Transport Layer Protocols -1

A/PROF. DUY NGO

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

Learning Objectives

3.1 transport-layer services

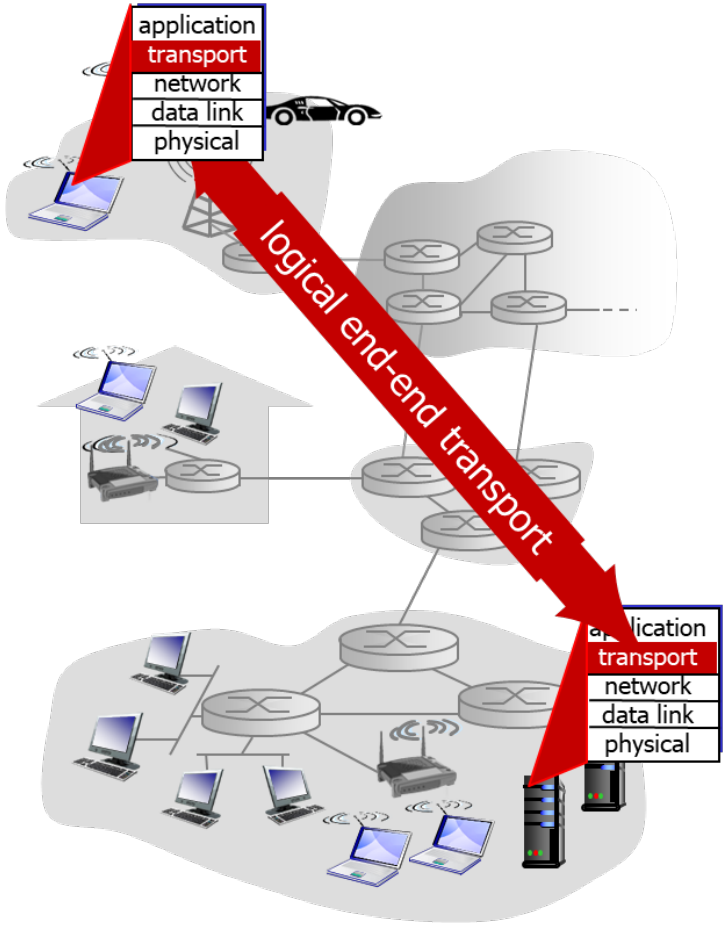
3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 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 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

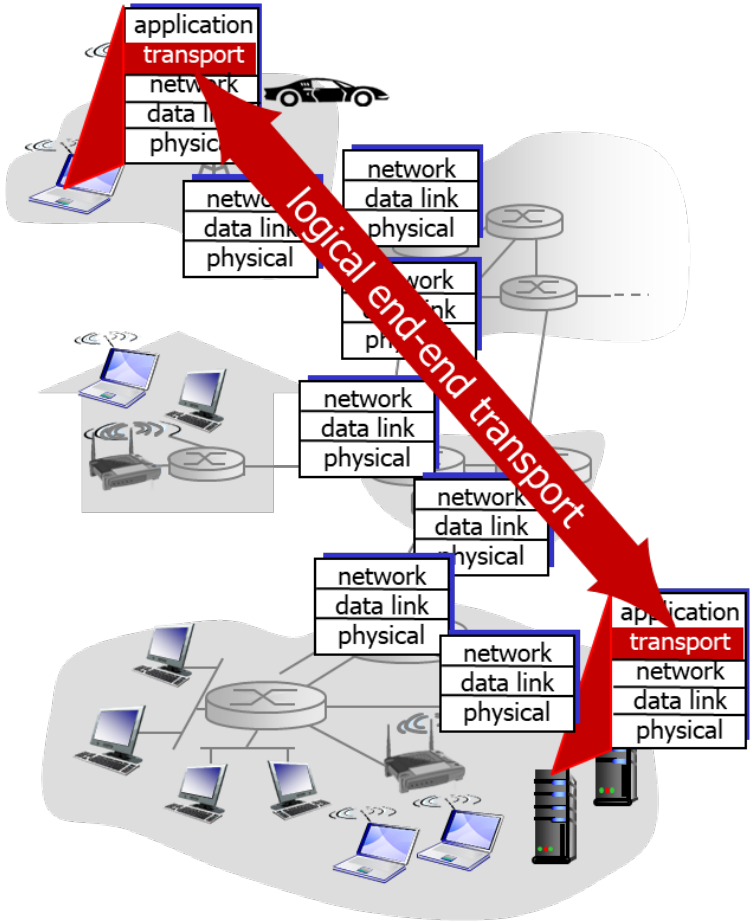
household analogy:

12 kids in house X sending letters to 12 kids in house Y:

- 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



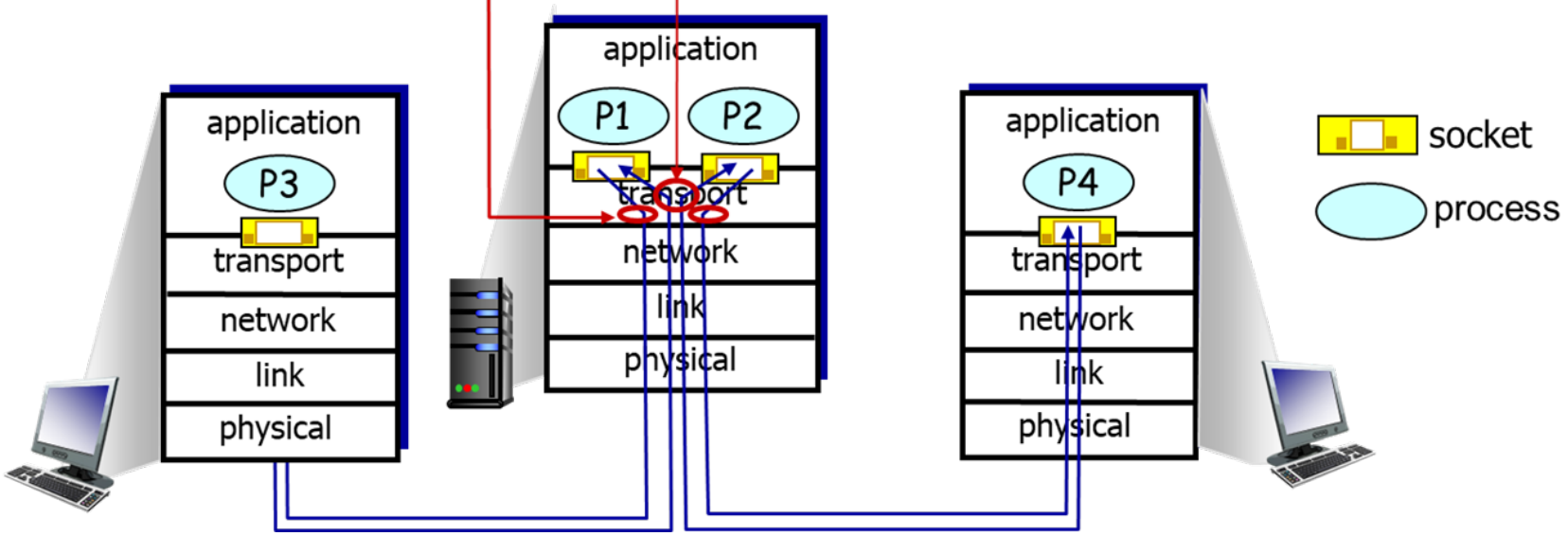
Multiplexing/Demultiplexing

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

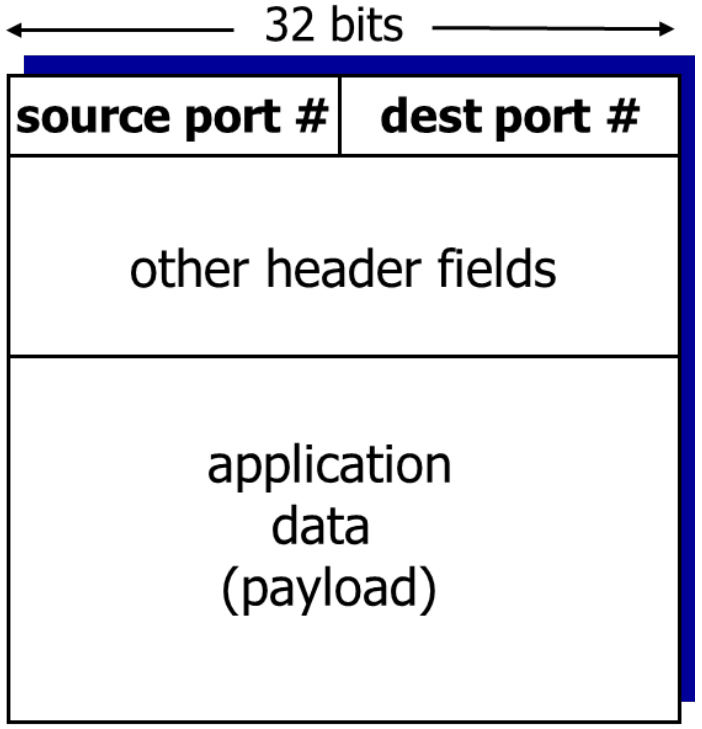


How Demultiplexing Works

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 #:

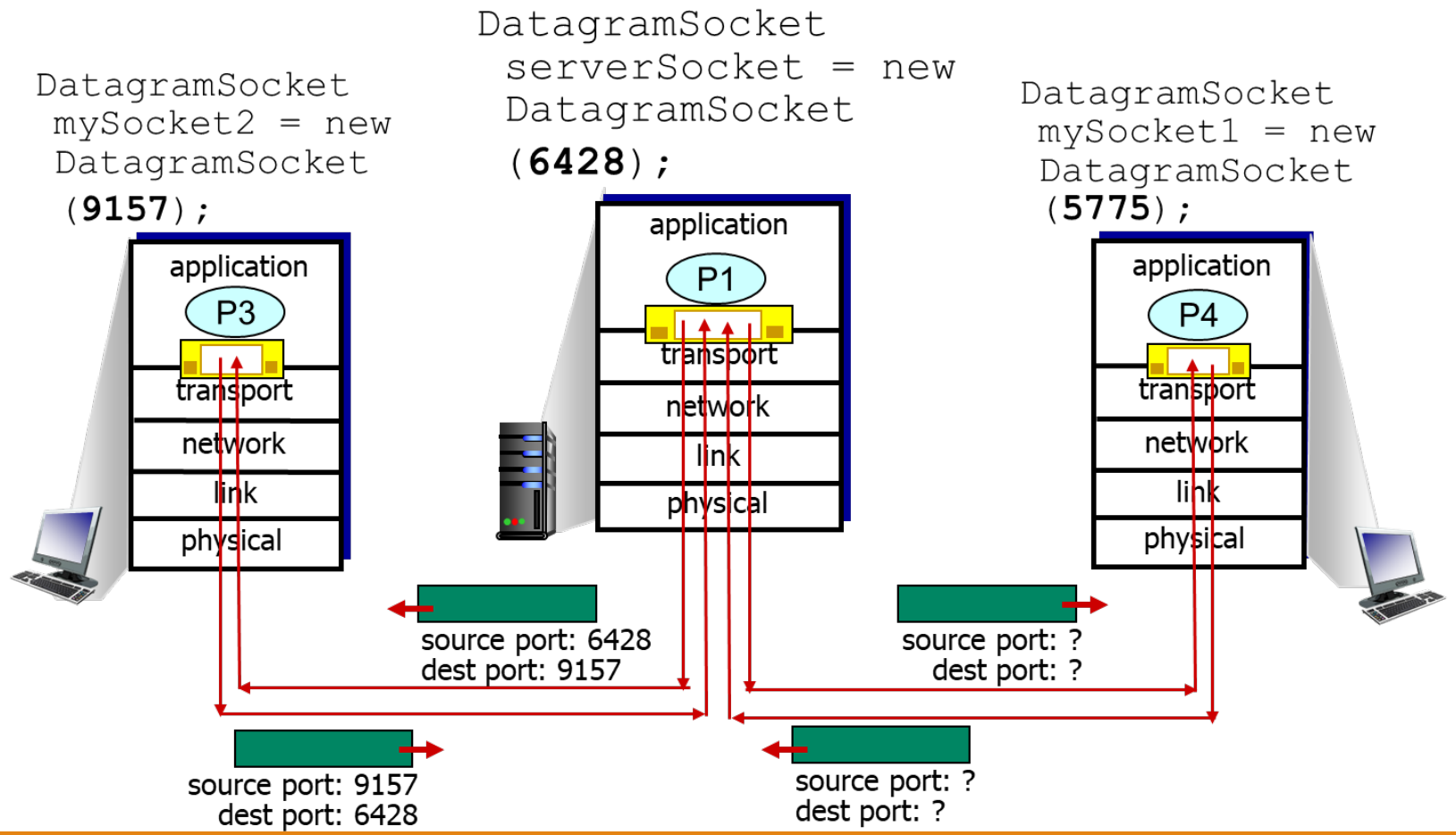
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

