Computer Networks and Applications

COMP 3331/COMP 9331

Week 2

CDN + Transport Layer Part 1

Reading Guide: Chapter 2, 2.6, 2.7 +
Chapter 3, Sections 3.1 – 3.4

Application Layer: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks (CDNs)

2.7 socket programming with UDP and TCP

Self study

Video Streaming and CDNs: context

- video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube: 37%, 16% of downstream residential ISP traffic
  - ~1B YouTube users, ~75M Netflix users
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution:* distributed, application-level infrastructure

Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)

frame *i*

temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

frame *i+1*

Multimedia: video

- CBR: (constant bit rate): video encoding rate fixed
- VBR: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples:
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, < 1 Mbps)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)

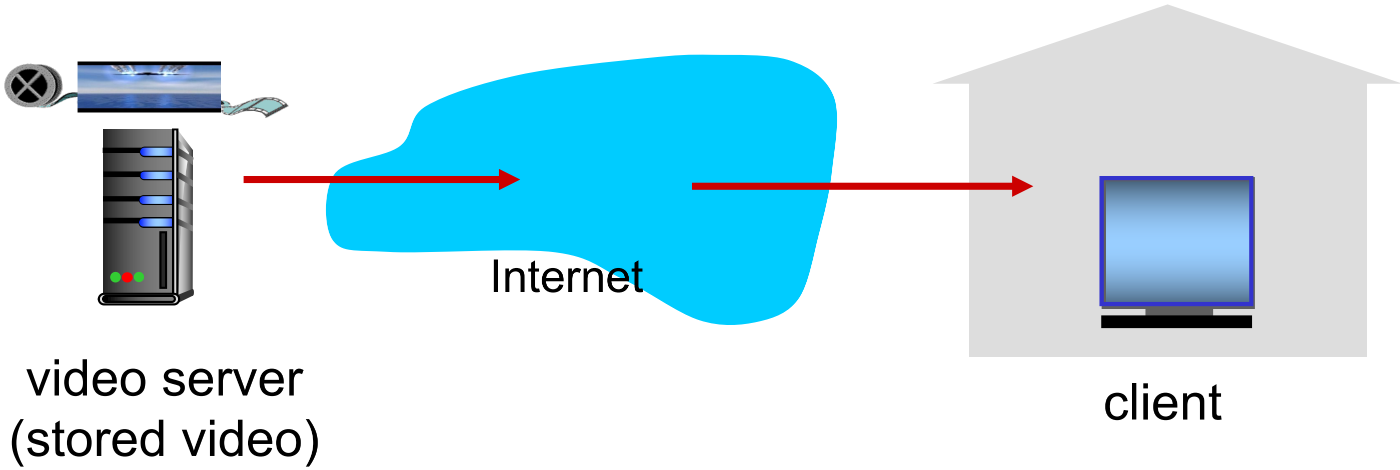
frame *i*

temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

frame *i+1*

Streaming stored video:

simple scenario:

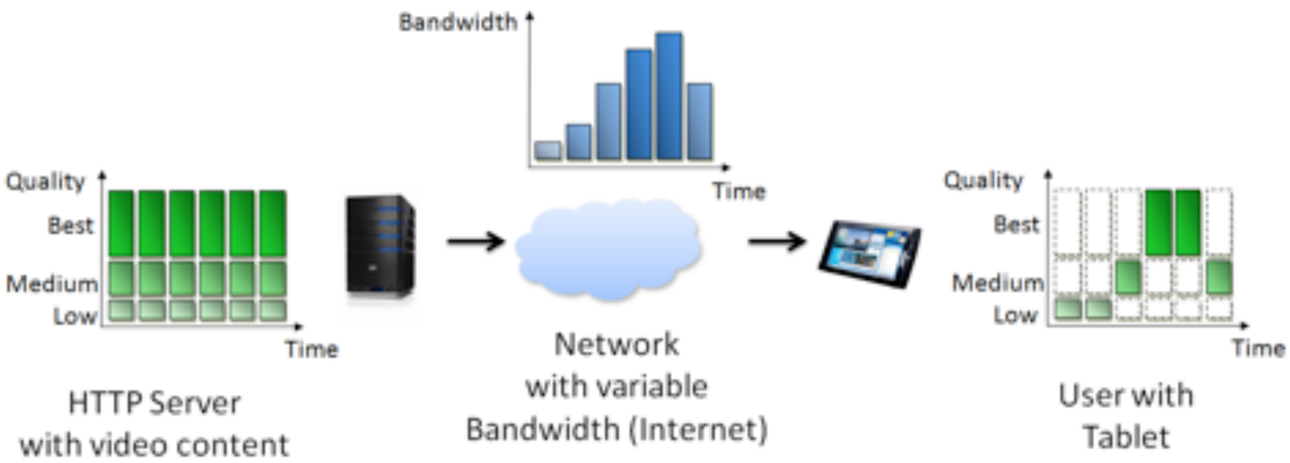


Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *server:*
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - *manifest file:* provides URLs for different chunks
- *client:*
  - periodically measures server-to-client bandwidth
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

Streaming multimedia: DASH

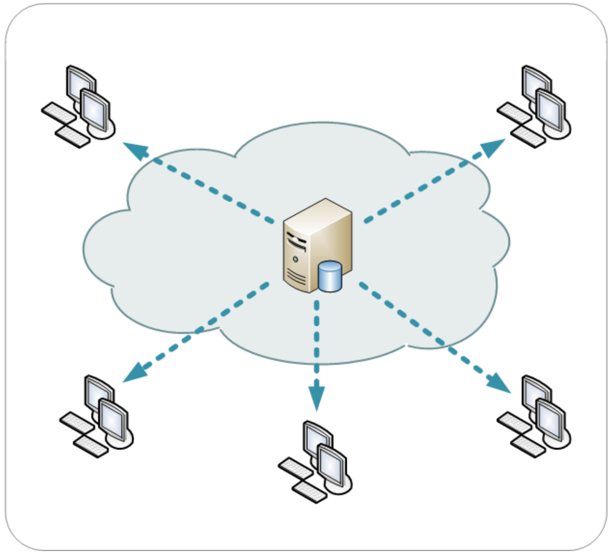
- *DASH: Dynamic, Adaptive Streaming over HTTP*
- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or has high available bandwidth)



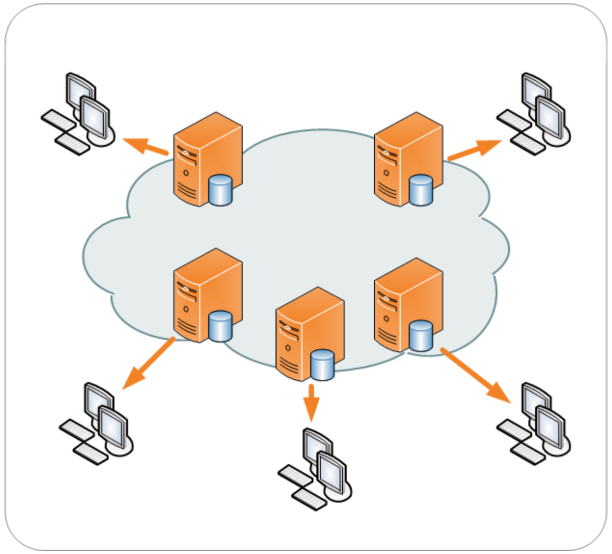
Content distribution networks

- Caching and replication as a service (amortise cost of infrastructure)
- Goal: bring content close to the user
- Large-scale distributed storage infrastructure (usually) administered by one entity
  - *e.g.*, Akamai has servers in 20,000+ locations
- Combination of (pull) caching and (push) replication
  - **Pull:** Direct result of clients' requests
  - **Push:** Expectation of high access rate

Single server



CDN



An example

```
bash-3.2$ dig www.mit.edu

; <<>> DiG 9.8.3-P1 <<>> www.mit.edu
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 27387
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 9, ADDITIONAL: 9

;; QUESTION SECTION:
;www.mit.edu.                   IN      A

;; ANSWER SECTION:
www.mit.edu.            1800    IN      CNAME   www.mit.edu.edgekey.net.
www.mit.edu.edgekey.net. 60     IN      CNAME   e9566.dscb.akamaiedge.net.
e9566.dscb.akamaiedge.net. 20   IN      A       23.77.150.125

;; AUTHORITY SECTION:
dscb.akamaiedge.net.    681     IN      NS      n4dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n5dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      a0dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n6dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n1dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n3dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n0dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n7dscb.akamaiedge.net.
dscb.akamaiedge.net.    681     IN      NS      n2dscb.akamaiedge.net.

;; ADDITIONAL SECTION:
a0dscb.akamaiedge.net.  7144    IN      AAAA    2600:1480:e800::c0
n0dscb.akamaiedge.net.  3048    IN      A       88.221.81.193
n1dscb.akamaiedge.net.  2752    IN      A       88.221.81.194
n2dscb.akamaiedge.net.  1380    IN      A       104.72.70.167
n3dscb.akamaiedge.net.  3048    IN      A       88.221.81.195
n4dscb.akamaiedge.net.  2810    IN      A       104.71.131.100
n5dscb.akamaiedge.net.  1326    IN      A       104.72.70.166
n6dscb.akamaiedge.net.  49      IN      A       104.72.70.174
n7dscb.akamaiedge.net.  2554    IN      A       104.72.70.175

;; Query time: 246 msec
;; SERVER: 129.94.172.11#53(129.94.172.11)
;; WHEN: Thu Mar  9 18:04:37 2017
;; MSG SIZE  rcvd: 463
```

Many well-known sites are hosted by CDNs. A simple way to check using dig is shown here.

