UNIVERSITY OF WATERLOO

CS 456/656 Computer Networks

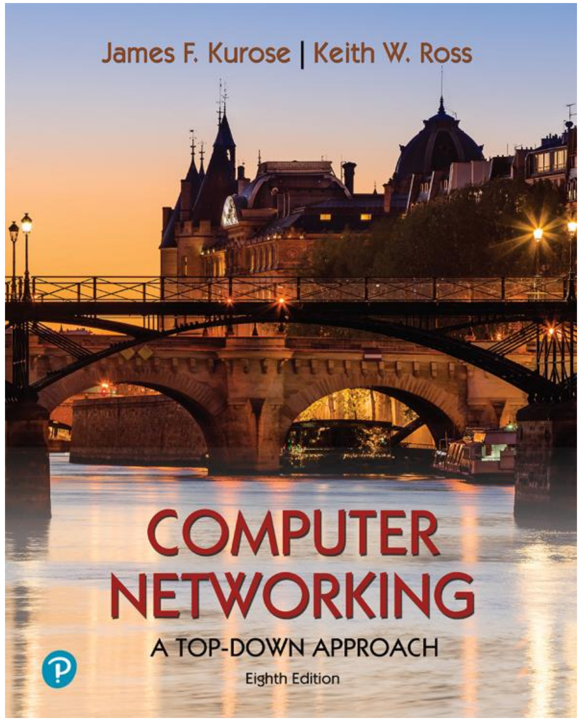
Lecture 4: Application Layer – Part 2

Mina Tahmasbi Arashloo and Bo Sun

Fall 2024

A note on slides

Adapted from the slides that accompany this book.



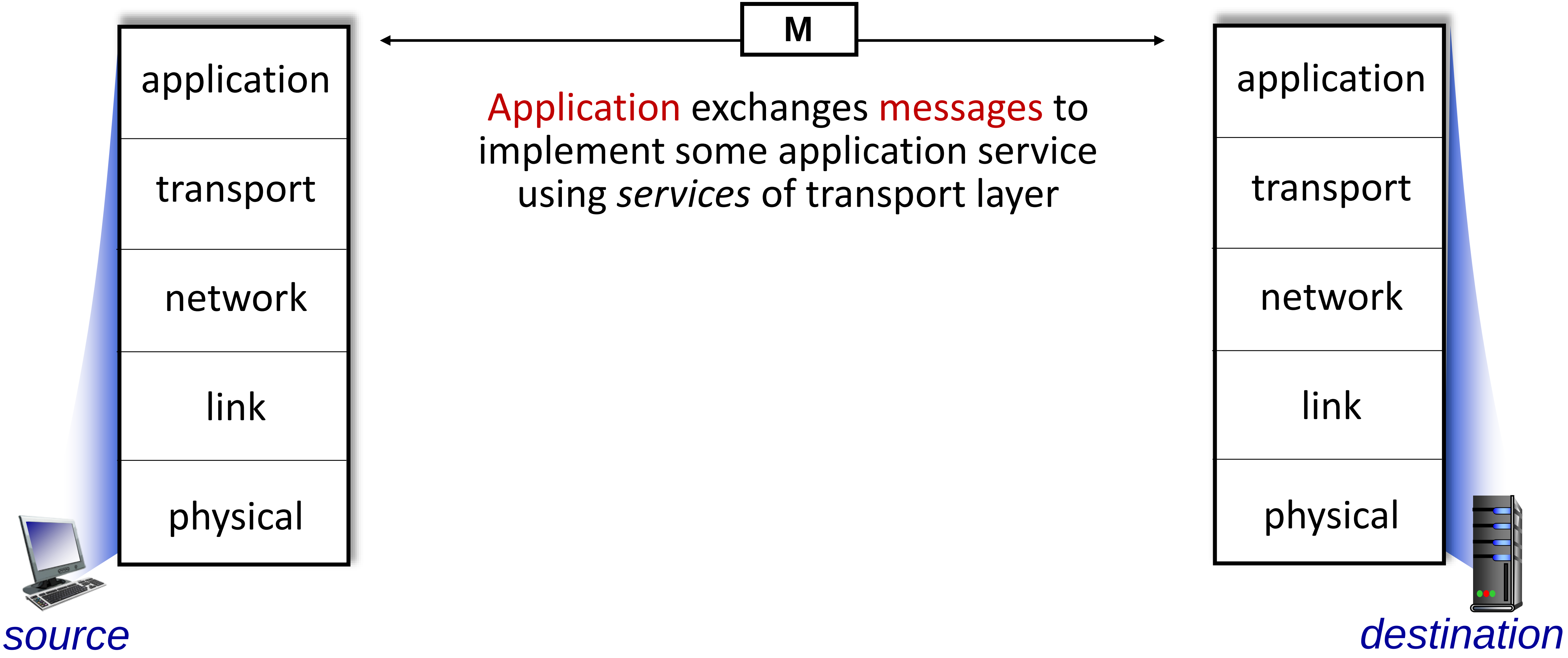
Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Examples applications we will discuss

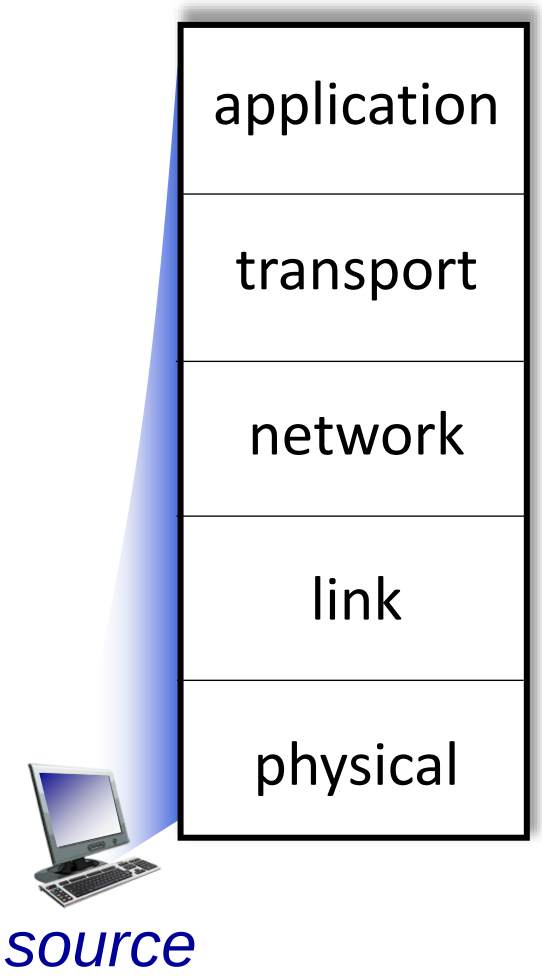
- Web applications: client-server
  - Fetching data for network applications from servers
  - Using a reliable connection-based **transport-layer service**
- Video streaming: client-server
- P2P file distribution: peer-to-peer
- E-Mail: client-server

How?

Communicating with the transport layer



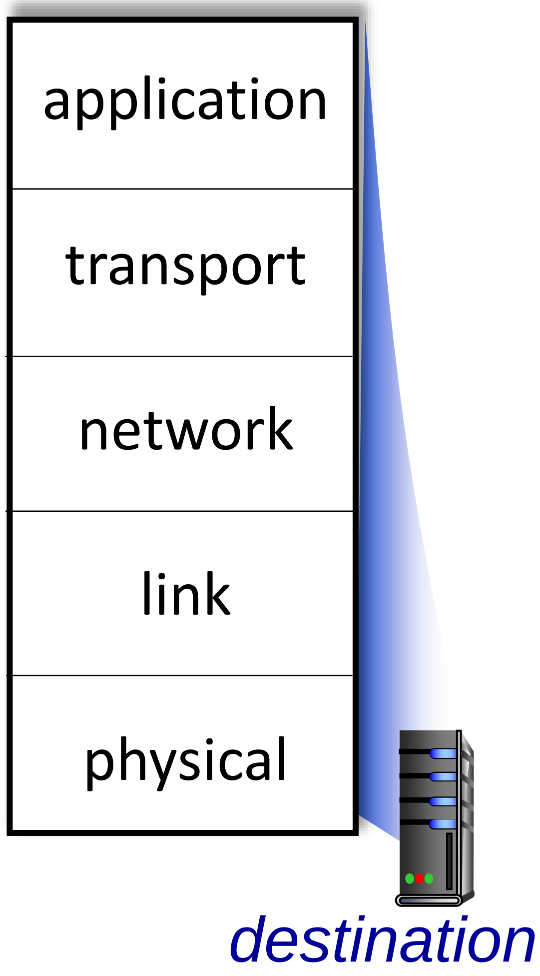
Communicating with the transport layer



M

Application exchanges messages to implement some application service <u>using *services* of transport layer</u>

How do applications communicate with the transport layer?



Communicating with the transport layer

The application should specify

- $\boxed{\text{The destination that will receive the data}}$ **??**
- What type of transport service it wants
  - Connection-based or connection-less?
  - Reliable or unreliable?
  - …
- The data that should be sent

Communication endpoints are processes

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by operating system)
- processes in different hosts communicate by exchanging messages over the network.

clients, servers

client process: process that initiates communication

server process: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* How do we find the IP address?

