Internet Protocols EBU5403

Module organiser: Richard Clegg
(r.clegg@qmul.ac.uk)

Michael Chai (michael.chai@qmul.ac.uk)

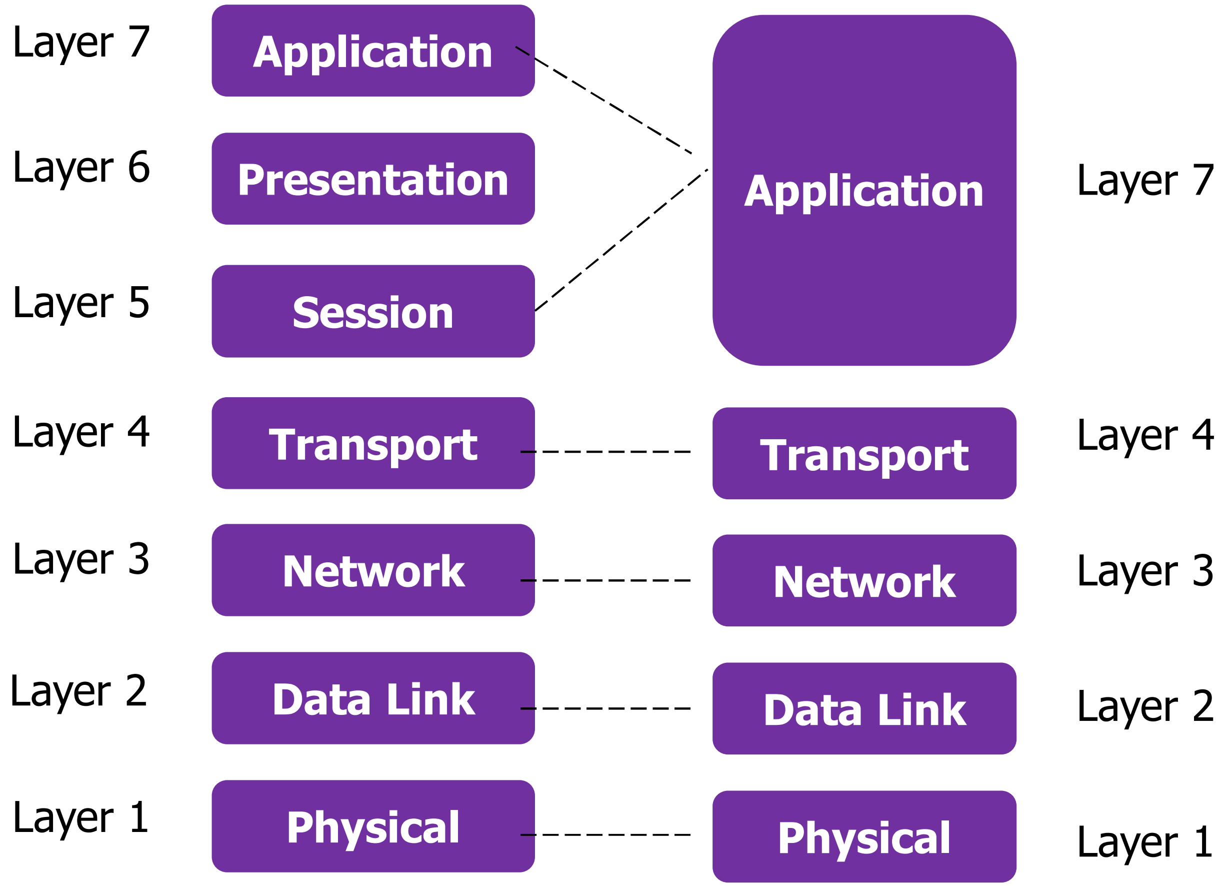
Cunhua Pan(c.pan@qmul.ac.uk)

	Part 1	Part 2	Part 3	Part 4
Ecommerce + Telecoms 1	Richard Clegg		Cunhua Pan	
Telecoms 2	Michael Chai			

Structure of course

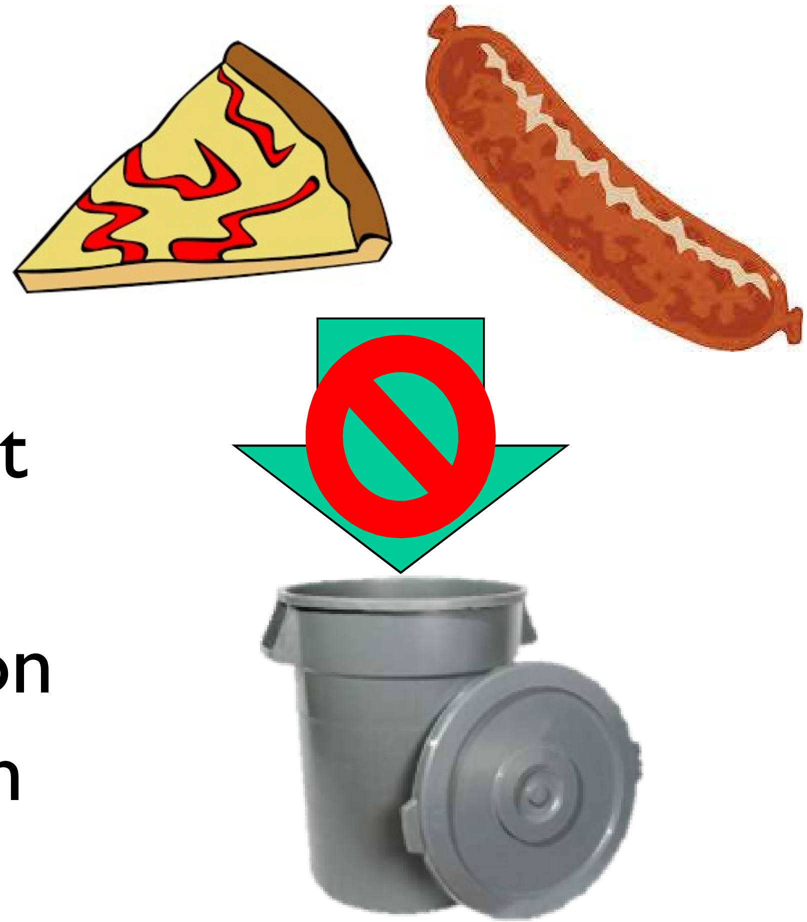
- Part A
  - Introduction to IP Networks
  - The Transport layer (part I)
- Part B
  - The Transport layer (part II)
  - The Network layer (part I)
  - Class test
- Part C
  - The Network layer (part II)
  - The Data link layer (part I)
  - Router lab tutorial (assessed lab work after this week)
- Part D
  - The Data link layer (part II)
  - Network management and security
  - Class test

ISO/OSI (left) vs TCP/IP (right)



How to remember the layers

- Please Do Not Throw Sausage + Pizza Away
- **P**lease – **P**hysical
- **D**o – **D**atalink
- **N**ot – **N**etwork
- **T**hrow – **T**ransport
- **S**ausage – **S**ession
- **P**izza – **P**resentation
- **A**way – **A**pplication

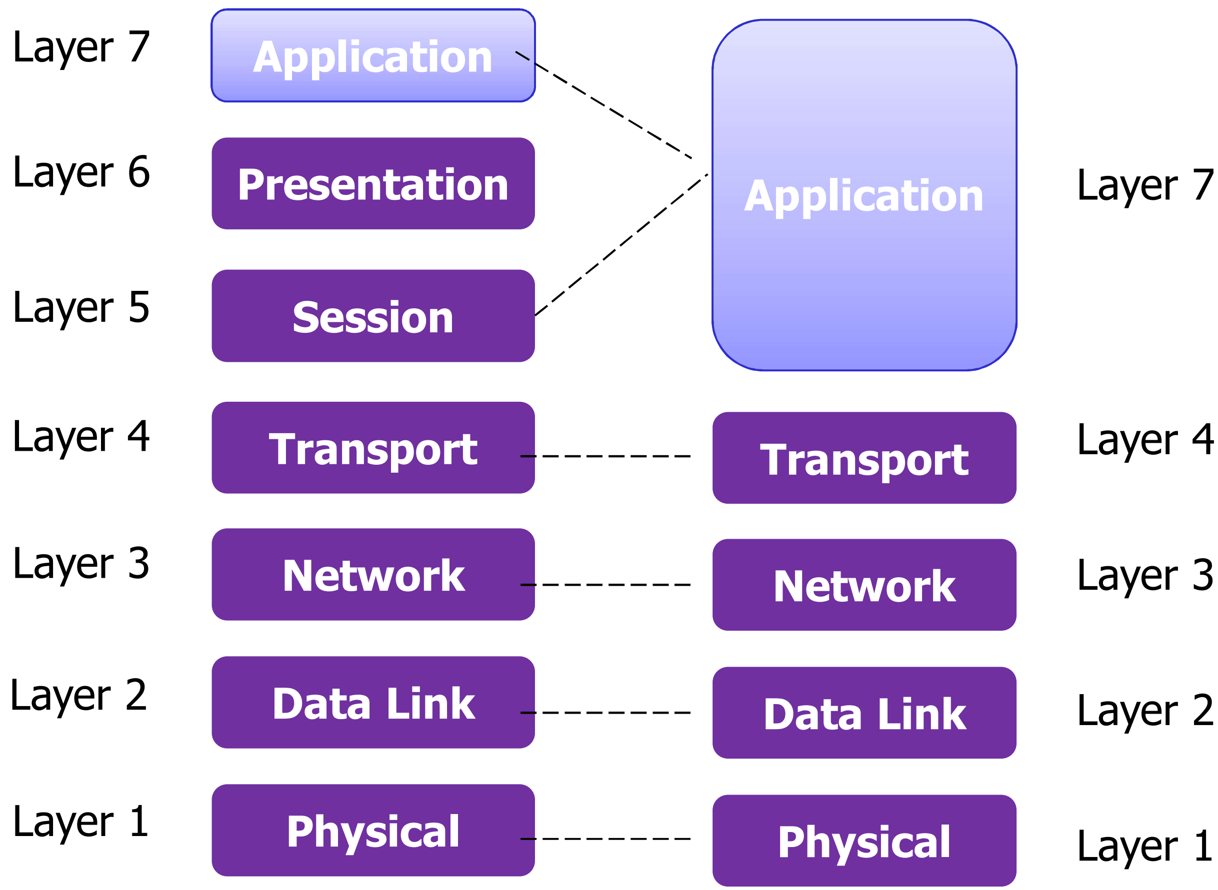


Part A: Transport Layer (part 1)

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

Application layer (very briefly)

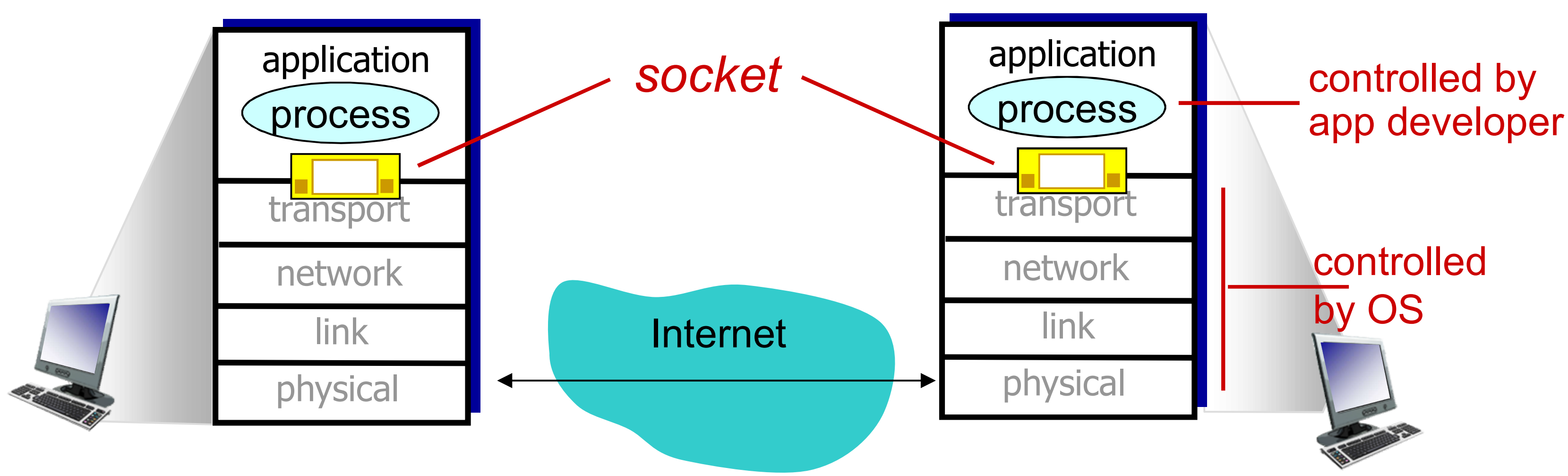


Application Layer (very briefly)

- This is covered in many other modules you will take.
- This is the layer you (mostly) work with as a programmer.
- At the application layer the "network" is abstracted away and you access a stream of data that arrives at a "socket".
- It is up to you to define the format of data your application writes and receives.
- Different applications have different formats:
  - HTTP (Hyptertext transfer protocol) world-wide web
  - FTP (File transfer protocol) moving data
  - SMTP (Send Mail transfer protocol) sending email
  - IMAP (Internet message access protocol) receiving email

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* no, *many* processes can be running on same host

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

4 different addresses in TCP/IP

- Physical address Layer 2
  - Also known as the link address, is the address of a node as defined by its LAN or WAN
- Logical address Layer 3 (32-bit , IPv4 128-bit IPv6)
  - Logical (IP) addresses are for universal communications that are independent of underlying physical networks
- Port address Layer 4 (16-bit)
  - Port addresses differentiate different processes
- Application-specific address Layer 7
  - Some applications have user-friendly addresses that are designed for that specific application, such as email address, URL.