Content distribution networks

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- *option 1:* single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

….quite simply: this solution *doesn't scale*

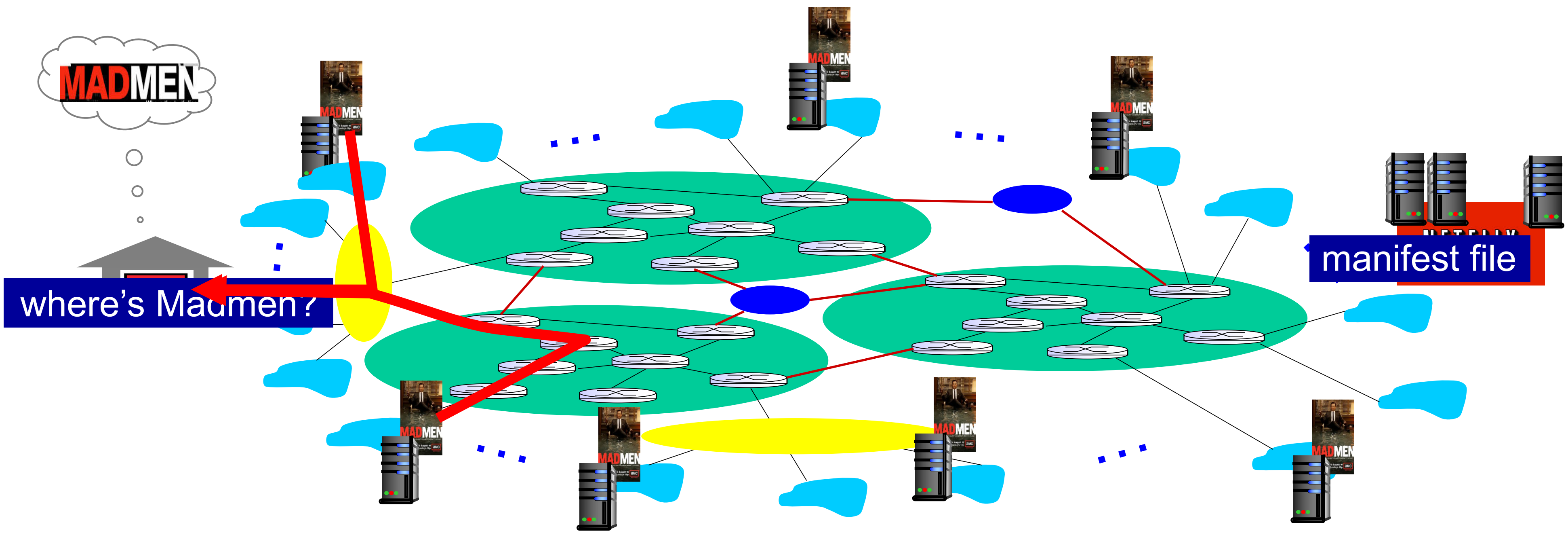
Content distribution networks

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?

- *option 2:* store/serve multiple copies of videos at multiple geographically distributed sites *(CDN)*
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - used by Akamai, thousands of locations
  - *bring home:* smaller number (10's) of larger clusters in POPs near (but not within) access networks
    - used by Limelight

Content Distribution Networks (CDNs)

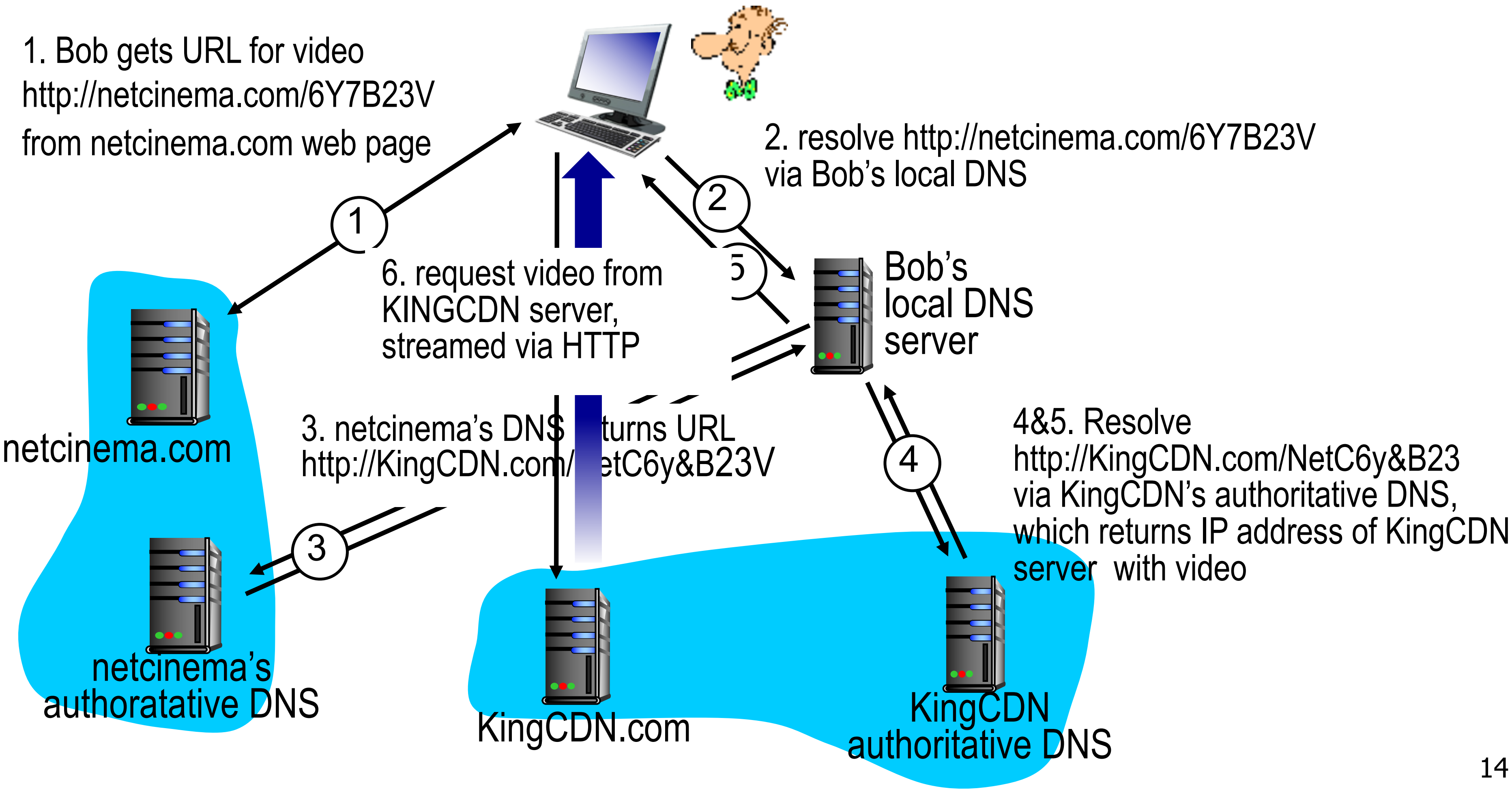
- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested



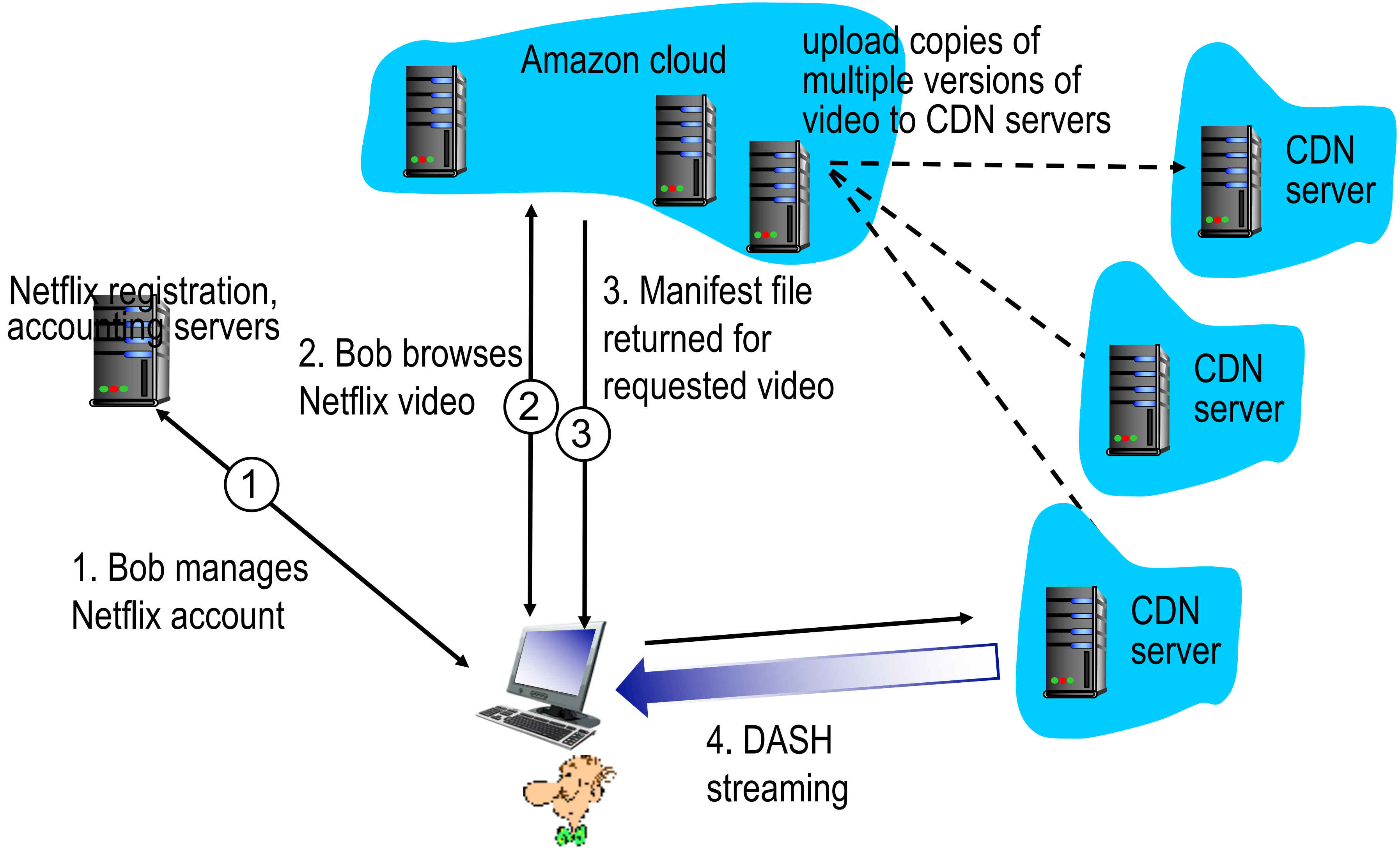
CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V

