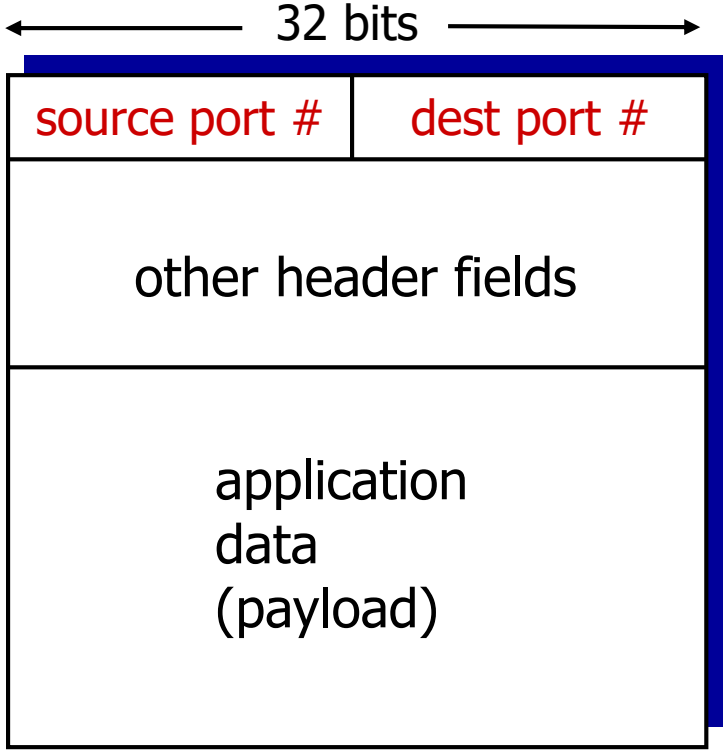
Demux at receiving time:

use **header** info to deliver received segments to correct socket



Demultiplexing at Receiver

- 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

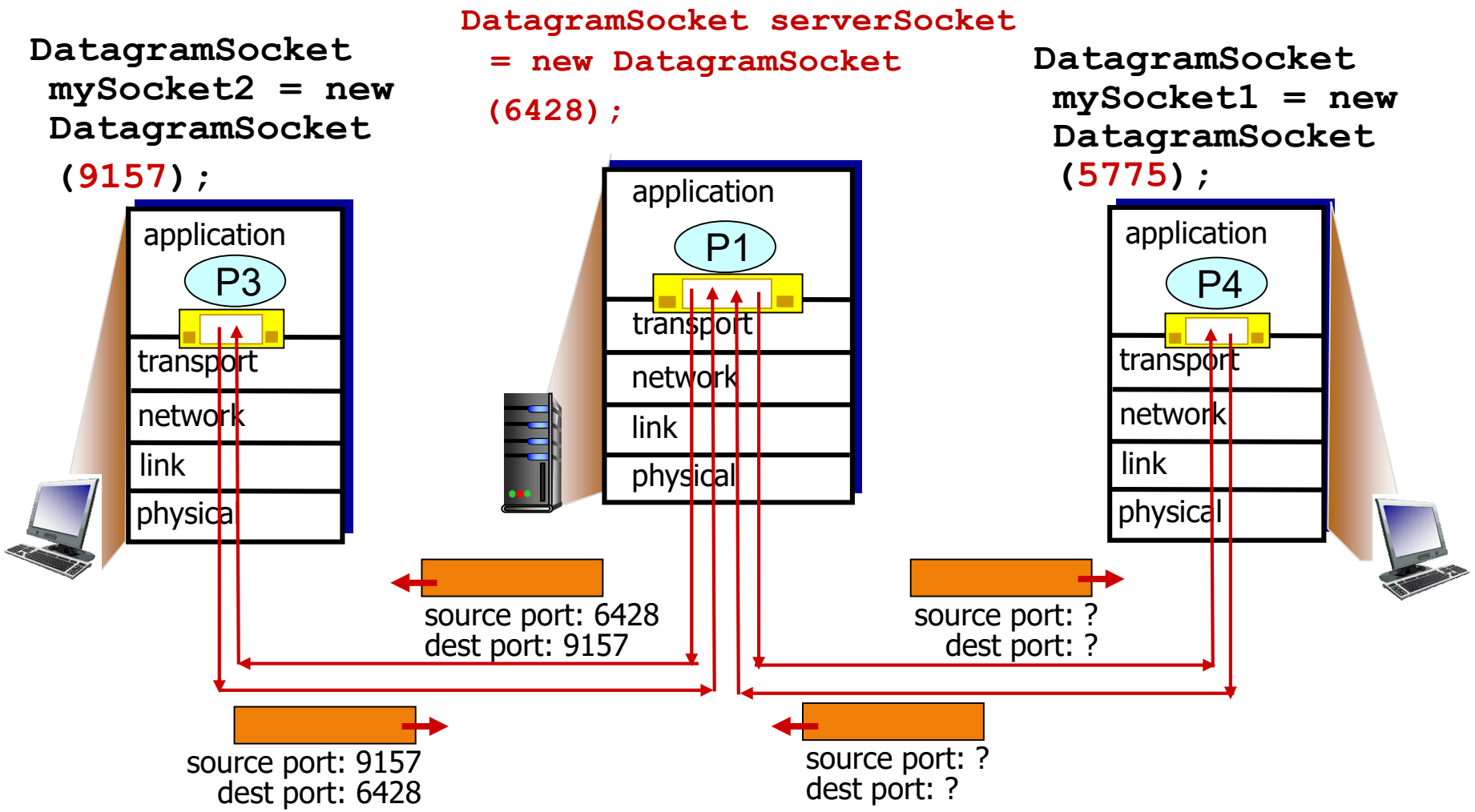
Connectionless (UDP) Demultiplexing

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

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

- Important to note that
  - IP datagrams with *same destination **port #**,* but different source IP addresses and/or source port numbers will be directed to *same socket* at destination