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

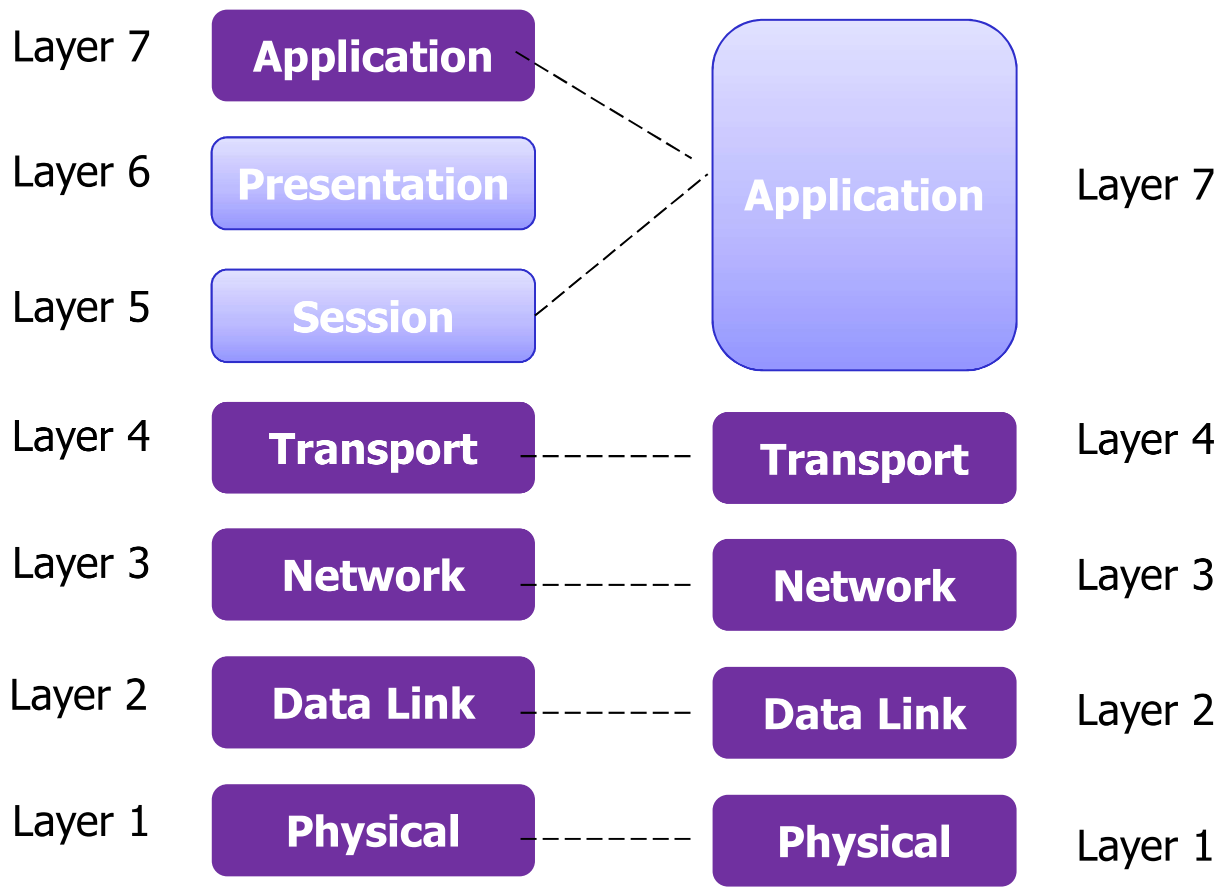
open protocols:

- defined in RFCs (request for comments)
- allows for interoperability
- e.g., HTTP (web), SMTP (email)

proprietary protocols:

- e.g., Skype

Session + Presentation layers



Session and Presentation Layers

- Theoretical only – not implemented in the Internet
- Session Layer (layer 5):
  - Takes care of a connection between two hosts for the lifetime of that connection
  - Authentication + authorisation (who is the user, what can they do)
  - In working Internet this is at the application layer.
- Presentation Layer (layer 6):
  - Translates data for an application.
  - For example takes care of detail of character set used to encode string of characters.
  - In working Internet this is at the application layer.

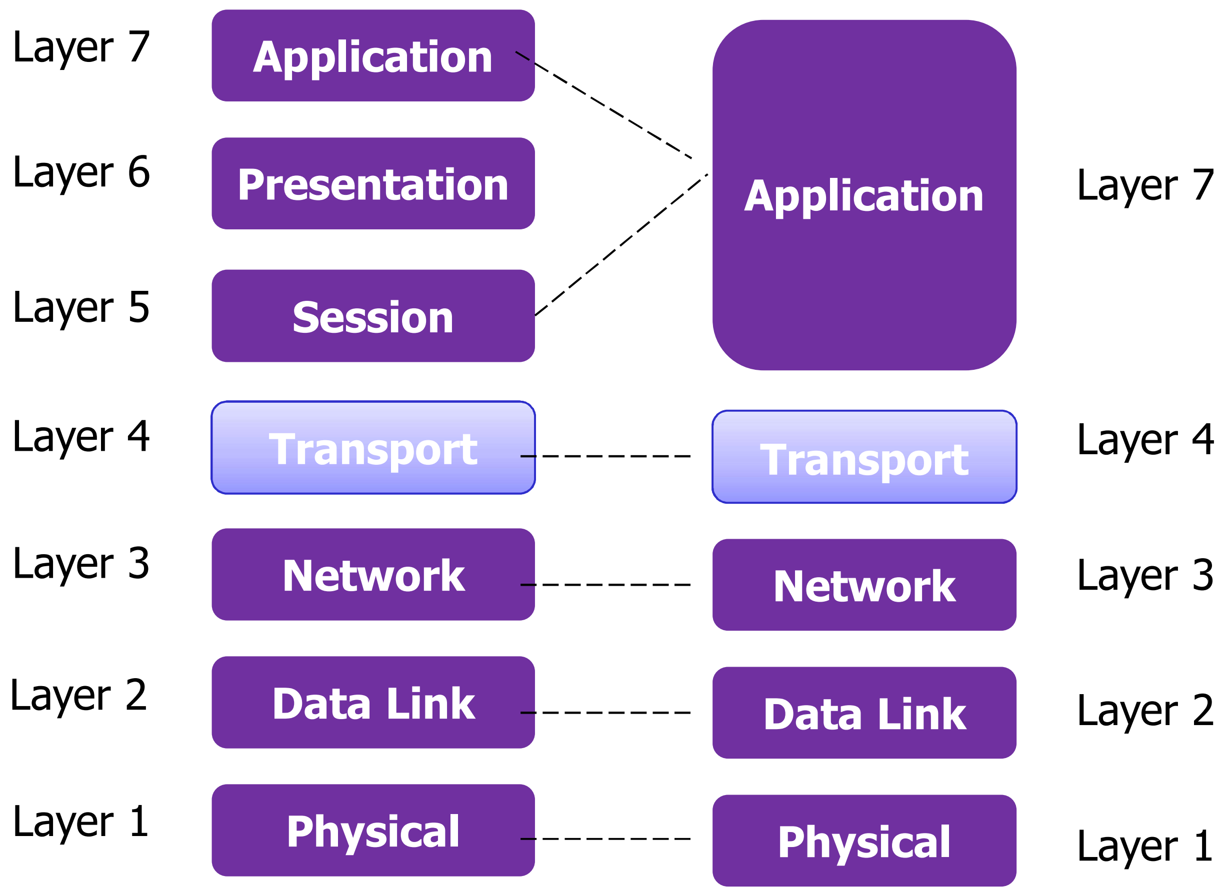
Test your understanding

- In ISO/OSI model which layer was planned to be responsible for:
- Options:
  - (A) Authentication and authorisation
  - (B) Sending and receiving email
  - (C) Encoding character sets
  - (D) Converting bits to a signal for WiFi
- Pause the recording and write down your answers.

Test your understanding

- In ISO/OSI model which layer was planned to be responsible for:
- Options:
  - (A) Authentication and authorisation (Session Layer – layer 5)
  - (B) Sending and receiving email (Application layer – layer 7)
  - (C) Encoding character sets (Presentation Layer – layer 6)
  - (D) Converting bits to a signal for WiFi (Physical Layer – layer 1)
- If you had problems review start of lecture.

Transport Layer



Transport layer: outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

Part I

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

Part II

Numbers relate to chapter numbers in Course text: Kurose + Ross

Transport Layer: outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

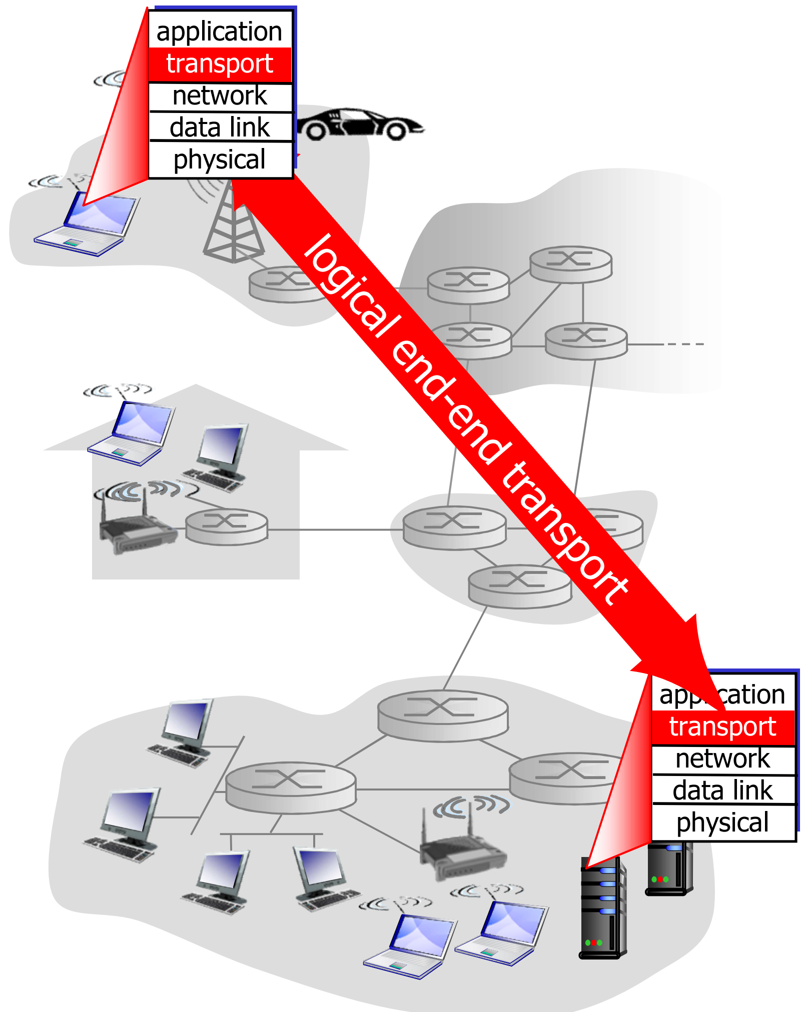
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

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

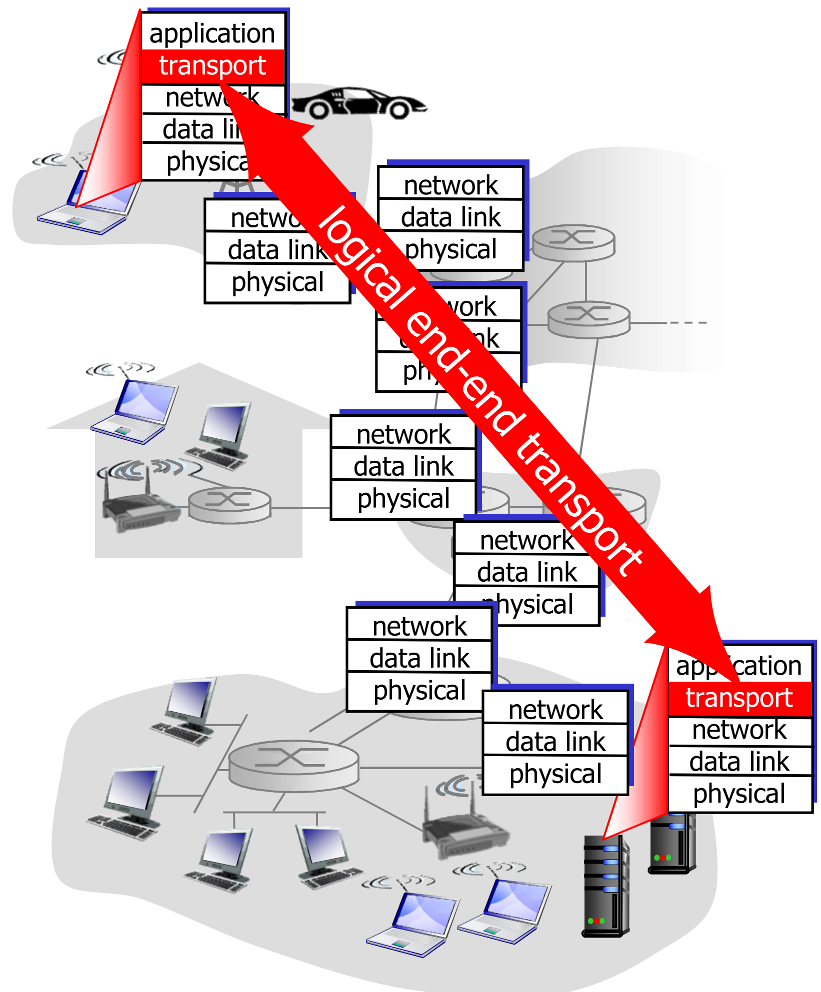
household analogy:

12 kids in Ann's house sending letters to 12 children in Bill's house:

- hosts = houses
- processes = children
- app messages = letters in envelopes
- transport protocol = Ann and Bill who give letter to correct child
- 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



How to remember the layers

- Please Do Not Throw Sausage + Pizza Away
- **P**lease – **P**hysical
- **D**o – **D**atalink
- **N**ot – **N**etwork
- **T**hrow – **T**ransport
- **S**ausage – **S**ession
- **P**izza – **P**resentation
- **A**way – **A**pplication

Transport Layer: 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

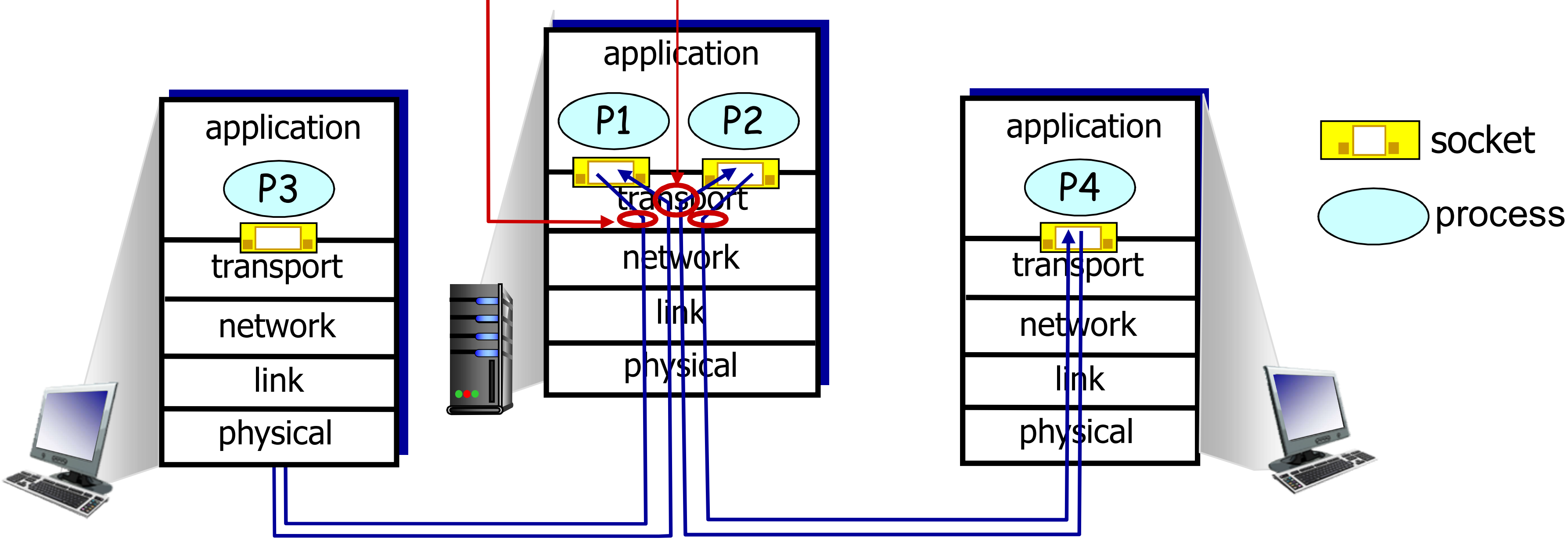
Multiplexing/demultiplexing

- Multiplexing (Mux):
  - Combining several streams of data into a single stream.
  - Example – your phone is browsing the web, refreshing your email, connecting to wechat at the same time. All these connections are sent over the same link.
- Demultipliexing (Demux)
  - The opposite process. A stream of data is separated out into its individual components.
  - The stream of packets the phone received is split up and sent to the appropriate program for web, email, wechat.