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

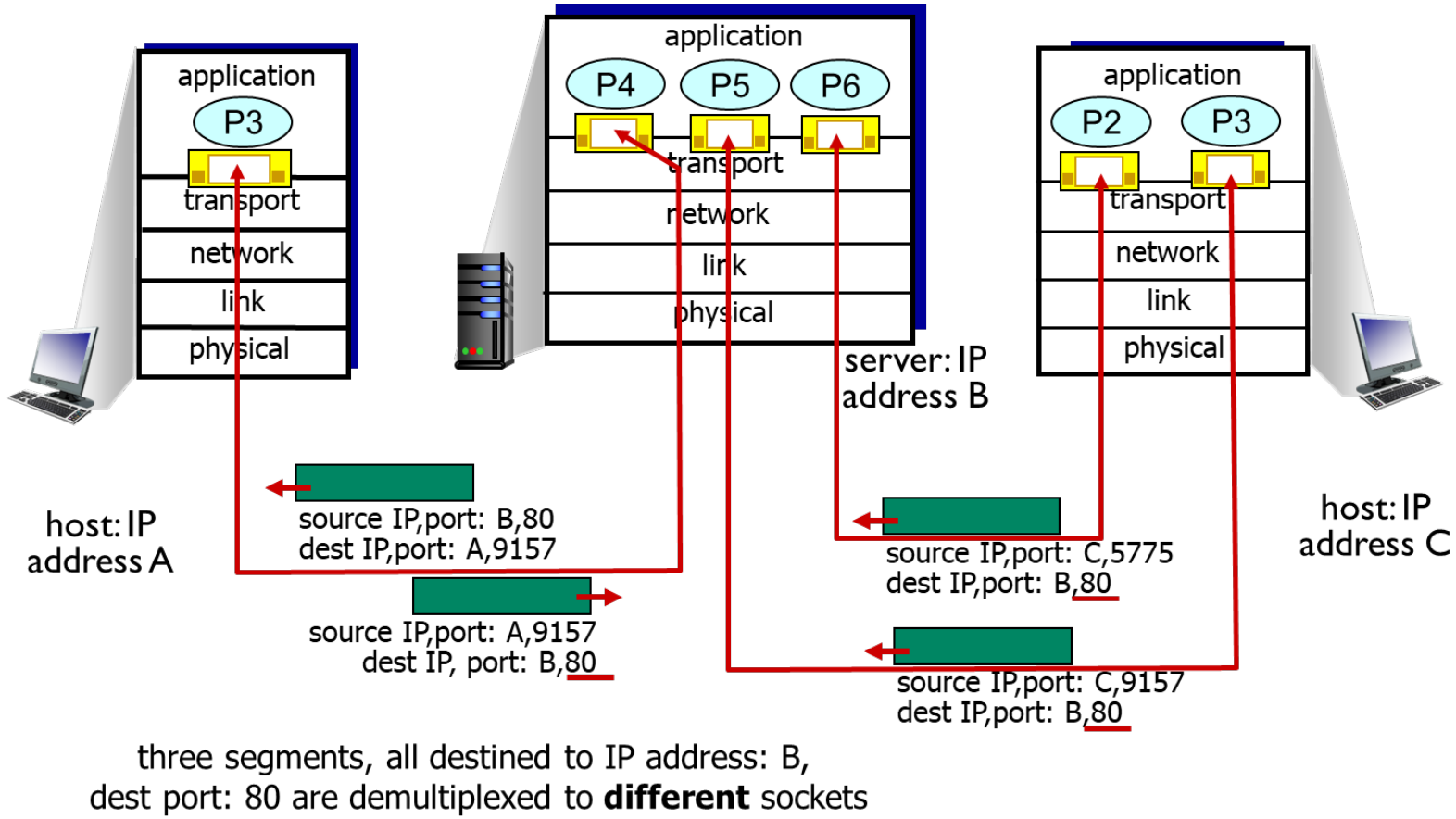
server host may support many simultaneous TCP sockets:

- each socket identified by its own 4-tuple

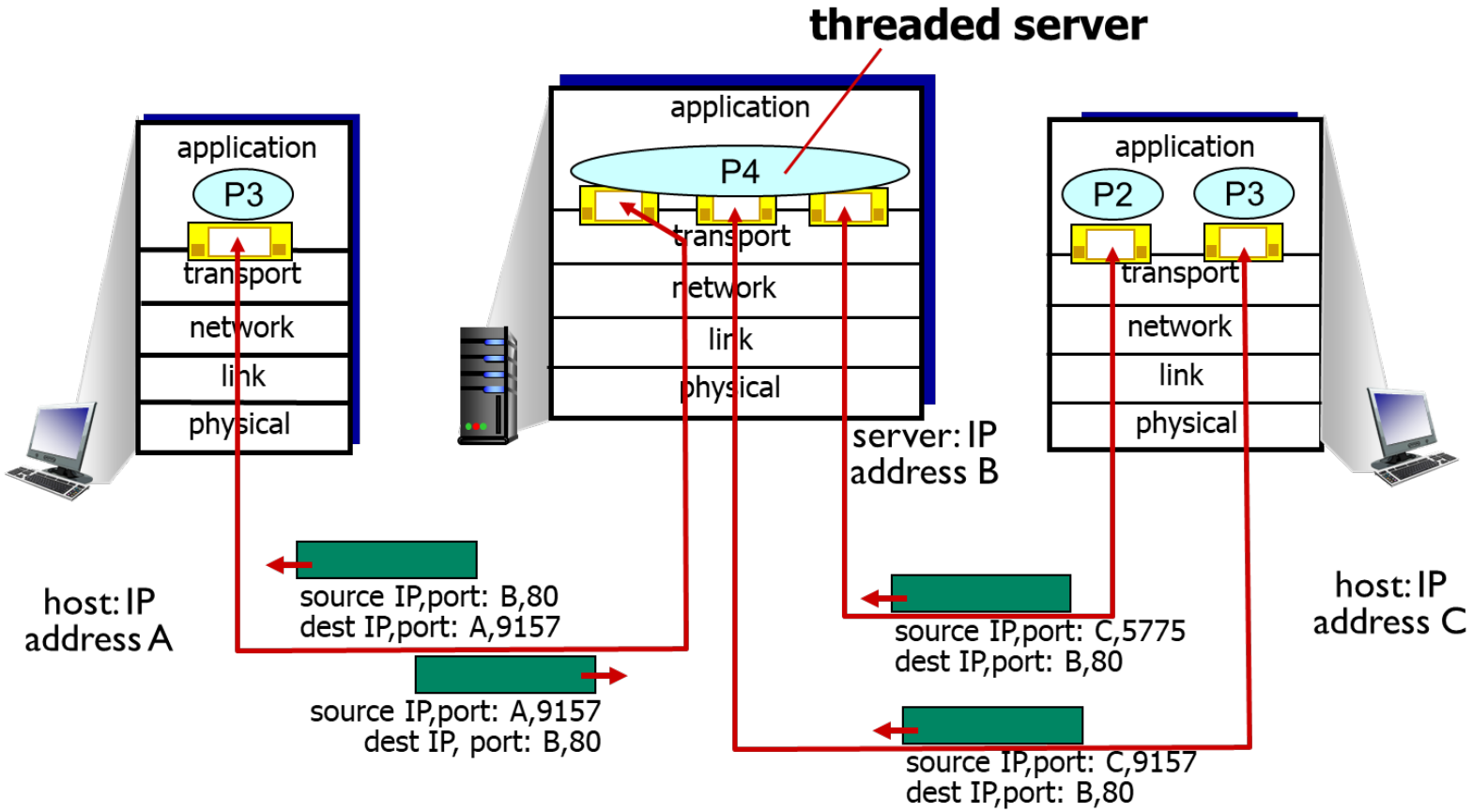
web servers have different sockets for each connecting client

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

Connection-Oriented Demux: Example (1 of 2)



Connection-Oriented Demux: Example (2 of 2)

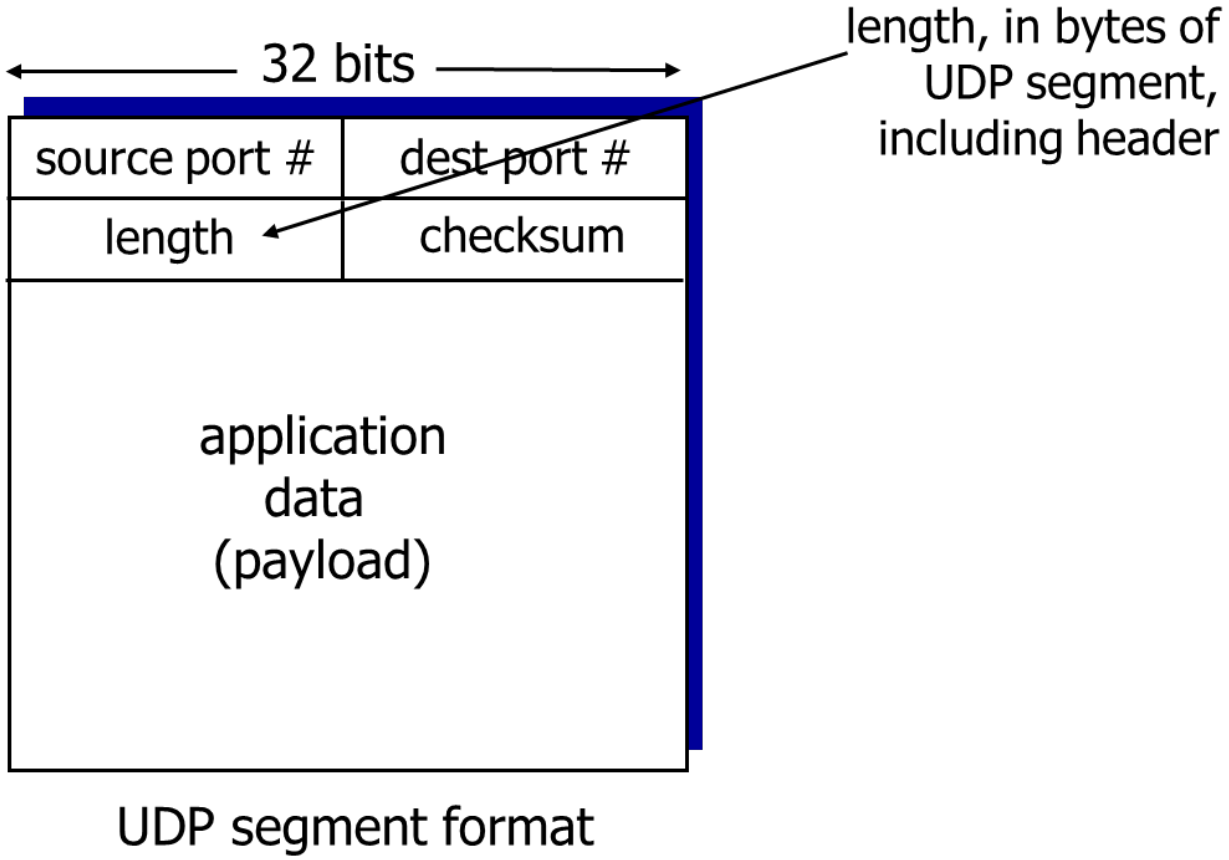


UDP: User Datagram Protocol [RFC 768]

