Internet Video

What is Multimedia Streaming?

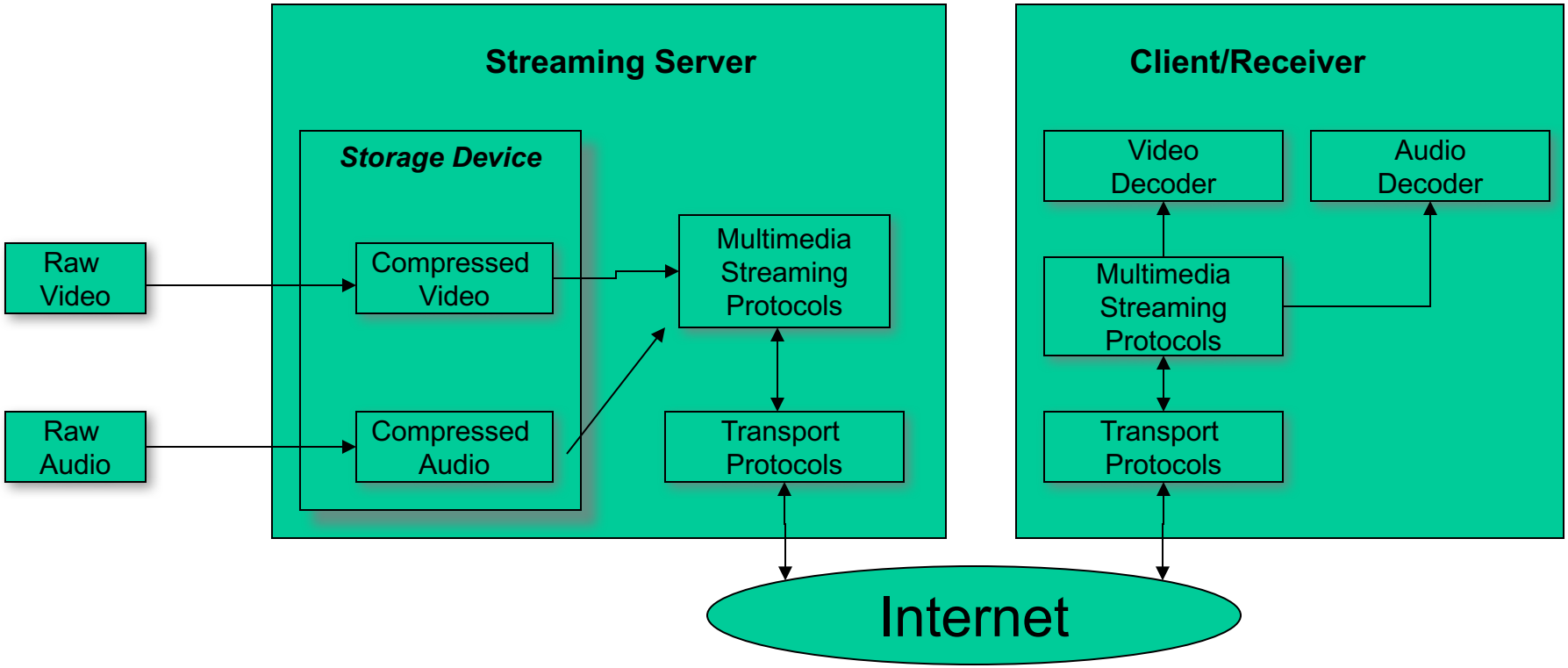
• Multimedia Streaming: Clients request audio/video files from servers and pipeline reception over the network and display

User's perspective:

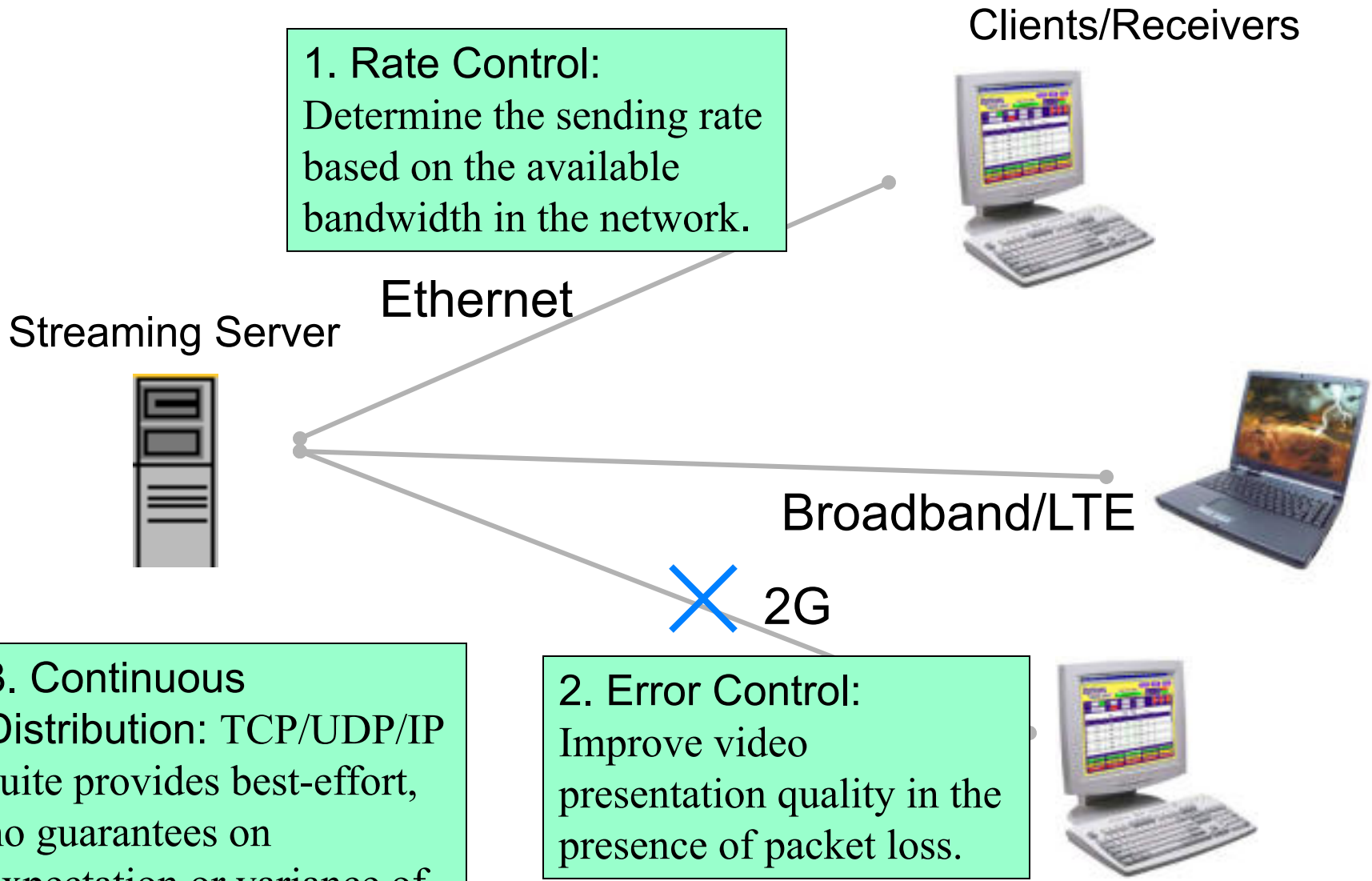
• Quick start without waiting for full download.

•Coming continuously without interruption.

•VCR operation (pause, resume, fast forward, rewind, etc.)



Challenges in Media Streaming Protocols



Techniques in Multimedia Streaming Protocols (1)

- Rate Control
  - Scalable compression
    - Base substream and enhancement substreams.
    - SNR scalability / spatial scalability / temporal scalability
  - Rate filter
    - Frequency filter
    - Frame-dropping filter
    - Re-quantization filter
  - QoS Feedback, e.g. RTCP.

Techniques in Multimedia Streaming Protocols (2)

- Error Control
  - Add redundant data in coding
    - MDC, (Multiple Description Coding)
    - FEC (Forward Error Coding)
  - Receiver End Error Concealment
    - Receiver conceal data loss.
    - Spatial interpolation, used in intra-coded frame.
    - Temporal interpolation, used in inter-coded frame.

A General View of the Classic Intenet Multimedia Streaming Protocols

- Stream description SDP, SMIL...

  Describe the session and content

- Stream control RTSP

  Remote control the session

- Media transport RTP

  Error control and flow control

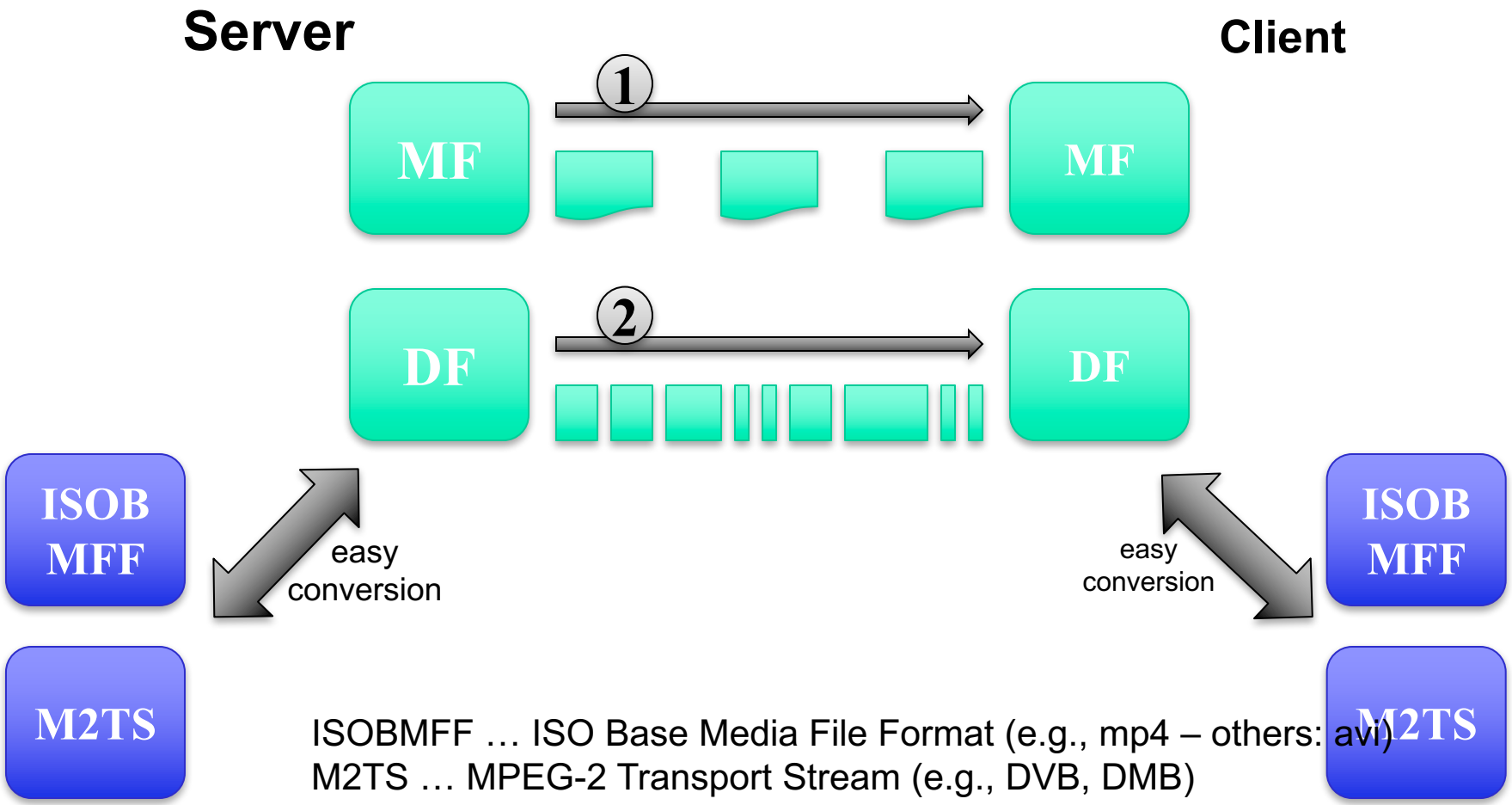
- Resource reservation (if any!): RSVP, DiffServ

  provide QoS for media streaming packets

HTTP-Based Internet Video

Use HTTP to scale video distribution

HTTP Streaming of Media



ISOBMFF … ISO Base Media File Format (e.g., mp4 – others: avi)
M2TS … MPEG-2 Transport Stream (e.g., DVB, DMB)
MF … Manifest Format (e.g., MPD, FMF)
DF … Delivery Format (e.g., F4F, 3gs)

Adaptive Streaming in Practice

move networks, inc.