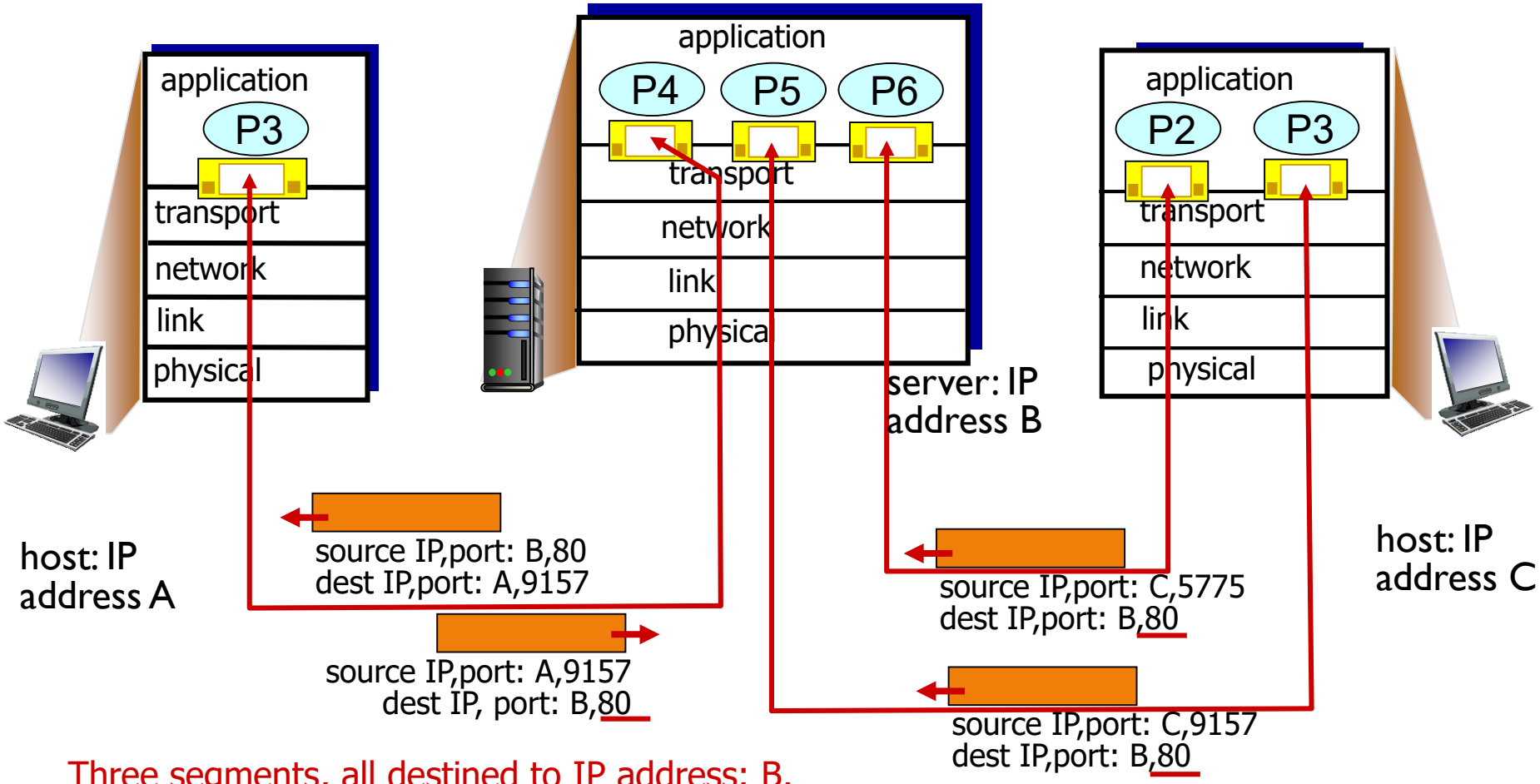
Example: Connectionless Demultiplexing



Connection Oriented Demultiplexing

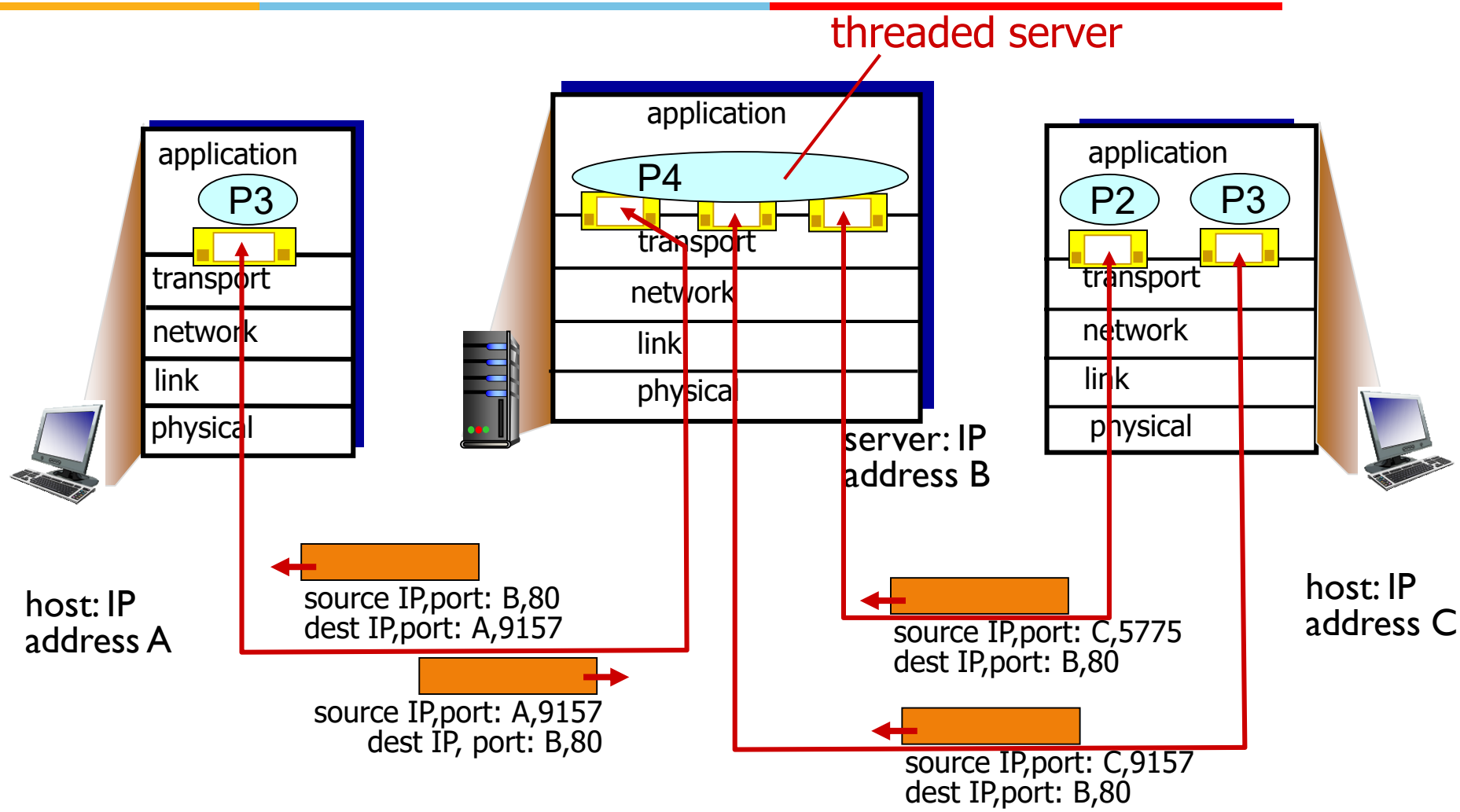
- TCP socket identified by 4-tuple:
  - Source IP address, source port #, dest IP address, dest port #
  - Demux: receiver uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - Each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - e.g., non-persistent HTTP will have different socket for each request