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

Communicating with the transport layer

The application should specify

- The destination that will receive the data
- What type of transport service it wants
  - Connection-based or connection-less?
  - Reliable or unreliable?
  - …
- The data that should be sent

Using Internet protocols

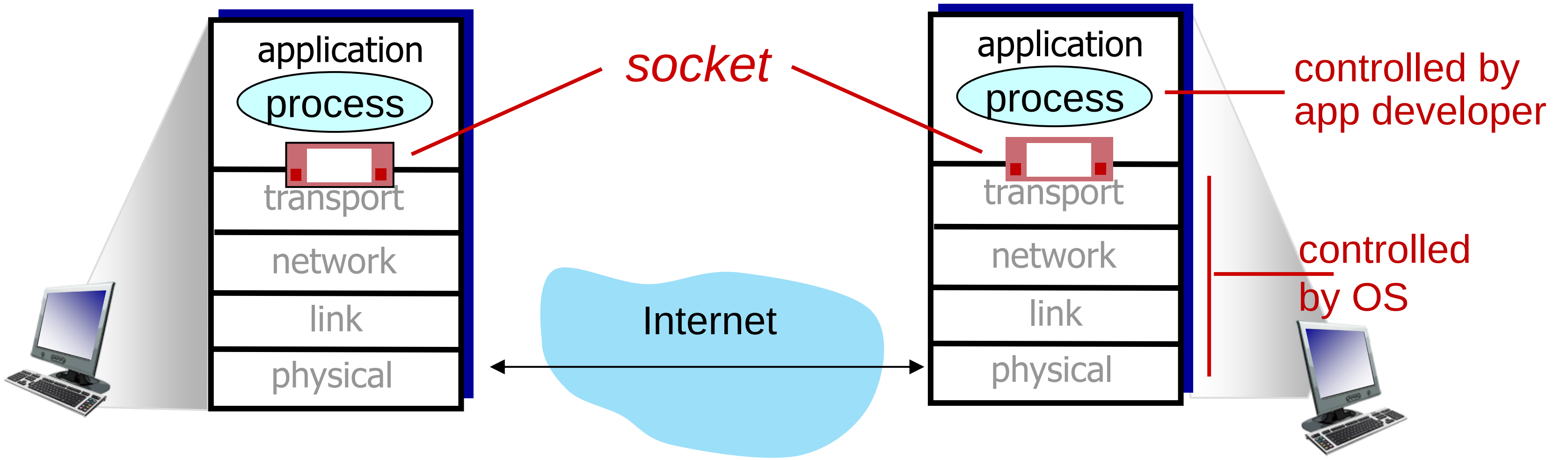
IP address and port

- TCP for reliable connection-based service
- UDP for unreliable connection-less service

- For applications using the Internet protocols, the common interface to the transport layer is the socket interface.

Sockets

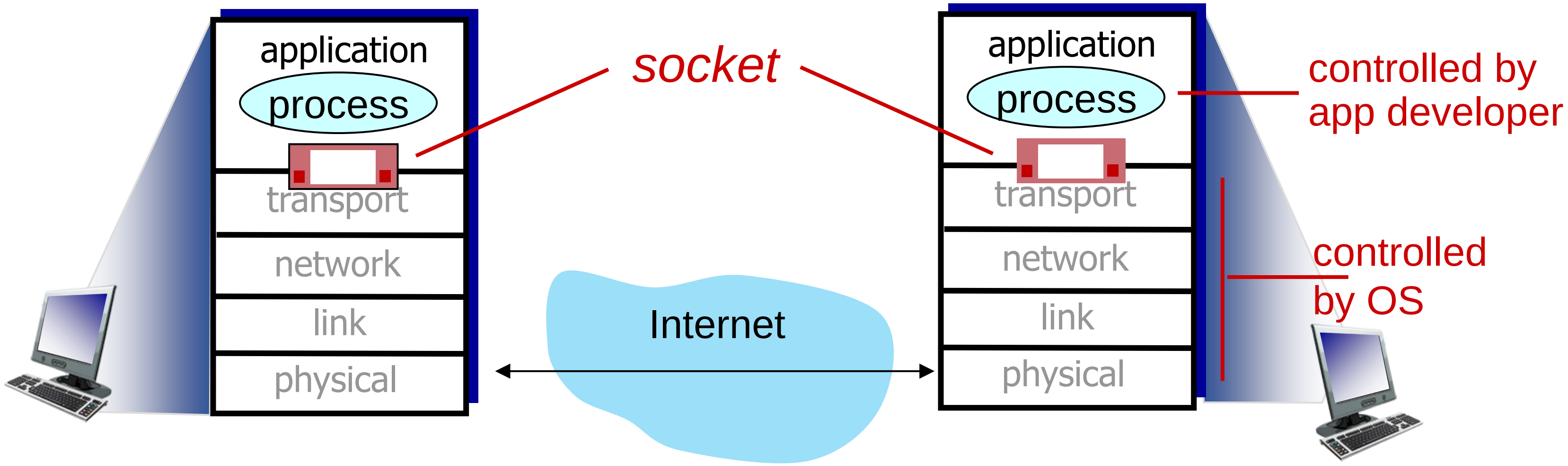
- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-to- end transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming with UDP

UDP: no "connection" between client and server:

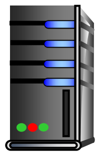
- no handshaking before sending data
- sender attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server processes

Client/server socket interaction: UDP



server (running on serverIP)

client

Server:

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

↓

read datagram from
serverSocket

↓

write reply to
serverSocket
specifying
client address,
port number

Client:

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

↓

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

↓

read datagram from
clientSocket

↓

close
clientSocket

Example app: UDP client

Python UDPClient

include Python's socket library → `from socket import *`

```
serverName = 'hostname'
serverPort = 12000
```

create UDP socket → `clientSocket = socket(AF_INET, SOCK_DGRAM)`

get user keyboard input → `message = input('Input lowercase sentence:')`

attach server name, port to message; send into socket → `clientSocket.sendto(message.encode(), (serverName, serverPort))`

read reply data (bytes) from socket → `modifiedMessage, serverAddress = clientSocket.recvfrom(2048)`

print out received string and close socket → `print(modifiedMessage.decode())`

```
clientSocket.close()
```

Note: this code update (2023) to Python 3

Example app: UDP server

Python UDPServer

create UDP socket →
bind socket to local port number 12000 →
loop forever →
Read from UDP socket into message, getting client's address (client IP and port) →
send upper case string back to this client →

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print('The server is ready to receive')
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

Note: this code update (2023) to Python 3

Socket programming with TCP

Client must contact server

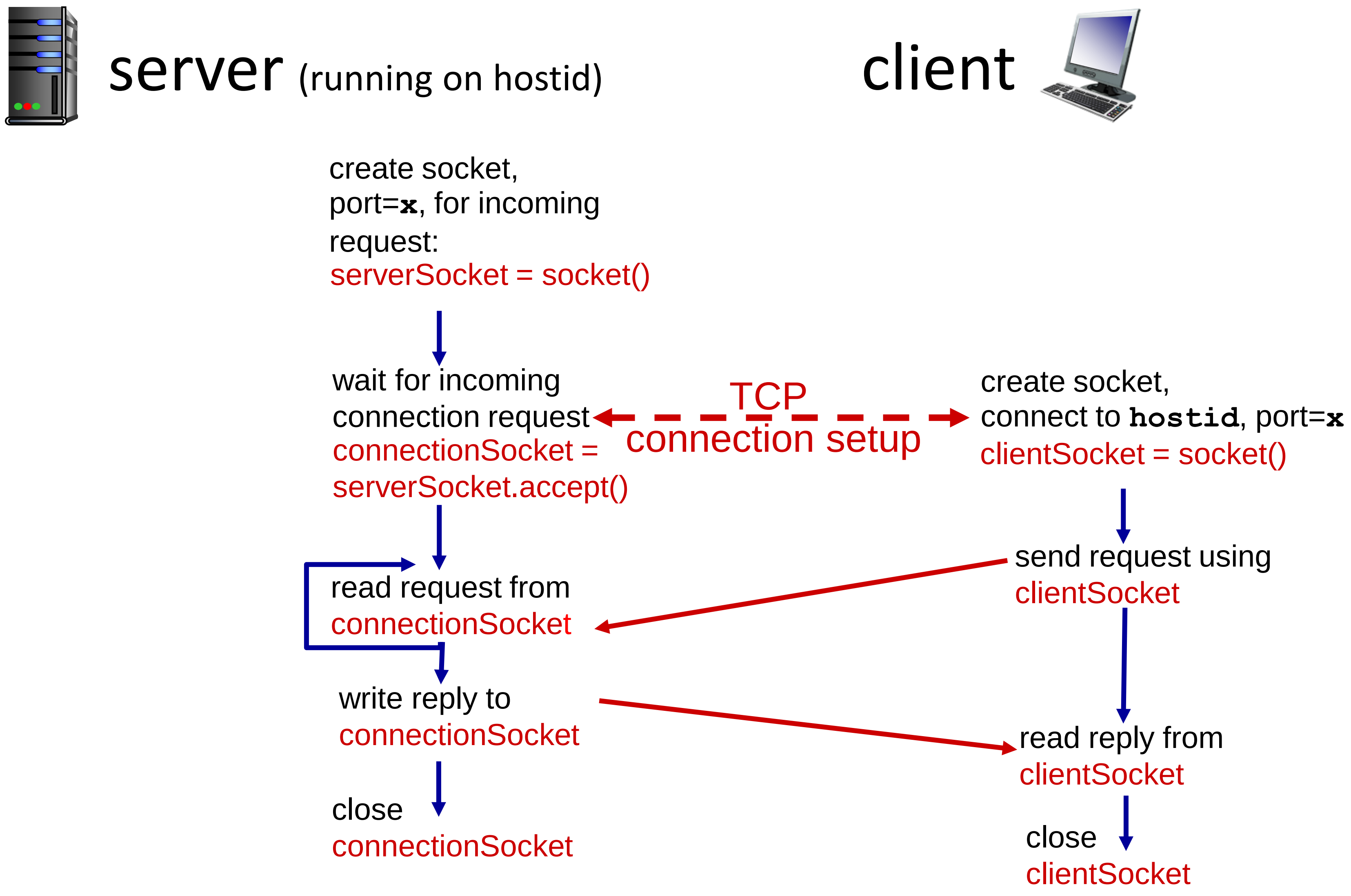
- To establish a connection
- server must have created a socket (door) that welcomes client's contact
- Client creates TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to establish connections with multiple clients
  - client source port # and IP address used to distinguish clients (more in Chap 3)