HTTP Live Streaming

Microsoft Silverlight

Smooth Streaming

Adobe HTTP Dynamic Streaming

Combined A/V streams only

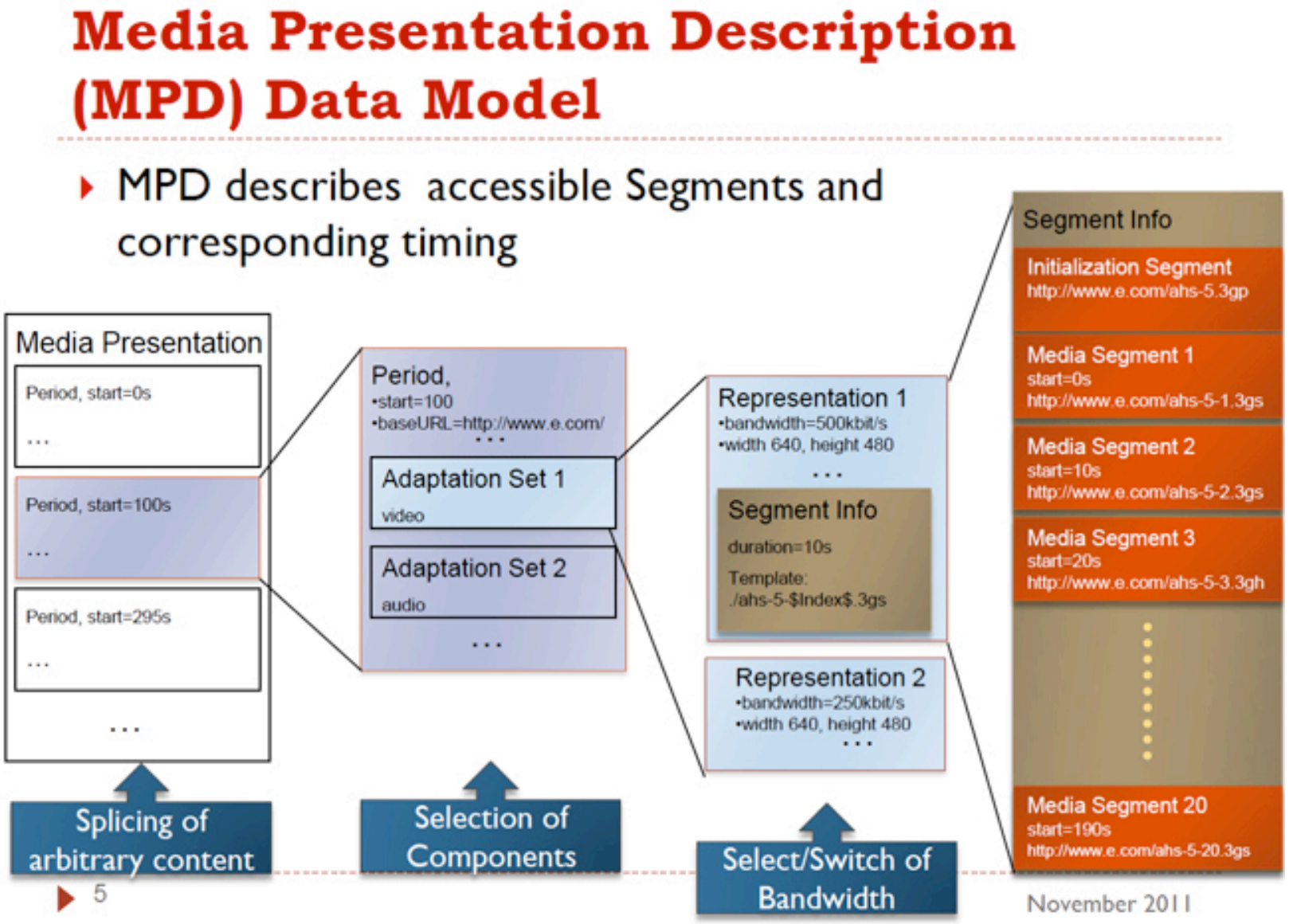
NETFLIX

MPEG DASH

Separate Audio/Video

Ack & ©: Mark Watson 2010/05/02

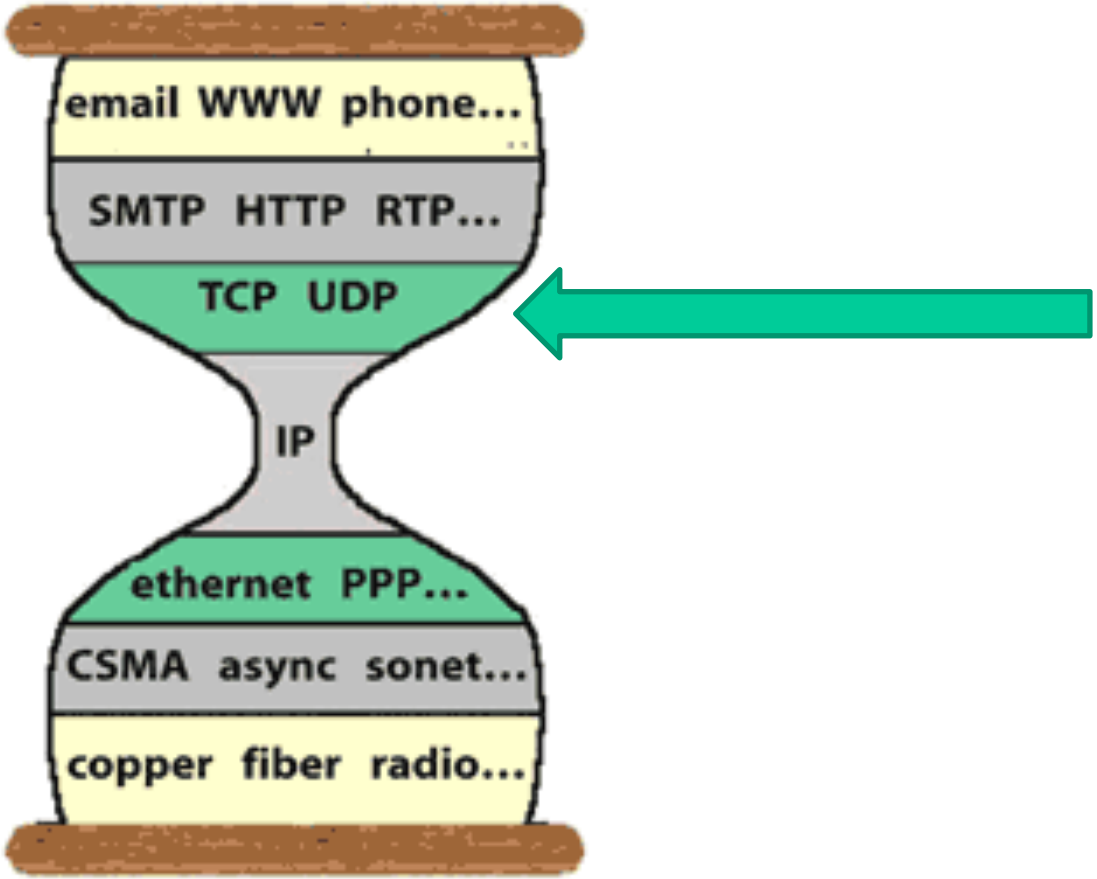
MPEG DASH Data Model



Media Presentation Description

- Redundant information of Media Streams for the purpose to initially select or reject Groups or Representations
  - Examples: Codec, DRM, language, resolution, bandwidth
- Access and Timing Information
  - HTTP-URL(s) and byte range for each accessible Segment
  - Earliest next update of the MPD on the server
  - Segment availability start and end time in wall-clock time
  - Approximated media start time and duration of a Media Segment in the media presentation timeline
  - For live service, instructions on starting playout such that media segments will be available in time for smooth playout in the future
- Switching and splicing relationships across Representations
- Relatively little other information

Moving to Transport Layer



Chapter 3

3.1 Transport-layer services

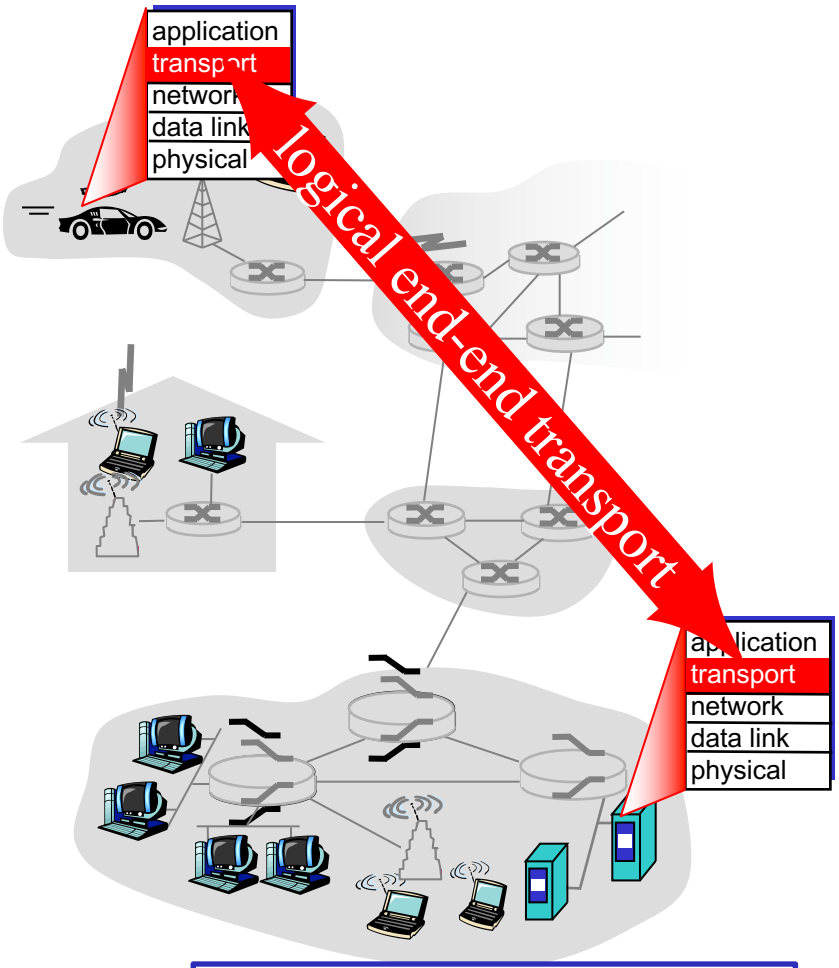
3.2 Multiplexing and de-multiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

Transport Layer

- Provide *logical communication* between app processes
- transport protocols run in end systems
  - send side: breaks app data into segments, passes to network layer
  - recv side: reassembles segments into original data format, passes to app layer
- Multiple transport protocols exist
  - Mostly used: TCP, UDP
  - Other transports: RTP, SCTP
  - Recent ones: QUIC, SPDY, HTTP/2



Check your `/etc/protocols`

Transport vs. network layer

- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services
- *network layer:* logical communication between hosts

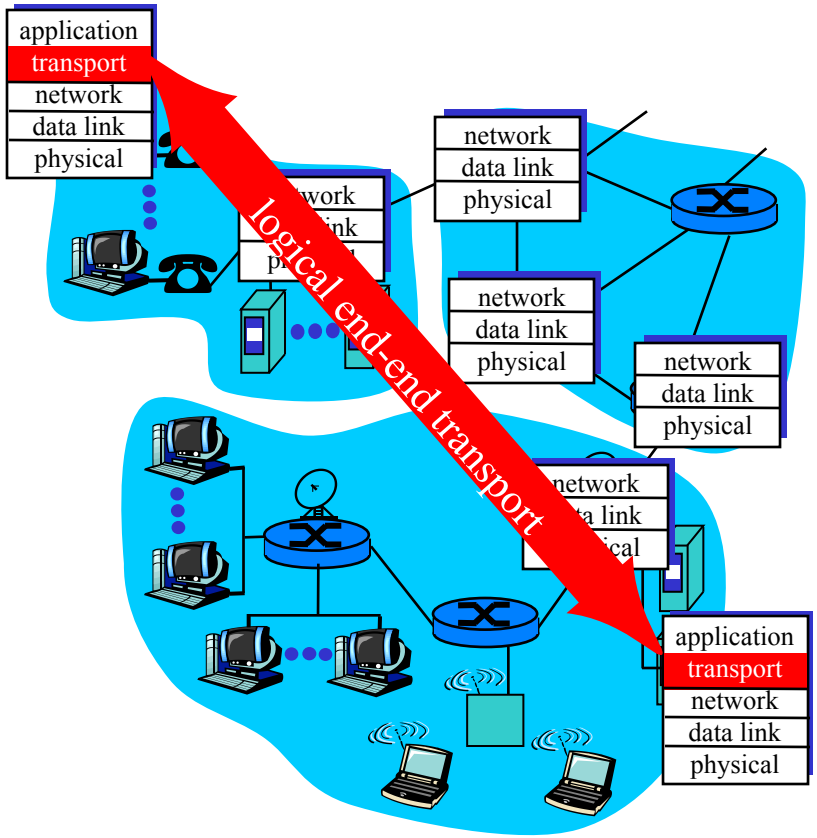
Household analogy:

12 young kids sending letters to 12 other kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = kids parents
- network-layer protocol = postal service

Internet transport-layer protocols

- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery: UDP
- FYI:
  - RTP: unreliable delivery with timing information
  - SCTP: reliable delivery of multiple substreams