Multiplexing/demultiplexing

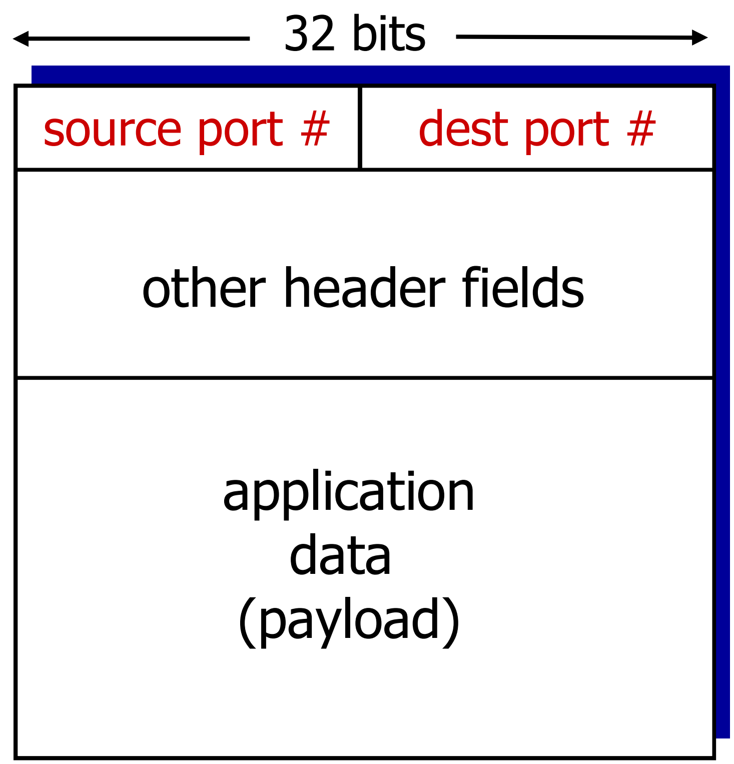
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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

Test your understanding

- Multiplexing (mux) or demultiplexing (demux)
- Options:
  - (A) Combining several traffic sources to one.
  - (B) Performed at the receiver.
  - (C) Taking one stream of traffic and breaking it up into several to be sent to different processes.

Pause the recording and write down your answers.

Test your understanding

- Multiplexing (mux) or demultiplexing (demux)
- Options:
  - (A) Combining several traffic sources to one. (Multiplexing)
  - (B) Performed at the receiver. (Demultiplexing)
  - (C) Taking one stream of traffic and breaking it up into several to be sent to different processes. (Demultiplexing).
- If you cannot understand the answers review the start of section 3.2

Connectionless demultiplexing

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

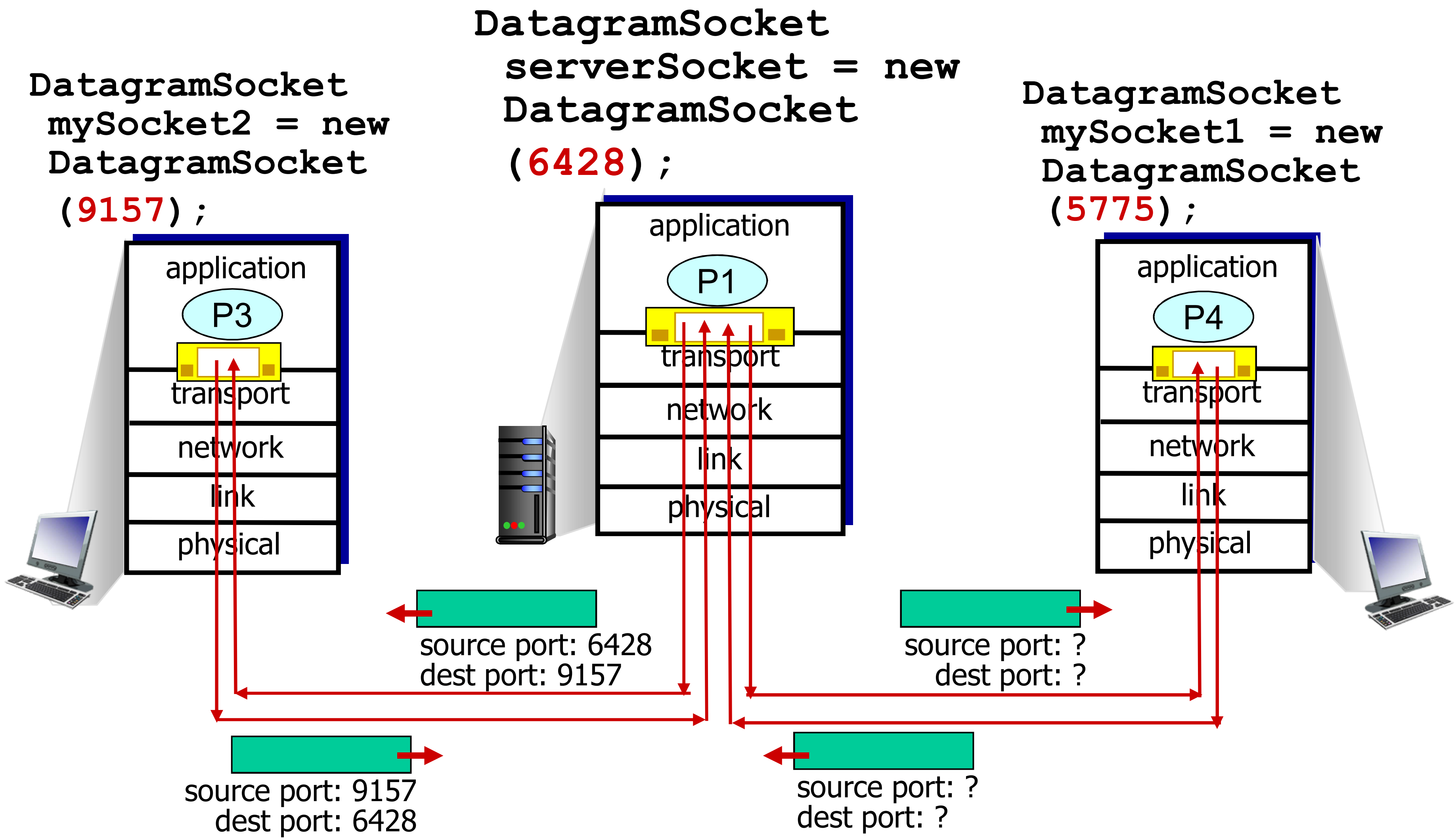
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *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

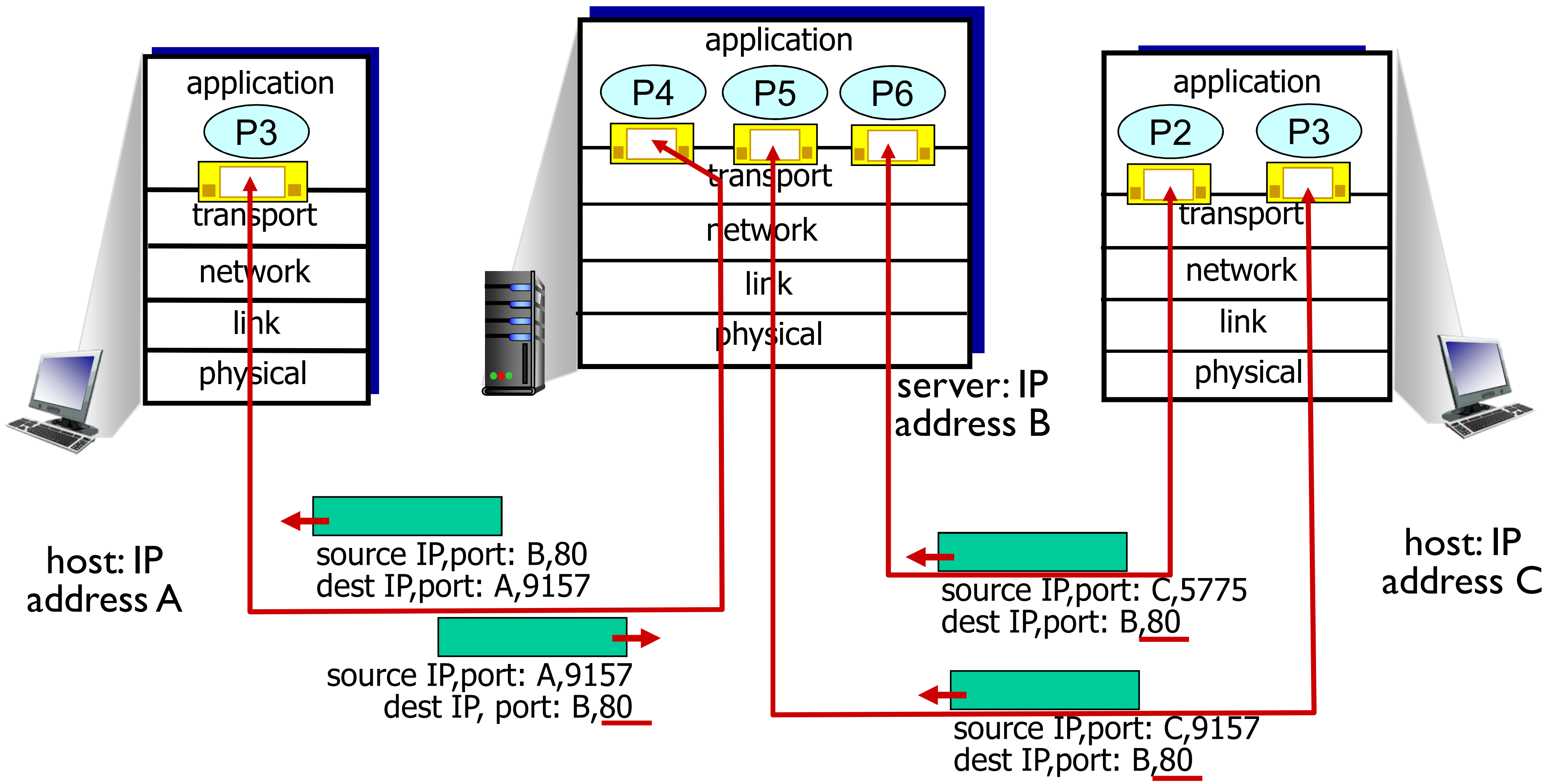
Connectionless demux: example



Connection-oriented demux

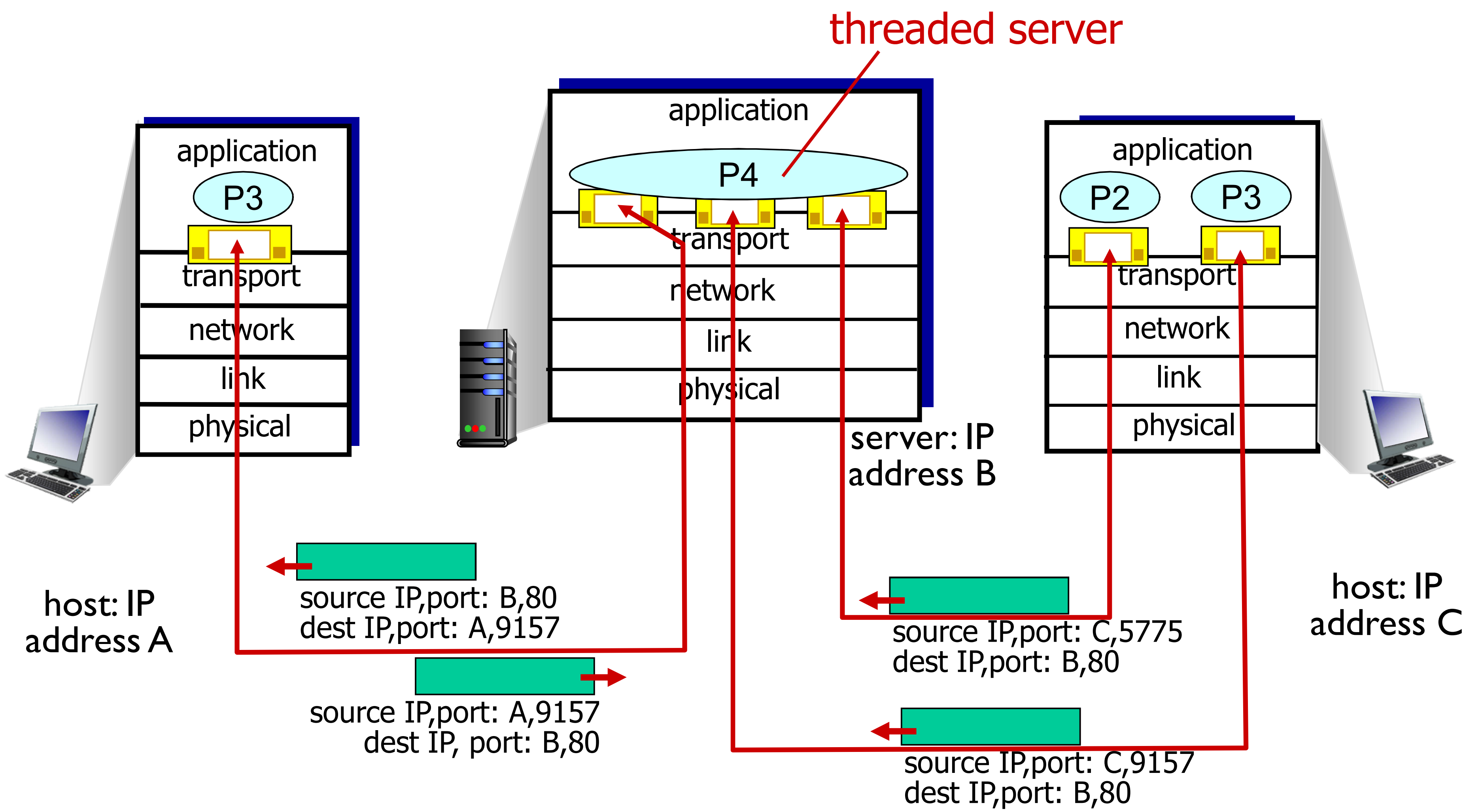
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



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

Connection-oriented demux: example



Typical port numbers for applications

- Port 80: Standard HTTP (Hypertext Transfer Protocol – browse the web
- Port 22: ssh (secure shell) log in from remote computer
- Port 25: SMTP (simple mail transfer protocol) – send mail via this host
- Port 143: IMAP (Internet Message Access Protocol) – read your email
- Port 443: HTTPS (secure HTTP) browse the web securely
- These are just conventions, you could change them if you so wished on your own host.