- video stored in CDN at http://KingCDN.com/NetC6y&B23V

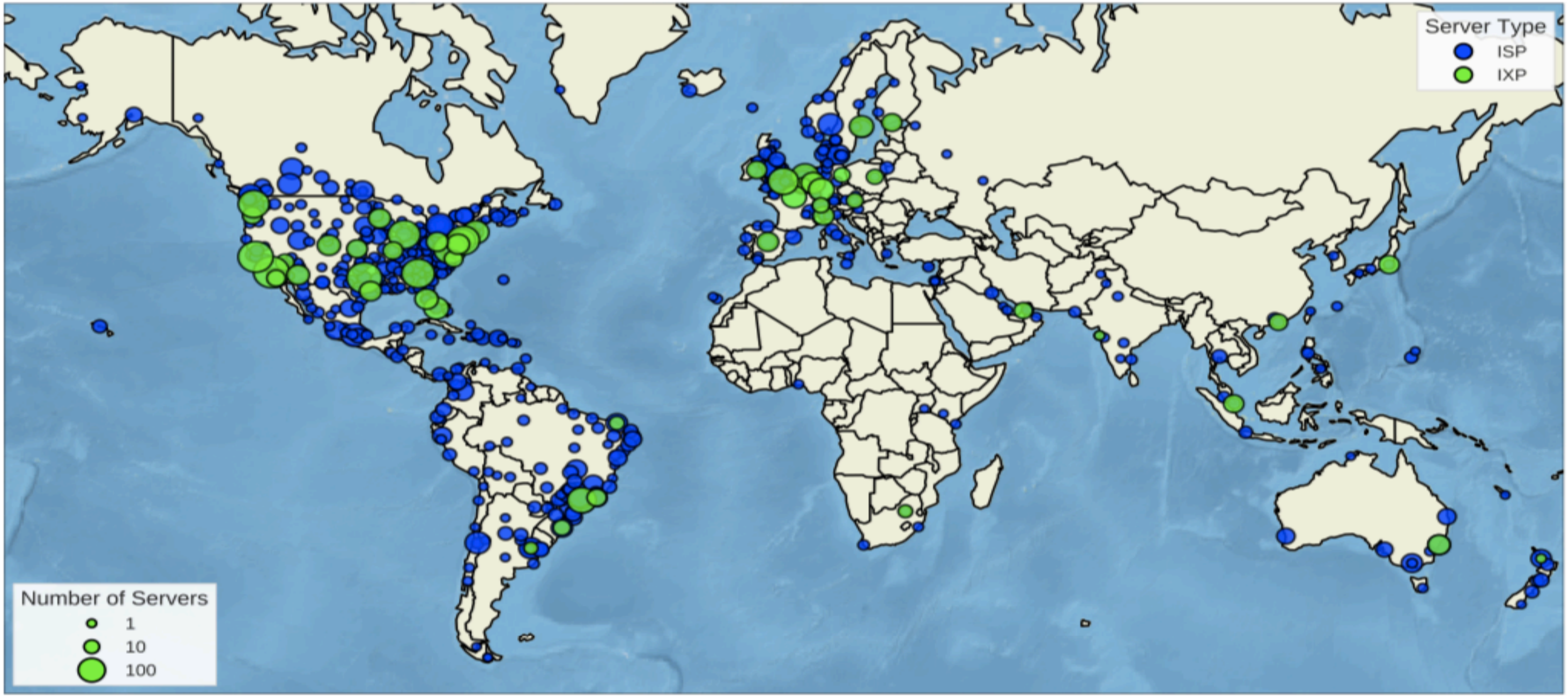


Case study: Netflix



Uses Push caching (during offpeak)
Preference to "deep inside" followed by "bring home"

NetFlix servers (snap shot from Jan 2018)



Researchers from Queen Mary University of London (QMUL) traced server names that are sent to a user's computer every time they play content on Netflix to find the location of the 8492 servers (4152 ISP, 4340 IXP).They have been found to be scattered across 578 locations around the world.

Quiz: CDN

- The role of the CDN provider's authoritative DNS name server in a content distribution network, simply described, is:
    a) to provide an alias address for each browser access to the "origin server" of a CDN website
    b) to map the query for each CDN object to the CDN server closest to the requestor (browser)
    c) to provide a mechanism for CDN "origin servers" to provide paths for clients (browsers)
    d) none of the above, CDN networks do not use DNS

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

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

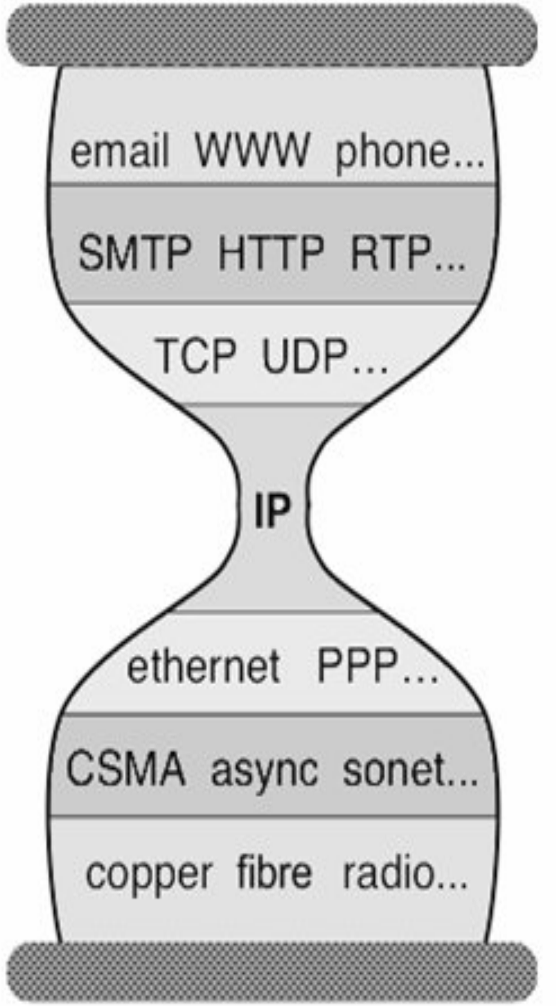
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Transport layer

- Moving “down” a layer
- Current perspective:
  - Application is the boss….
  - Usually executing within the OS Kernel
  - The network layer is ours to command !!



Network layer (context)

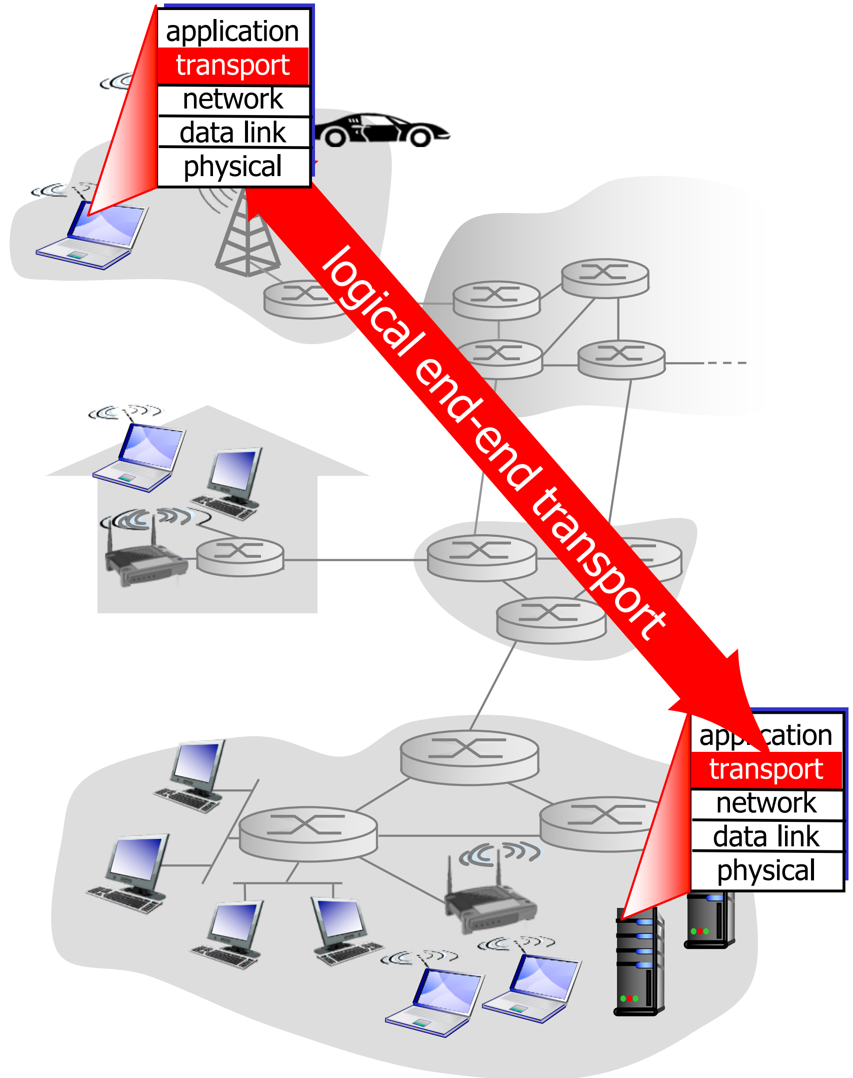
- What it does: finds paths through network
  - Routing from one end host to another

- What it doesn't:
  - Reliable transfer: "best effort delivery"
  - Guarantee paths
  - Arbitrate transfer rates

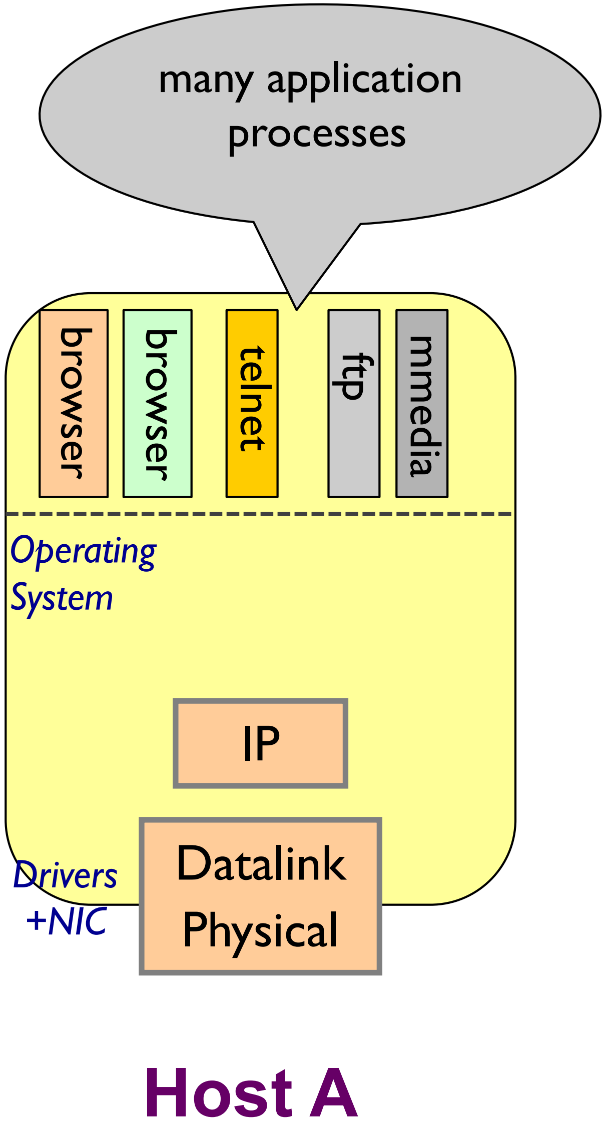
- For now, think of the network layer as giving us an "API" with one function: *sendtohost(data, host)*
  - Promise: the data will go to that (usually!!)