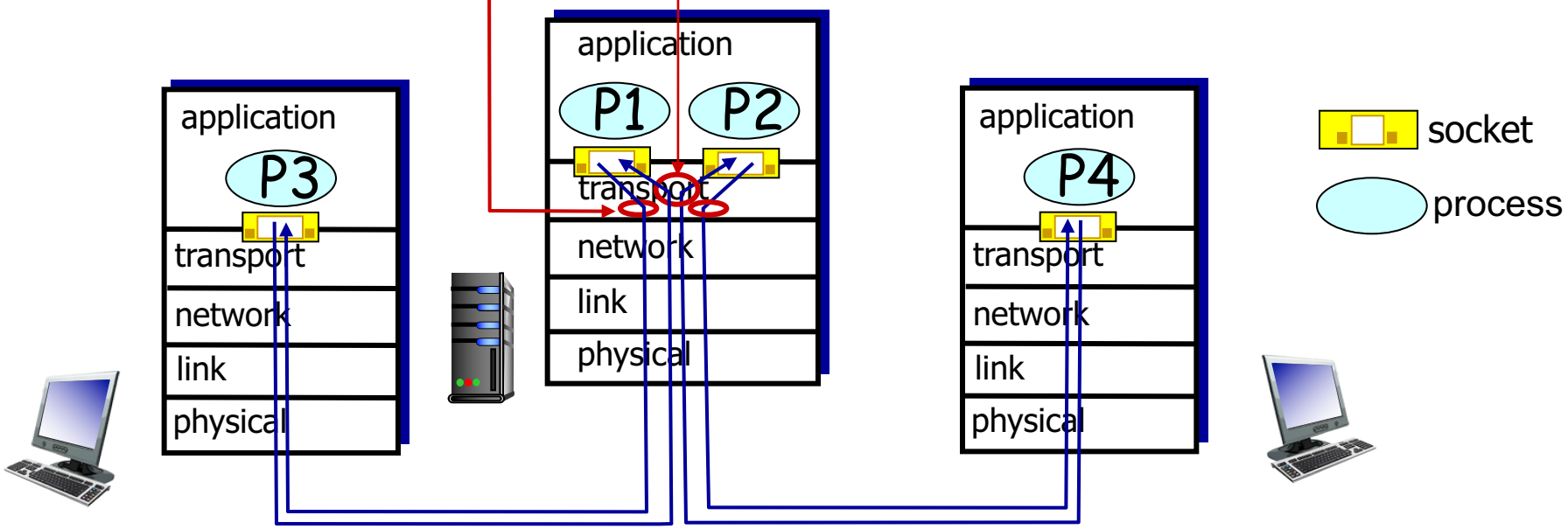


Common function among all the above:
multiplexing/demultiplexing

Multiplexing/demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

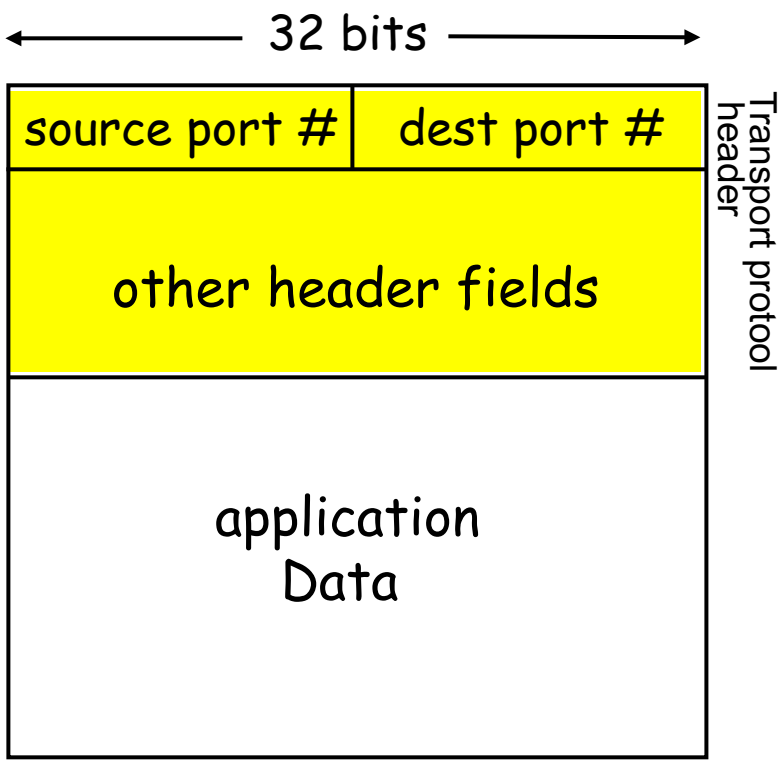
demultiplexing at receiver:
use header info to deliver received segments to correct socket



Each process is identified by IP address and port#

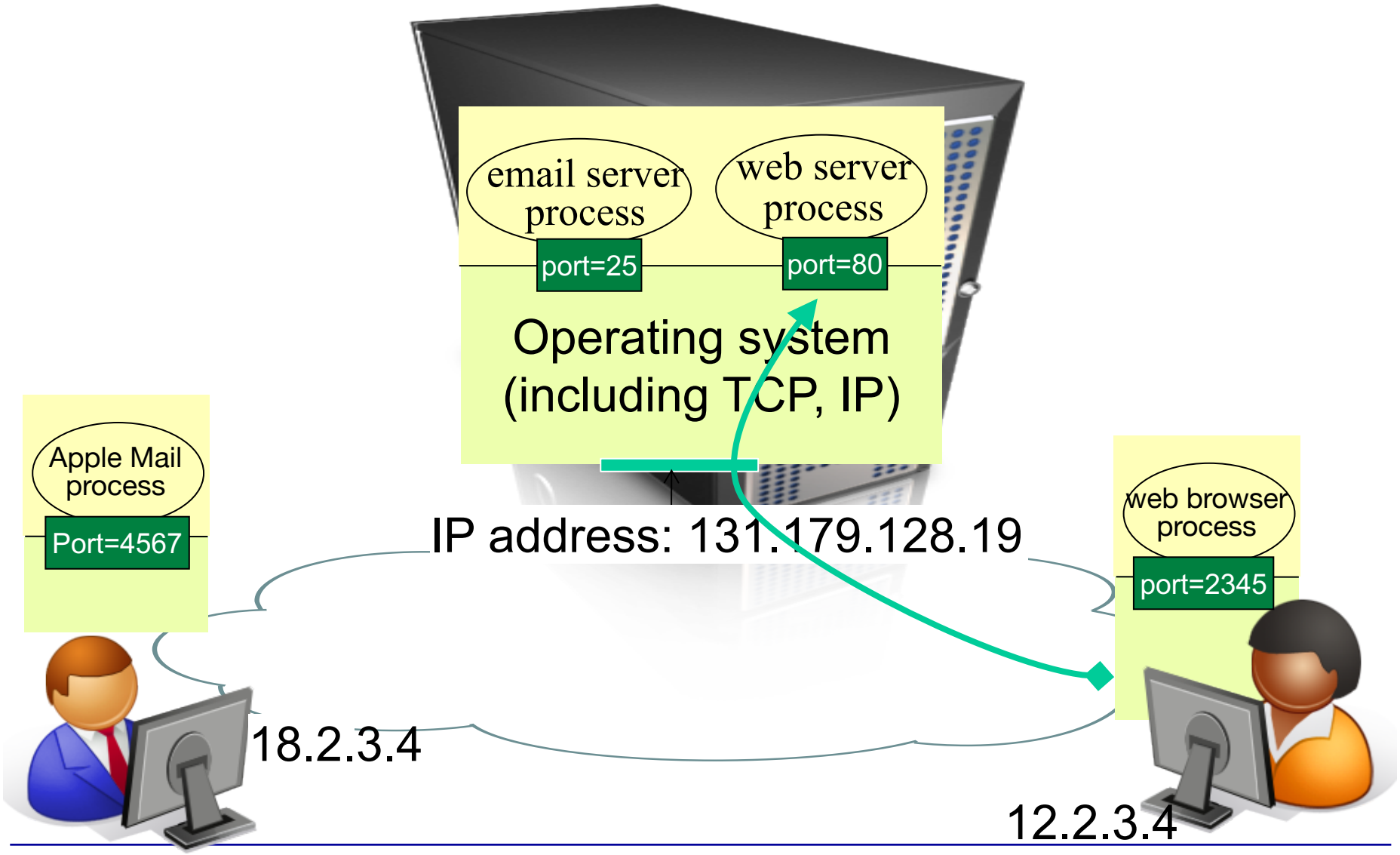
How Demultiplexing Works

- 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

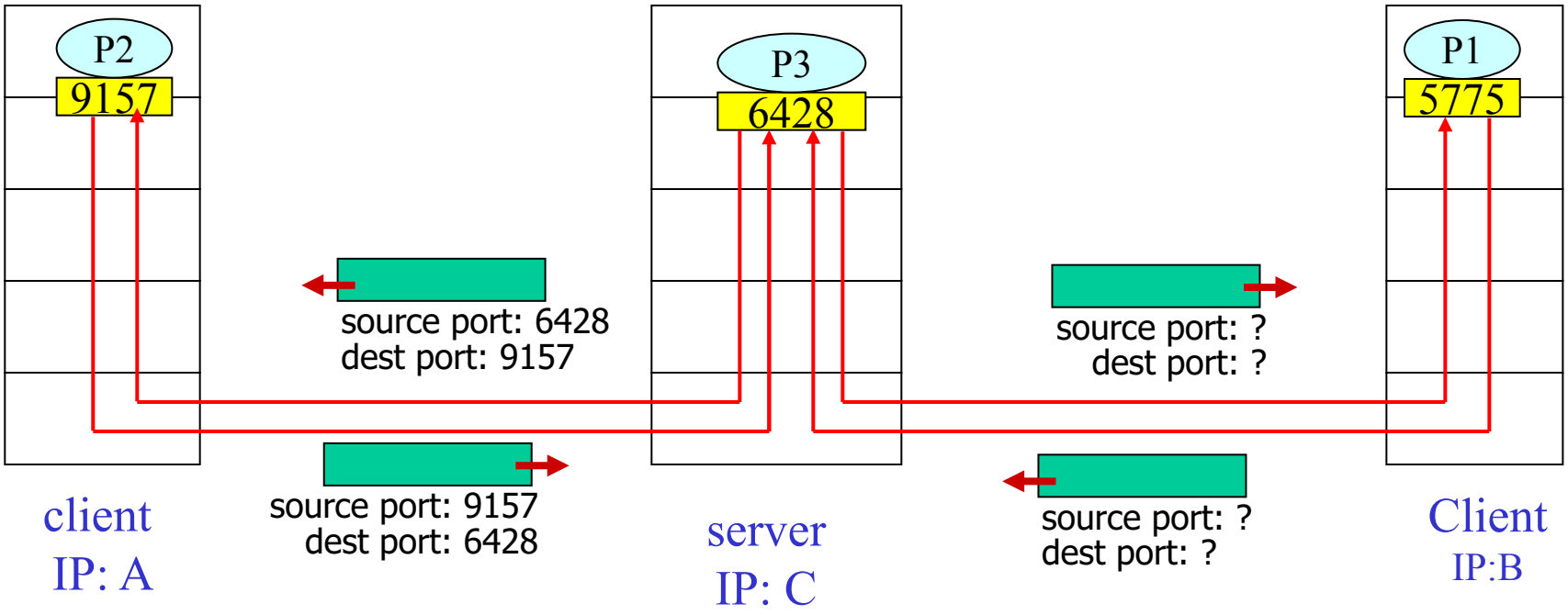
IP address, TCP connection, port number, processes, and sockets



Connectionless Demultiplexing

- when creating a packet to send to a UDP socket, must specify
  - destination IP address
  - destination port #
- When host receives a UDP packet:
  - checks destination port# in the packet
  - directs the packet to socket with that port#
- IP packets with *same dest. port #* are directed to the same socket at the destination host
  - They may have different source IP addresses and/or source port#s

Connectionless demux (cont)



How can a server know where to return a reply?

FYI: From UDP spec (RFC768) "UDP module must be able to determine the source and destination internet addresses and the protocol field from the internet header. One possible UDP/IP interface would return the whole internet datagram including all of the internet header in response to a receive operation"

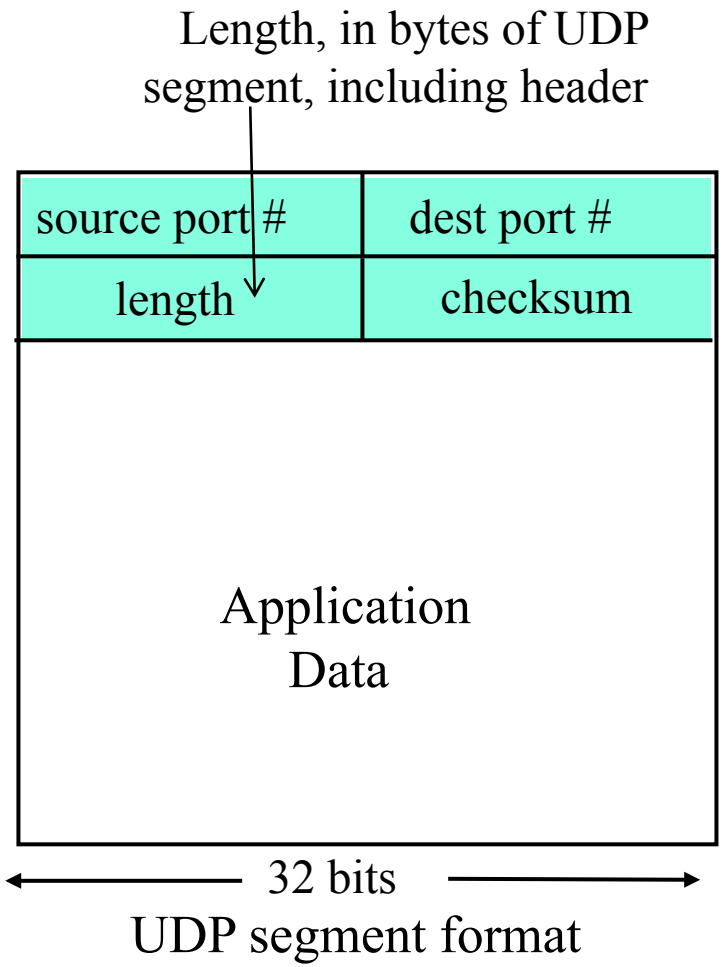
UDP: User Datagram Protocol [RFC 768]

- Best effort service: UDP segments may be lost, duplicated, or delivered out of order to app. processes
- *connectionless*:
  - no prior handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- UDP use:
  - DNS
  - streaming multimedia apps (loss tolerant, rate sensitive)
- For reliable transfer: add reliability at application layer

UDP header format

Why is there a UDP?