Application viewpoint

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

Note: this code update (2023) to Python 3

Example app: TCP server

Python TCPServer

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                          encode())
    connectionSocket.close()
```

Note: this code update (2023) to Python 3

Communicating with the transport layer

- Is the socket interface the only interface?
  - It is the most common, but there are others
  - Applications have evolved quite a lot since the Socket API was created
  - They want options more than just reliable vs unreliable service
    - E.g., performance, security, semi-reliability, etc.
  - Research question: What is a good interface for the application to tell the transport layer about their needs?
  - We’ll talk more about this when we discuss the transport layer

Examples applications we will discuss

- Web applications: client-server
- Video streaming: client-server
- P2P file distribution: peer-to-peer
- E-Mail: client-server

Example: Video Streaming

- 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*

Example: Video Streaming

- 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, 64Kbps – 12 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)

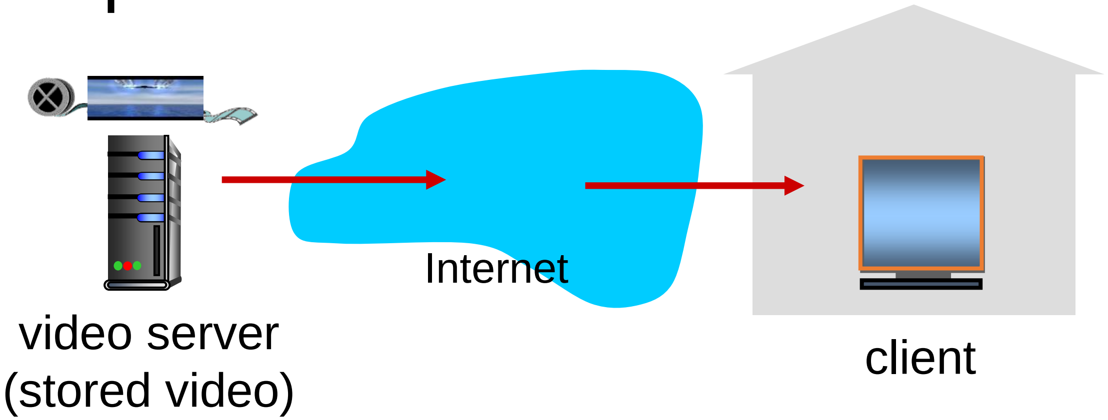
frame *i*

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

frame *i+1*

Streaming stored video

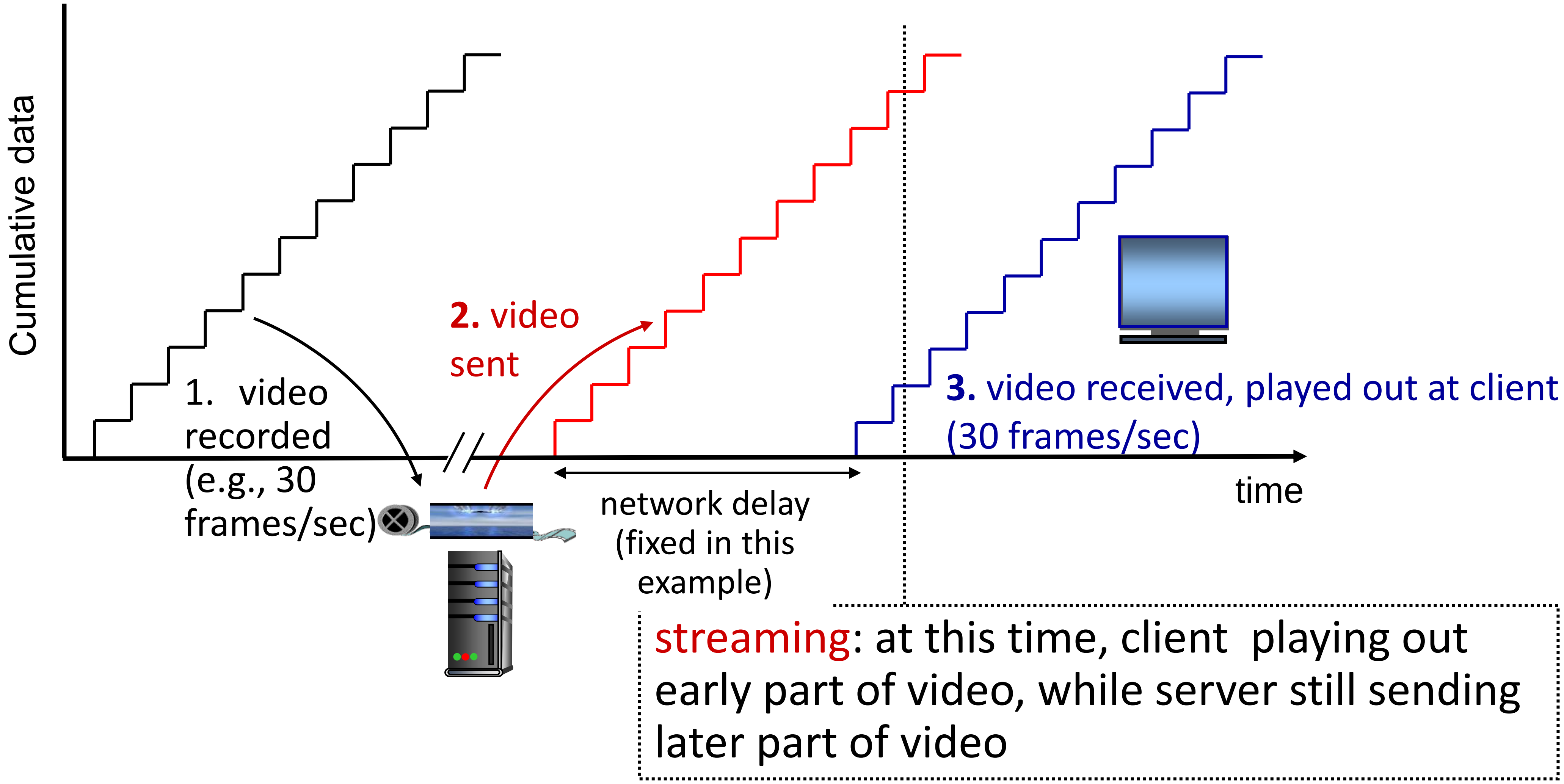
simple scenario:



Main challenges:

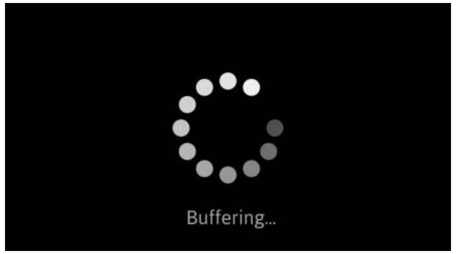
- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, access network, network core, video server)
- packet loss, delay due to congestion will delay playout, or result in poor video quality

Streaming stored video



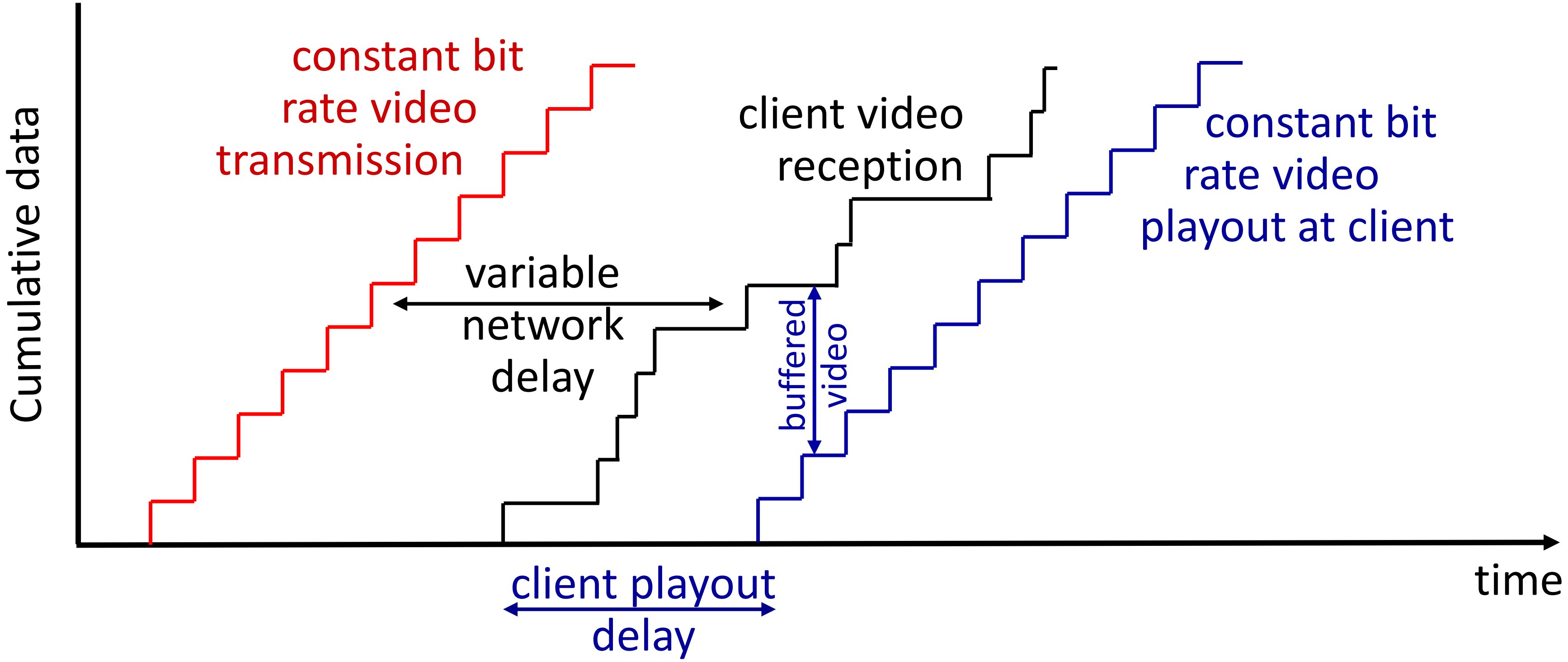
Streaming stored video: challenges

- continuous playout constraint: during client video playout, playout timing must match original timing
  - … but network delays are variable (jitter), so will need client-side buffer to match continuous playout constraint



- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted

Streaming stored video: playout buffering



- *client-side buffering and playout delay:* compensate for network-added delay, delay jitter

Video streaming in practice

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- *challenge:* scale - how to efficiently get content to millions of users?
- *challenge:* heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; high vs low bandwidth)

Idea 1: Content distribution networks (CDNs)