- "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
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: Segment Header (1 of 2)



UDP segment format

UDP: Segment Header (2 of 2)

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" (example, flipped bits) in transmitted segment

sender:

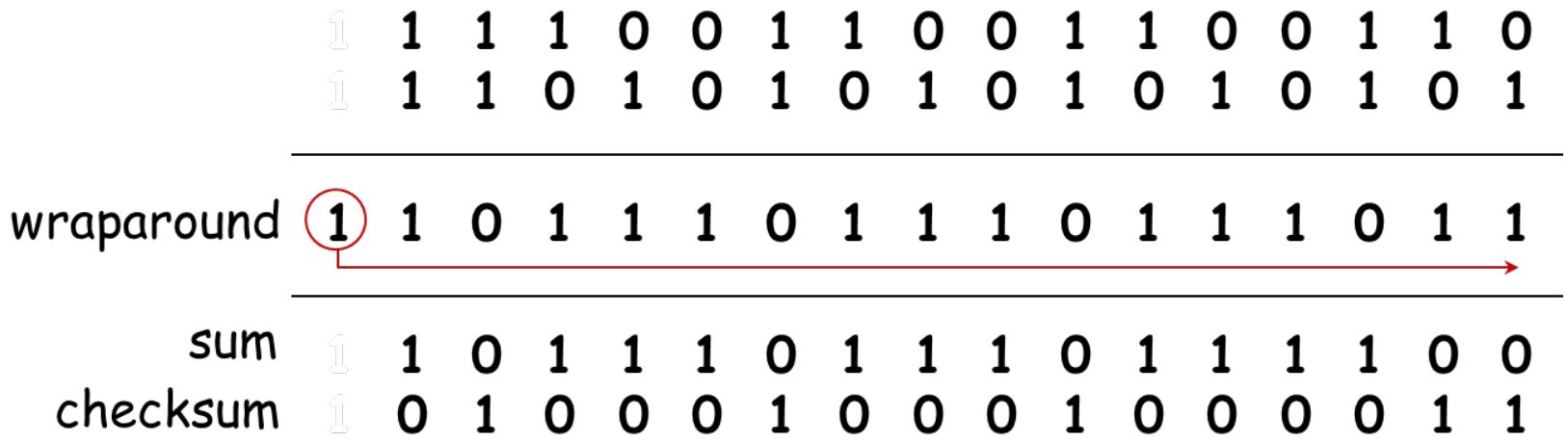
- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