- no connection establishment
- simple: no connection state at sender, receiver
- small header size
- ⬇ no congestion control: UDP can blast away as fast as desired



UDP segment format

UDP checksum

Goal: detect bit errors in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

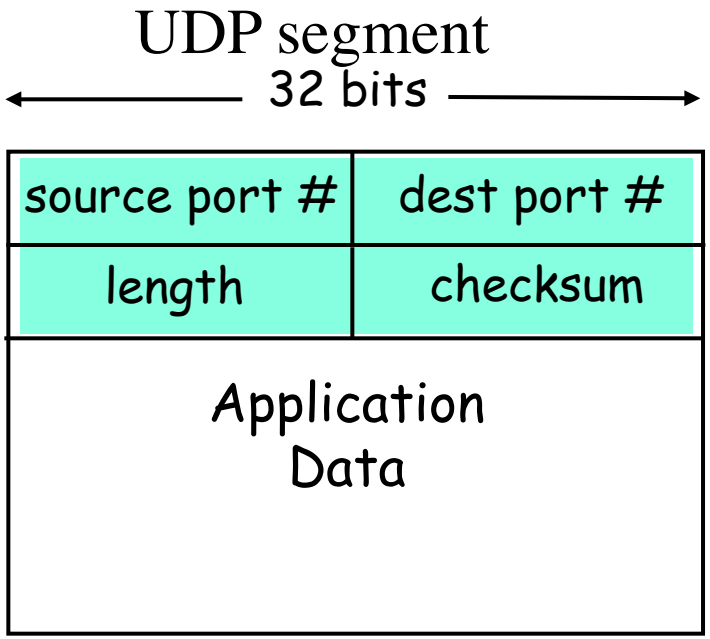
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error *detected*

UDP Checksum

- https://en.wikipedia.org/wiki/User_Datagram_Protocol#Checksum_computation
- The 16-bit checksum field is used for error-checking received packet
- **16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets**
  - *For computing, the value of the checksum field is zero*

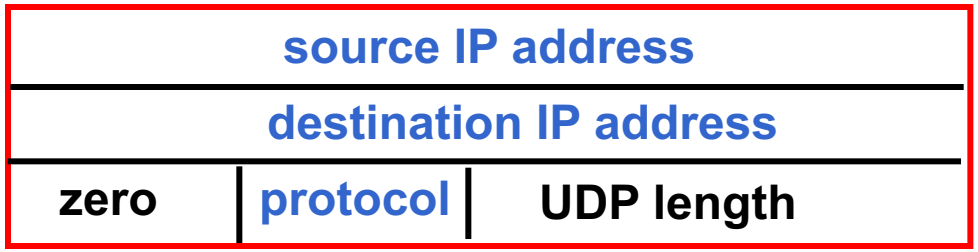
How to Calculate UDP Checksum

- UDP header
  - Length: # of bytes (including both header & data)
  - checksum: computed over
    - the pseudo header, and
    - UDP header and data.
    - if checksum field=0: no checksum



- pseudo header: UDP's self-protection against misdelivered IP packets

  pseudo header is not carried in UDP packet, nor counted in the length field



IPv4 Checksum

- https://en.wikipedia.org/wiki/IPv4#Header_Checksum
- The 16-bit checksum field is used for error-checking of the IPv4 header
  - If value of the carried checksum don't match the calculated, the router discards the packet
  - Different from TCP and UDP checksums
- ***16-bit one's complement of the one's complement sum of all 16-bit words in the IPv4 ***header****
  - *For computing, the value of the checksum field is zero*

IPv4 Checksum Example

- 4500 0030 4422 4000 8006 0000 8C7C 19AC AE24 $1E2B_{16}$ (20 bytes IP header)
- Sum up every 16 bits
  - $4500_{16} + 0030_{16} + 4422_{16} + 4000_{16} + 8006_{16} + 0000_{16} + 8C7C_{16} + 19AC_{16} + AE24_{16} + 1E2B_{16} = 0002BBCF$ (32-bit sum)
- 1's complement 16-bit sum (carry out)
  - $0002_{16} + BBCF_{16} = BBD1_{16} = 1011101111010001_2$
  - Can do this carry out while summing up
- 1's complement of 1's complement 16-bit sum
  - $\sim BBD1_{16} = 0100010000101110_2$

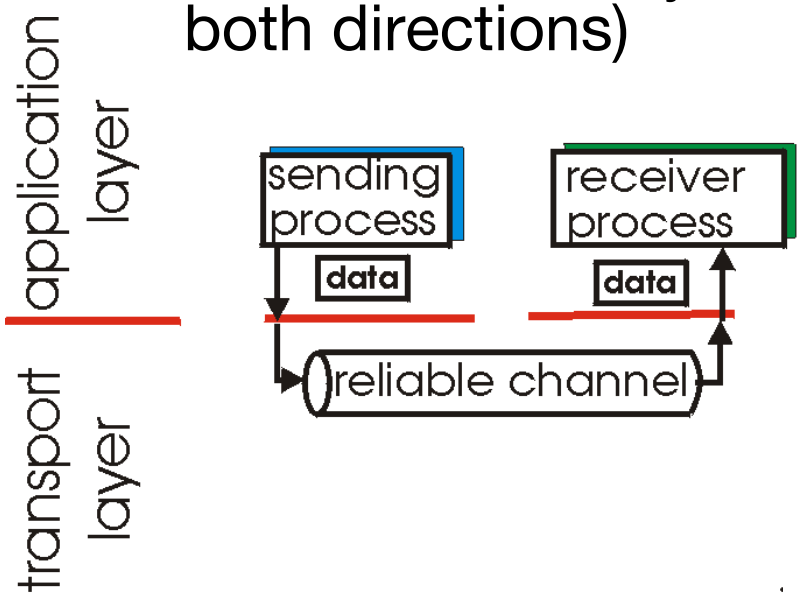
http://www.thegeekstuff.com/2012/05/ip-header-checksum/

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Principles of Reliable Data Transfer

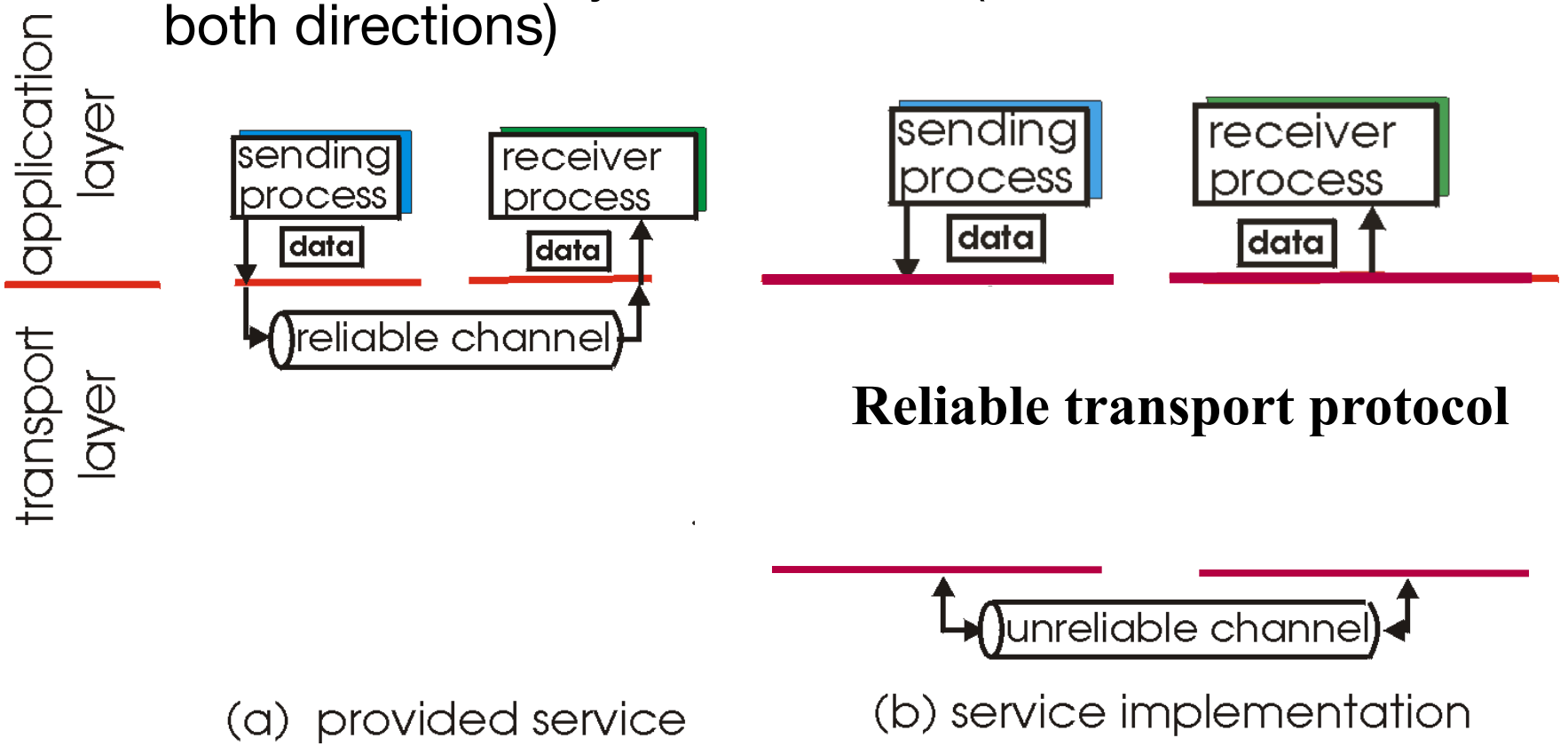
- characteristics of unreliable channel determines complexity of a reliable data transfer protocol (rdt)
- incrementally develop sender, receiver sides of rdt
  - consider one-way data transfer (control info will flow in both directions)



(a) provided service

Principles of Reliable Data Transfer

- characteristics of unreliable channel determines complexity of a reliable data transfer protocol (rdt)
- incrementally develop sender, receiver sides of rdt
  - consider one-way data transfer (control info will flow in both directions)



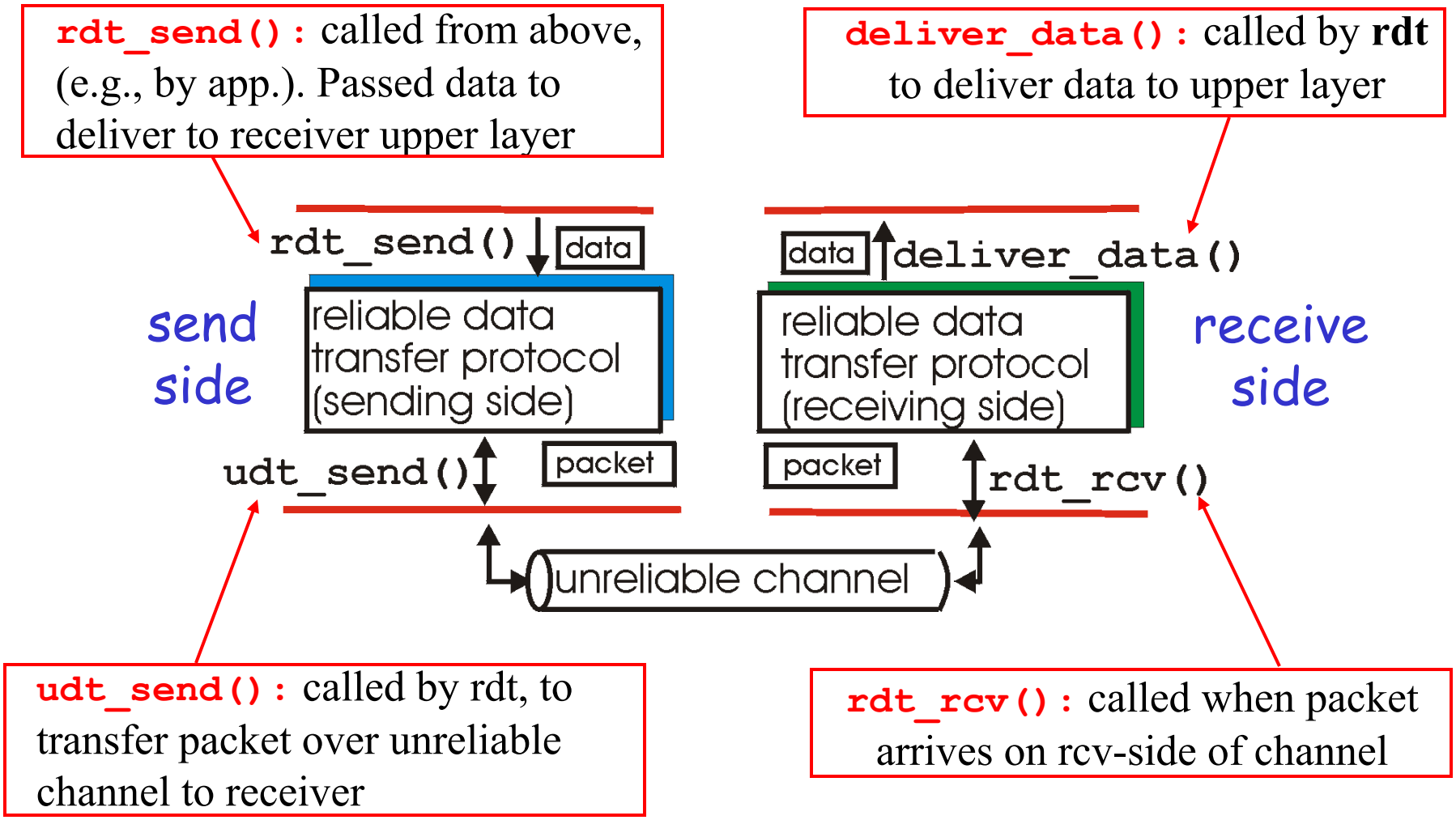
A simplified version of Principles of Reliable Data Transfer