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 (and possibly congested) path to distant clients

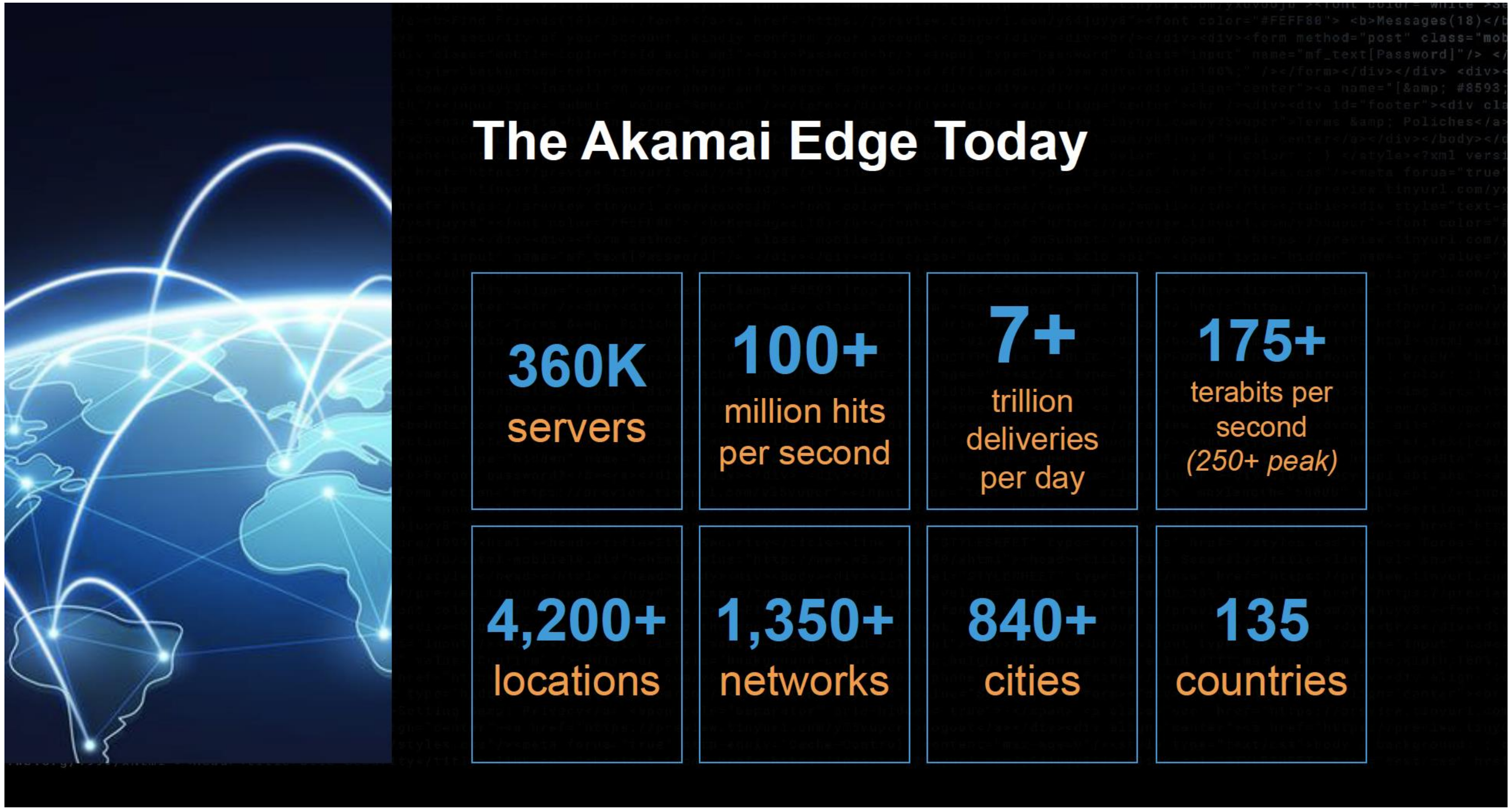
....quite simply: this solution *doesn't scale*

Idea 1: Content distribution networks (CDNs)

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
    - Akamai: 240,000 servers deployed in > 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in Internet exchange points (IXPs)
    - used by Limelight

Example CDN: Akamai

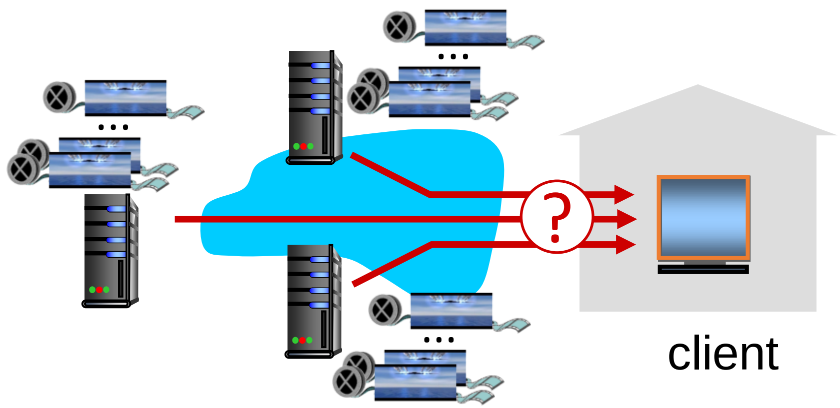


Source: https://networkingchannel.eu/living-on-the-edge-for-a-quarter-century-an-akamai-retrospective-downloads/

Idea 2: DASH (Dynamic Adaptive Streaming over HTTP)

server:

- divides video file into multiple chunks
- each chunk encoded at multiple different rates
- different rate encodings stored in different files
- files replicated in various CDN nodes
- *manifest file:* provides URLs for different chunks

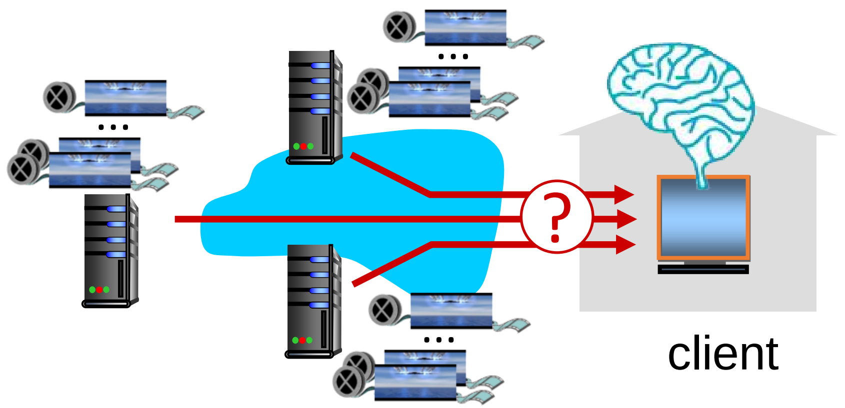


client:

- periodically estimates 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), and from different servers

Idea 2: DASH (Dynamic Adaptive Streaming over HTTP)

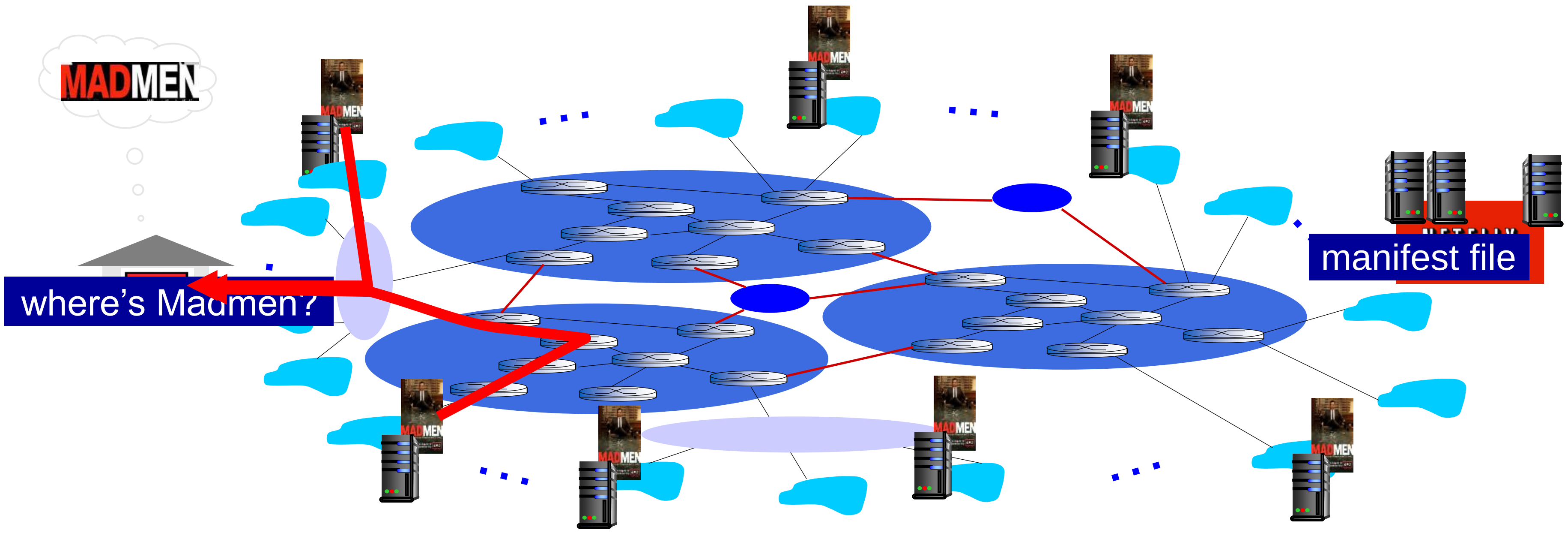
- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or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