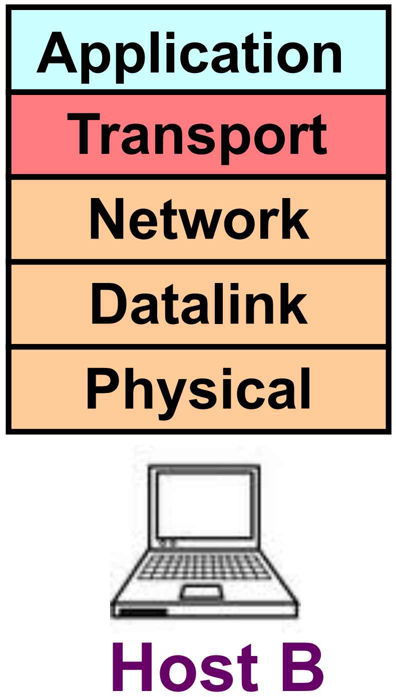
Transport services and protocols

- 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
  - Exports services to application that network layer does not provide

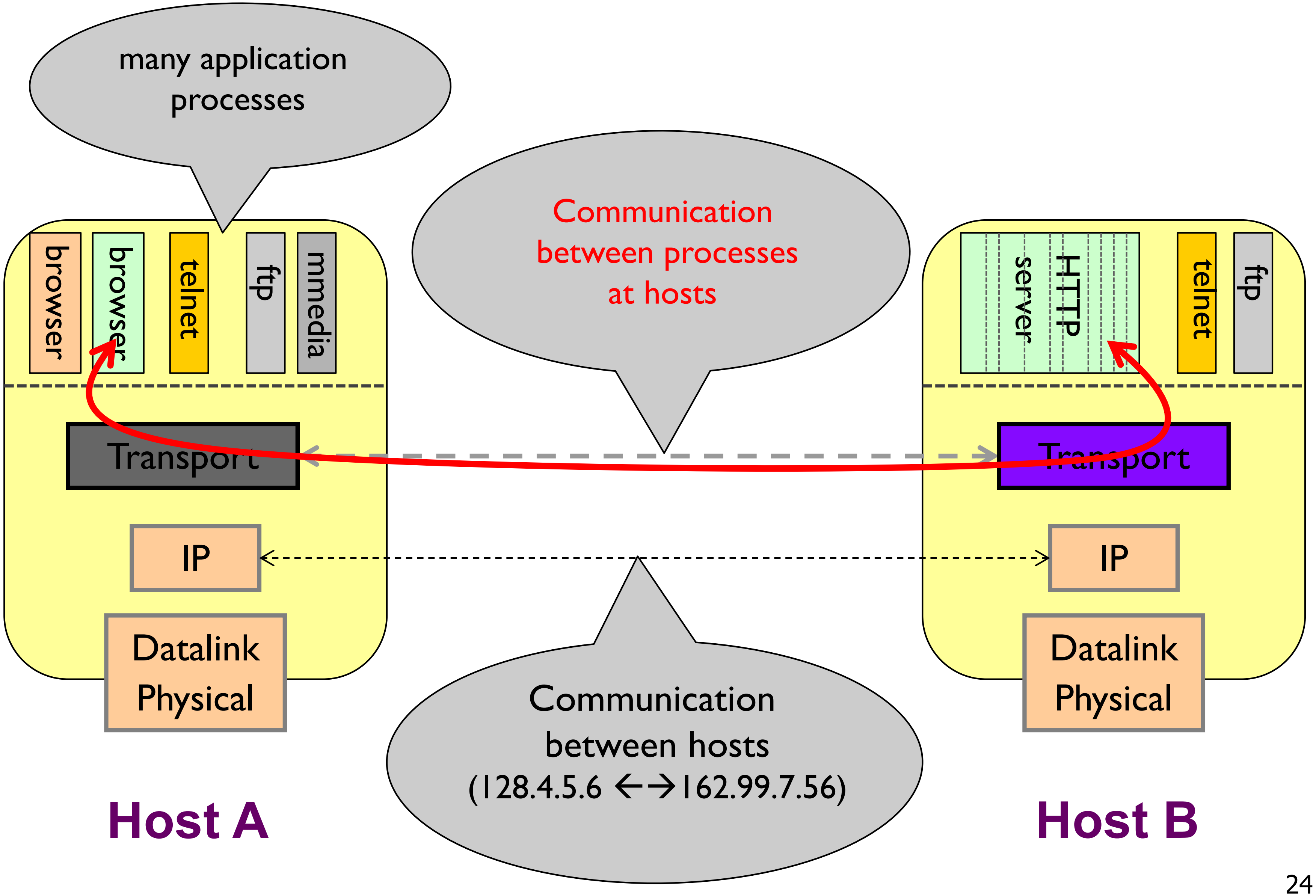


Why a transport layer?





Why a 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

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

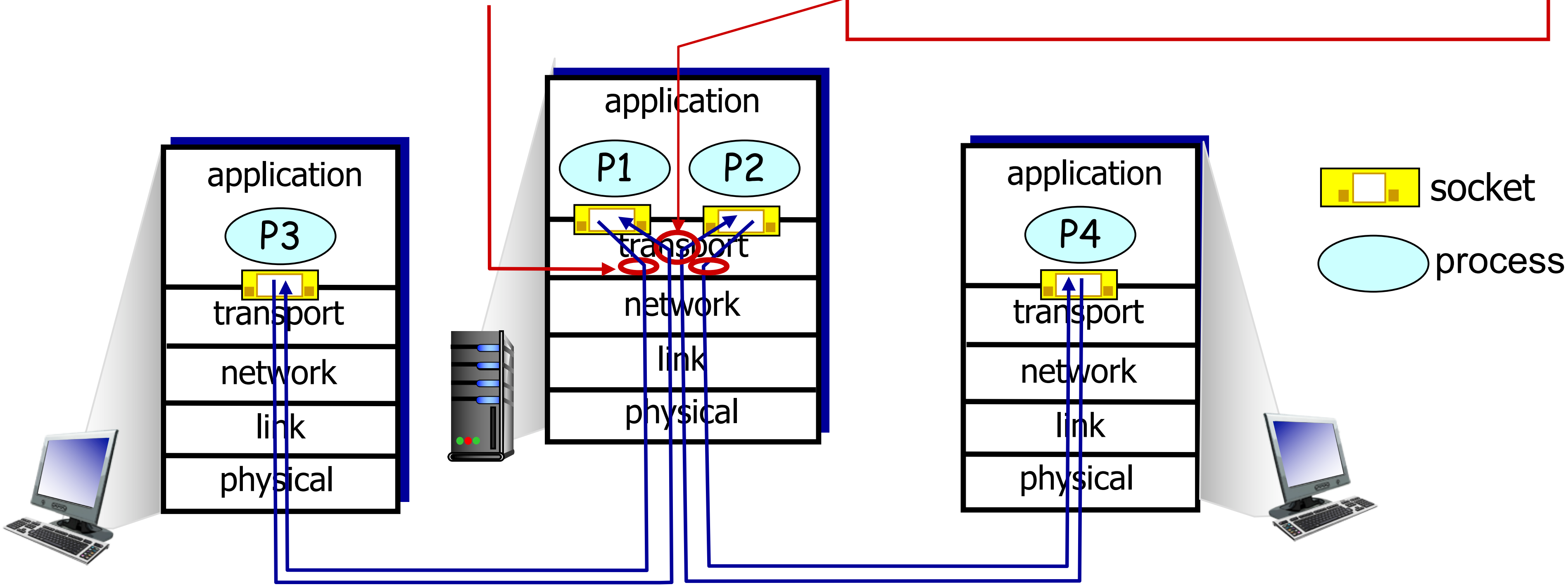
3.6 principles of congestion control

3.7 TCP congestion control

Multiplexing/demultiplexing

multiplexing 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



Note: The network is a shared resource. It does not care about your applications, sockets, etc.

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 #

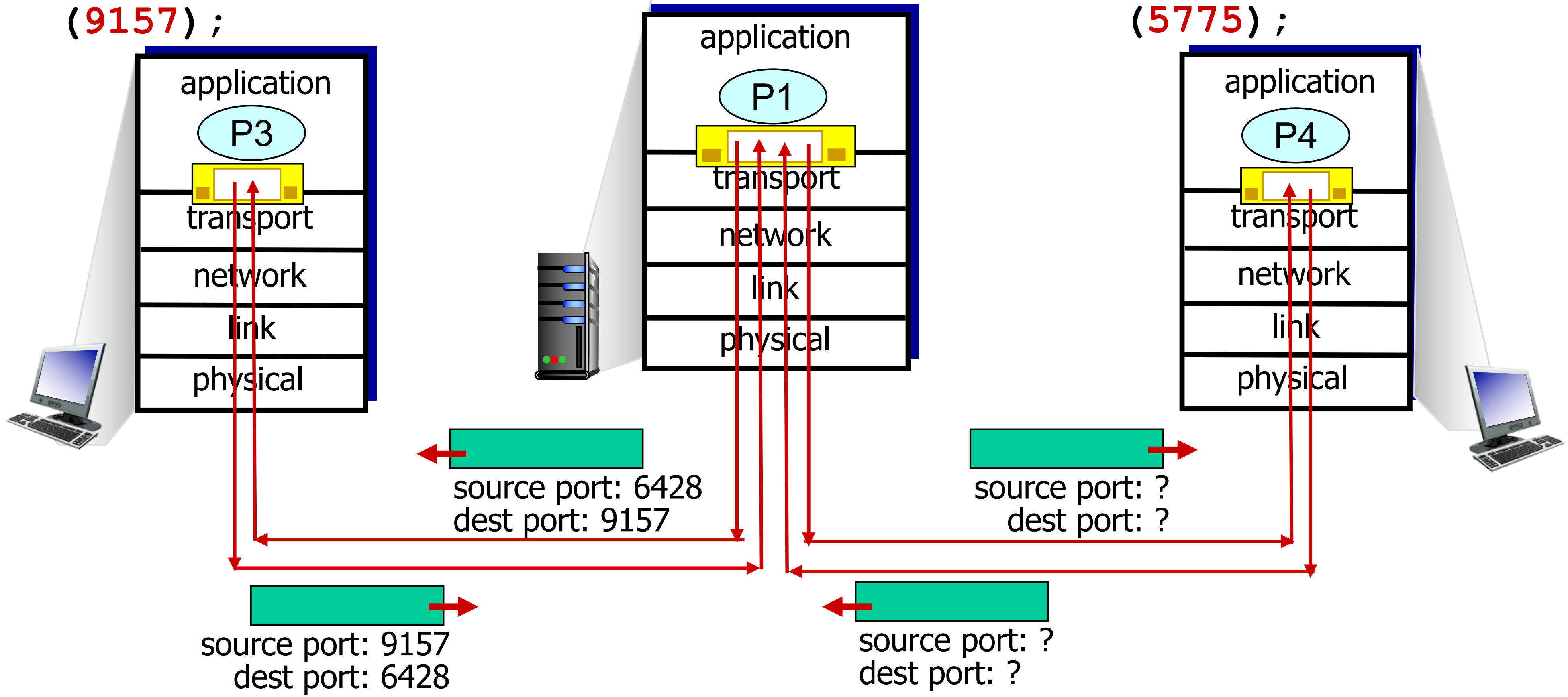
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
 serverSocket = new
 DatagramSocket
 (6428);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```