Example: Connection Oriented Demux



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

Example

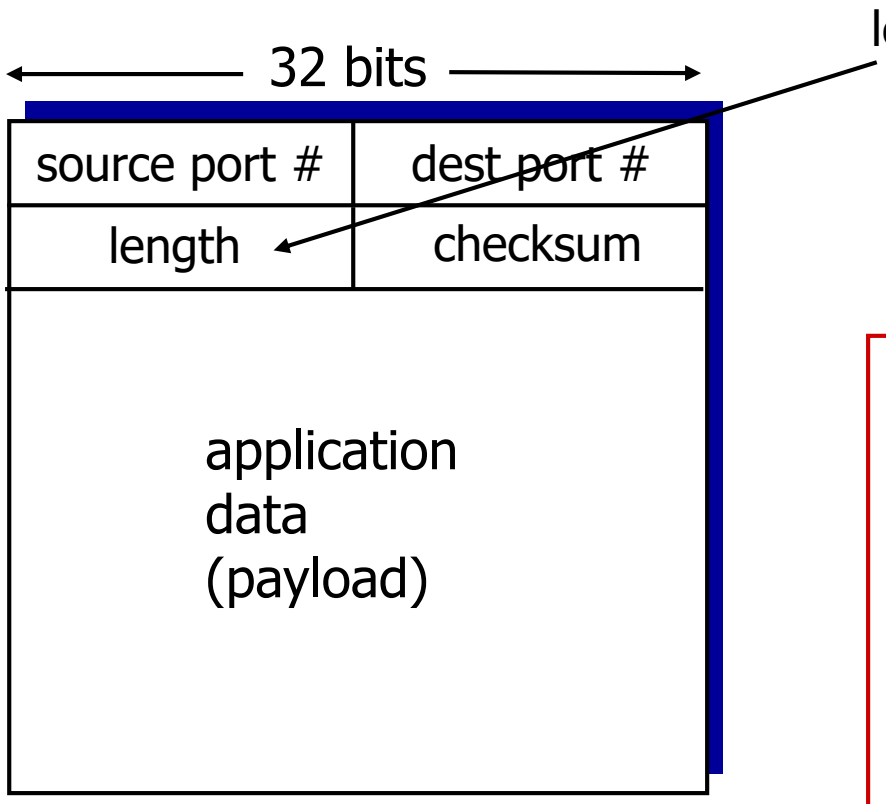

threaded server
application
P3
transport
network
link
physical
host: IP
address A
application
P4
transport
network
link
physica
server: IP
address B
application
P2
P3
transport
network
link
physical
host: IP
address C
source IP,port: B,80
dest IP,port: A,9157
source IP,port: A,9157
dest IP, port: B,80
source IP,port: C,5775
dest IP,port: B,80
source IP,port: C,9157
dest IP,port: B,80

User Datagram Protocol [RFC 768]

- Best effort service
  - UDP segment may lost, delivered out of order to app
- Connectionless
  - No handshaking between sender and receiver

- Each UDP segment handled independently of others

UDP Segment Header



UDP segment format

Why is there a UDP?

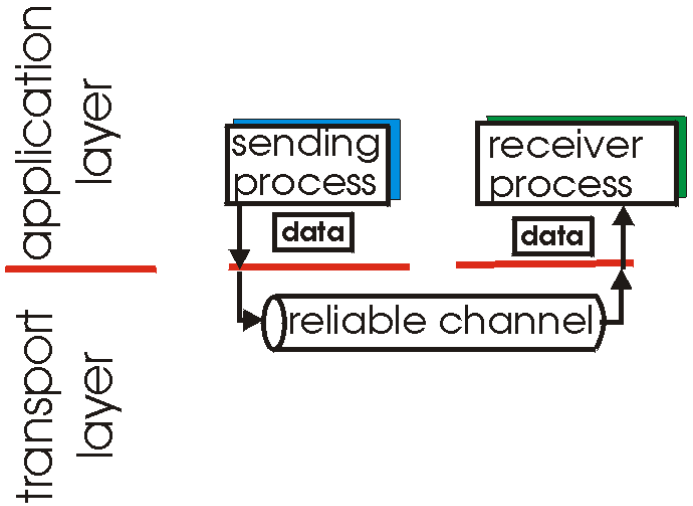
- No connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP Checksum

- Treat segment contents (with header fields) as a sequence of 16-bit integers at sender
  - Sum all such 16-bit words in the segment
  - One's complement of the sum is put in checksum field
- At the receiver, all 16-bit words are added (including checksum) to detect error in segment