Streaming video = encoding + DASH + playout buffering

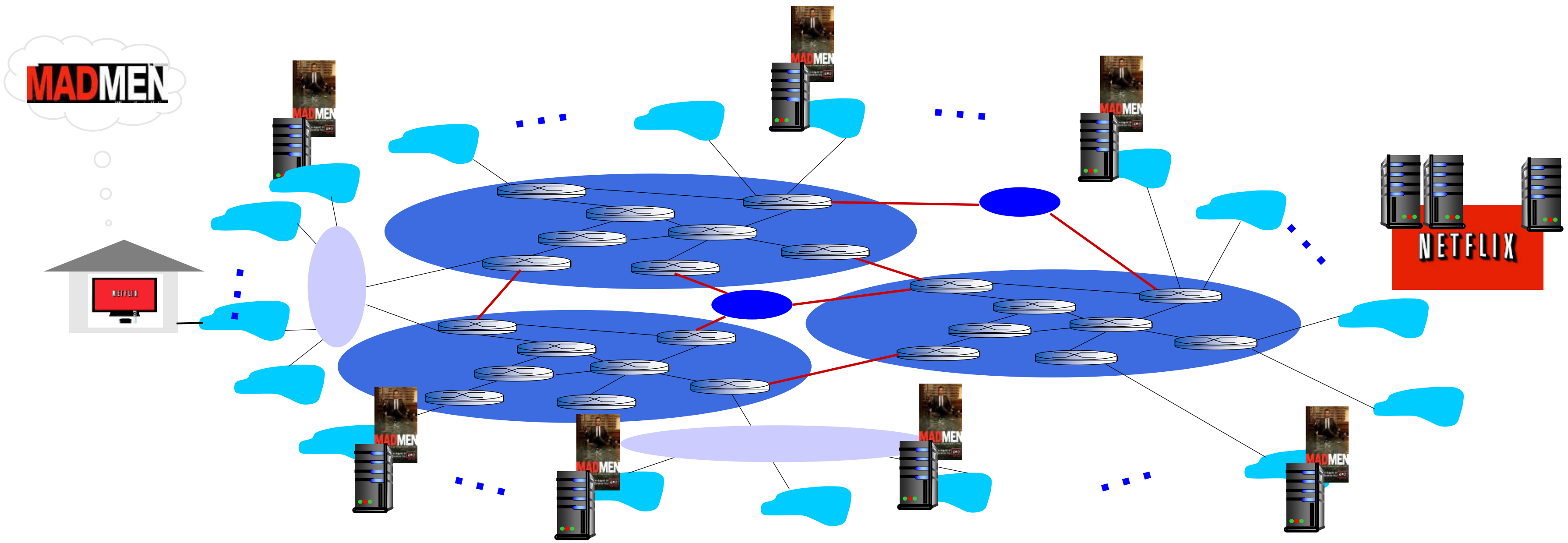
Video streaming example: Netflix

- Netflix: stores copies of content (e.g., MADMEN) at its (worldwide) OpenConnect CDN nodes
- subscriber requests content, service provider returns manifest
  - using manifest, client retrieves content at highest supportable rate
  - may choose different rate or copy if network path congested

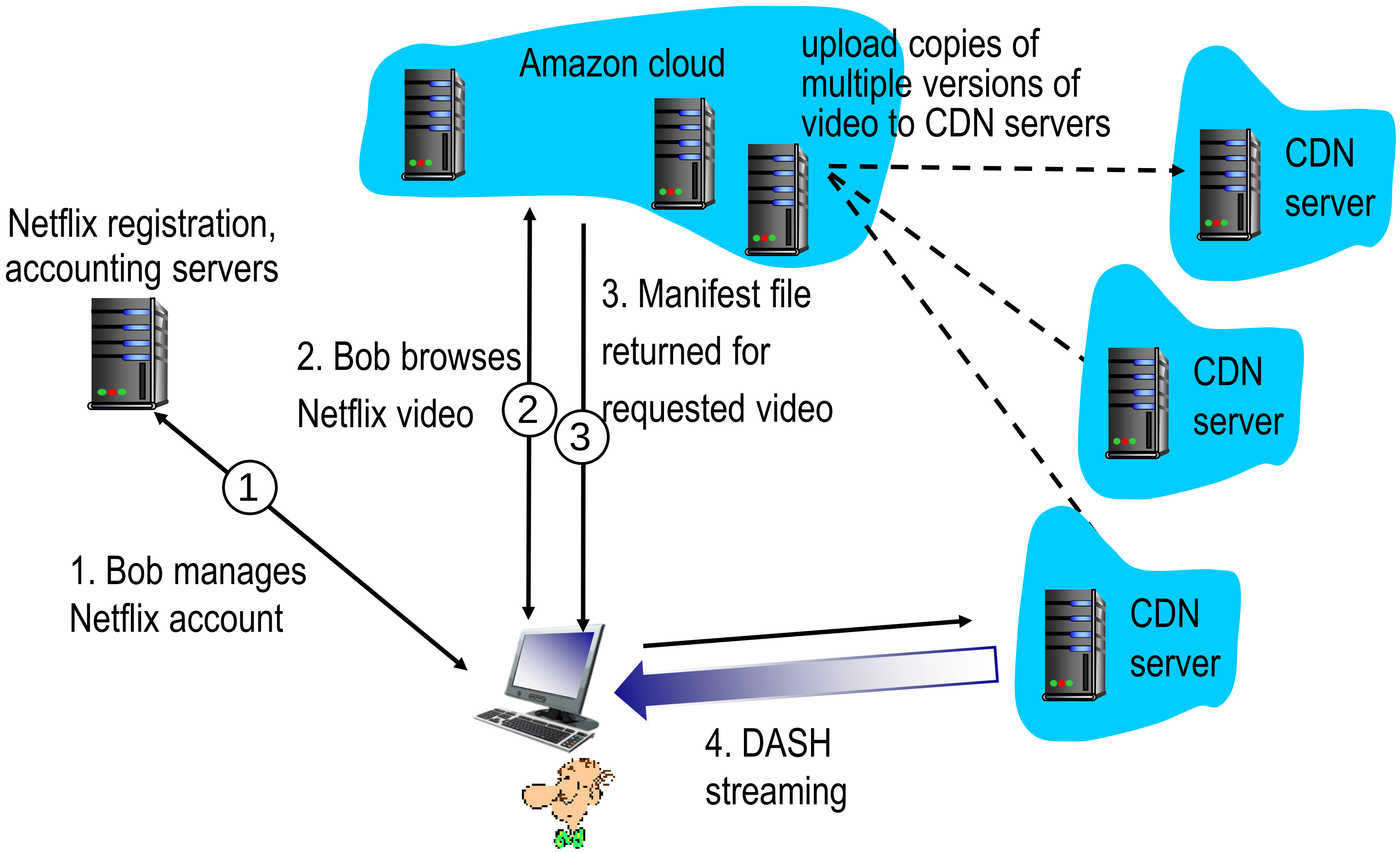


Video streaming example: Netflix

- Some interesting design decisions services like Netflix need to make:
  - What content to place in which CDN nodes?
  - From which CDN node to retrieve content? At which rate?



Video streaming example: Netflix

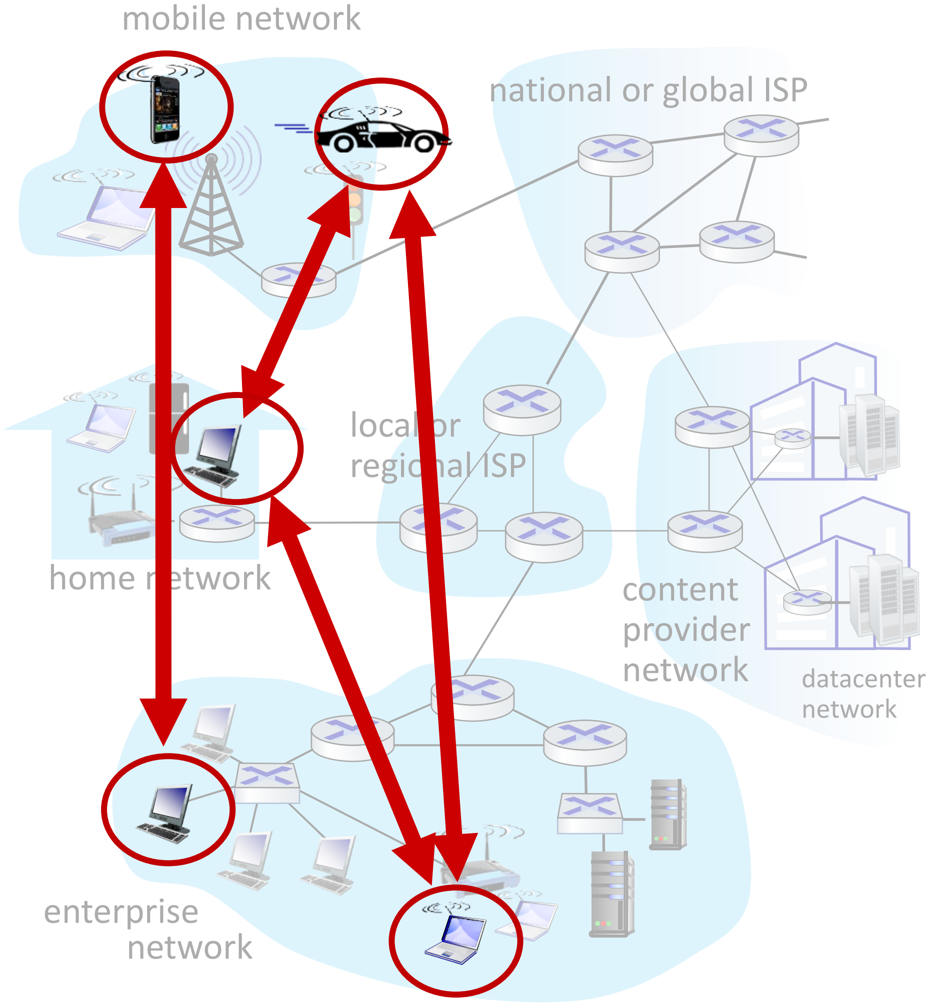


Examples applications we will discuss

- Web applications: client-server
- Video streaming: client-server
- P2P file distribution: peer-to-peer
- E-Mail: client-server

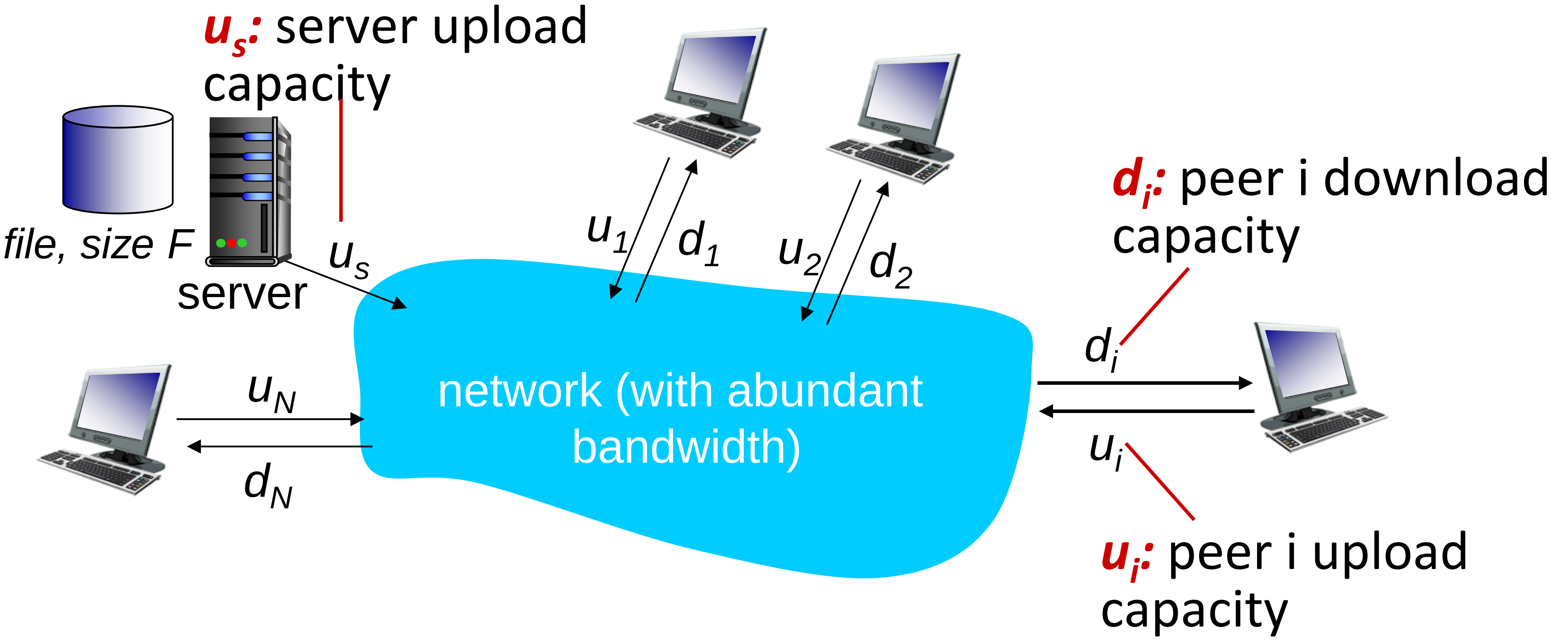
Reminder: Peer-to-peer (P2P) architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change network addresses
  - complex management
- examples: P2P file sharing (BitTorrent)



File distribution

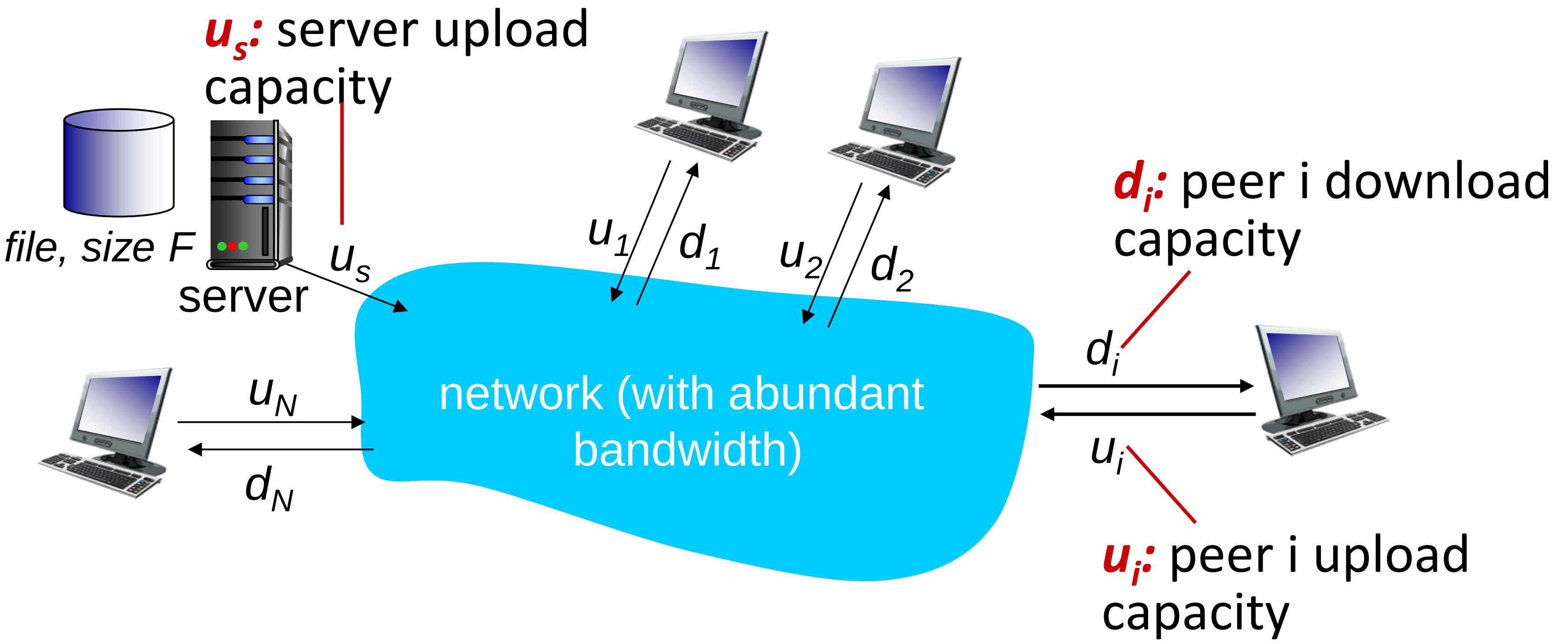