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

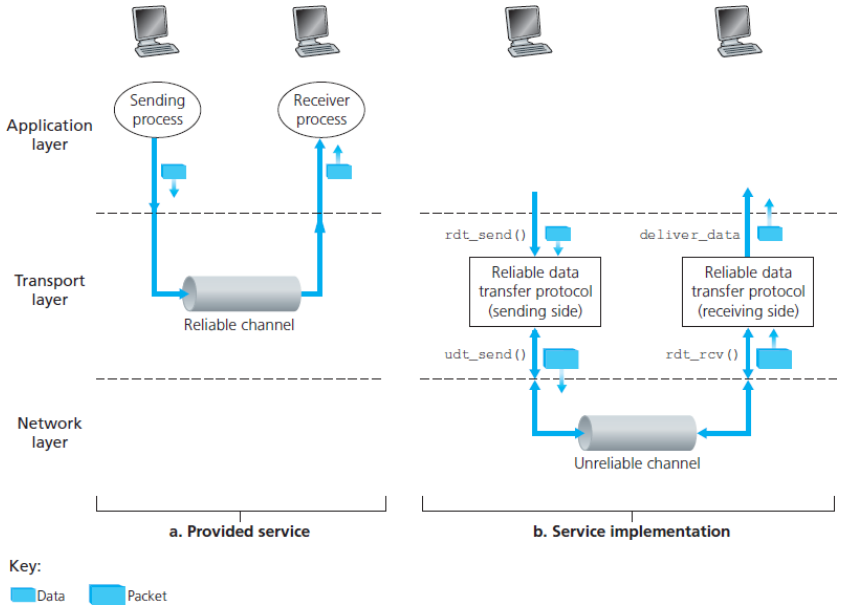
example: add two 16-bit integers



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

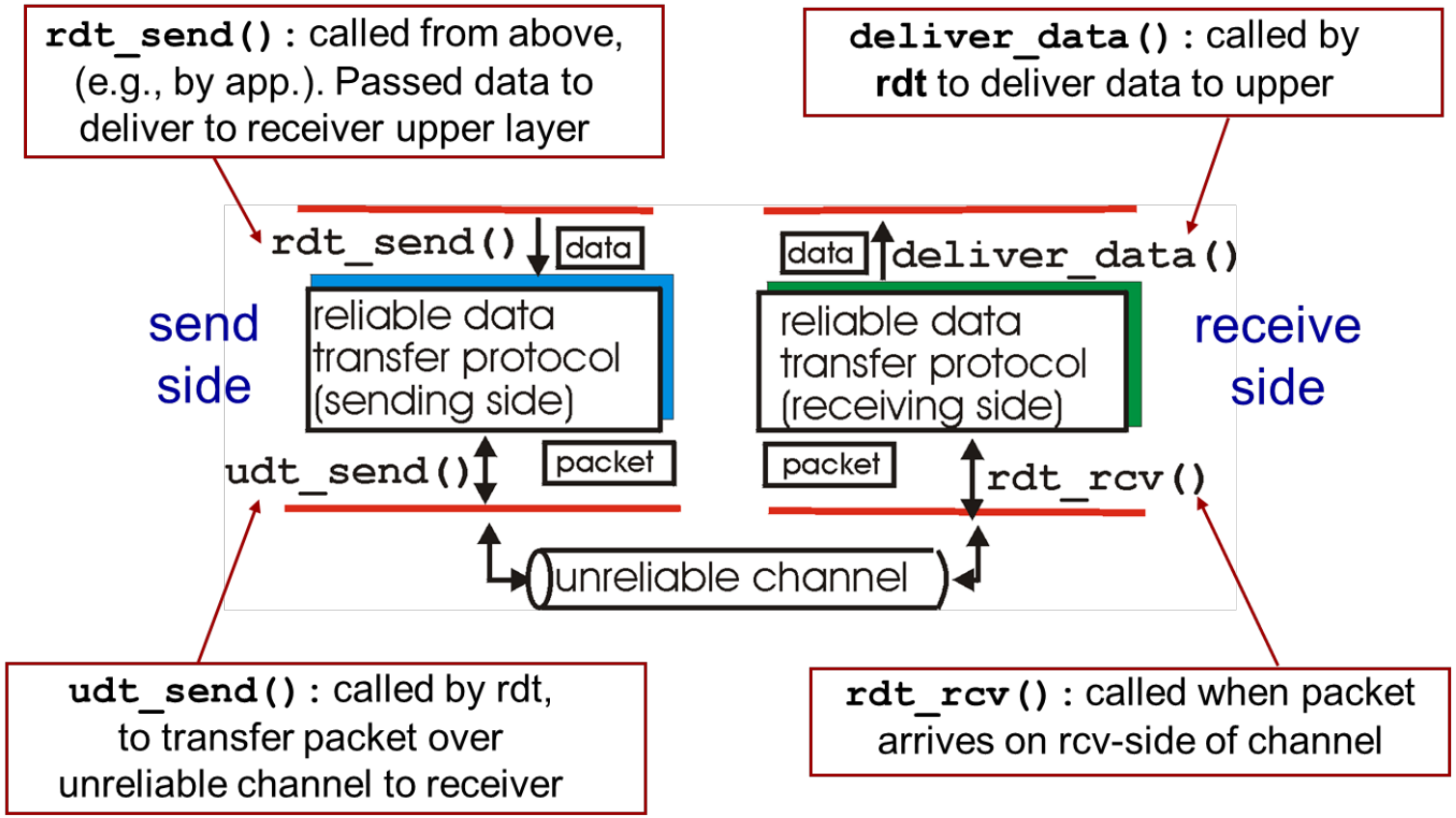
Principles of Reliable Data Transfer

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



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

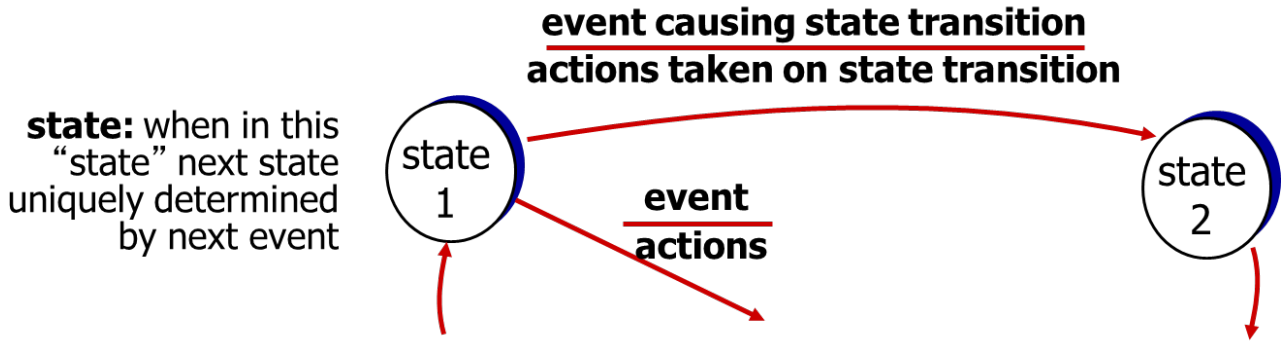
Reliable Data Transfer: Getting Started (1 of 2)



Reliable Data Transfer: Getting Started (2 of 2)

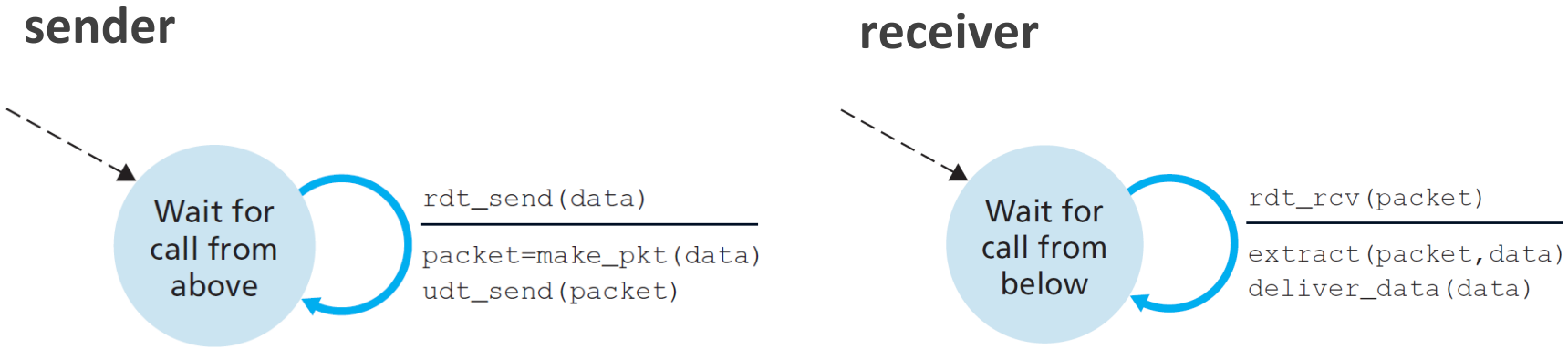
we'll:

- incrementally develop sender, receiver sides of **r**eliable **d**ata **t**ransfer 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 (1 of 2)

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

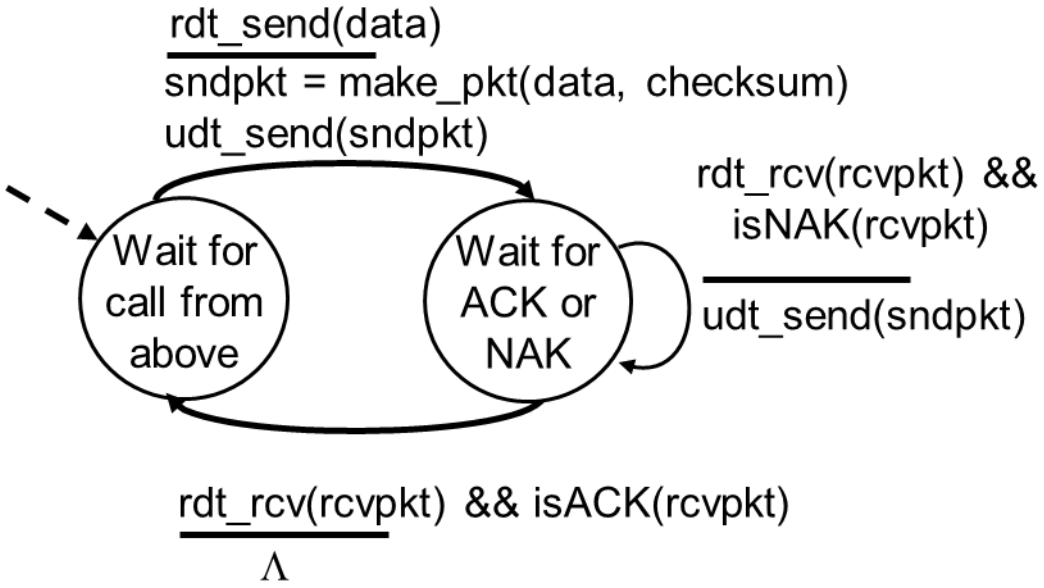
How do humans recover from "errors" during conversation?

rdt2.0: Channel with Bit Errors (2 of 2)

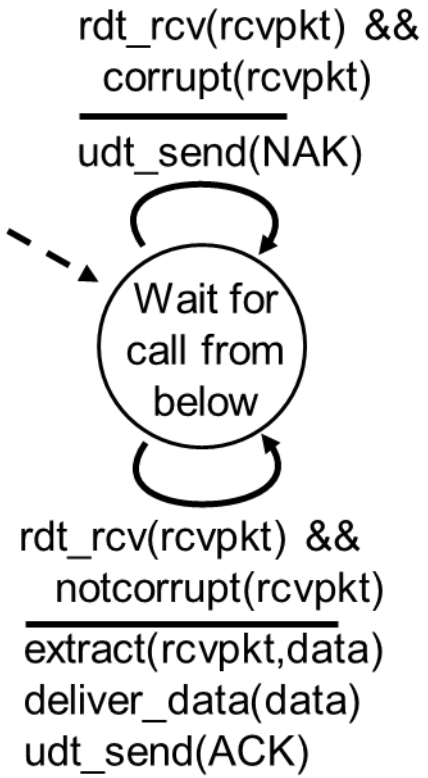
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- **the** question: how to recover from errors:
  - **acknowledgements (ACKs):** receiver explicitly tells sender that pkt received OK
  - **negative acknowledgements (NAKs):** receiver explicitly tells sender that pkt had 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 (Finite State Machine) Specification