Principles of Reliable Data Transfer

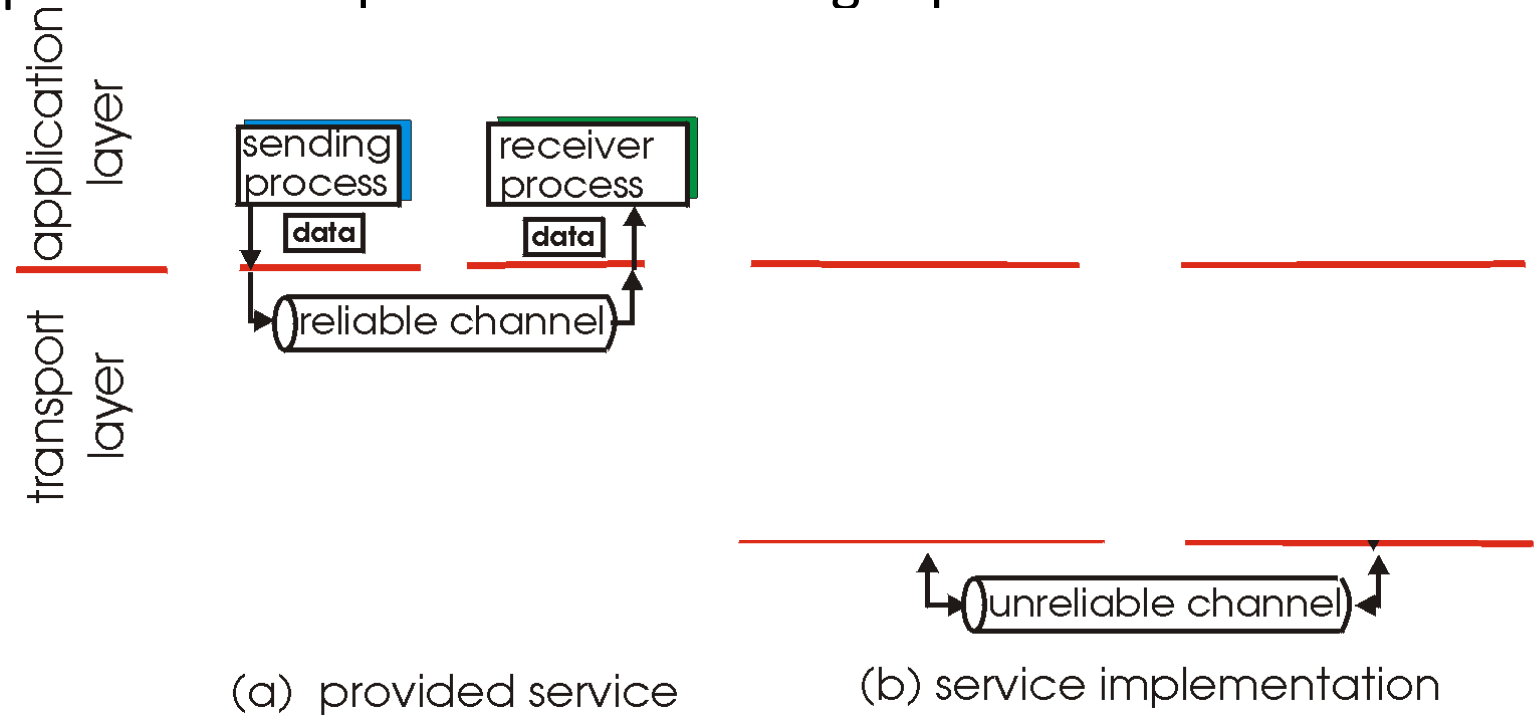
- Important in application, transport, link layers
- Top-10 list of important networking topics!