- 3 questions
  - How many different types of errors?
  - How to detect each?
  - How to recover?
- 3 types of errors, and how to detect each
  - Corrupted bits in a packet: detected by checksum
  - Packet loss: detected by alarm timer (at sender end) in absence of ACK
  - Packets arrived out of order: detected by assigning each packet a sequence number
- Recovery: retransmitting the error/lost packet

Three basic components in reliable data delivery by retransmission

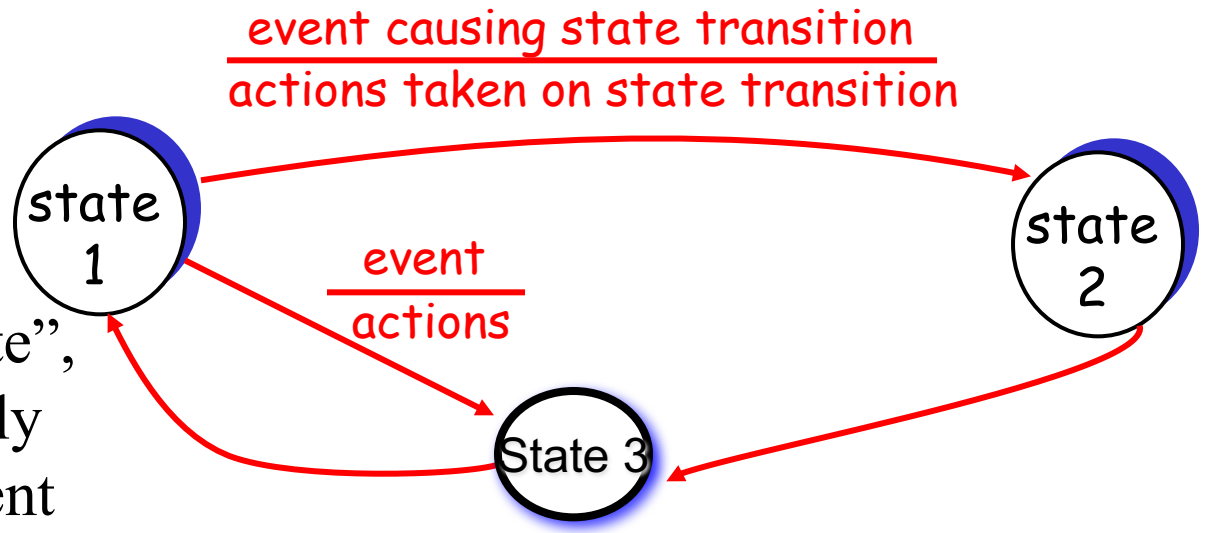
- sequence #: used to uniquely identify individual piece of data
- Acknowledgment (ACK): reception report sent by receiver to the sender
- Retransmission timer set by the sender for the already sent, but has not been acknowledged packet
  - Retransmit the packet when timer expires

Reliable Data Transfer (rdt): getting started



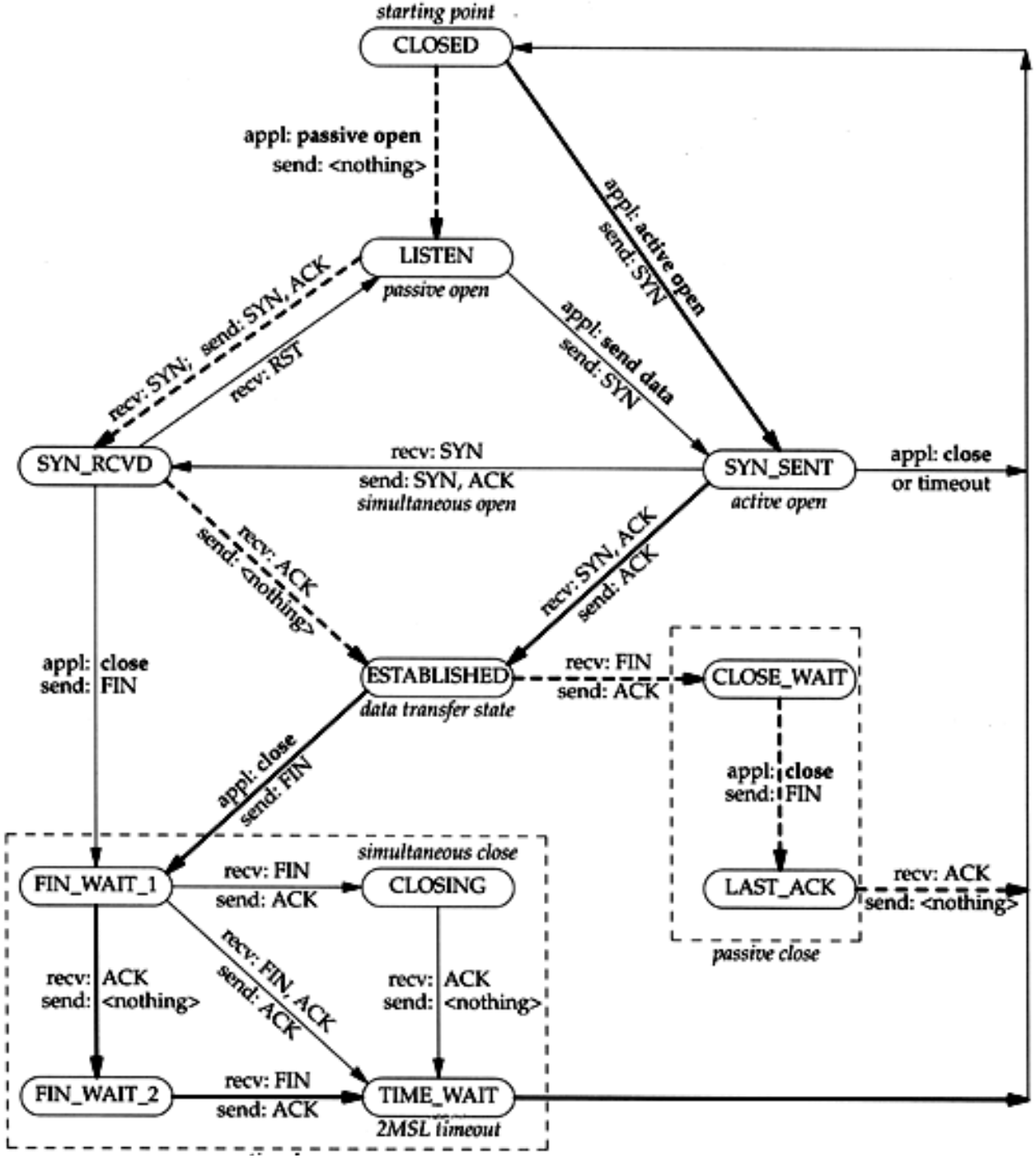
Reliable data transfer: getting started

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



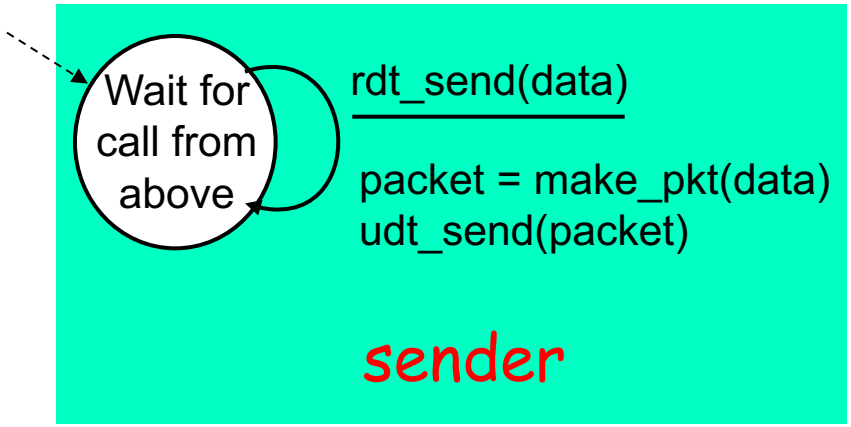
state: when in this "state", the next state is uniquely determined by next event

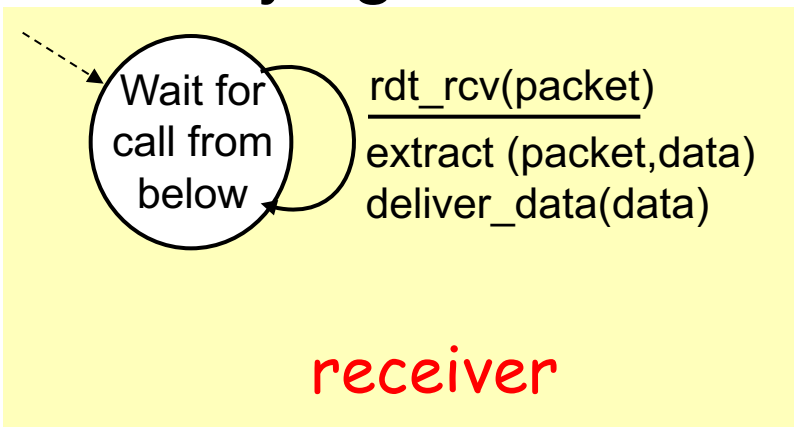
Very "simple" state diagram of TCP



Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no packet losses
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel

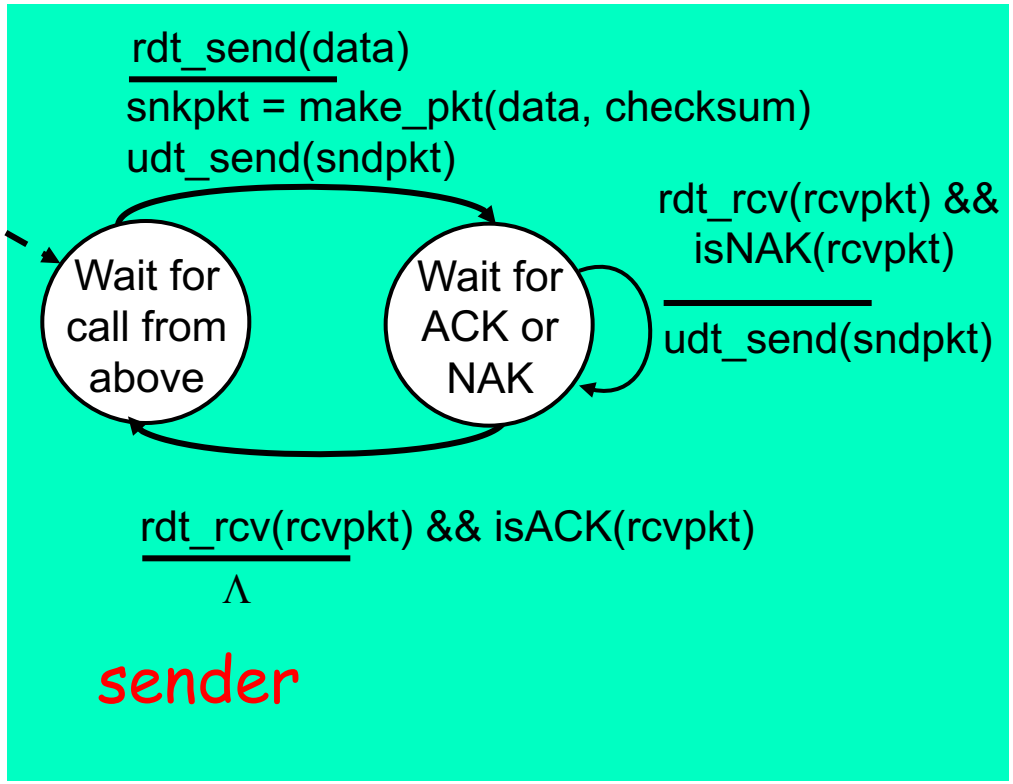


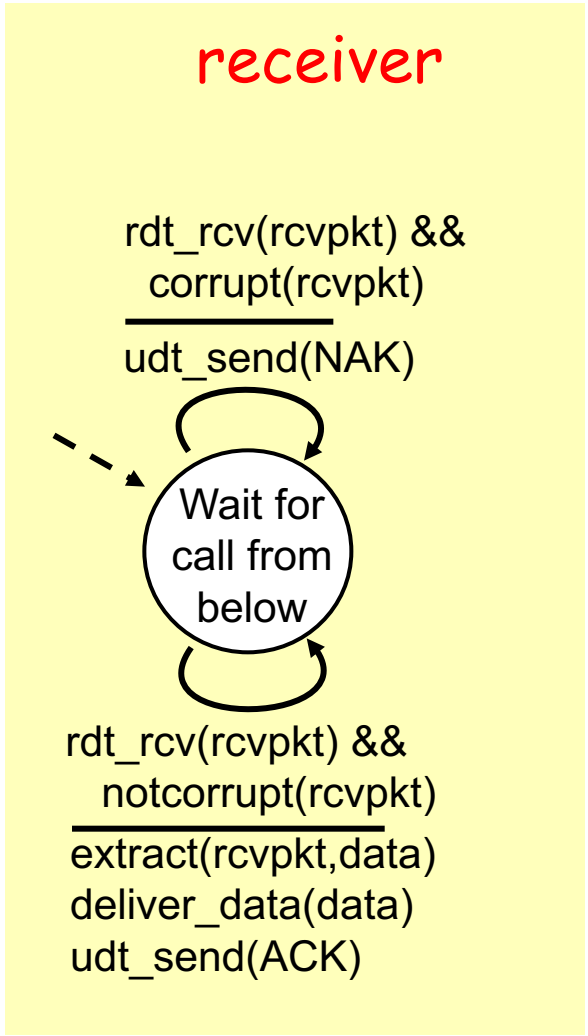


Rdt2.0: channel with bit errors

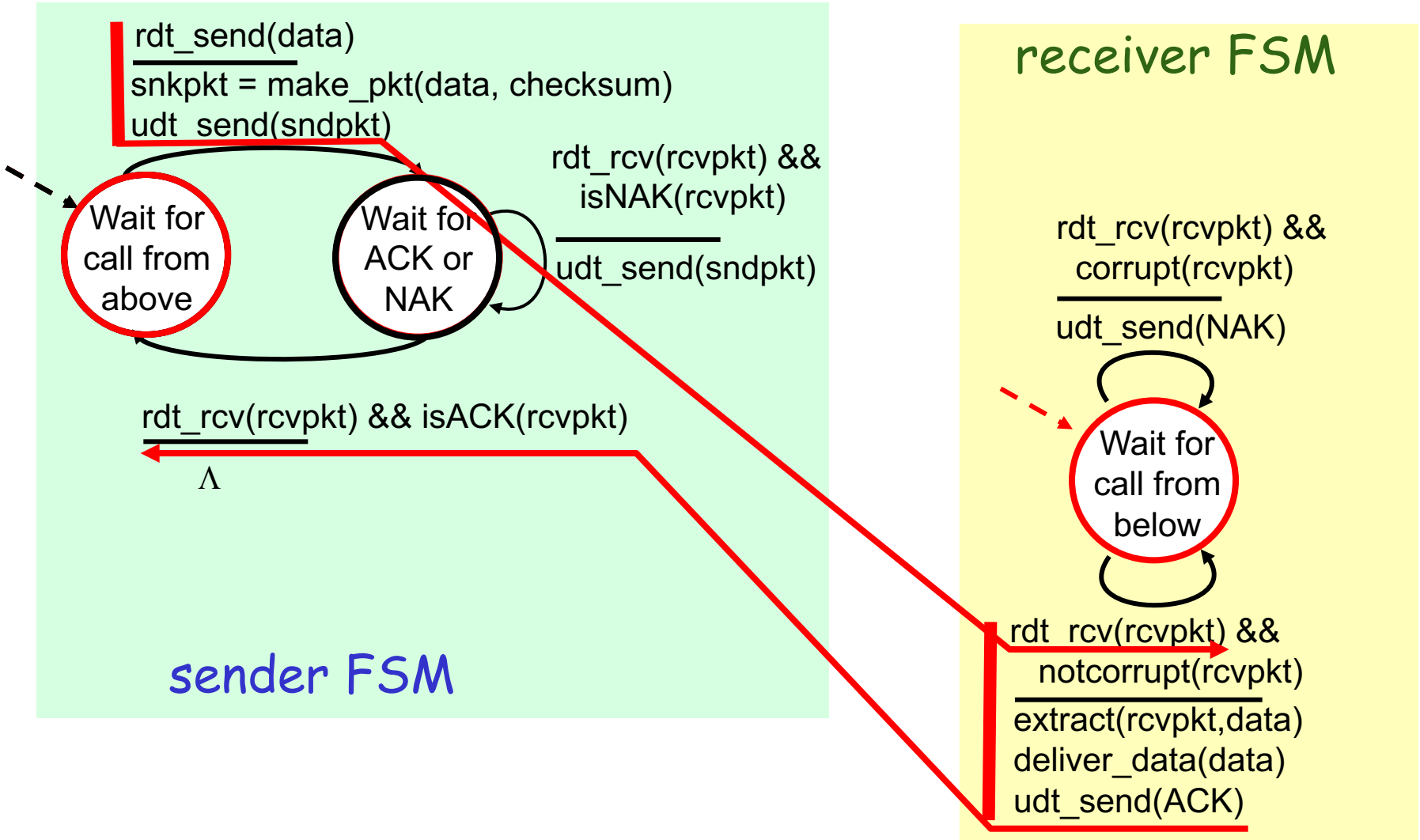
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors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

rdt2.0: FSM specification

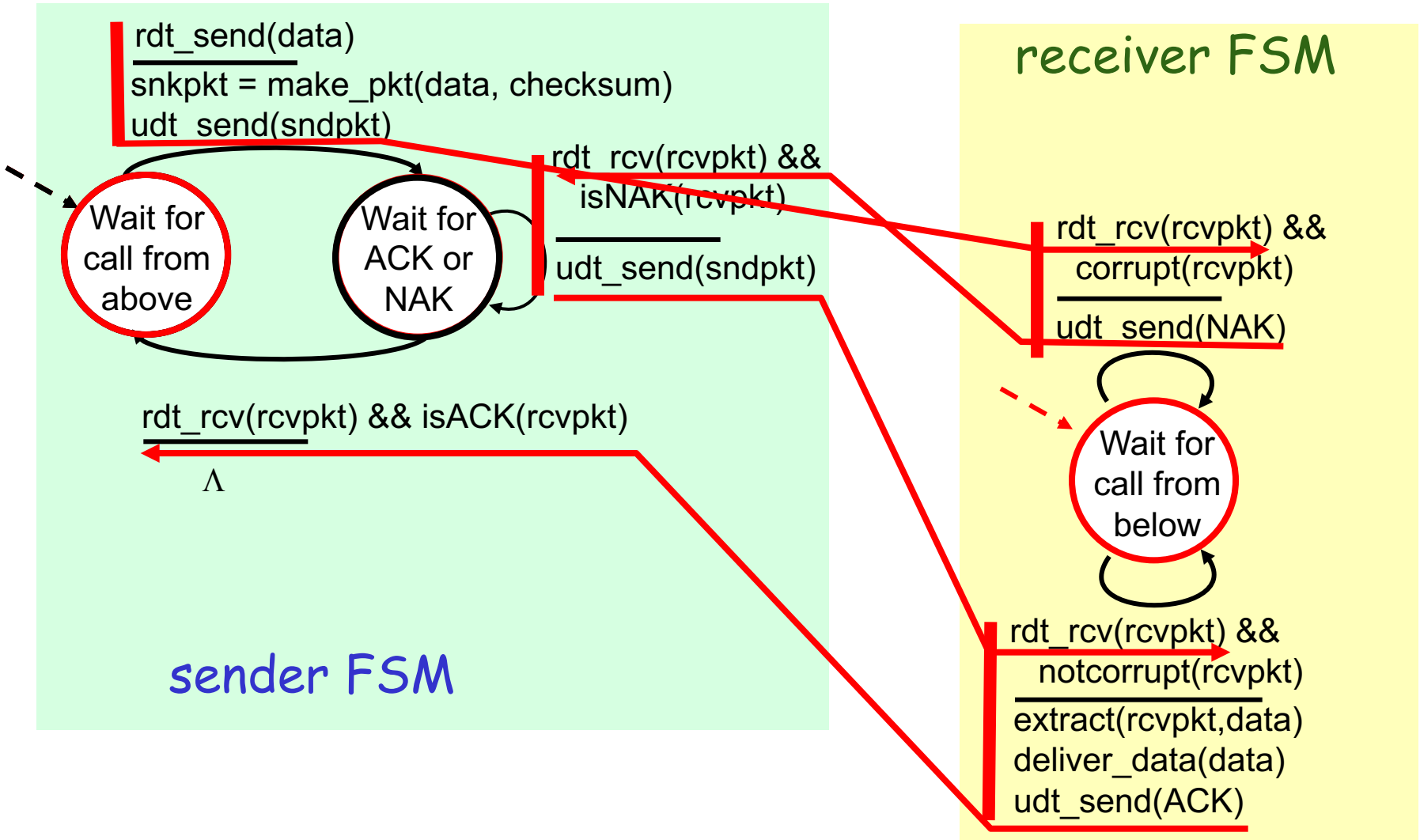




rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

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

→Need a way to detect duplicate

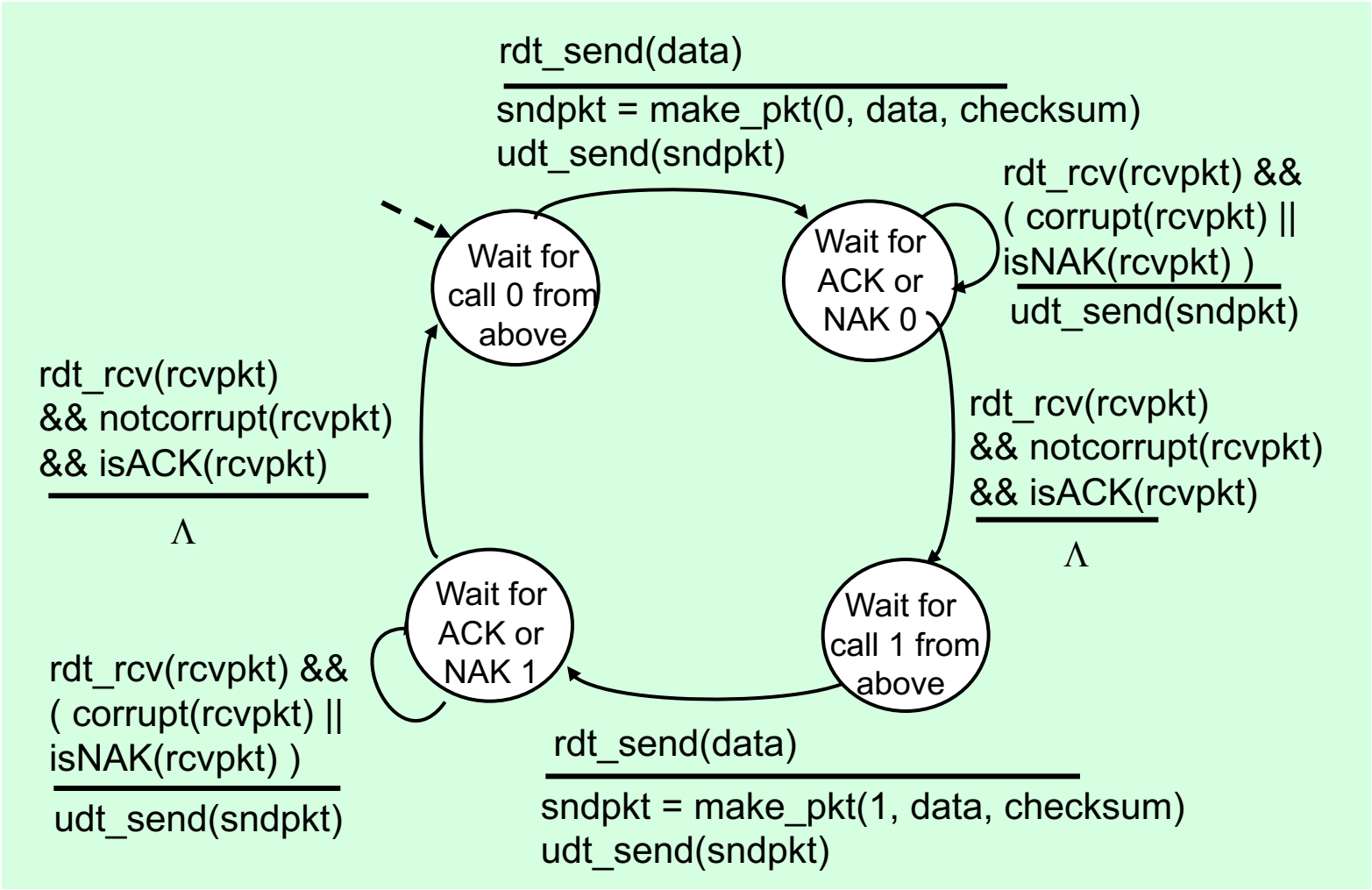
Handling duplicates:

- sender retransmits current pkt if ACK/NAK garbled
- sender adds *sequence number* to each pkt
- receiver discards duplicate pkt