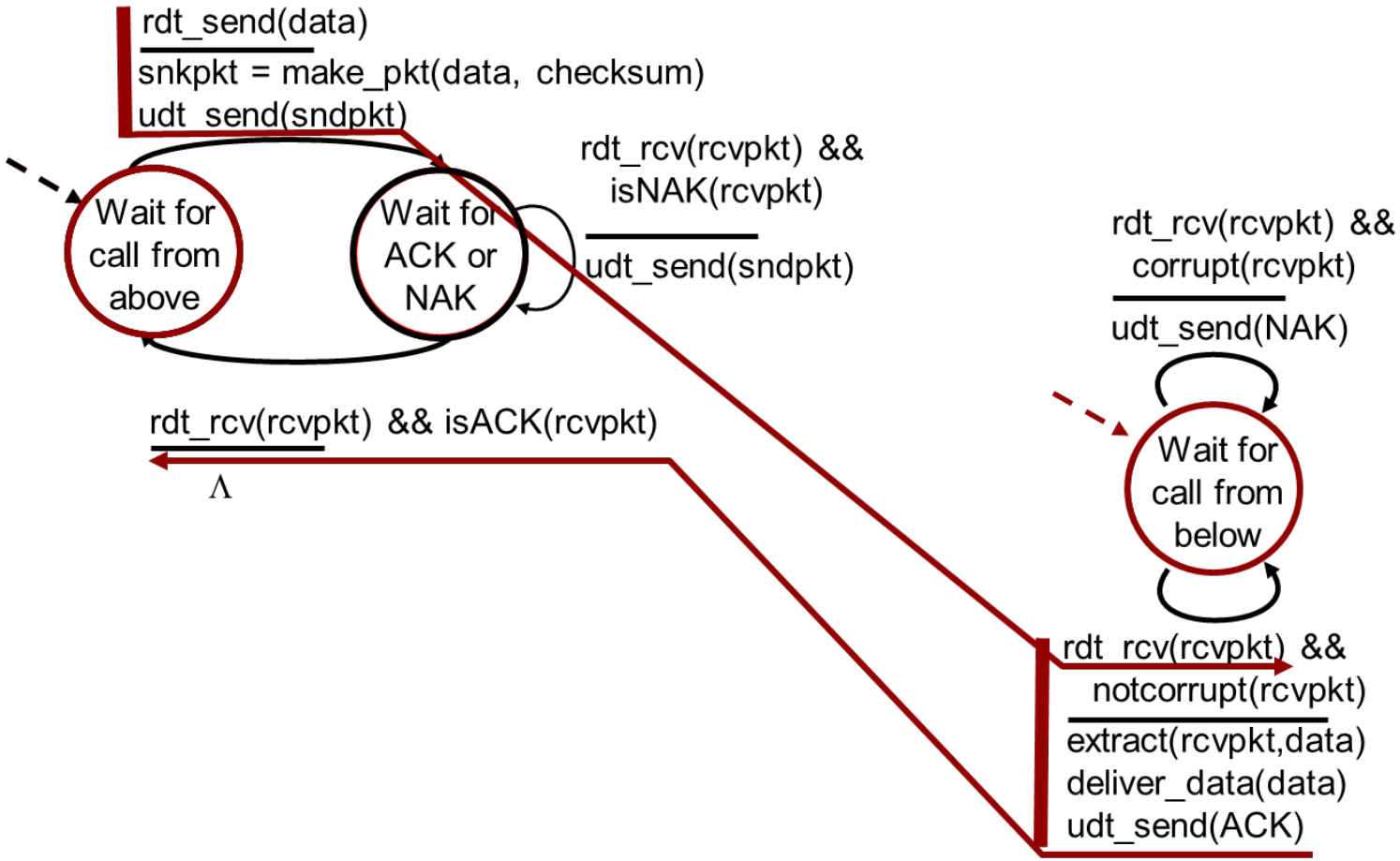
sender



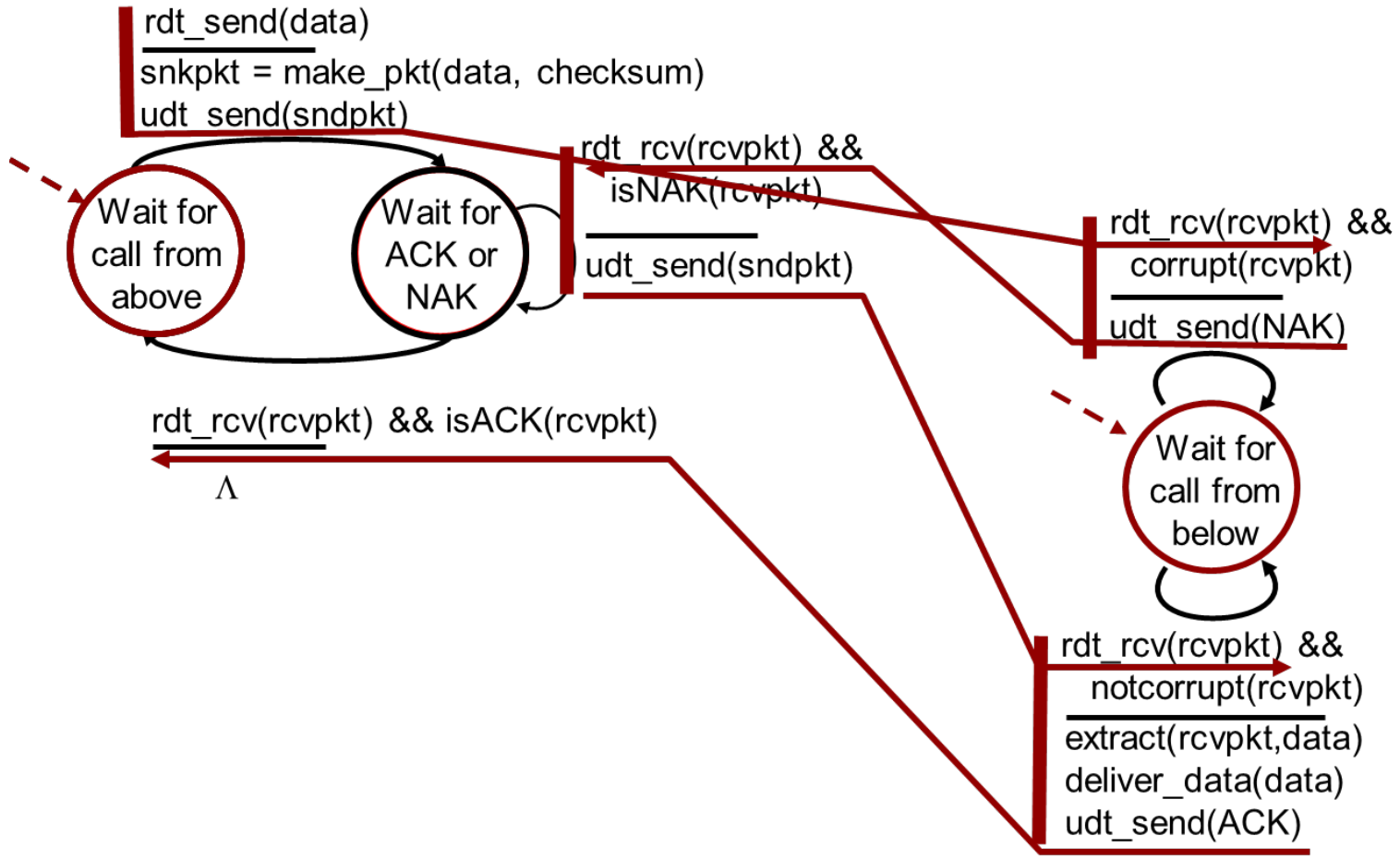
receiver



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

stop and wait

sender sends one packet, then waits for receiver response

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