Test your understanding

- True or false?
- Options:
  - (A) Multiplexing is performed when traffic is received.
  - (B) In a connection-oriented demux only one process can receive data set to a given port.
  - (C) In a connection-oriented demux we need 4 pieces of information to identify the process to receive a packet.
  - (D) Connectionless demux is more efficient.

Test your understanding

- True or false?
- Options:
  - (A) Multiplexing is performed when traffic is received.
  - False (multiplexing is combining)
  - (B) In a connection-oriented demux only one process can receive data set to a given port.
  - False (one port can connect to many processes).
  - (C) In a connection-oriented demux we need 4 pieces of information to identify the process to receive a packet.
  - True (source+dest port plus source + dest IP address)
  - (D) Connectionless demux is more efficient.
  - False (it depends on circumstances which is efficient)
- If you did not get the answers right review section 3.2

Transport layer 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

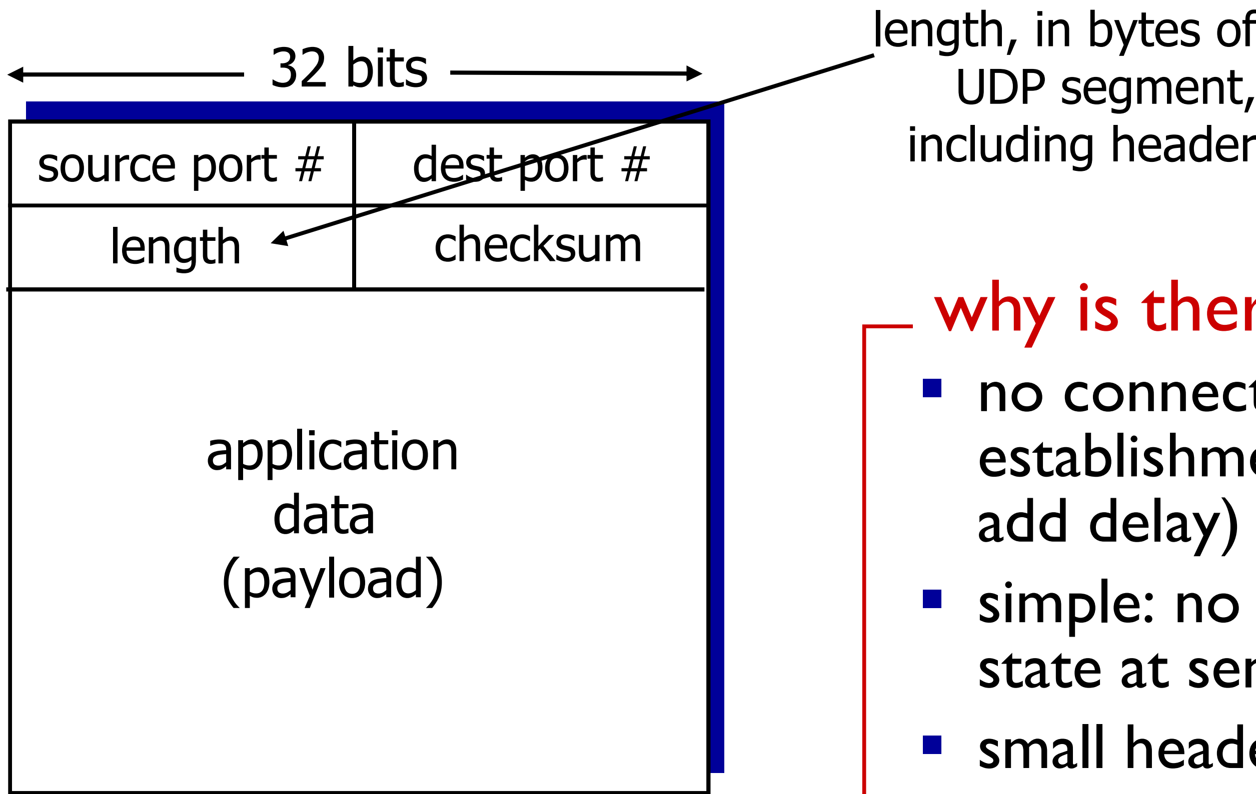
3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

- Simplest usable Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no "handshaking" between UDP sender, receiver (can send immediately without asking first)
  - each UDP packet 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



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 – 0 becomes 1 or 1 becomes 0) in transmitted segment

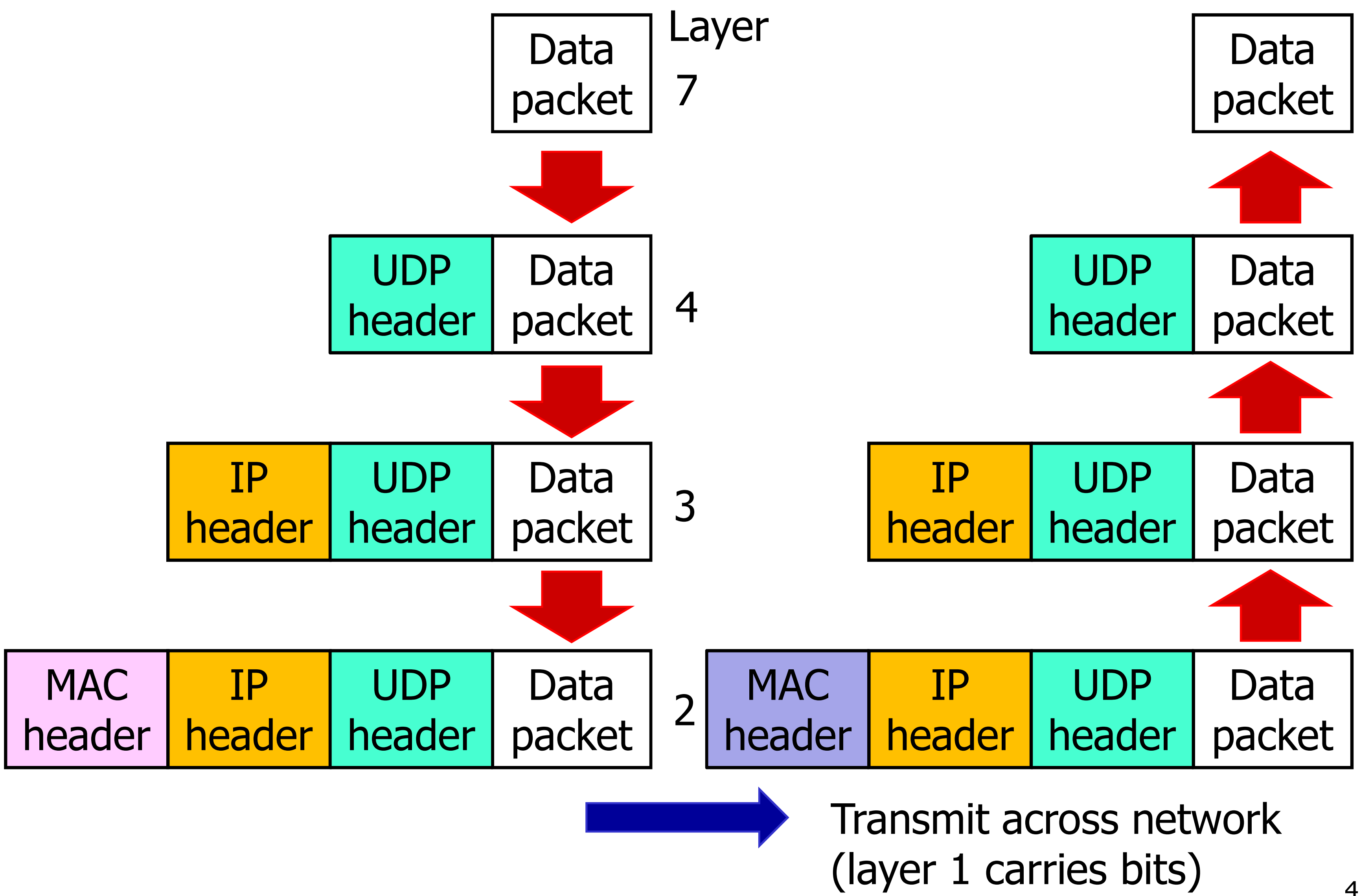
sender:

- 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 ….

UDP Encapsulation/decapsulation



Test your understanding

- UDP questions:
  - (A) When it is examined by the transport layer does a UDP packet have layer 2 and 3 headers?
  - (B) What is the purpose of the UDP checksum?
  - (C) What are the four things in a UDP header apart from data?
  - (D) Is UDP connection oriented or connectionless?

Pause the video and write down

Test your understanding

- UDP questions:
  - (A) When it is examined by the transport layer does a UDP packet have layer 2 and 3 headers?

    No – the network layer removes those headers before sending the packet up. The application layer could not add them.
  - (B) What is the purpose of the UDP checksum?

    Helps us tell if the packet is corrupted
  - (C) What are the four things in a UDP header apart from data?

    Checksum, packet length, source port, dest port
  - (D) Is UDP connection oriented or connectionless?

    Connectionless.
- If you had problems with these questions review section 3.3

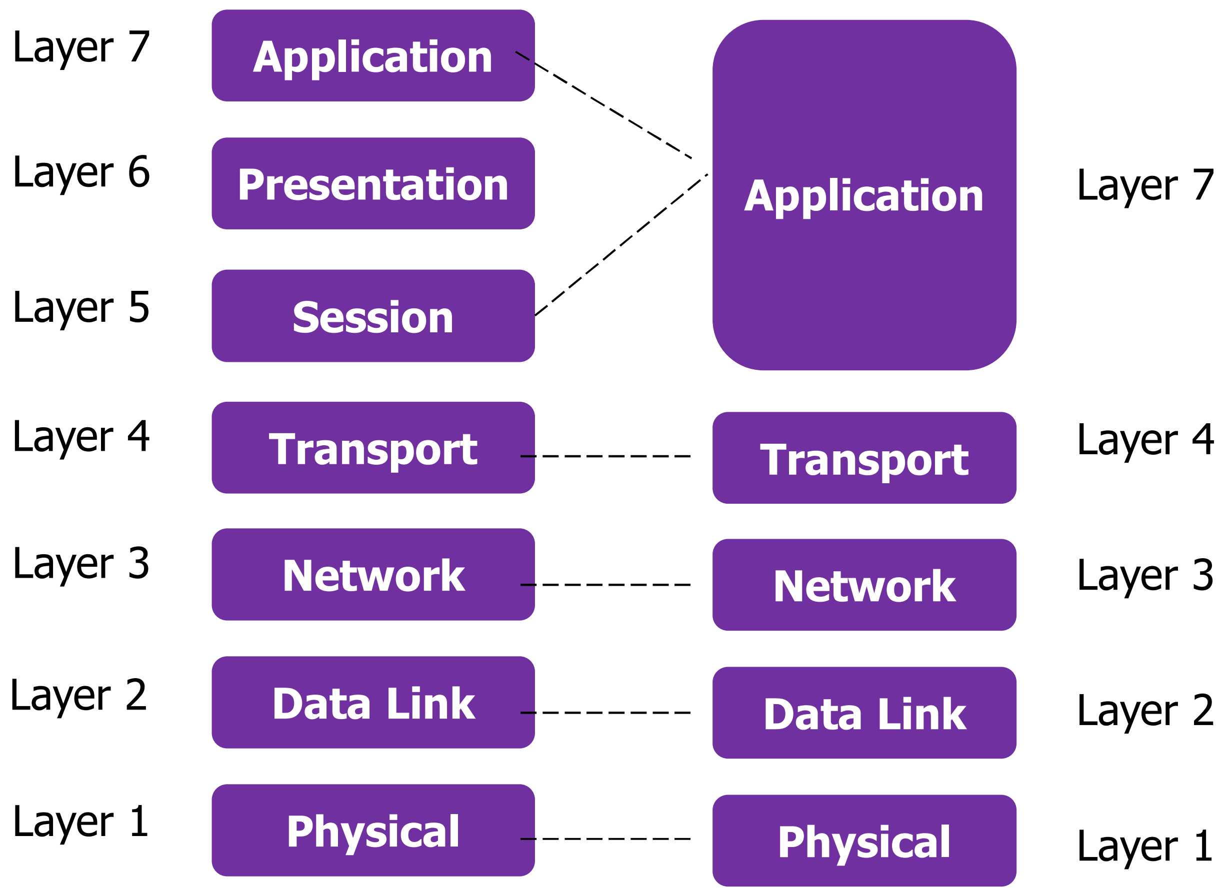
What have we learned?