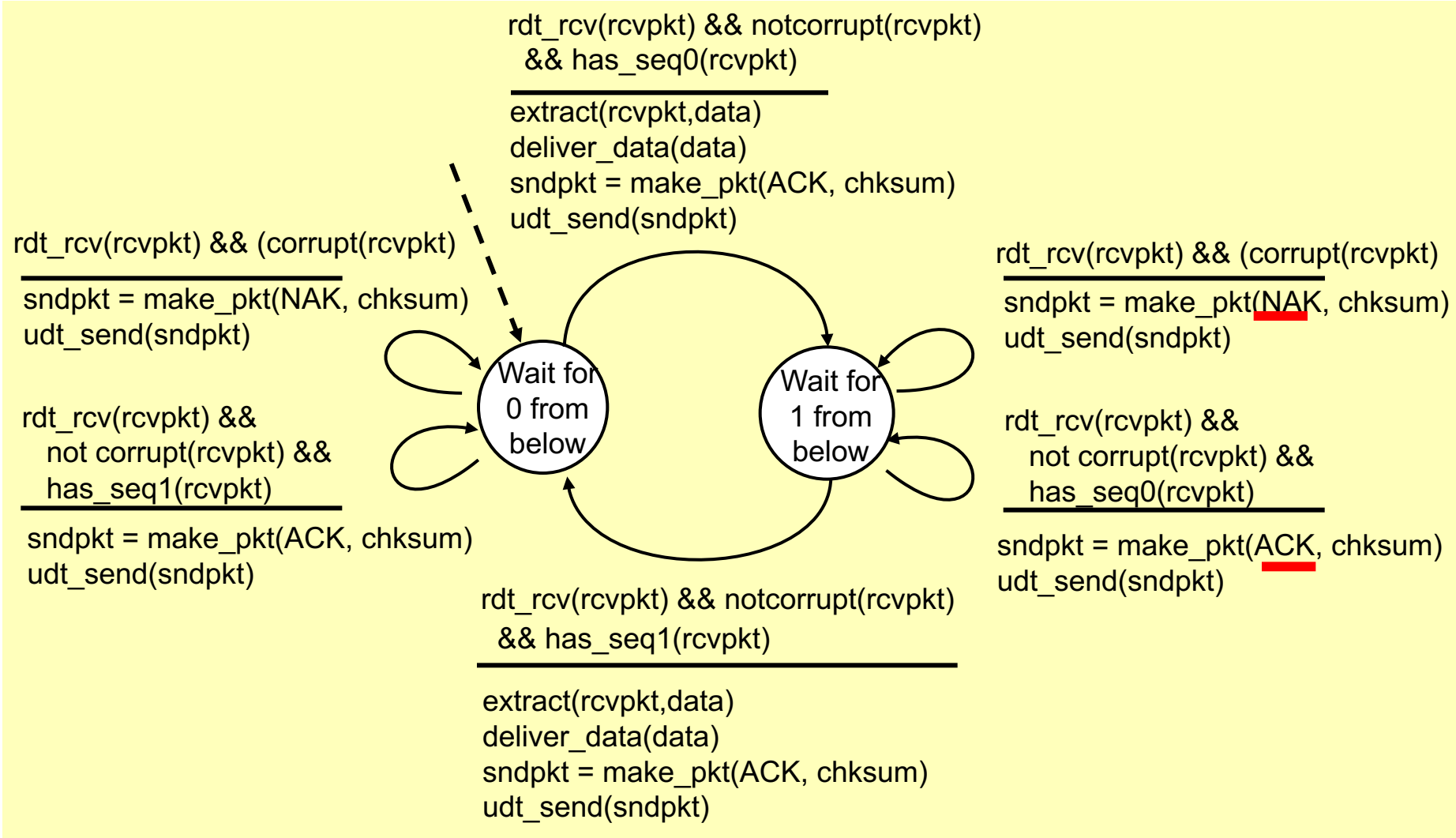
stop and wait

Sender sends one packet, then waits for receiver's response

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. #

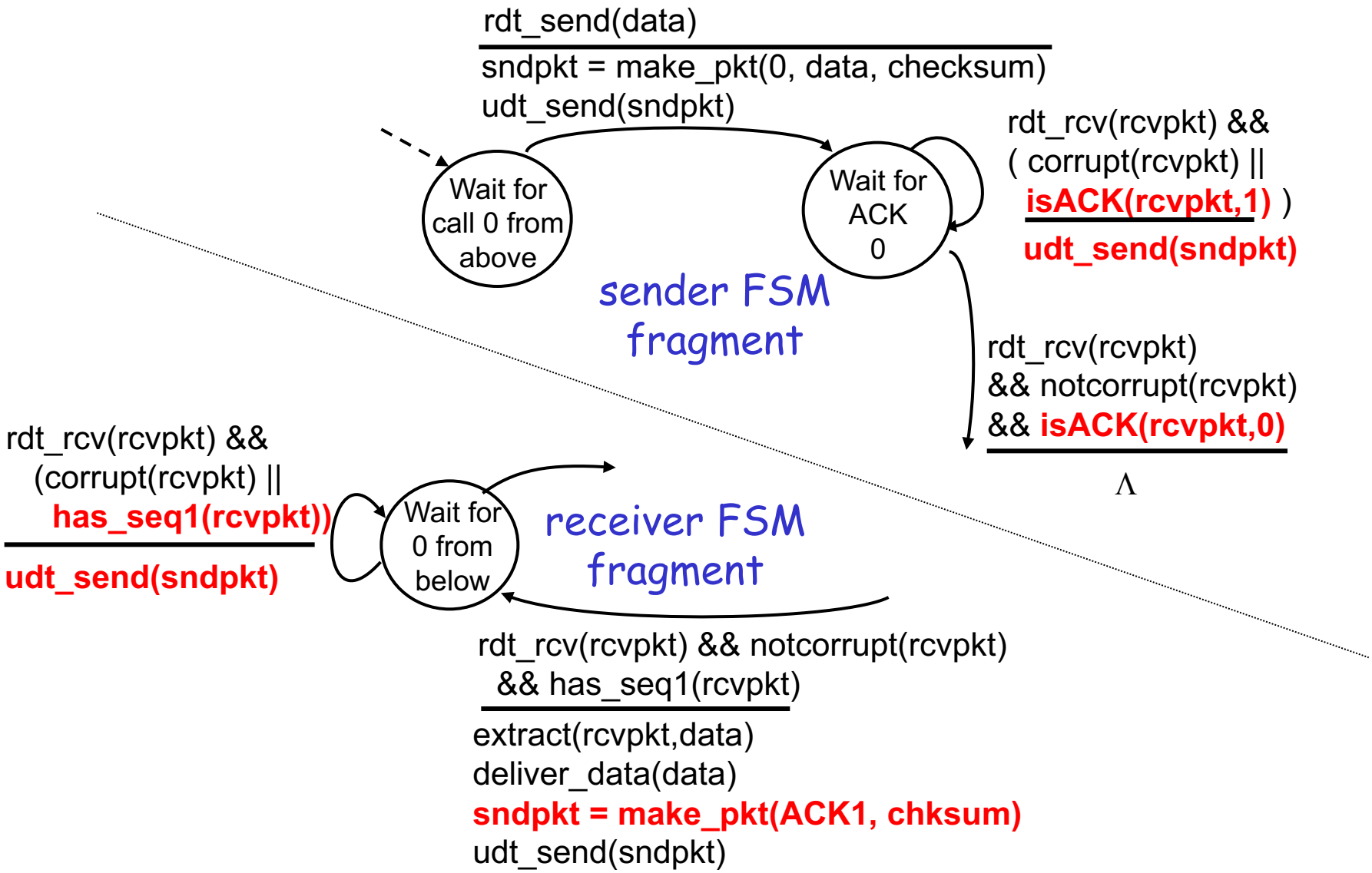
Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- 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*

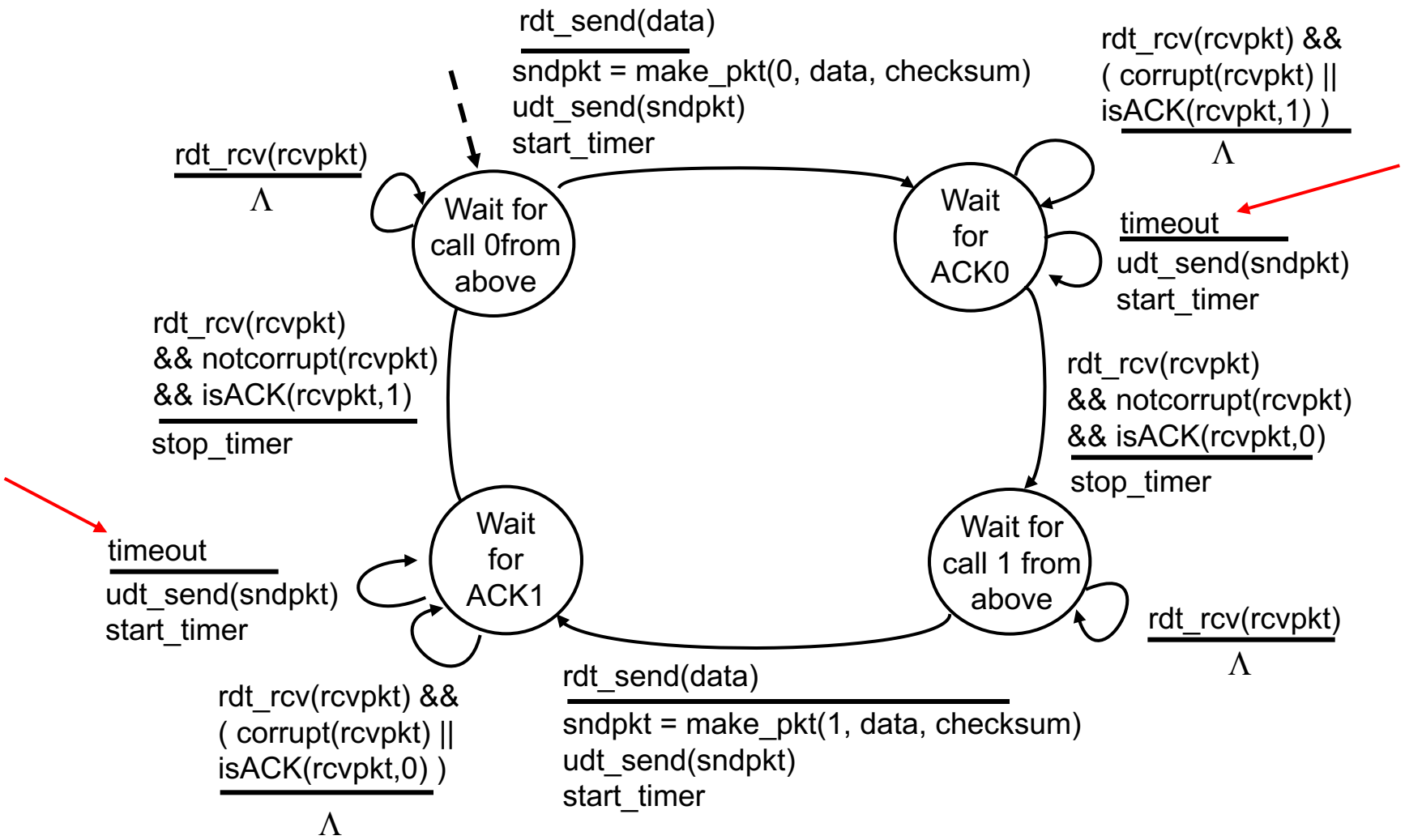
rdt2.2: sender, receiver fragments



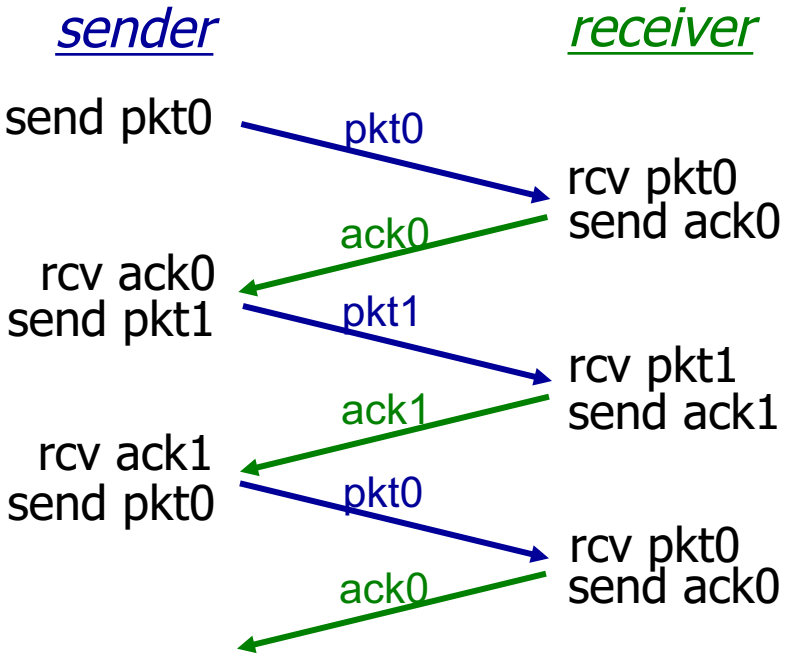
rdt3.0: channel with bit errors & packet loss

- After sending out a packet, the sender waits for ACK from receiver
  - Set up a retransmission timer
- When the timer expires: retransmits the packet
- In case the packet (or ACK) just delayed but not lost
  - Retransmitted packet will be a duplicate
  - Sequence number can detect this