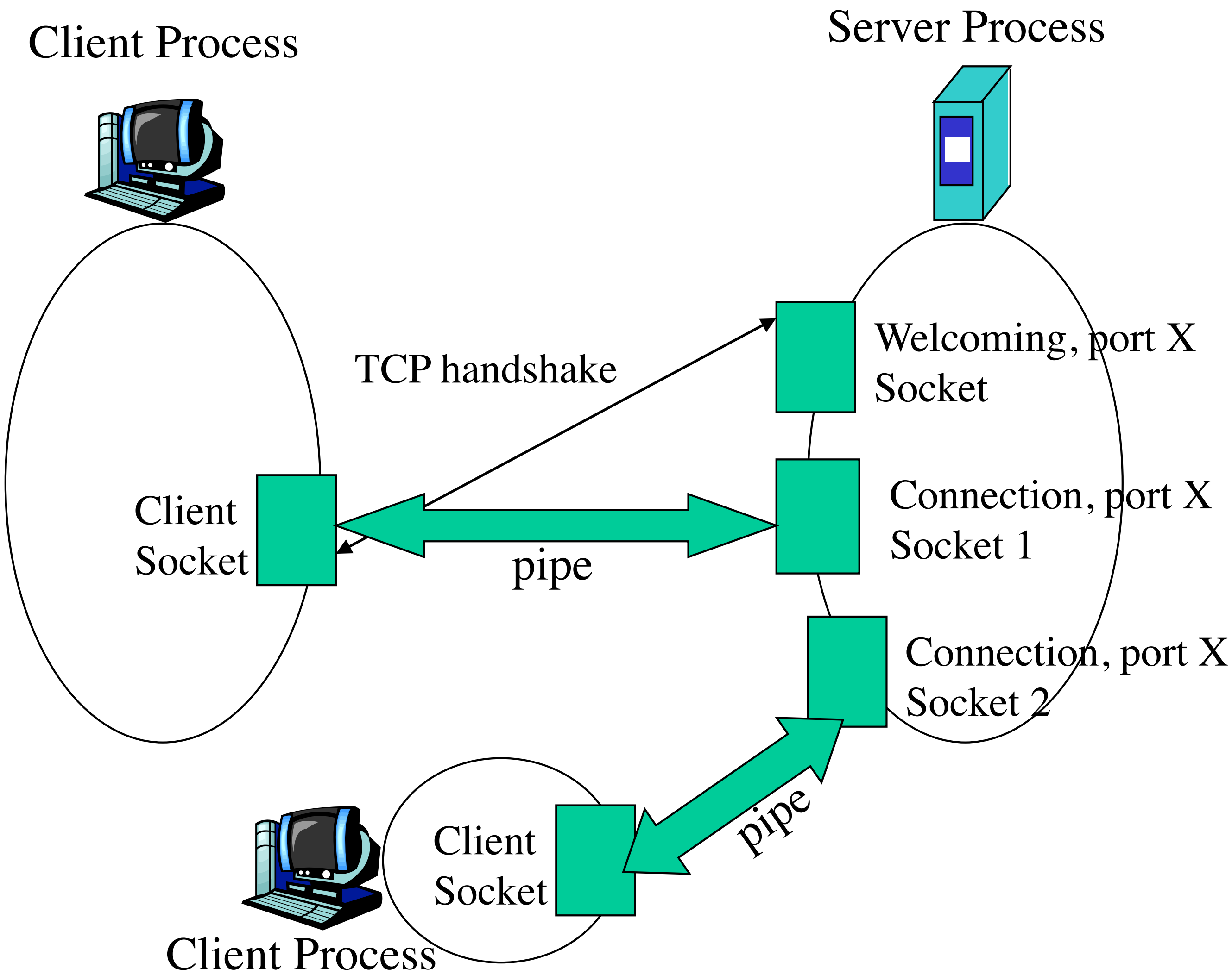


Connection-oriented demux

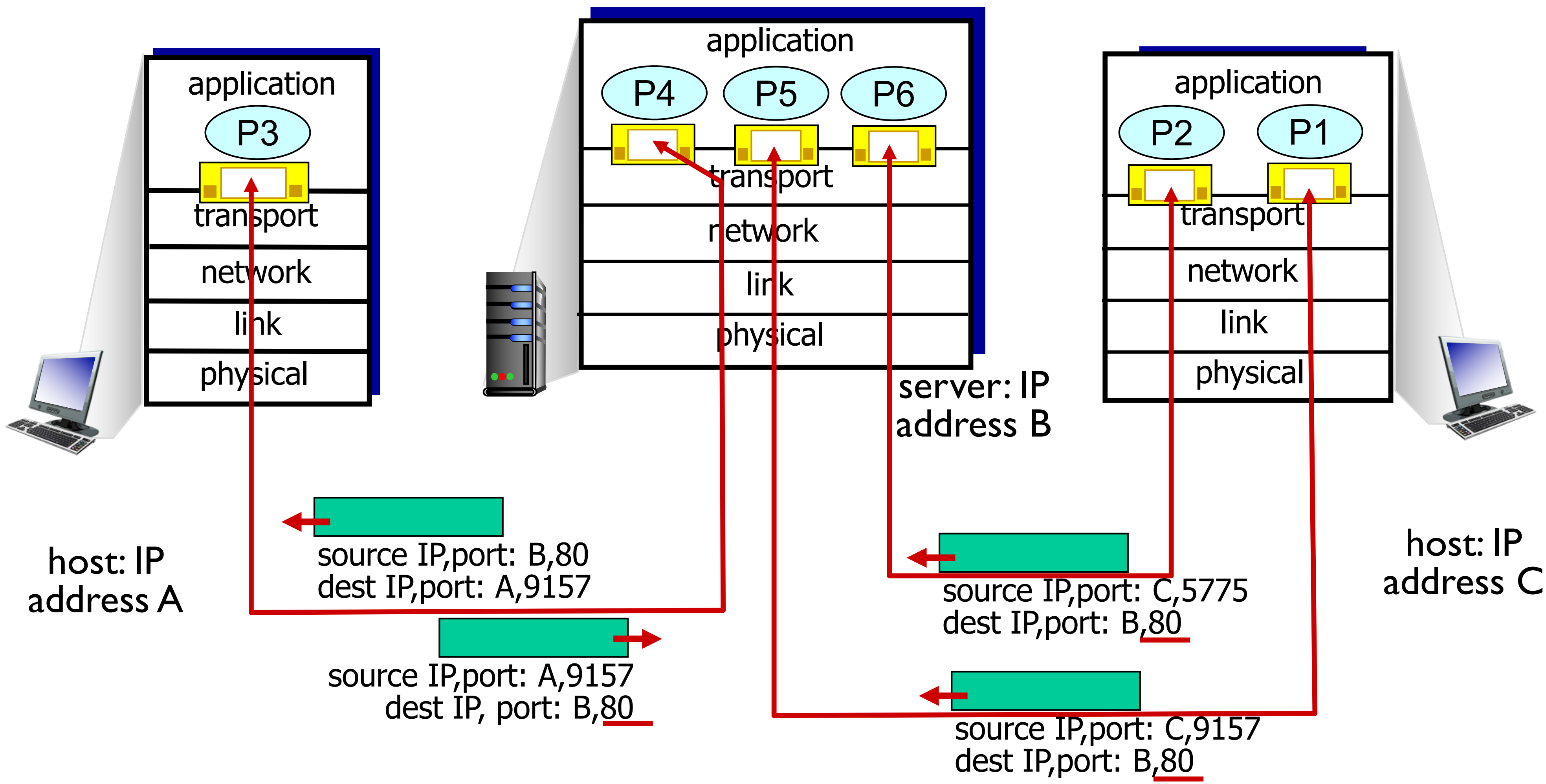
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- 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

Revisiting TCP Sockets



Connection-oriented demux: example



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

Quiz: Sockets

- Suppose that a Web Server runs in Host C on port 80. Suppose this server uses persistent connections, and is currently receiving requests from two different Hosts, A and B.

  - Are all of the requests being sent through the same socket at host C ?

  - If they are being passed through different sockets, do both of the sockets have port 80?