- Brief look at Application Layer (layer seven) protocols which programmers use.
- Transport Layer introduction (layer four).
  - Deliver data to application sockets themselves.
  - Responsible for “end-to-end” connection.
- User Datagram Protocol (transport layer):
  - Simple, connectionless.
  - Packets may be lost and reordered.
  - Useful for simple situations or real-time.

Structure of course

- Part A
  - Introduction to IP Networks
  - The Transport layer (part 1)
- Part B
  - The Transport layer (part II)
  - The Network layer (part I)
  - Class test
- Part C
  - The Network layer (part II)
  - The Data link layer (part I)
  - Router lab tutorial (assessed lab work after this week)
- Part D
  - The Data link layer (part II)
  - Network management and security
  - Class test

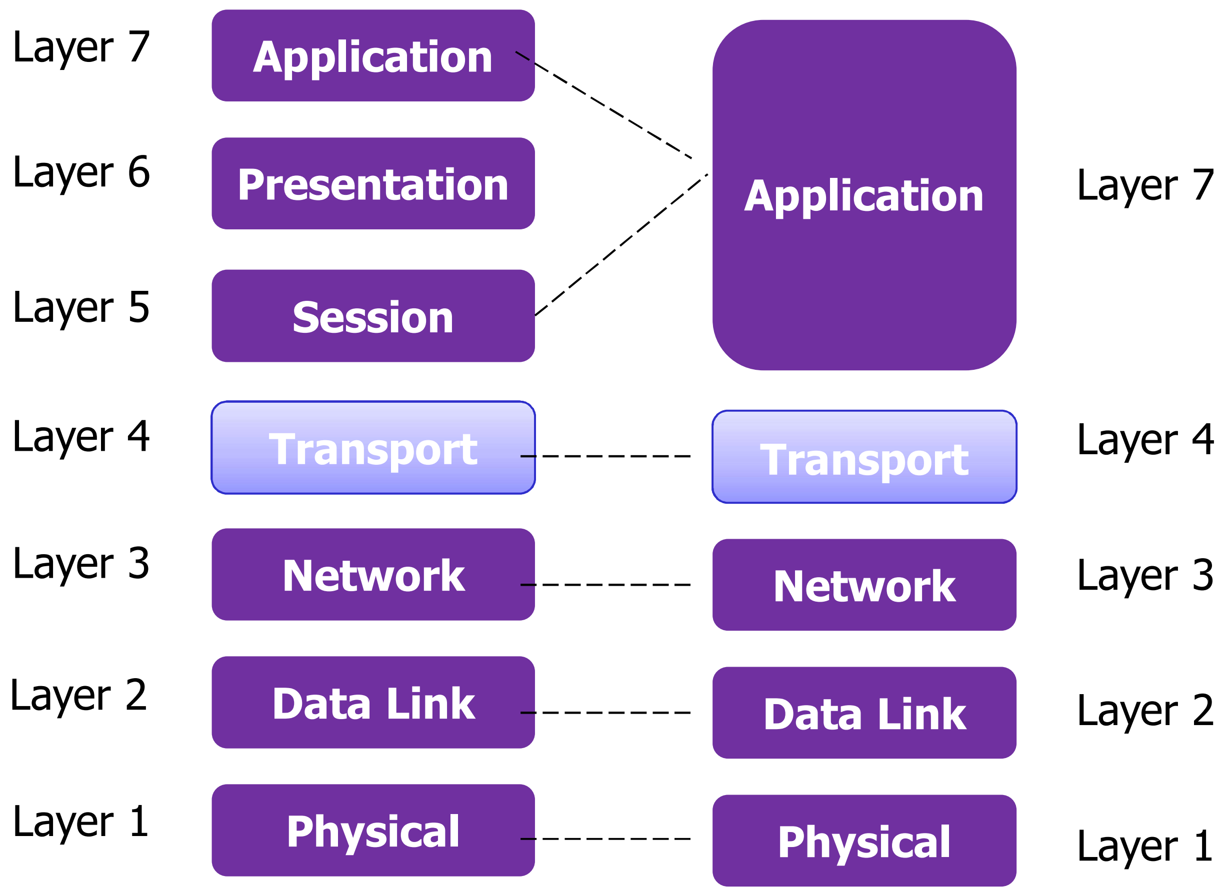
ISO/OSI (left) vs TCP/IP (right)



How to remember the layers

- Please Do Not Throw Sausage + Pizza Away
- **P**lease – **P**hysical
- **D**o – **D**atalink
- **N**ot – **N**etwork
- **T**hrow – **T**ransport
- **S**ausage – **S**ession
- **P**izza – **P**resentation
- **A**way – **A**pplication

Transport Layer



Part A: Transport Layer (part II)

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 layer outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

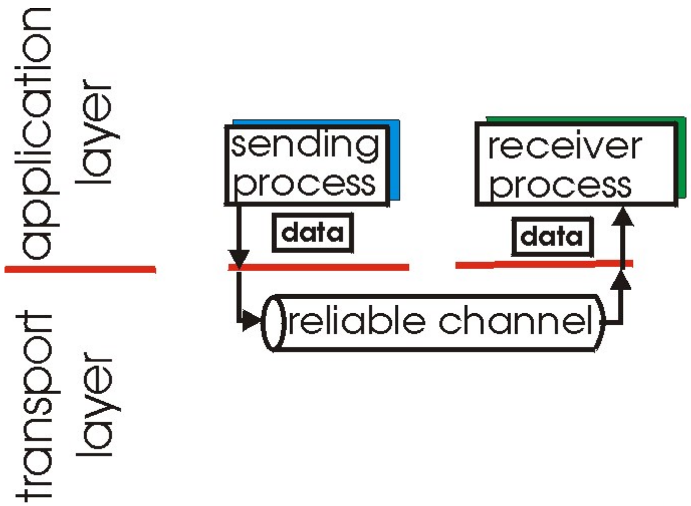
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

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

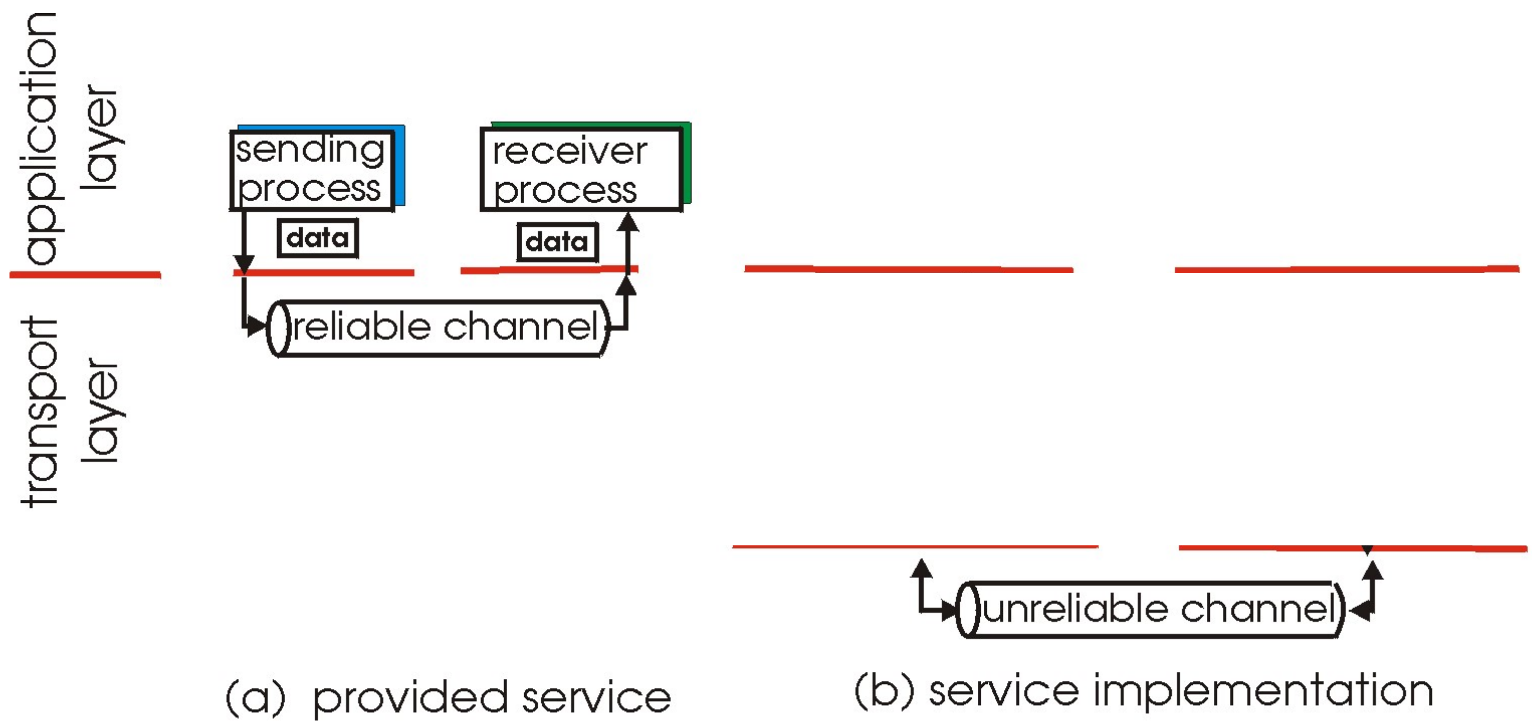


(a) provided service

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

Principles of reliable data transfer

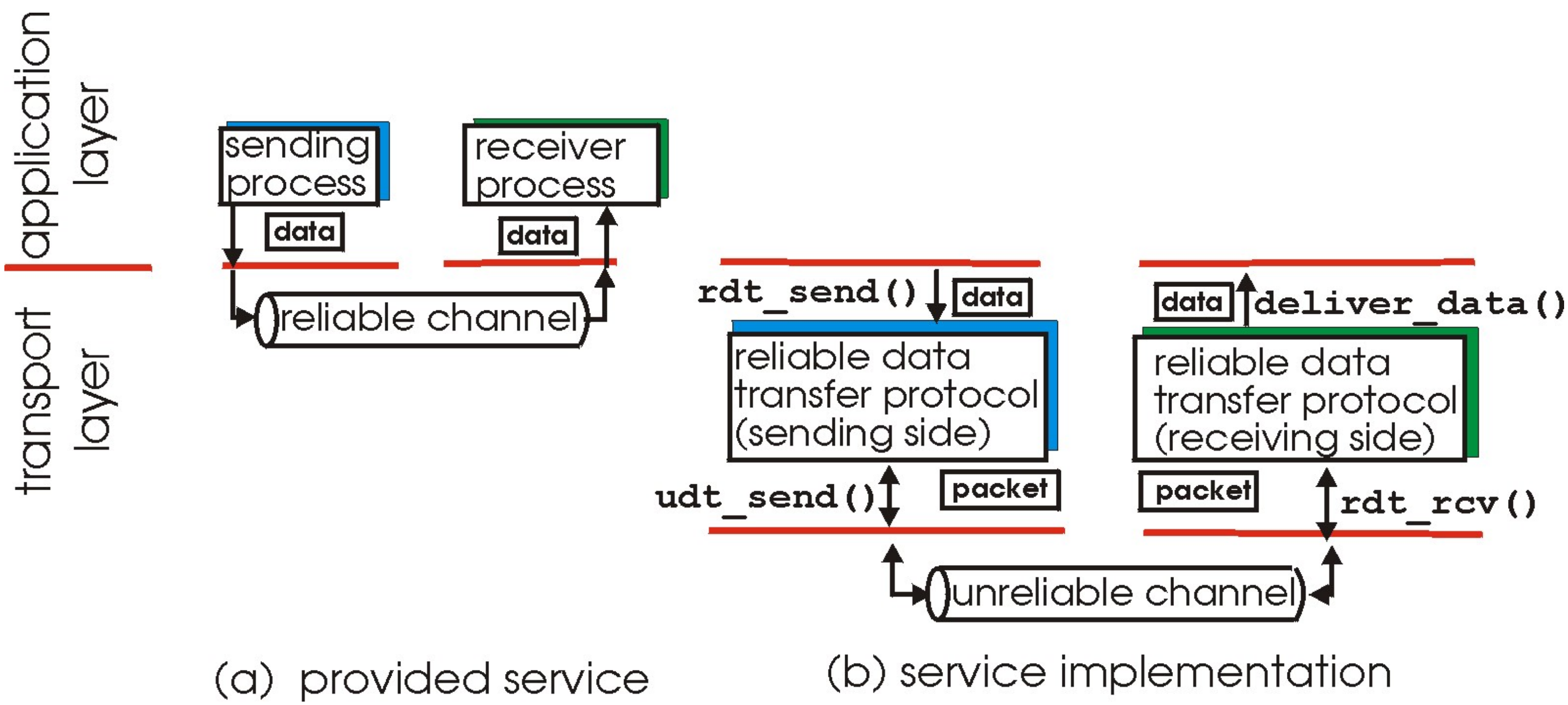
- important in application, transport, link layers
  - top-10 list of important networking topics!



- 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!