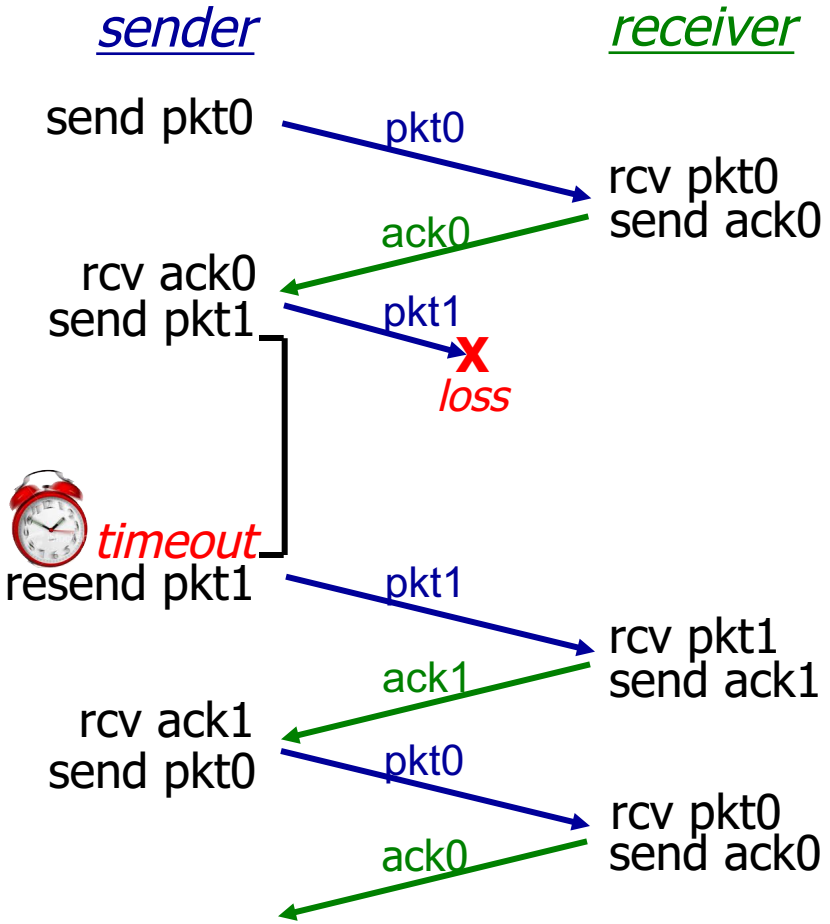
rdt3.0 sender



rdt3.0 in action

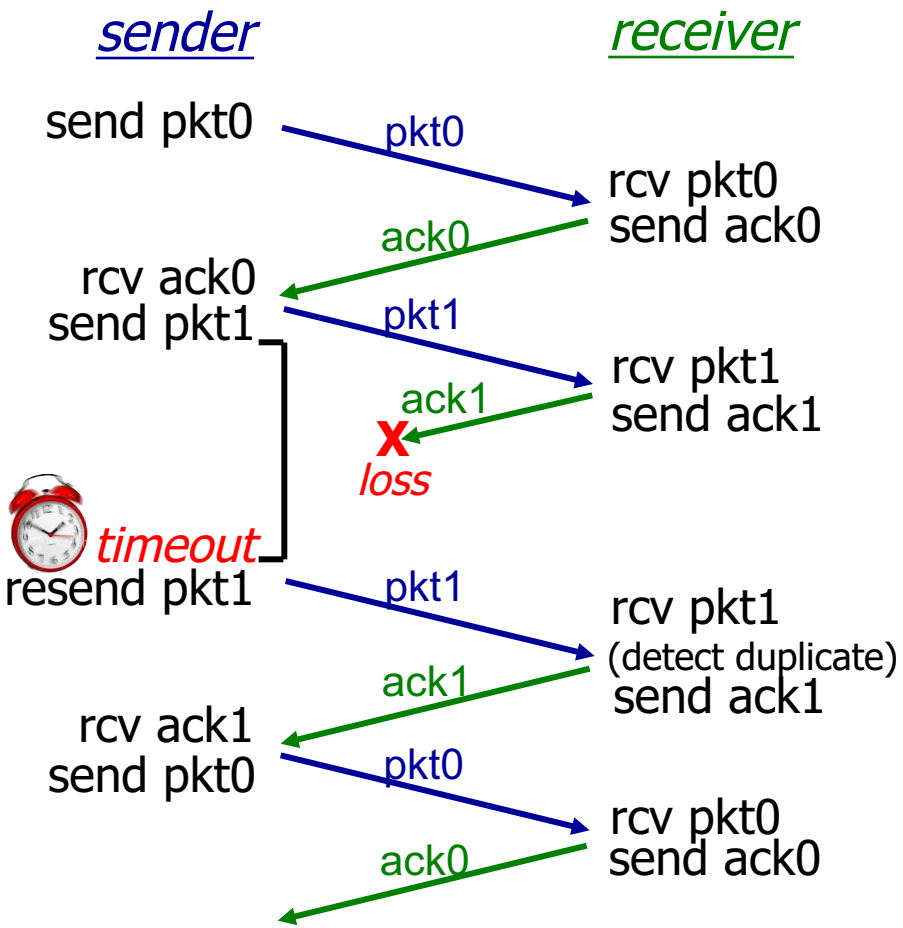


(a) no loss

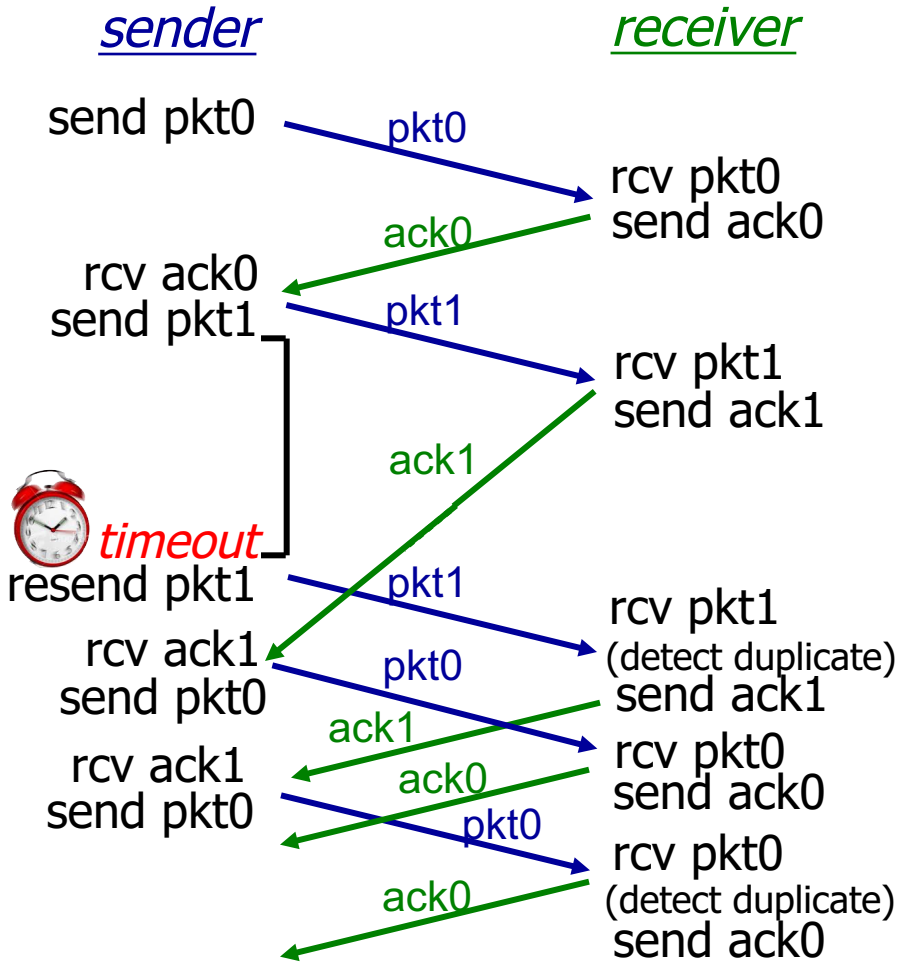


(b) packet loss

rdt3.0 in action



(c) ACK loss

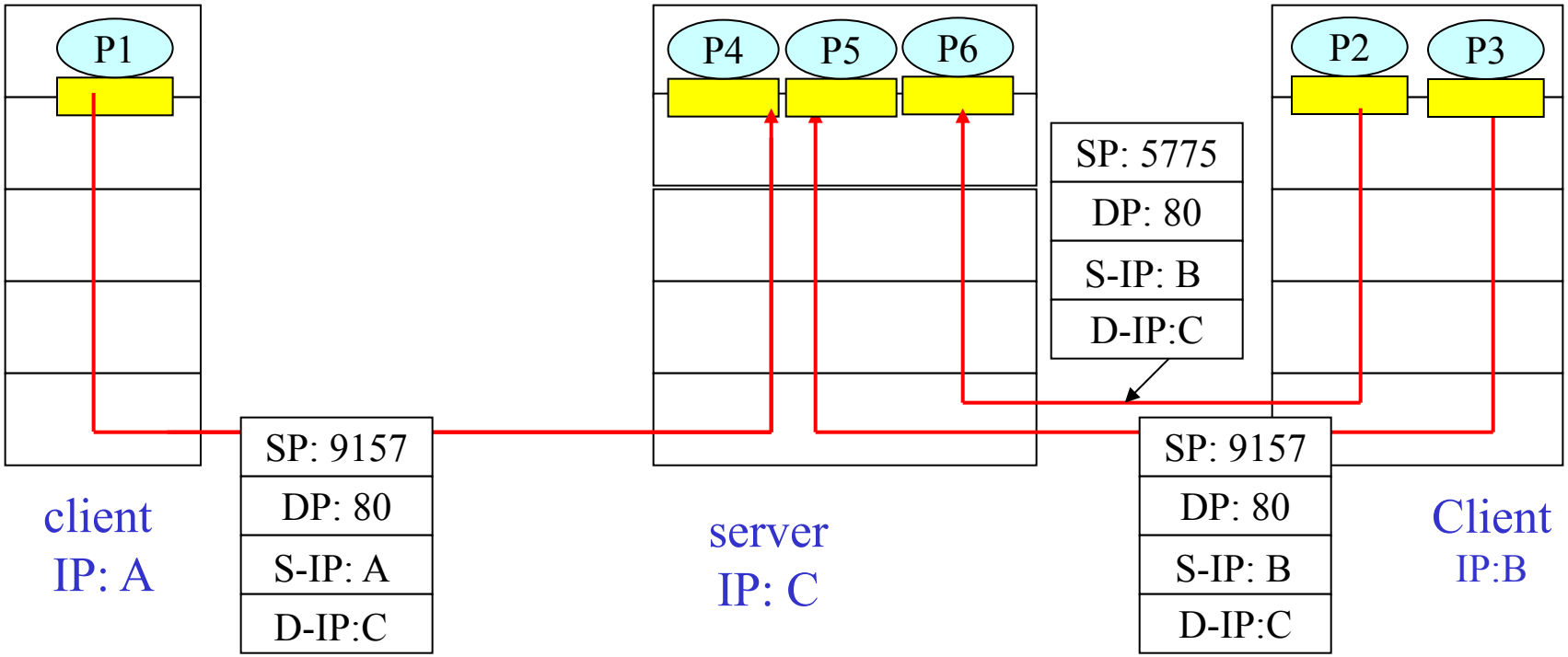


(d) premature timeout due to delayed ACK: duplicate transmissions

Connection-oriented Demultiplex

- a TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- receiving host uses all four values to direct segment to appropriate socket
- A server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- e.g. a web serve creates separate sockets for each connecting client

Connection-oriented demux (cont)



A server process can tell apart

- data from different hosts by IP addresses
- Data from the same host but different processes by source port numbers

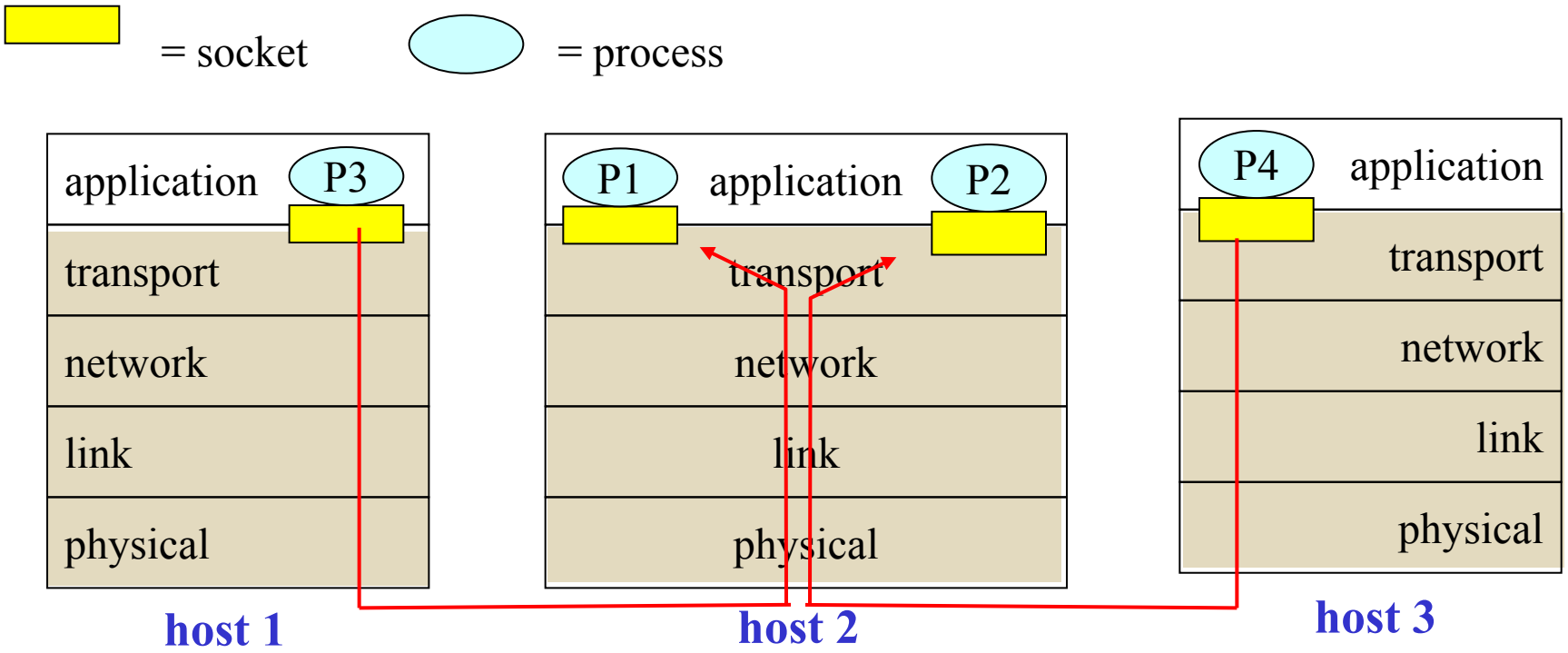
Multiplexing/demultiplexing

Multiplexing at sender:

gathering data from sockets, enveloping data with header (used for demultiplexing later)

Demultiplexing at receiver:

delivering received segments to correct socket



Each process is identified by IP address and port#