May I scan your ports?

http://netsecurity.about.com/cs/hackertools/a/aa121303.htm

- Servers wait at open ports for client requests
- Hackers often perform *port scans* to determine open, closed and unreachable ports on candidate victims
- Several ports are well-known
  - <1024 are reserved for well-known apps
  - Other apps also use known ports
    - MS SQL server uses port 1434 (udp)
    - Sun Network File System (NFS) 2049 (tcp/udp)
- Hackers can exploit known flaws with these known apps
  - Example: Slammer worm exploited buffer overflow flaw in the SQL server
- How do you scan ports?
  - Nmap, Superscan, etc

http://www.auditmypc.com/

https://www.grc.com/shieldsup

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

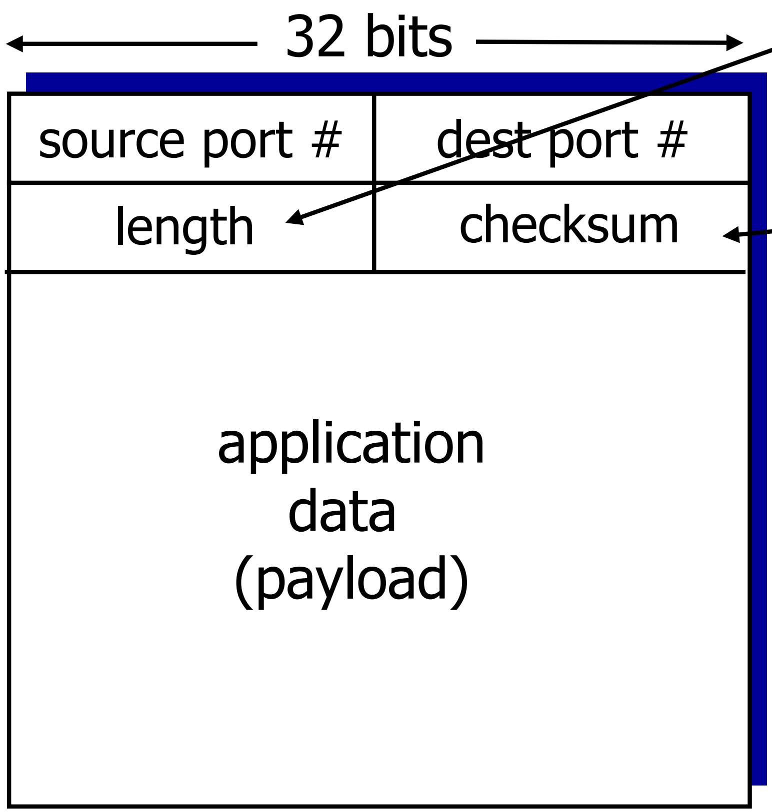
3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

- "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: segment header



UDP segment format

length, in bytes of UDP segment, including header

2 bytes Optional Checksum

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
  - Router memory errors
  - Driver bugs
  - Electromagnetic interference

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:

- Add all the received together as 16-bit integers
- Add that to the checksum
- If the result is not 1111 1111 1111 1111, there are errors !

UDP: Checksum

- *Checksum is the16-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.
- Checksum header, data and pre-pended IP pseudo-header
- But the header contains the checksum itself?
- What's IP pseudo-header?

bits	0 – 7	8 – 15	16 – 23	24 – 31
0	Source address			
32	Destination address			
64	Zeros	Protocol	UDP length	
96	Source Port		Destination Port	
128	Length		Checksum	
160+	Data			

IP pseduo-header

UDP header

UDP payload

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

4500 003C 1C46 4000 4006 B1E6 AC10 0A63 AC10 0A0C

4500 -> 0100010100000000
003C -> 0000000000111100
1C46 -> 0001110001000110
4000 -> 0100000000000000
4006 -> 0100000000000110
0000 -> 0000000000000000
AC10 -> 1010110000010000
0A63 -> 0000101001100011
AC10 -> 1010110000010000
0A0C -> 0000101000001100

4500 -> 0100010100000000
003c -> 0000000000111100
453C -> 0100010100111100

453C -> 0100010100111100
1c46 -> 0001110001000110
6182 -> 0110000110000010

6182 -> 0110000110000010
4000 -> 0100000000000000
A182 -> 1010000110000010

A182 -> 1010000110000010
4006 -> 0100000000000110
E188 -> 1110000110001000

E188 -> 1110000110001000
AC10 -> 1010110000010000
18D98 -> 11000110110011000

18D98 -> 11000110110011000
8D99 -> 1000110110011001

8D99 -> 1000110110011001
0A63 -> 0000101001100011
97FC -> 1001011111111100

97FC -> 1001011111111100
AC10 -> 1010110000010000
1440C -> 10100010000001100

1440C -> 10100010000001100
440D -> 0100010000001101

440D -> 0100010000001101
0A0C -> 0000101000001100
4E19 -> 0100111000011001

B1E6 ->1011000111100110

UDP Applications

- Latency sensitive/time critical
  - Quick request/response (DNS, DHCP)
  - Network management (SNMP)
  - Routing updates (RIP)
  - Voice/video chat
  - Gaming (especially FPS)

- Error correction unnecessary (periodic messages)

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

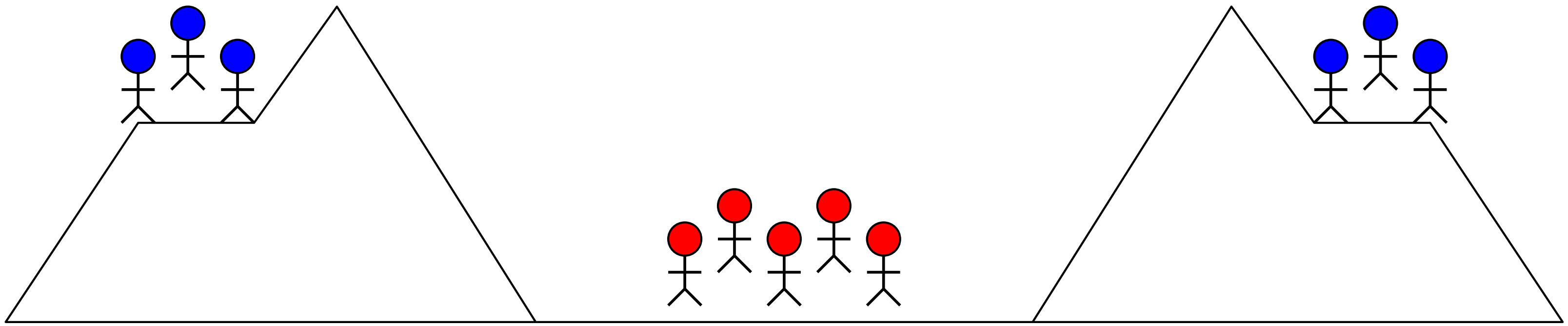
3.6 principles of congestion control

3.7 TCP congestion control

Reliable Transport

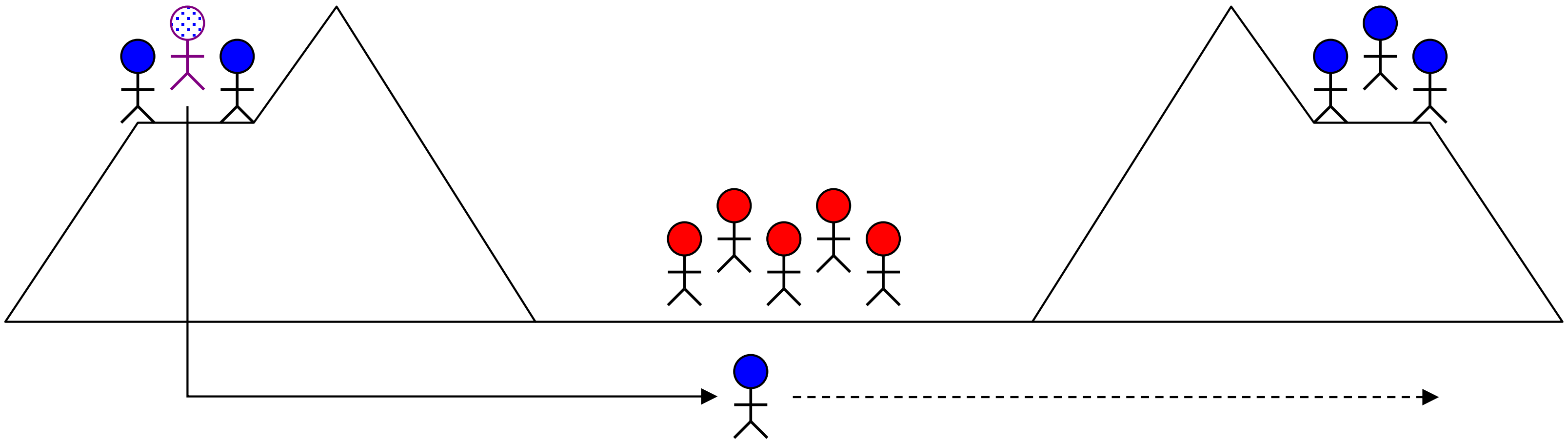
- In a perfect world, reliable transport is easy
- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost
  - a packet is delayed (*why?*)
  - packets are reordered (*why?*)
  - a packet is duplicated (*why?*)

The Two Generals Problem



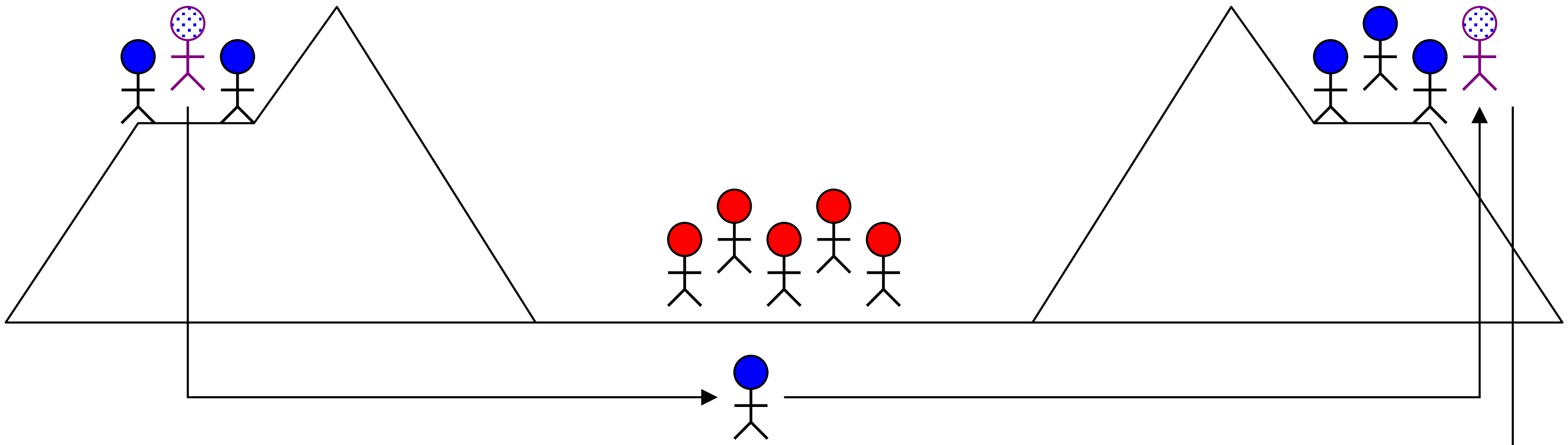
- Two army divisions (blue) surround enemy (red)
  - Each division led by a general
  - Both must agree when to simultaneously attack
  - If either side attacks alone, defeat
- Generals can only communicate via messengers
  - Messengers may get captured (unreliable channel)

The Two Generals Problem



- How to coordinate?
  - Send messenger: "Attack at dawn"
  - What if messenger doesn't make it?