- A server distributes one copy of a large file to each of the N hosts (peers)
  - u_s: upload rate of the server's access link
  - u_i: upload rate of the i-th peer's access link
  - d_i: download rate of the i-th peer's access link



File distribution: client-server vs P2P

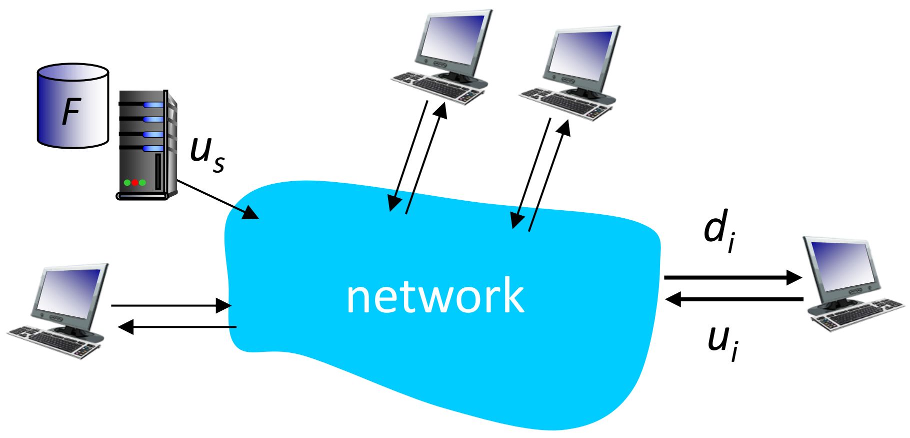
Q: how much time to distribute file (size *F*) from one server to *N peers*?

- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must sequentially send (upload) *N* file copies:
  - No one else “helps” in uploading
  - time to send one copy: F/u_s
  - time to send *N* copies: NF/u_s
- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}



Lower bound, but can be achieved in certain scenarios.

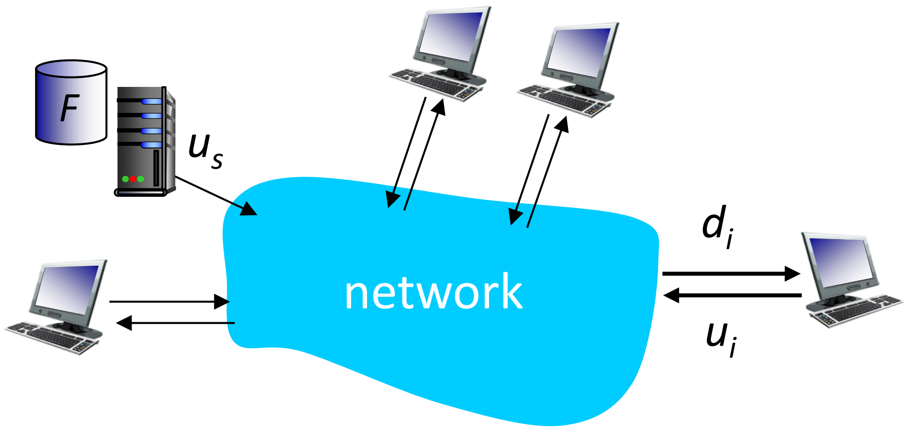
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} \geq max\{NF/u_s, F/d_{min}\}$$

increases linearly in N

File distribution time: P2P

- *server transmission:* must upload at least one copy:
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *Server and clients:* as a whole, the system must deliver (upload) a total of NF bits (F bits to each of the N peers)
  - max upload rate is $u_s + \Sigma u_i$



Lower bound, but can be achieved in certain scenarios.

time to distribute F to N clients using P2P approach