(a) provided service

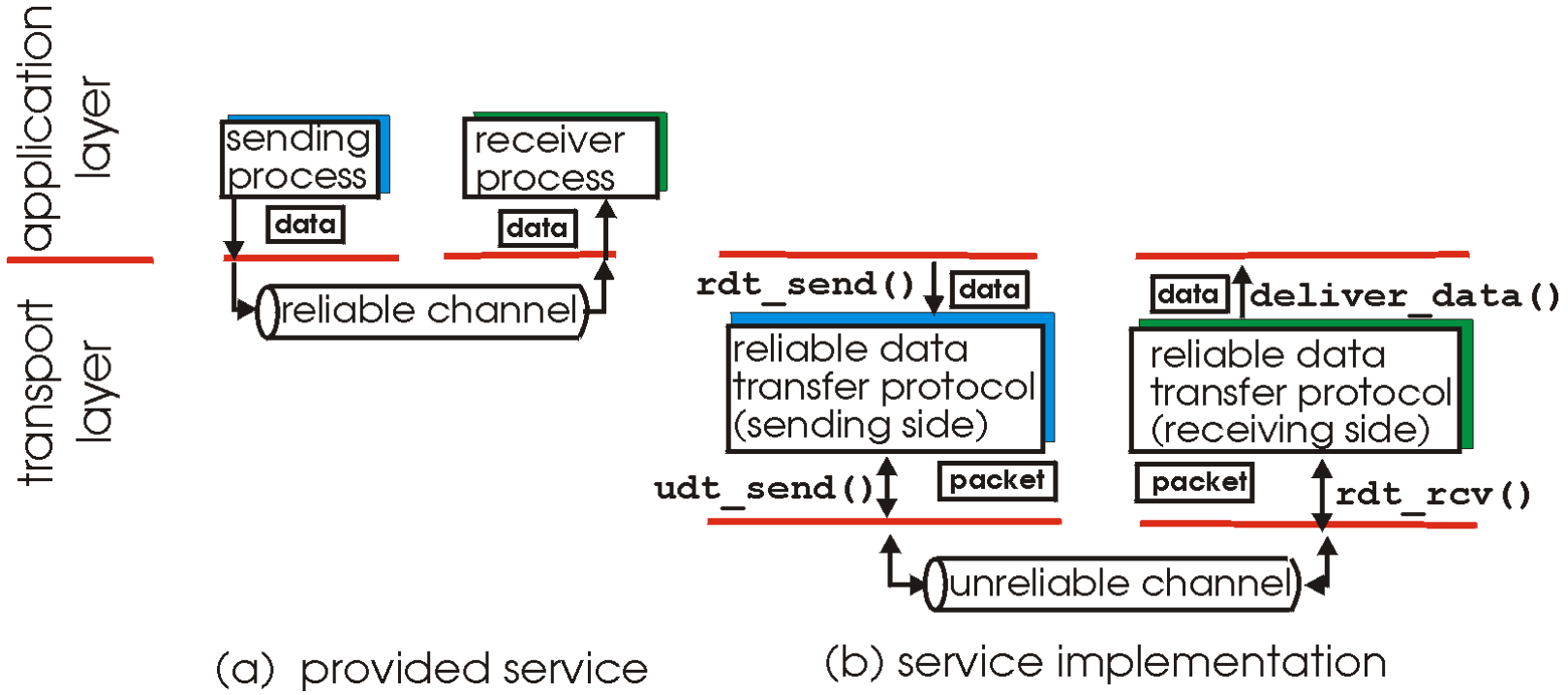
Principles of Reliable Data Transfer

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



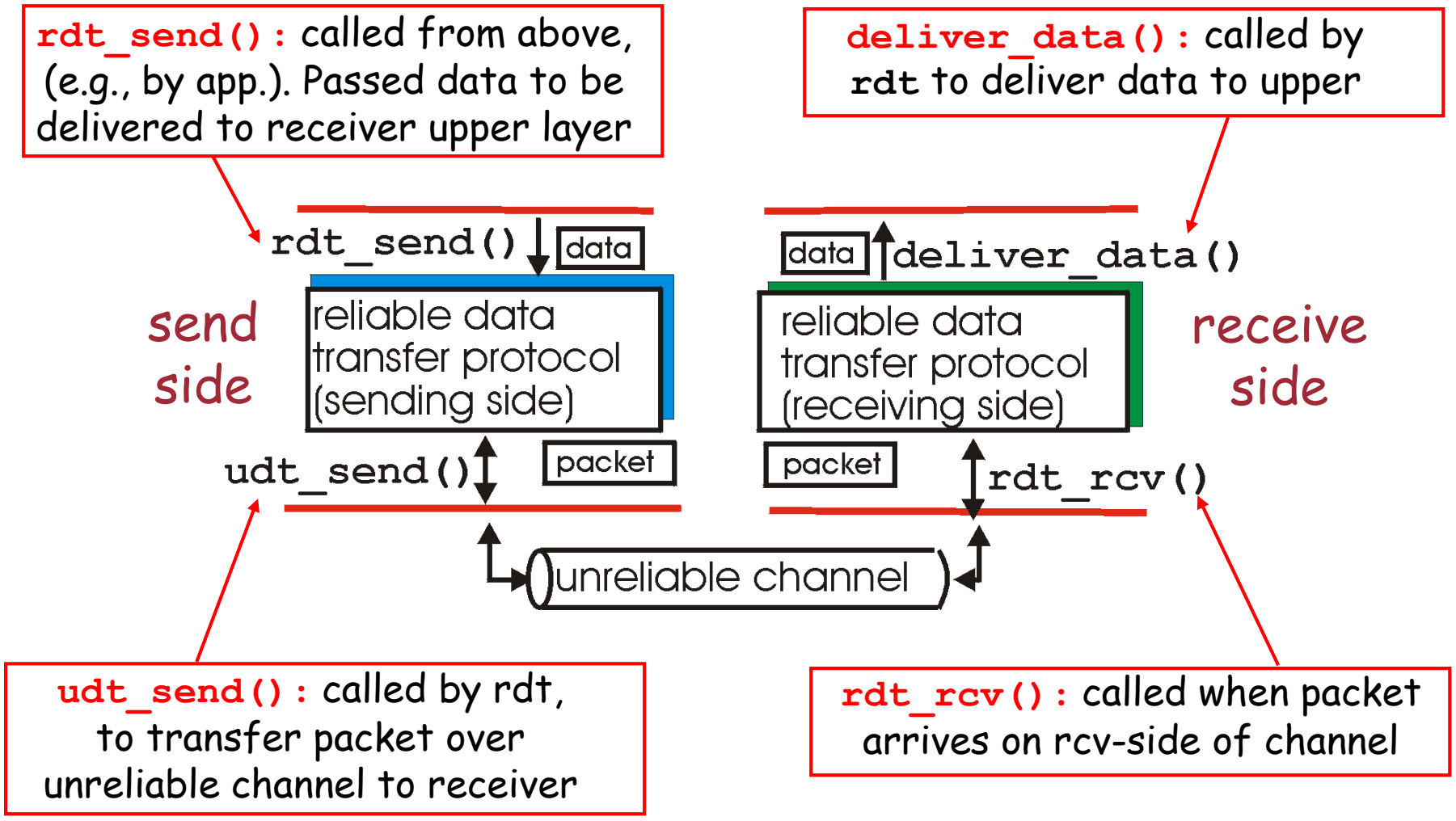
Principles of Reliable Data Transfer

- 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

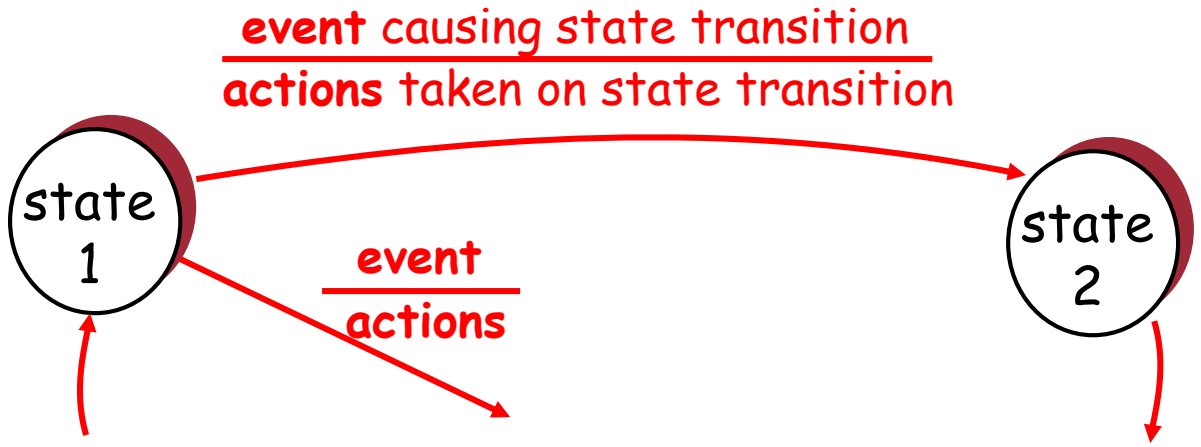


Reliable Data Transfer: getting started

We will:

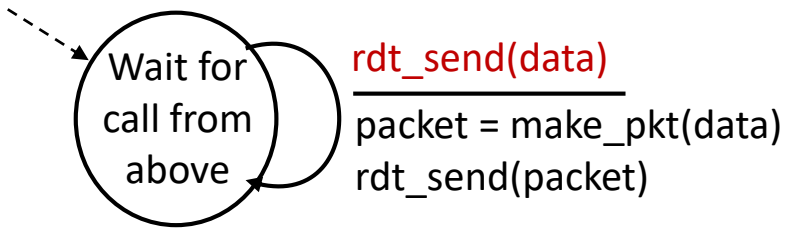
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - But control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver