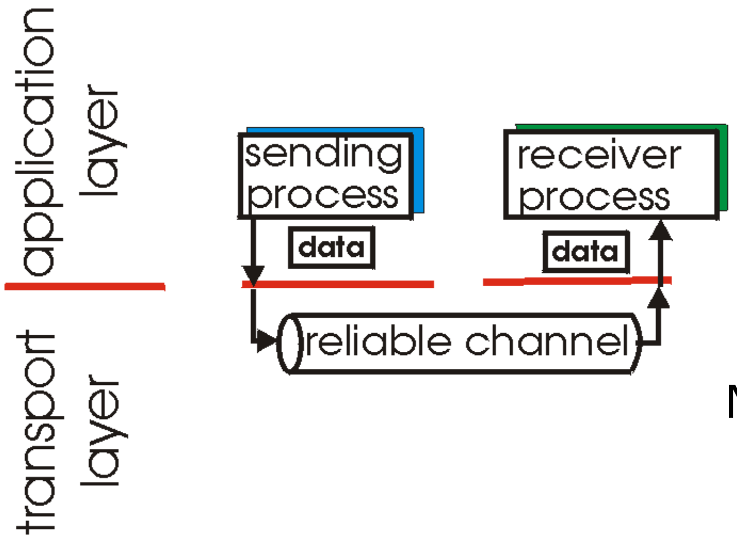
The Two Generals Problem



- How to be sure messenger made it?
  - Send acknowledgement: “We received message”

Principles of reliable data transfer

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



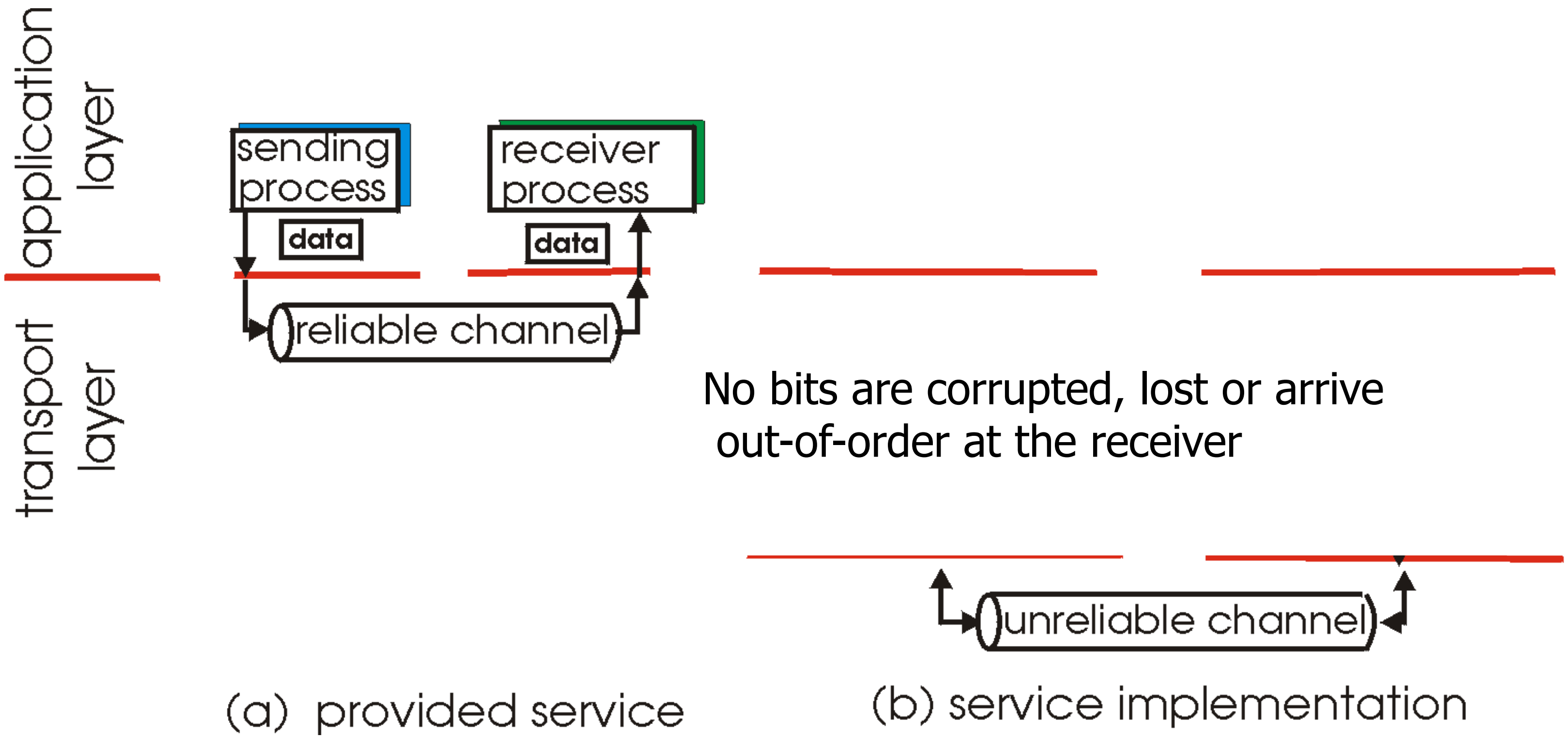
No bits are corrupted, lost or arrive out-of-order at the receiver

(a) provided service

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

Principles of reliable data transfer

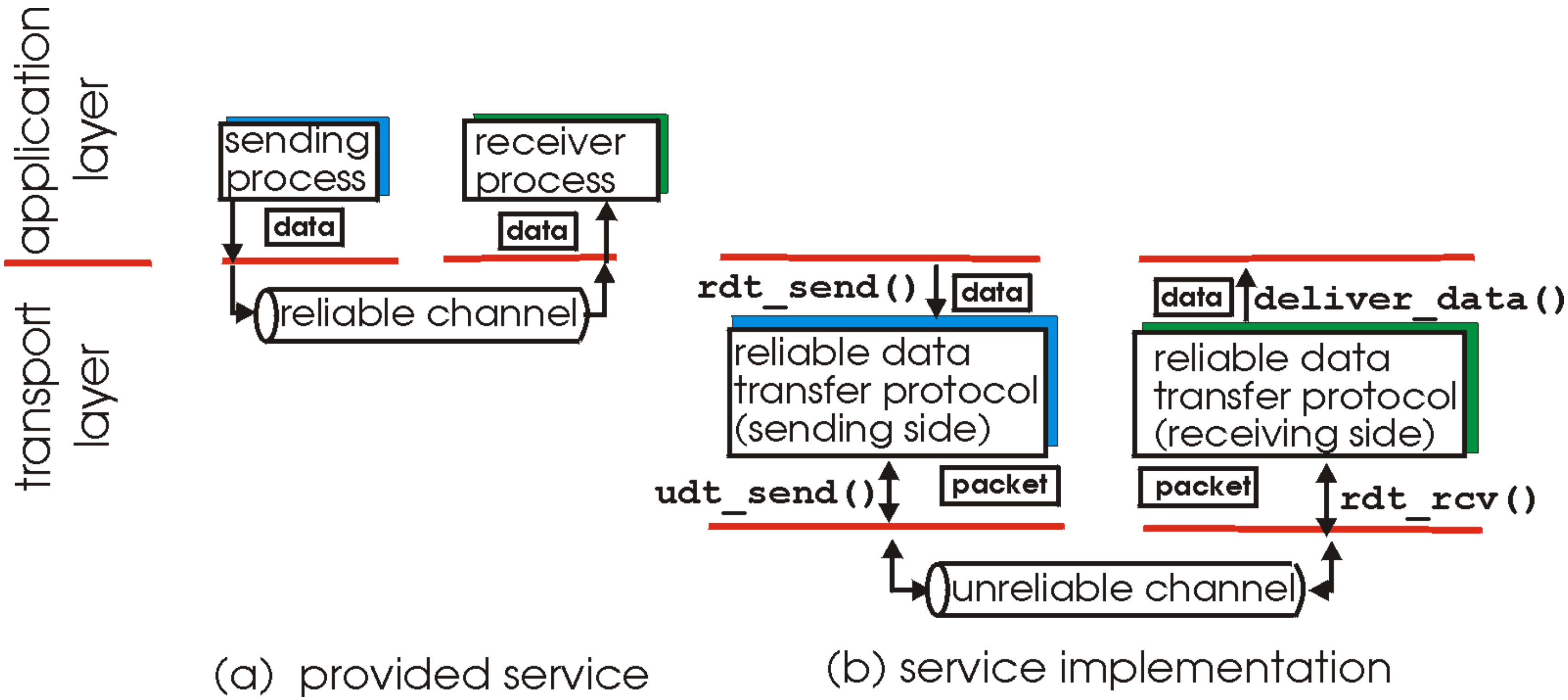
- important in application, transport, link layers
  - top-10 list of important networking topics!