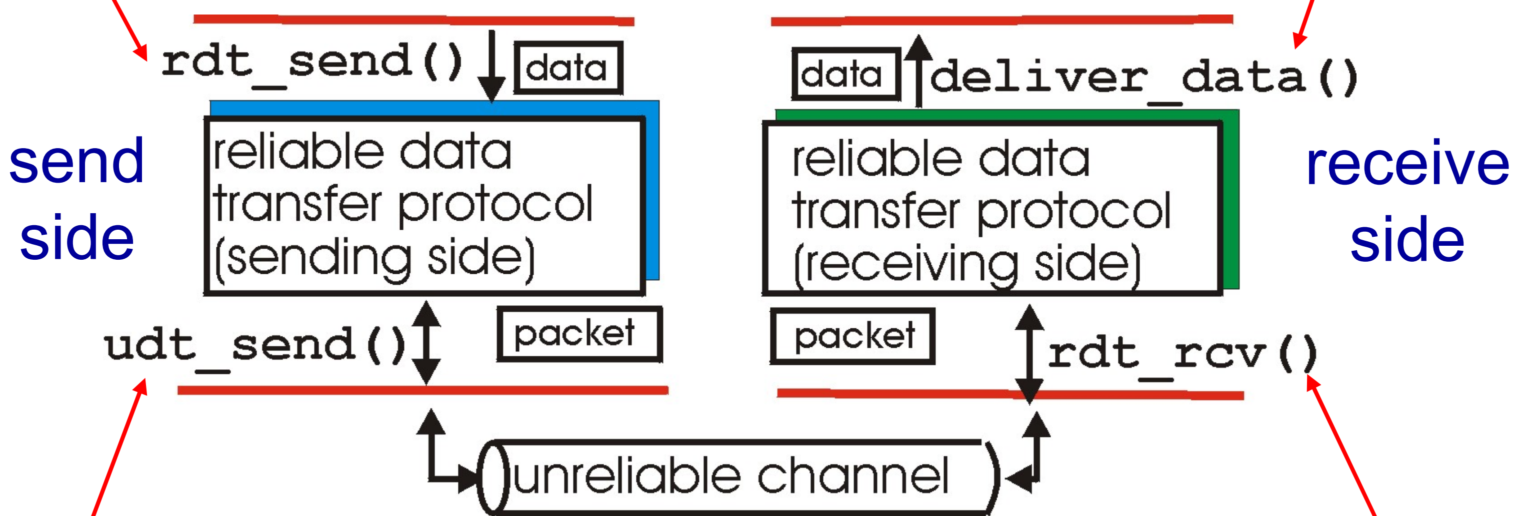


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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

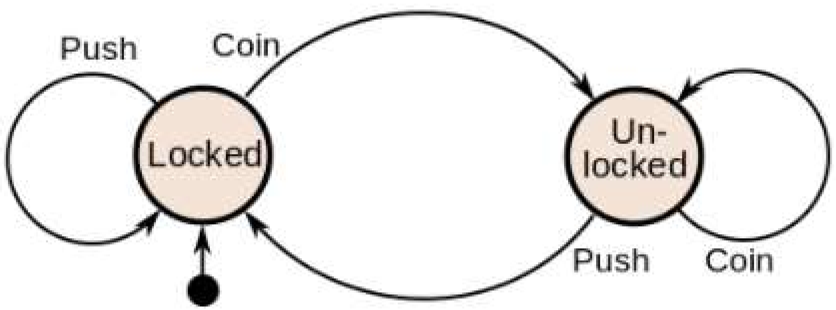


udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

What is a Finite State Machine

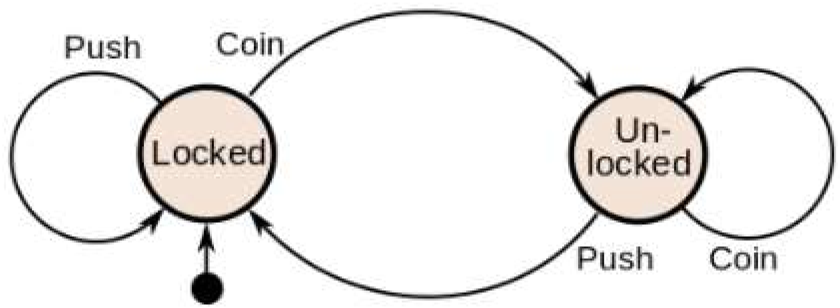
- A finite state machine shows how a system works with states and transitions between them.
- Consider a coin operated turnstile – we put in a coin, it allows us in.
- It has two states "locked" "unlocked".
- It has two events "push" and "insert coin"



Diagrams from wikipedia

Test your understanding

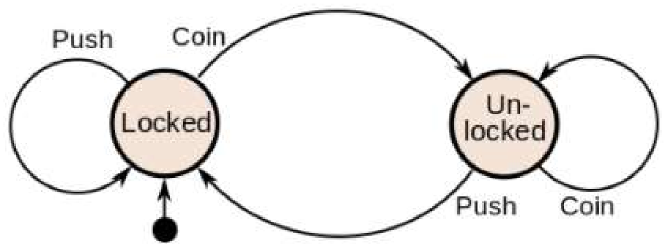
- State machine (starts in state "locked"):



  - (A) After inserting two coins what state is the machine in?
  - (B) After inserting a coin and pushing and inserting a coin what state is the machine in?

Test your understanding

- State machine (starts in state "locked"):



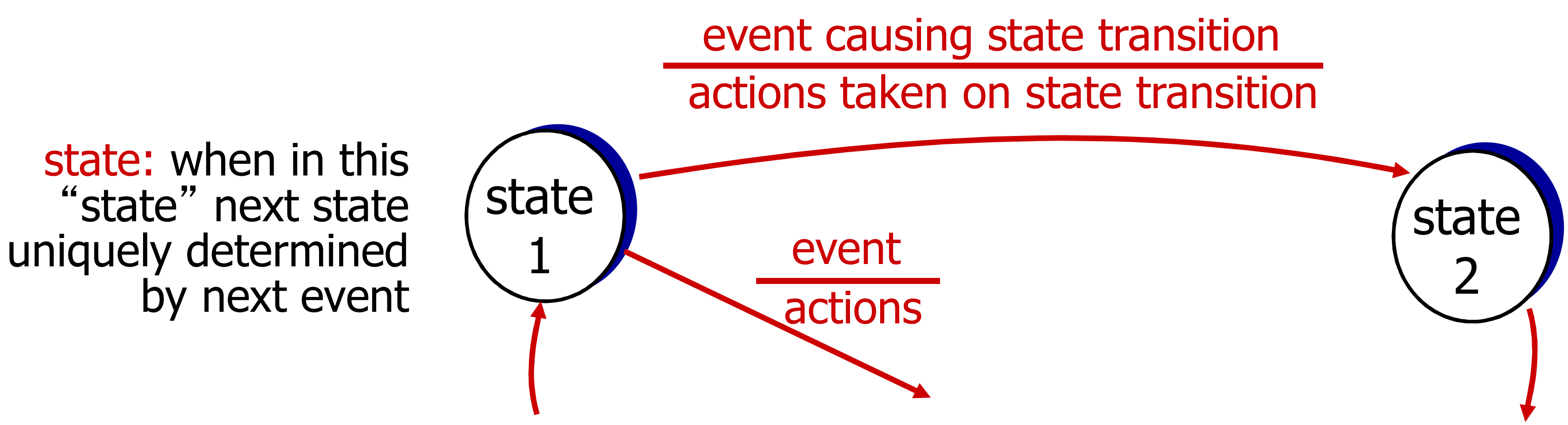
  - (A) After inserting two coins what state is the machine in?
    - (Unlocked)
  - (B) After inserting a coin and pushing and inserting a coin what state is the machine in?
    - (Unlocked)

If you got it wrong review state machines

Reliable data transfer: getting started

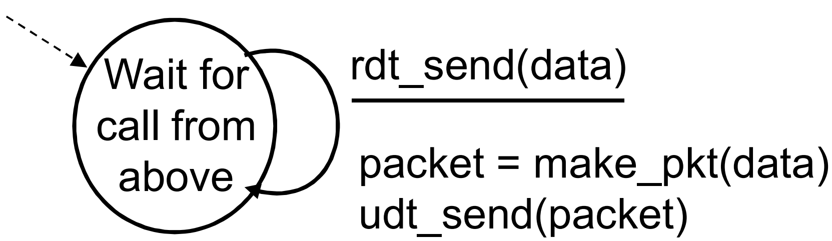
We'll:

- 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

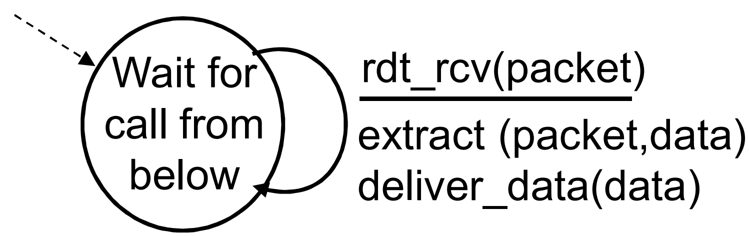


rdt1.0: reliable transfer over a reliable channel

- 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

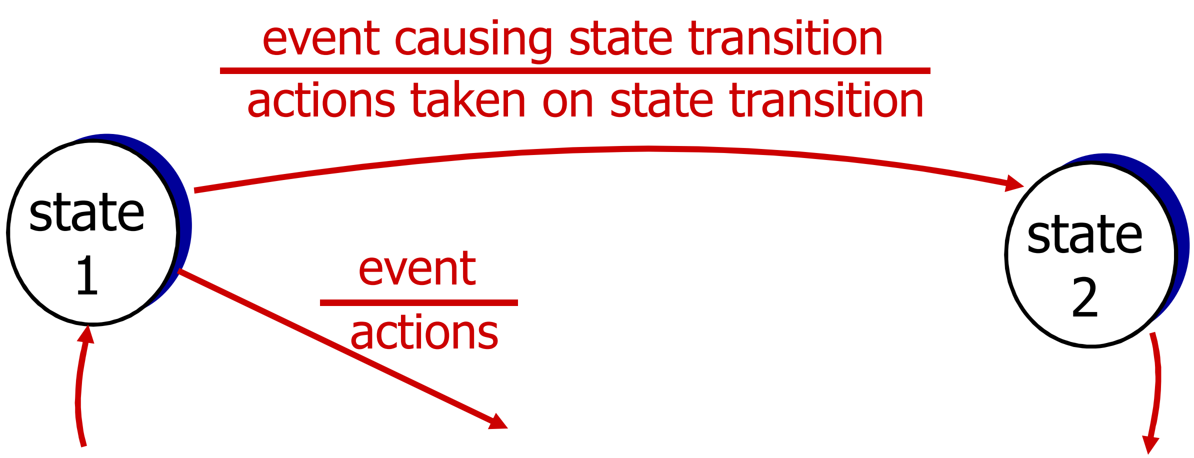
rdt2.0: channel with bit errors

- We drop the assumption that the channel has no errors.
- Now the channel could have bit errors.
- We need to be able to detect and recover from errors in transmission.
- Consider this question:
  - How do “humans” recover from errors in conversation?

Reliable data transfer: getting started

- 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

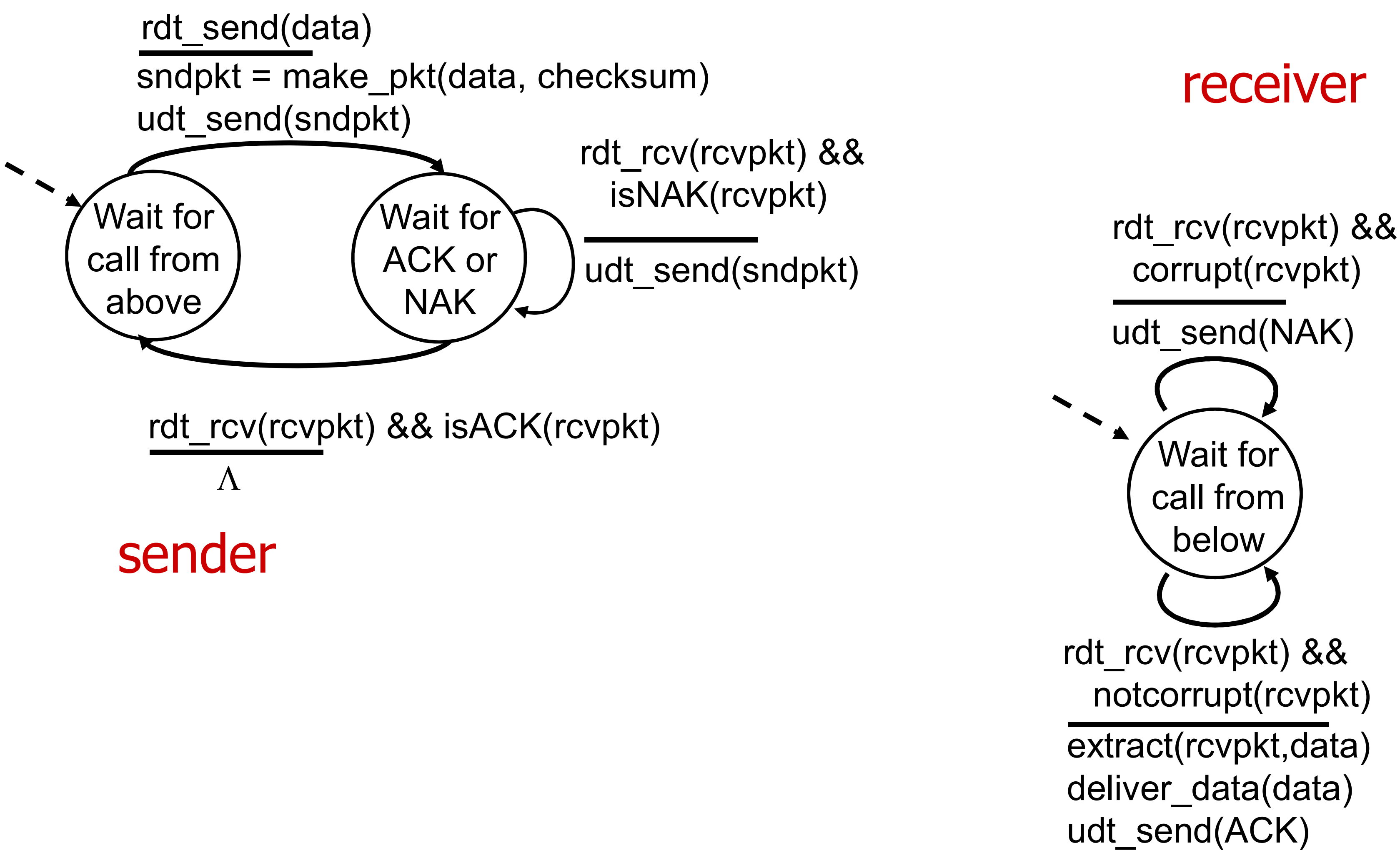
state: when in this "state" next state uniquely determined by next event