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

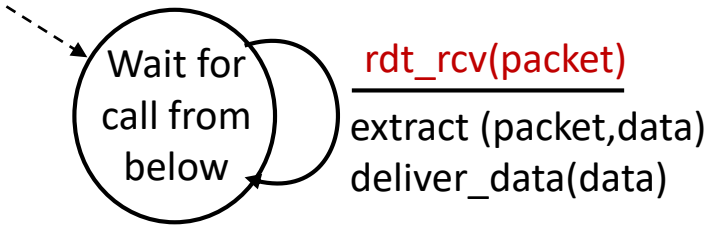


rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - No bit errors, No loss of packets
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver read data from underlying channel



sender



receiver

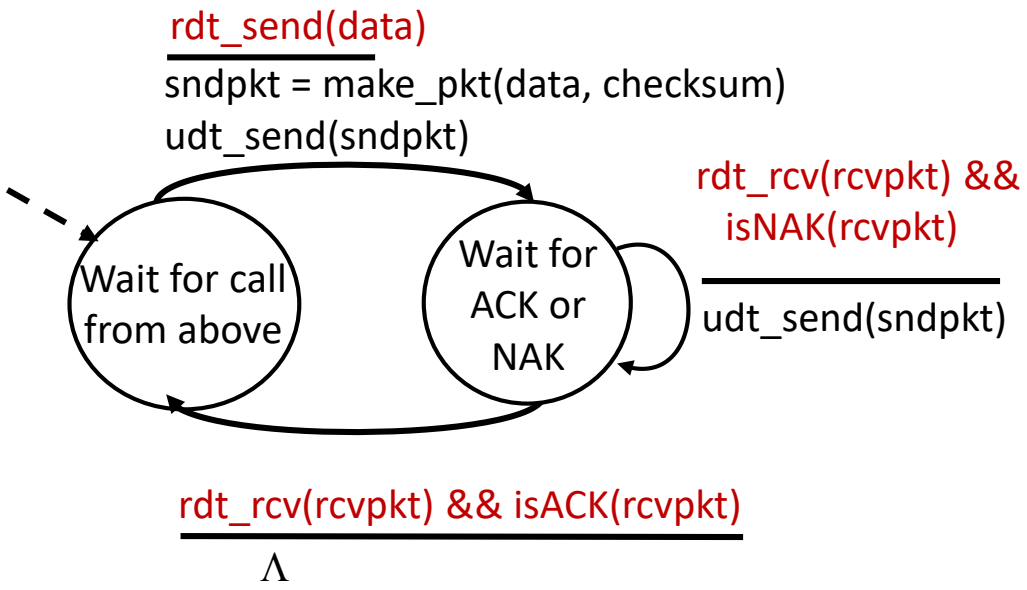
rdt2.0: channel with bit errors

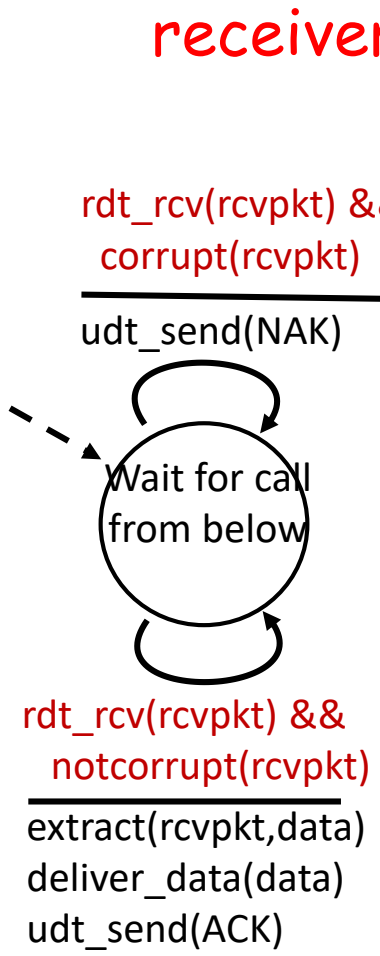
- Underlying channel may flip bits in packet
  - **Don't worry… Checksum is there to detect bit errors**

- **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

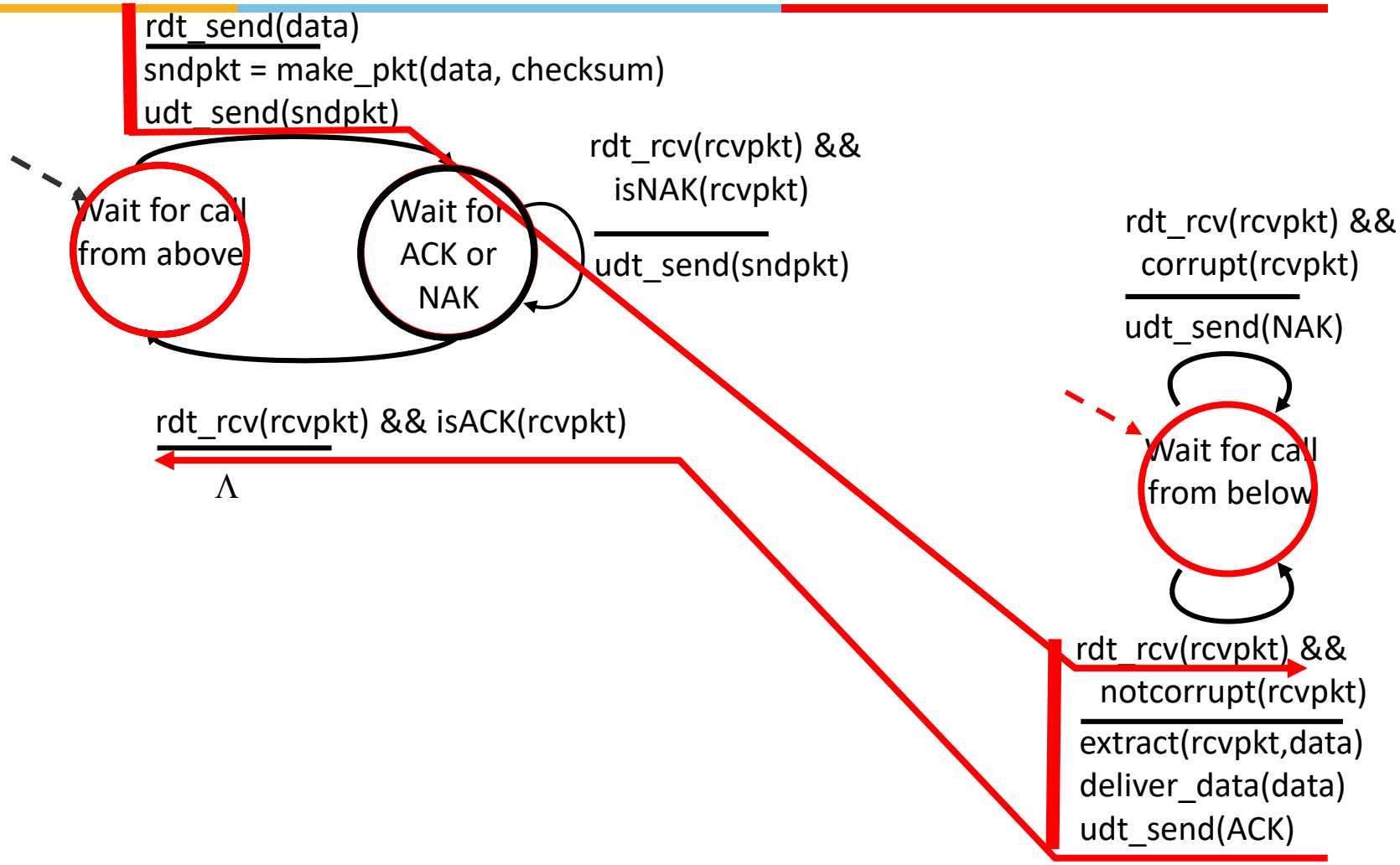
- New mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - Error detection
  - Receiver feedback: control msgs (ACK,NAK) rcvr->sender