- 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

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!
- Channel will not re-order packets

rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Transport layer does nothing !

rdt2.0: channel with bit errors

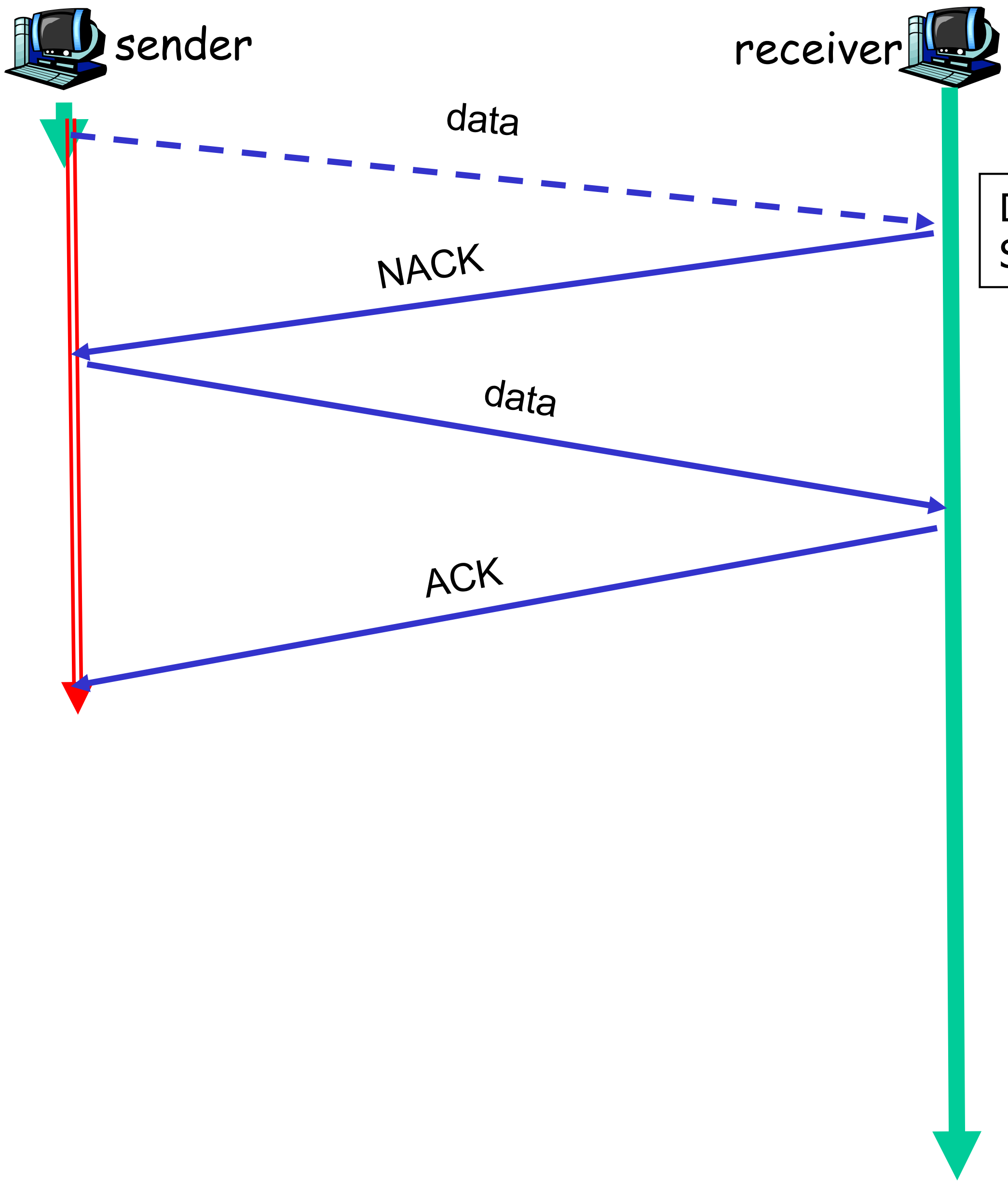
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

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
  - retransmission

Global Picture of rdt2.0



Dotted line: erroneous transmission
Solid line: error-free transmission

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: 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 "expected" pkt should have seq # of 0 or 1

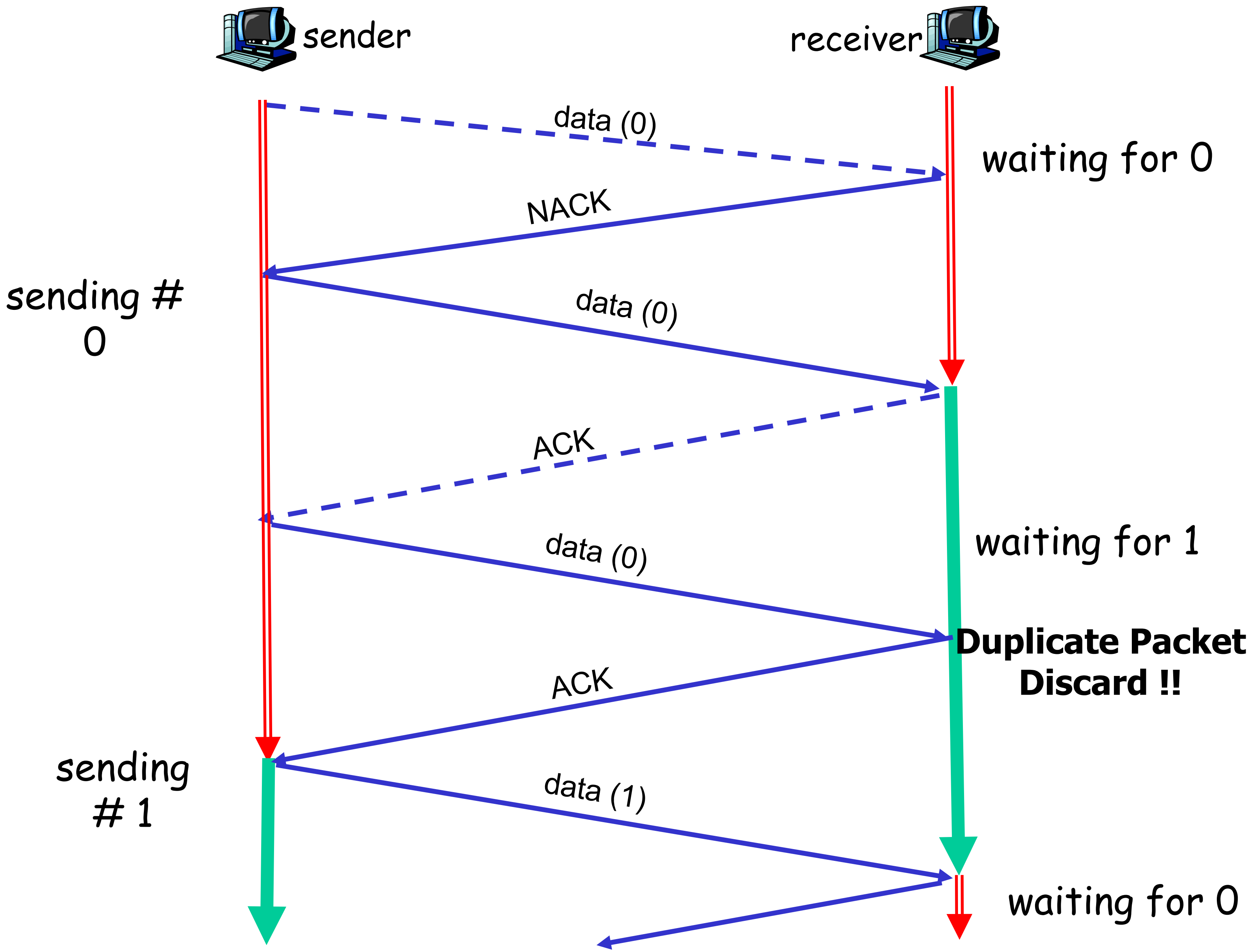
receiver:

- 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

- New Measures: Sequence Numbers, Checksum for ACK/NACK, Duplicate detection

Another Look at rdt2.1

Dotted line: erroneous transmission
Solid line: error-free transmission

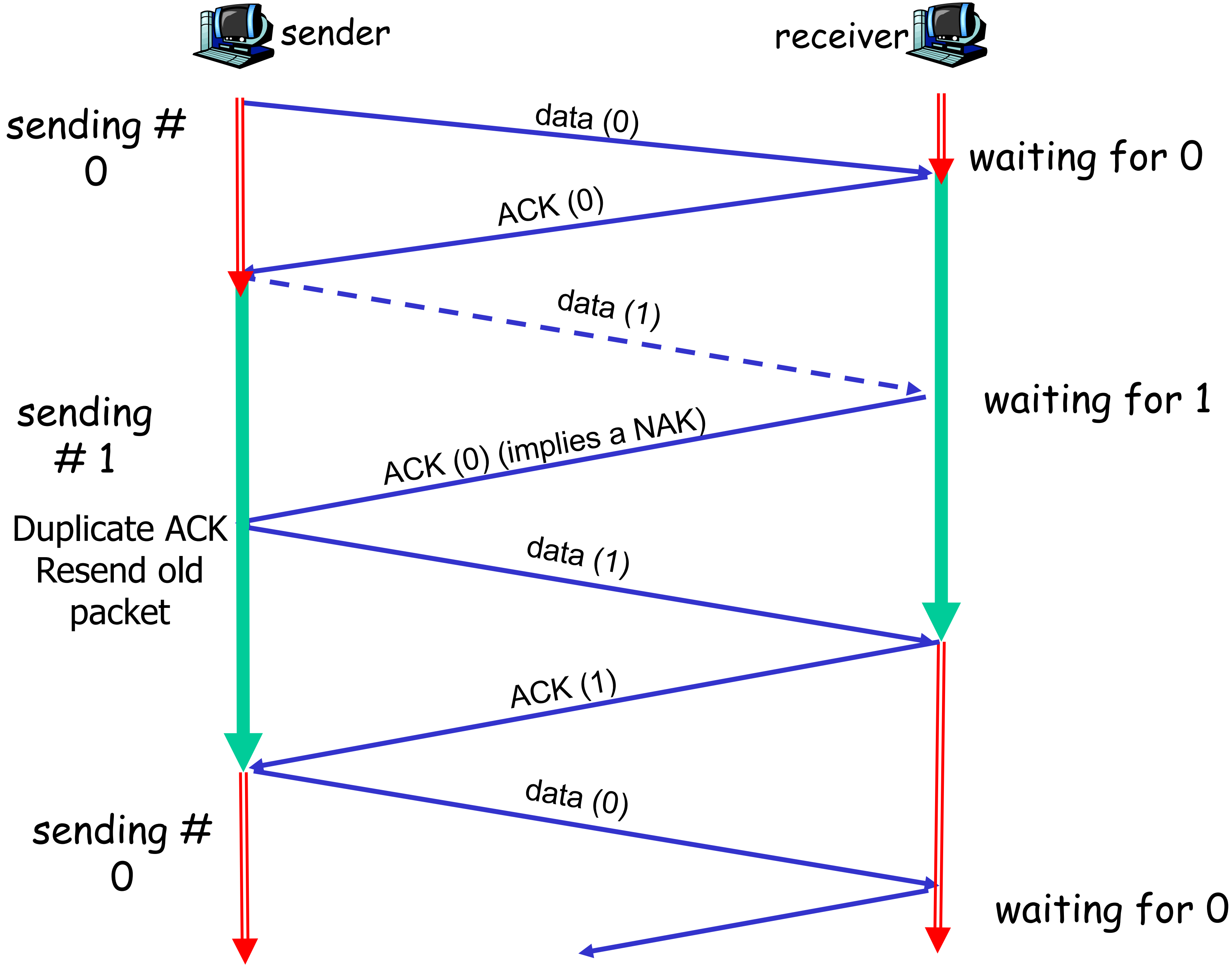


rdt2.2: a NAK-free protocol

- 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: Example

Dotted line: erroneous transmission
Solid line: error-free transmission



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

(continued in next lecture)

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