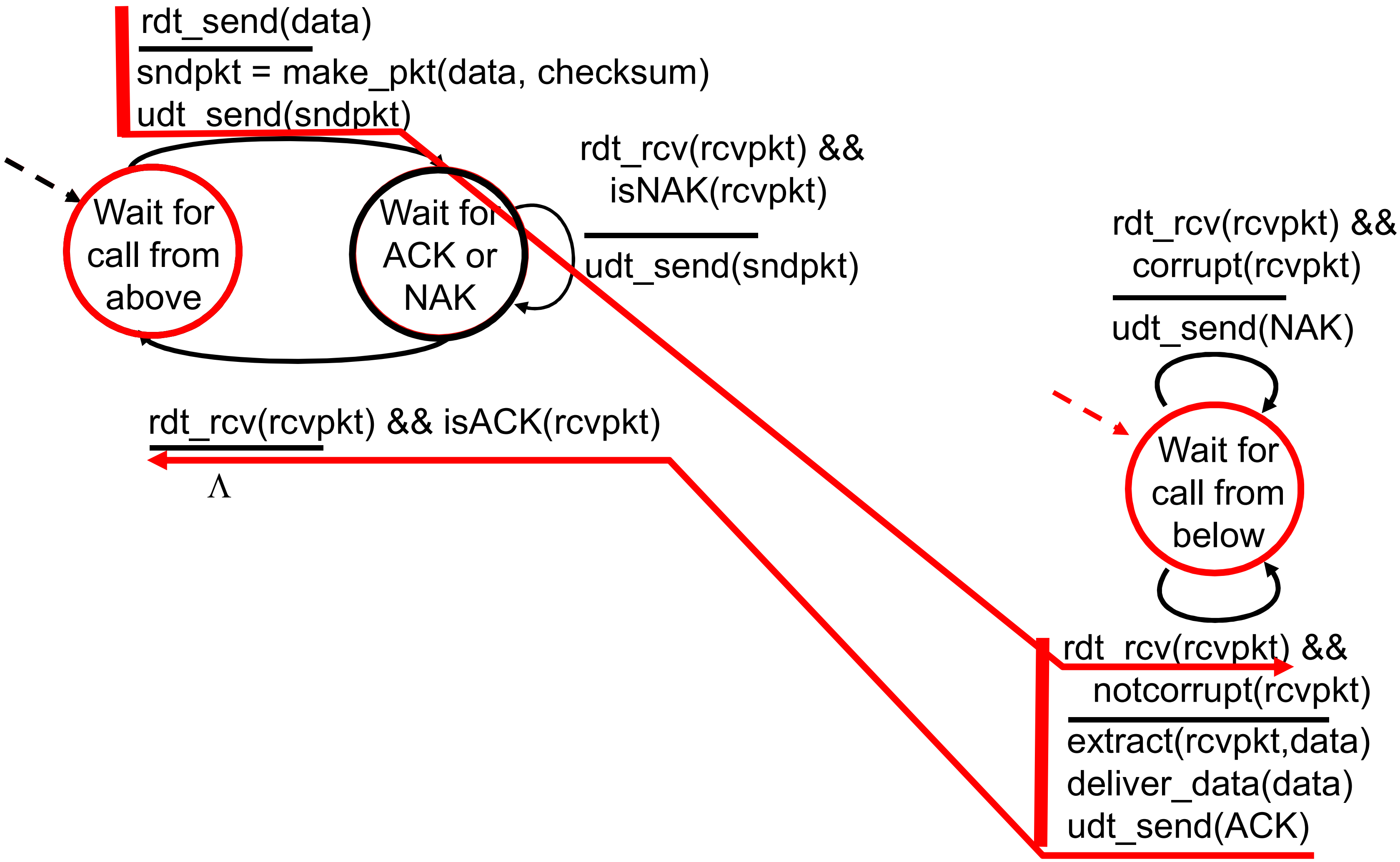
rdt2.0: channel with bit errors

- 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

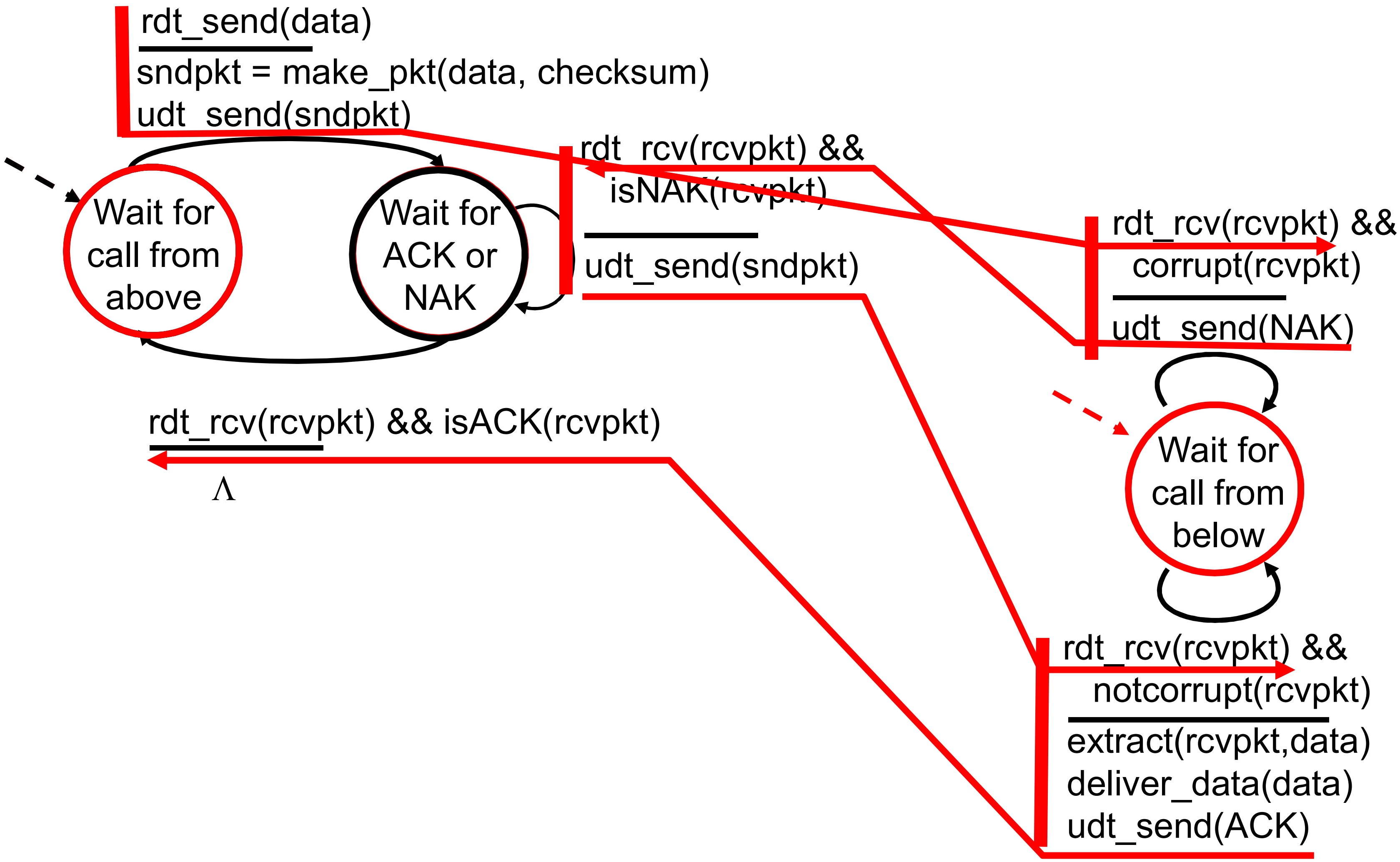
rdt2.0: FSM specification



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* (seq) to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet, then waits for receiver response

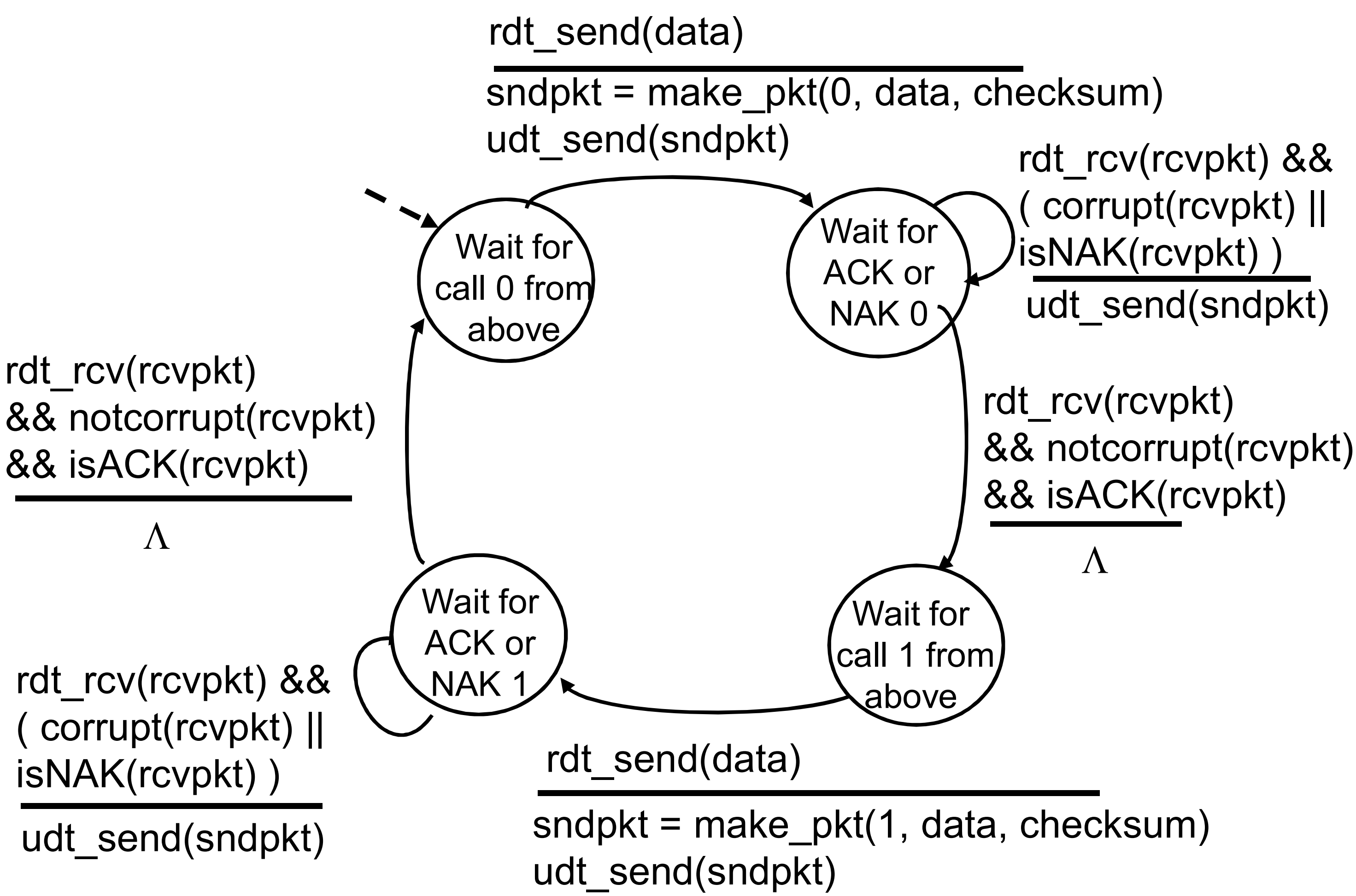
Test your understanding

- In rdt2.0:
  - A) What does the receiver send if it gets a corrupt packet?
  - B) If the sender is in "Wait for call from above" and receives a NACK what does it do?
  - C) If the sender is in "Wait for ACK or NACK" and receives a NACK what does it do?

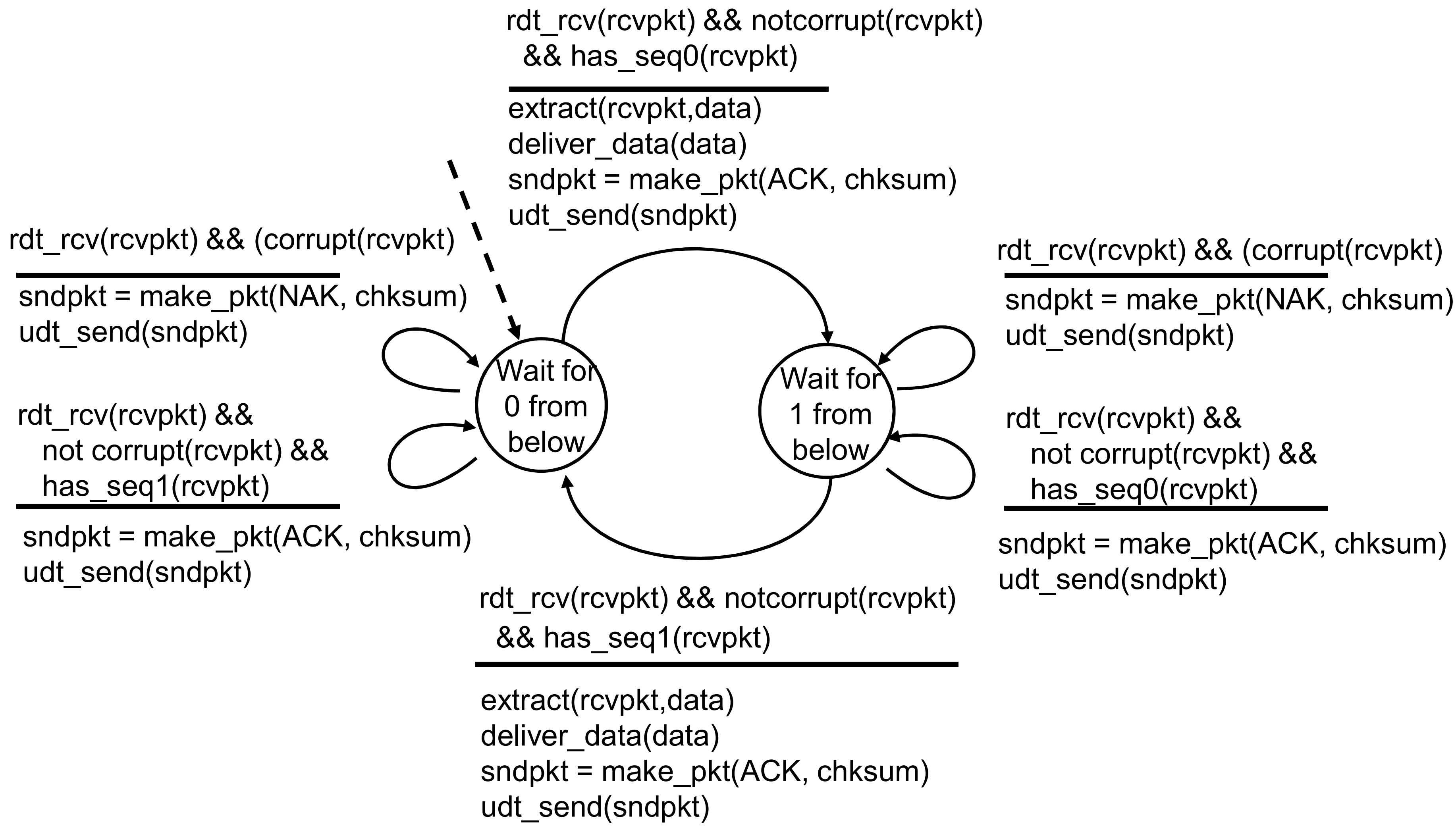
Test your understanding

- In rdt2.0:
  - A) What does the receiver send if it gets a corrupt packet?
    - Sends a NACK and stays in same state.
  - B) If the sender is in “Wait for call from above” and receives a NACK what does it do?
    - Nothing (this should never happen)
  - C) If the sender is in “Wait for ACK or NACK” and receives a NACK what does it do?
    - Sends another packet and stays in this state.
  - If you got it wrong review rdt2.0