rdt2.0: FSM Specification

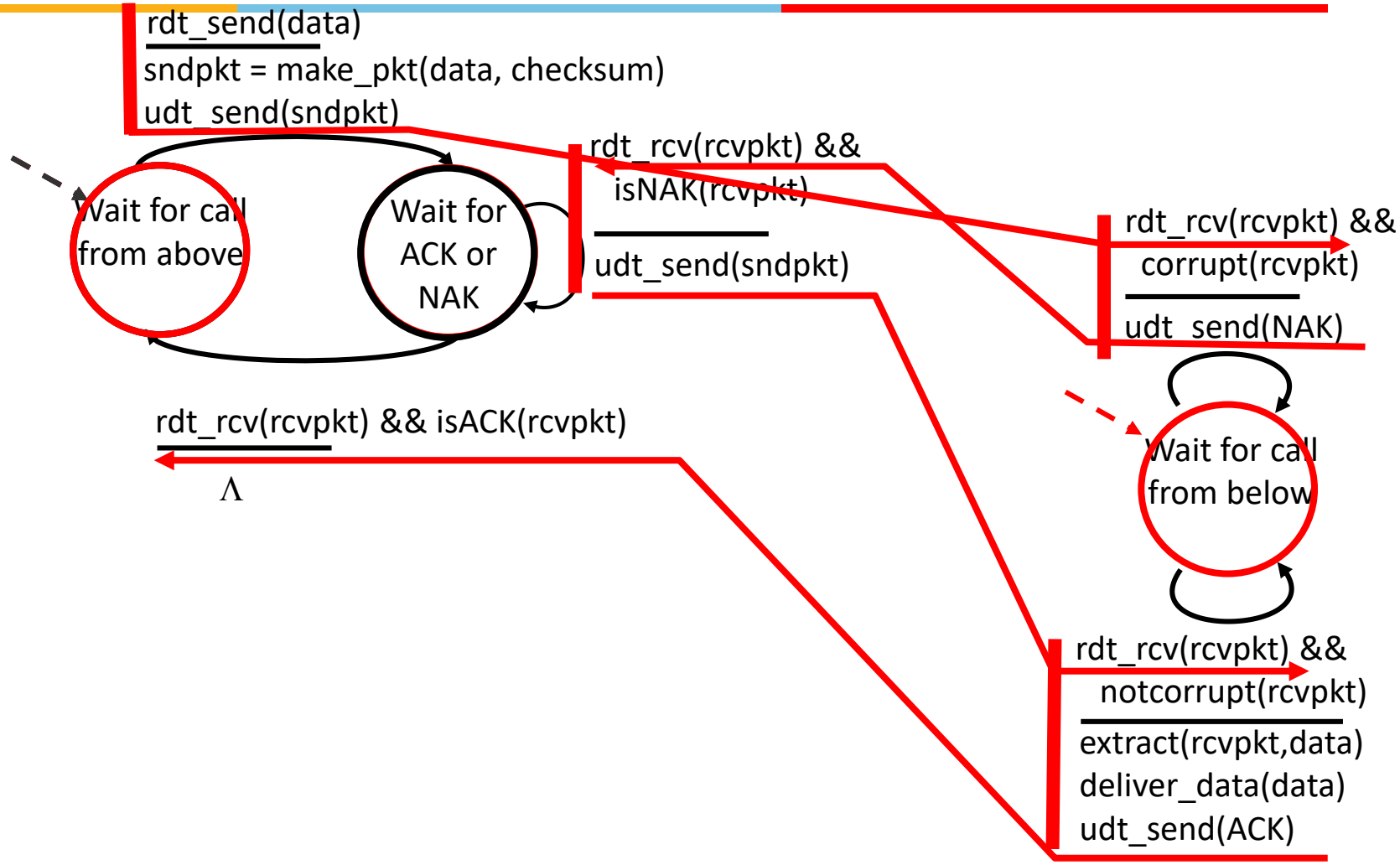




rdt2.0: Operation with no Errors



rdt2.0: Error Scenario

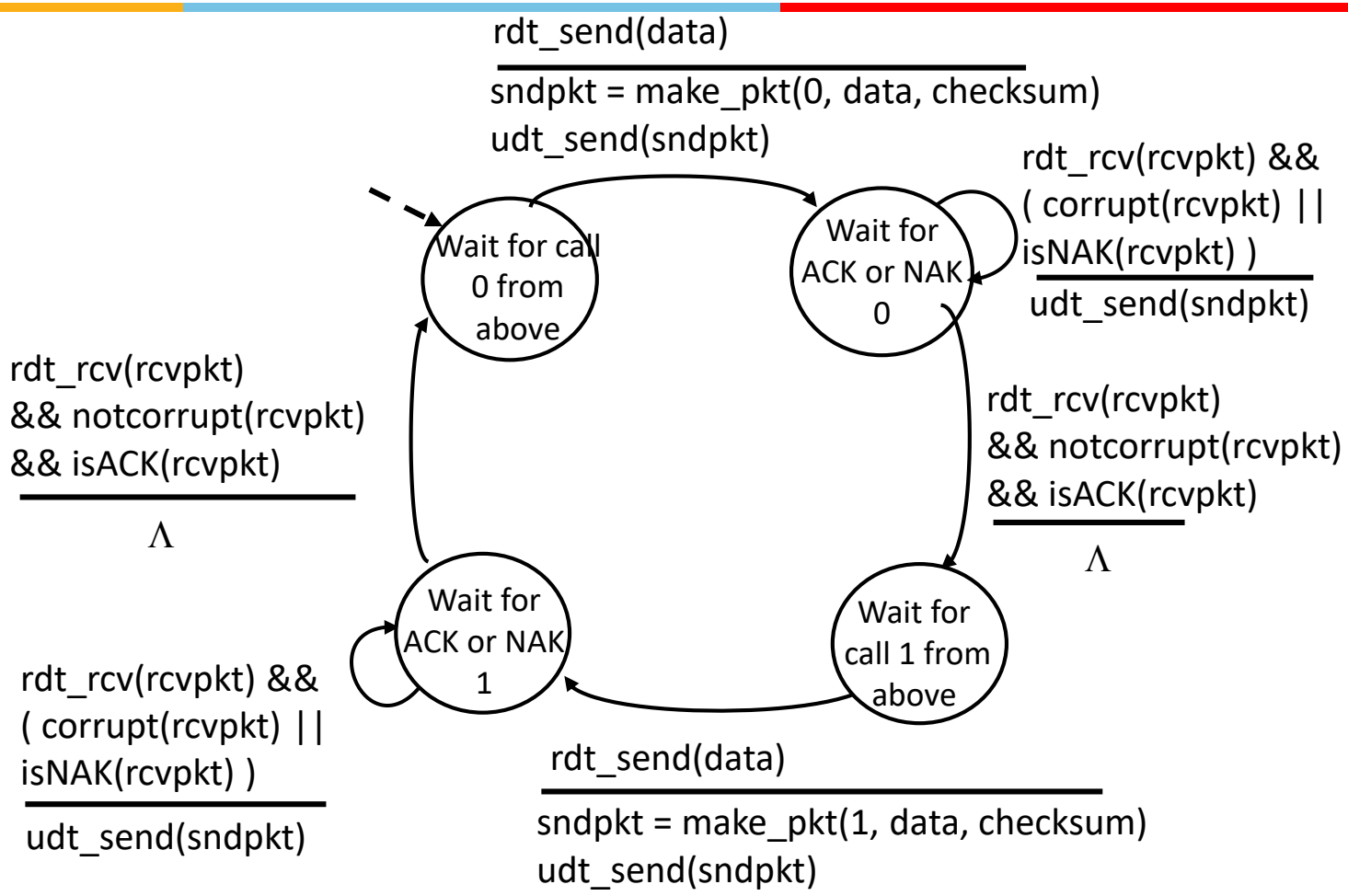


rdt2.0 Has a fatal flaw!

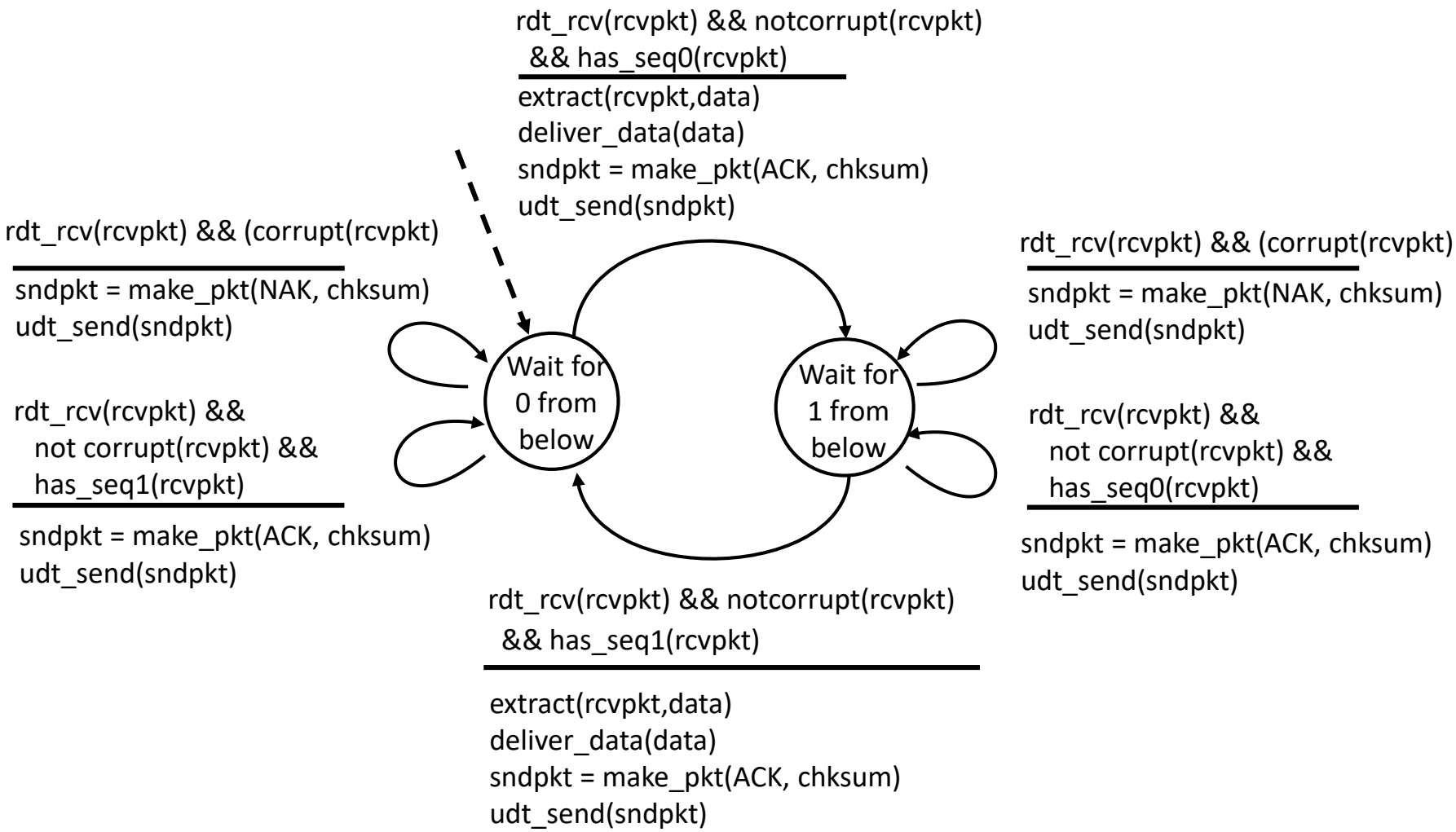
- What happens if ACK/NAK corrupted?
  - Sender doesn't know what happened at receiver!
  - Simple, just retransmit.

- How to handle duplicates?
  - 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



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 "current" pkt has 0 or 1 seq. #

Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected pkt seq #
  - For an out of order received packet, it sends ACK for it

- Note: Receiver can *not* know if its last ACK/NAK received OK at sender

Thank You!