rdt2.1: sender, handles corrupt ACK/NAKs



rdt2.1: receiver, handles corrupt ACK/NAKs



rdt2.1: discussion

<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

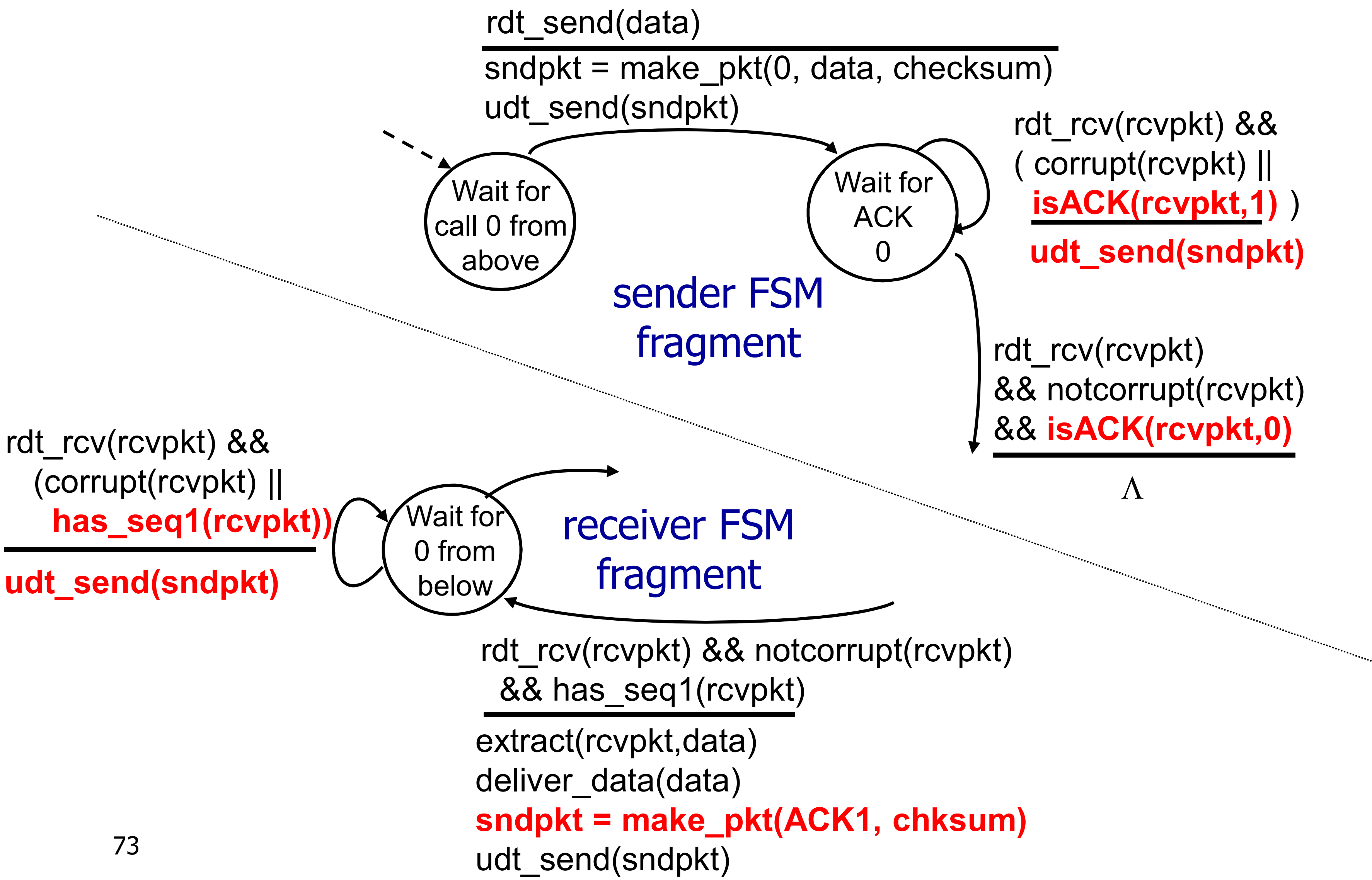
<u>receiver:</u>

- 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

rdt2.2: 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



Test your understanding

- In rdt2.2:
  - A) What does the sender do if it is in "Wait for ACK 1" and it receives ACK 0?
  - B) What does the receiver do if it is in "Wait for 1 from below" and it receives a packet with SEQ 0?
  - C) What will cause rdt2.2 to send a NACK?

Test your understanding

- In rdt2.2:
  - A) What does the sender do if it is in "Wait for ACK 1" and it receives ACK 0?
    - This is either an ACK for an old packet or a NACK. Send packet again and stay in same state.
  - B) What does the receiver do if it is in "Wait for 1 from below" and it receives a packet with SEQ 0?
    - This is an old packet. Stay in same state.
  - C) What will cause rdt2.2 to send a NACK?
    - It can never send a NACK – if it wants to indicate NACK it sends an ACK with the "wrong" number.

rdt3.0: channels with errors *and* loss

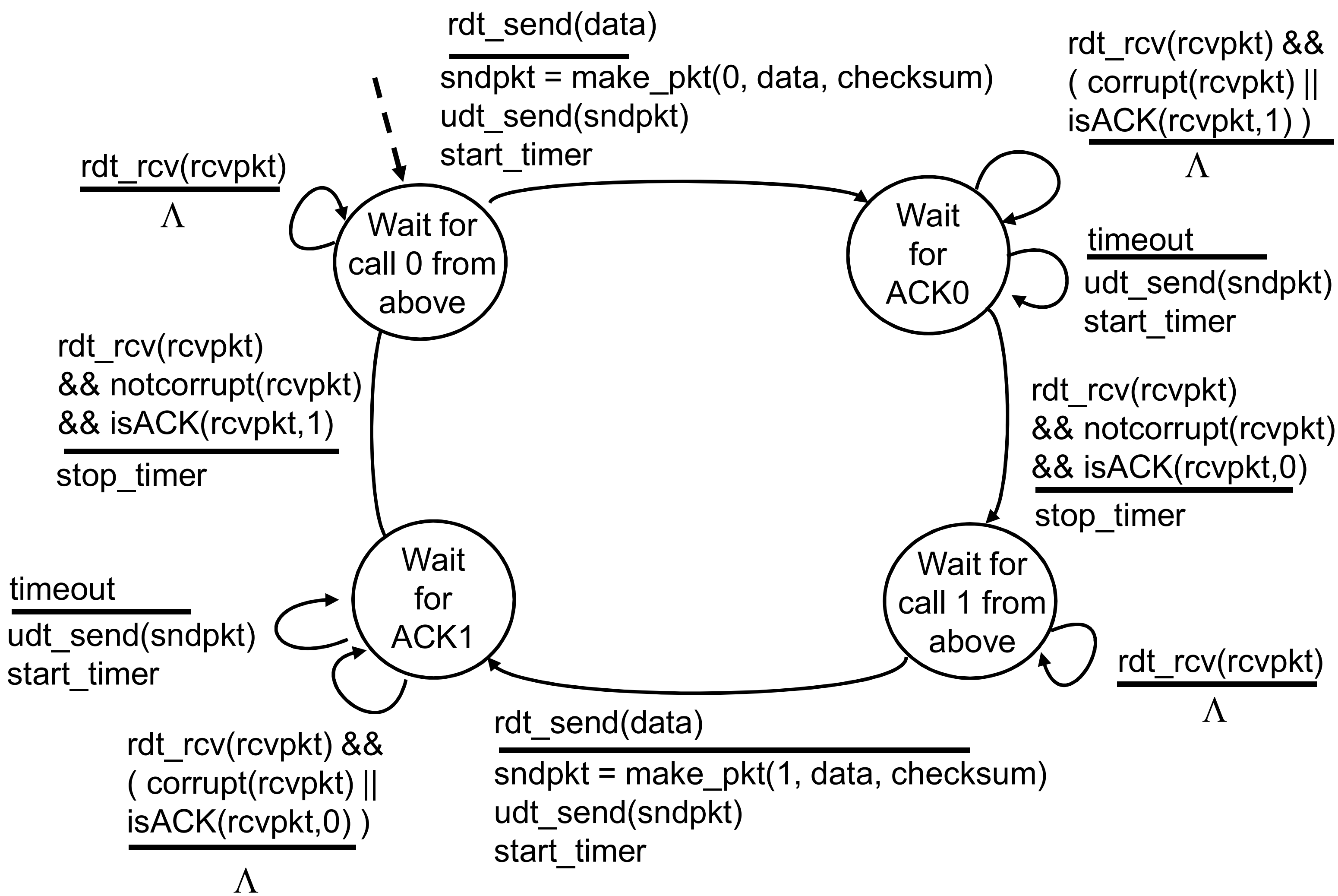
new assumption: underlying channel can also lose packets (data, ACKs)

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

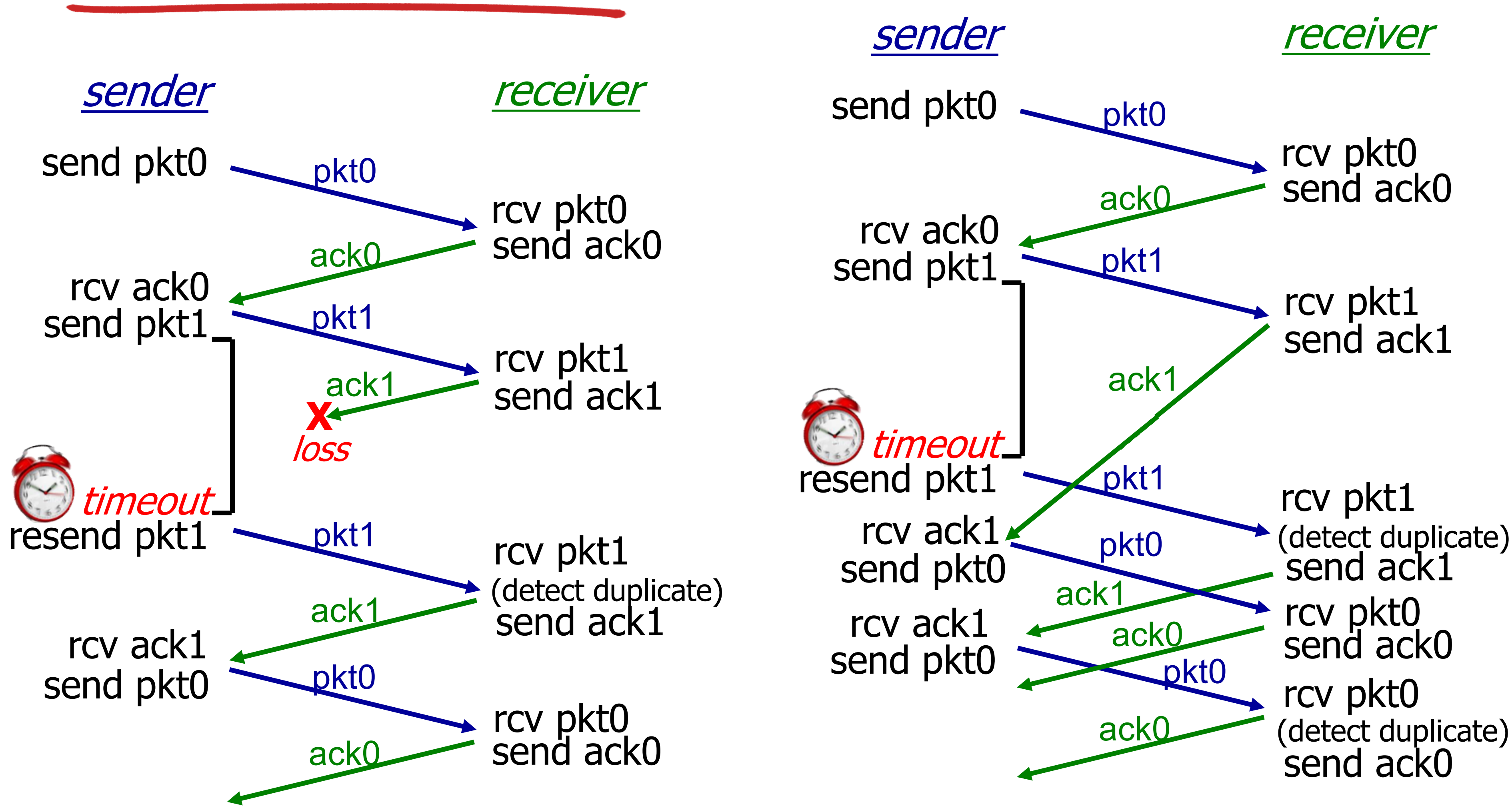
approach: sender waits "reasonable" amount of time for ACK

- 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 in action



(c) ACK loss

(d) premature timeout/ delayed ACK

Test your understanding

- In rdt3.0:
  - A) Name the two types of error rdt3.0 can deal with?
  - B) How many sequence numbers does rdt3.0 use?
  - C) How does rdt3.0 receiver indicate a packet is received but was corrupted?

Test your understanding

- In rdt3.0:
  - A) Name the two types of error rdt3.0 can deal with?
    - Packet is corrupted. Packet did not arrive at all.
  - B) How many sequence numbers does rdt3.0 use?
    - Two (0 and 1)
  - C) How does rdt3.0 receiver indicate a packet is received but was corrupted?
    - Does not use NACK. Uses ACK with the "wrong" number – 0 if it was waiting for 1 and vice versa.

What have we learned?

- We are starting on our journey to create "reliable data transfer" over an unreliable network.
- Key concepts:
  - Sequence number – measures which packet we are currently sending
  - ACK (acknowledgement) – numbered, says that a specific packet has definitely been arrived
  - Timeout – a time to wait before we think a packet is lost

Transport layer summary

- So far:
  - 3.1 transport-layer services
  - 3.2 multiplexing and demultiplexing
  - 3.3 connectionless transport: UDP
  - 3.4 principles of reliable data transfer
- Next part of lectures (part B):
  - 3.4 principles of reliable data transfer (continued)
  - 3.5 connection-oriented transport: TCP
  - 3.6 principles of congestion control
  - 3.7 TCP congestion control