$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

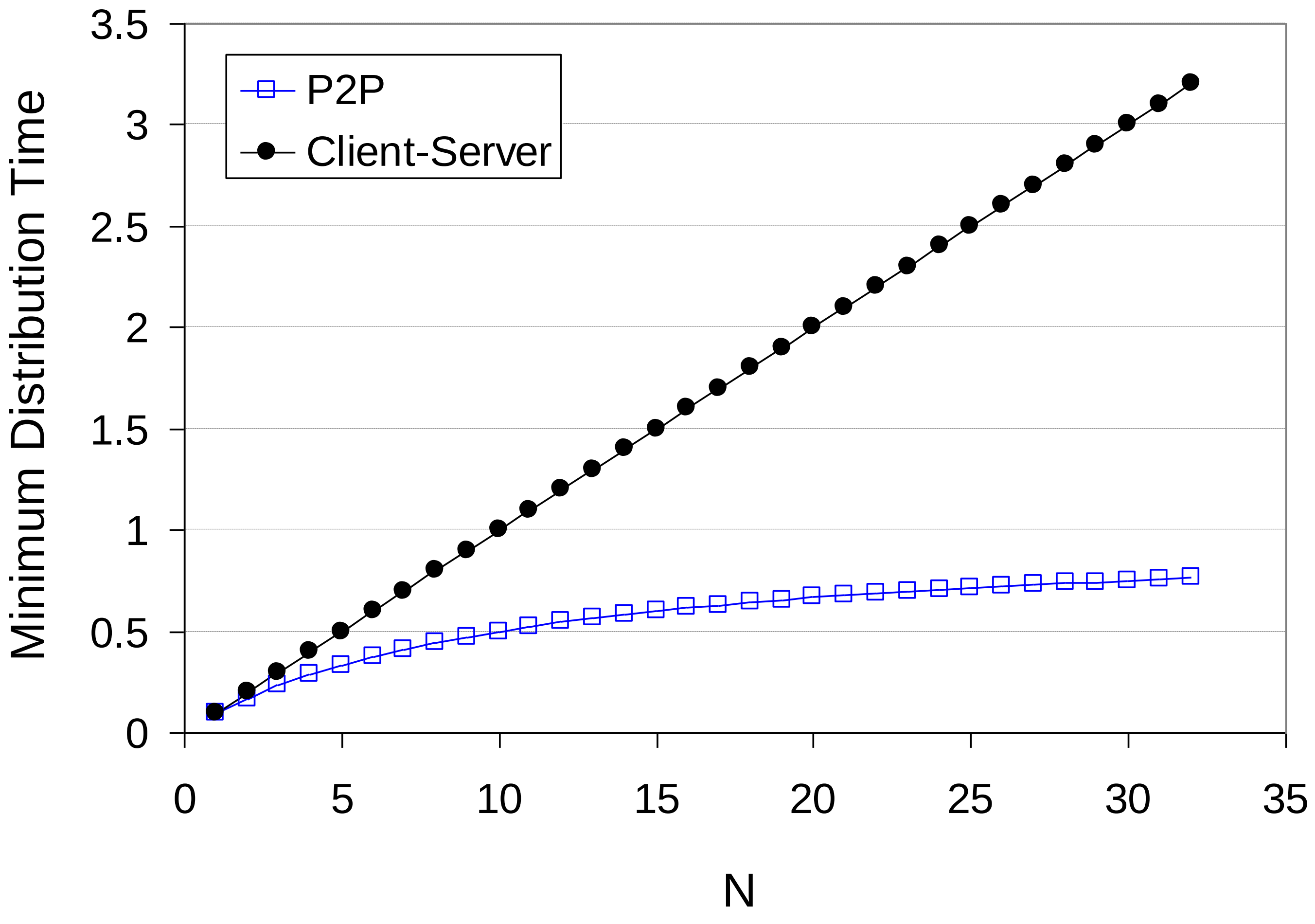
… but so does this, as each peer brings service capacity

In-class exercise: file distribution time

- Consider distributing a file of $F = 360$ Mbits to 20 peers. The server has an upload rate of 1 Mbps, and each peer has upload rate of 100kbps and download rate of 1 Mbps. What is the minimum file distribution time for client-server and P2P distributions respectively?

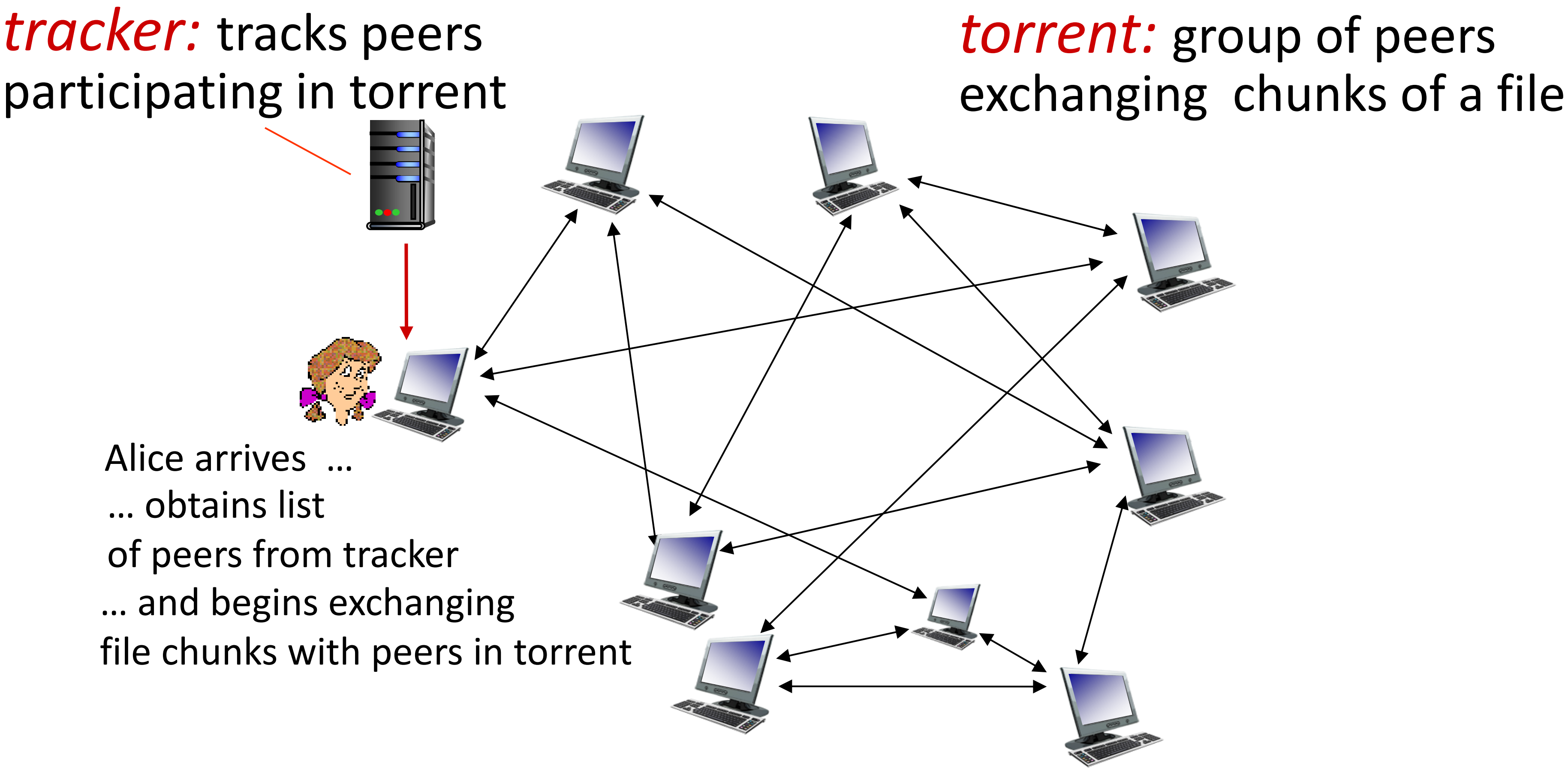
Client-server vs. P2P: example

client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$

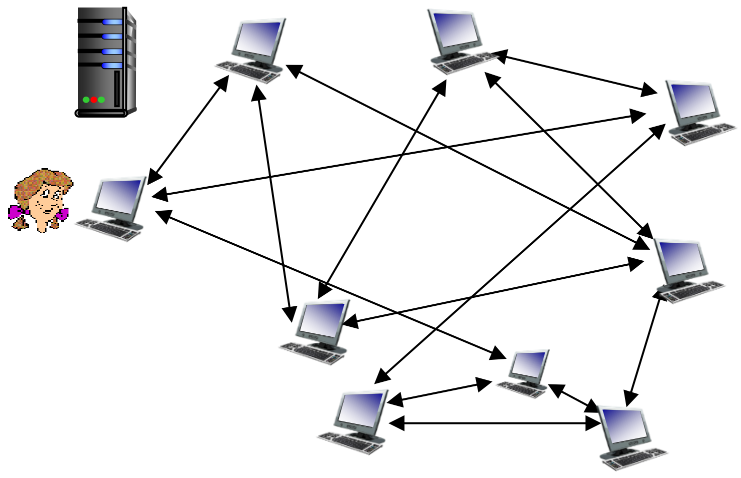


P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks



P2P file distribution: BitTorrent



- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

Requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

Sending chunks: tit-for-tat

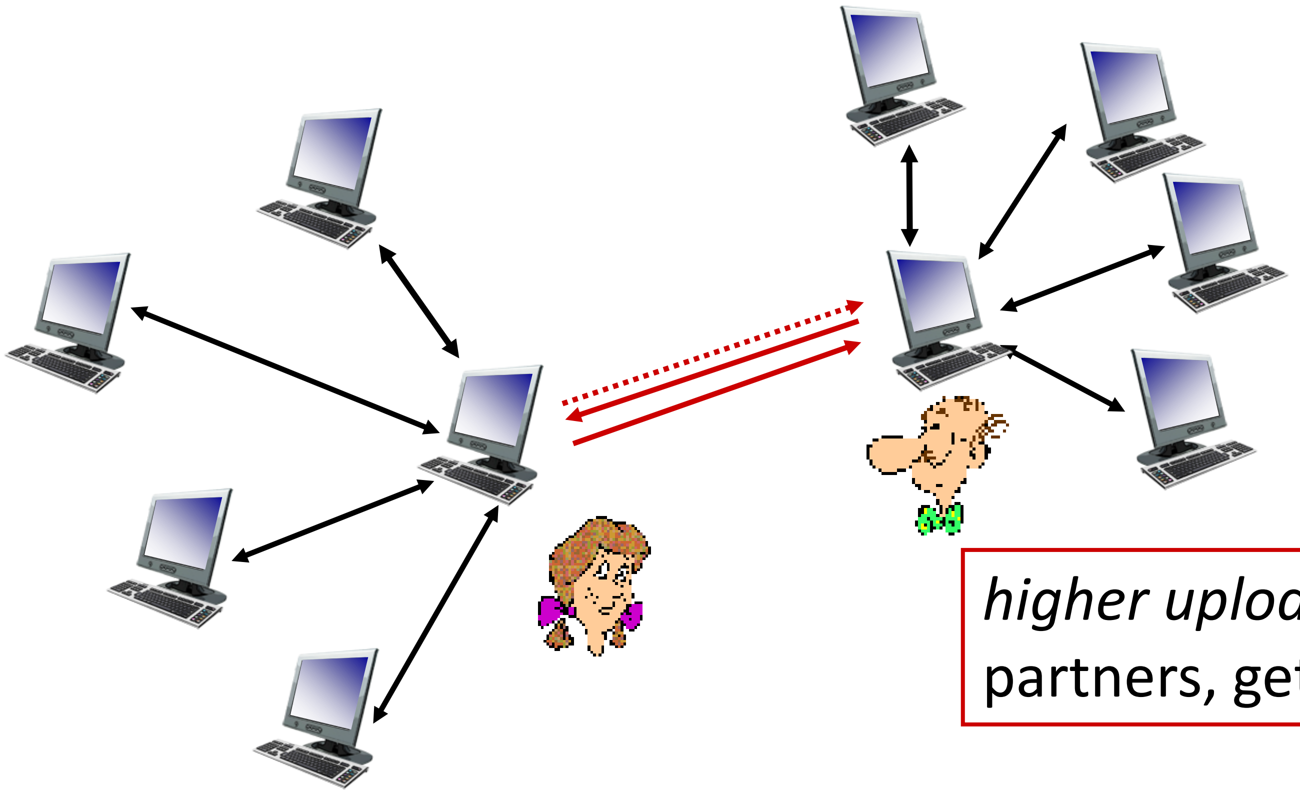
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers



higher upload rate: find better trading partners, get file faster !

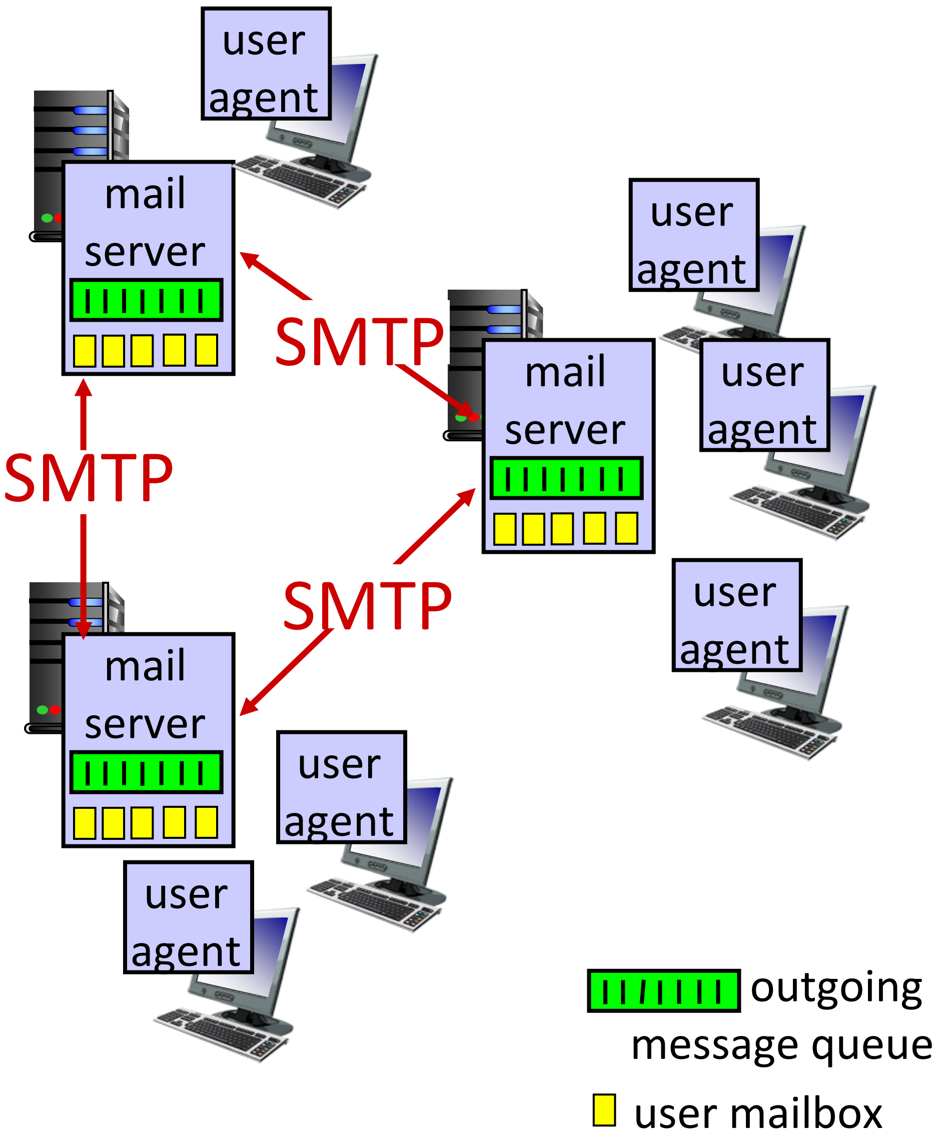
Examples applications we will discuss

- Web applications: client-server
- Video streaming: client-server
- P2P file distribution: peer-to-peer
- E-Mail: client-server

Example: E-mail

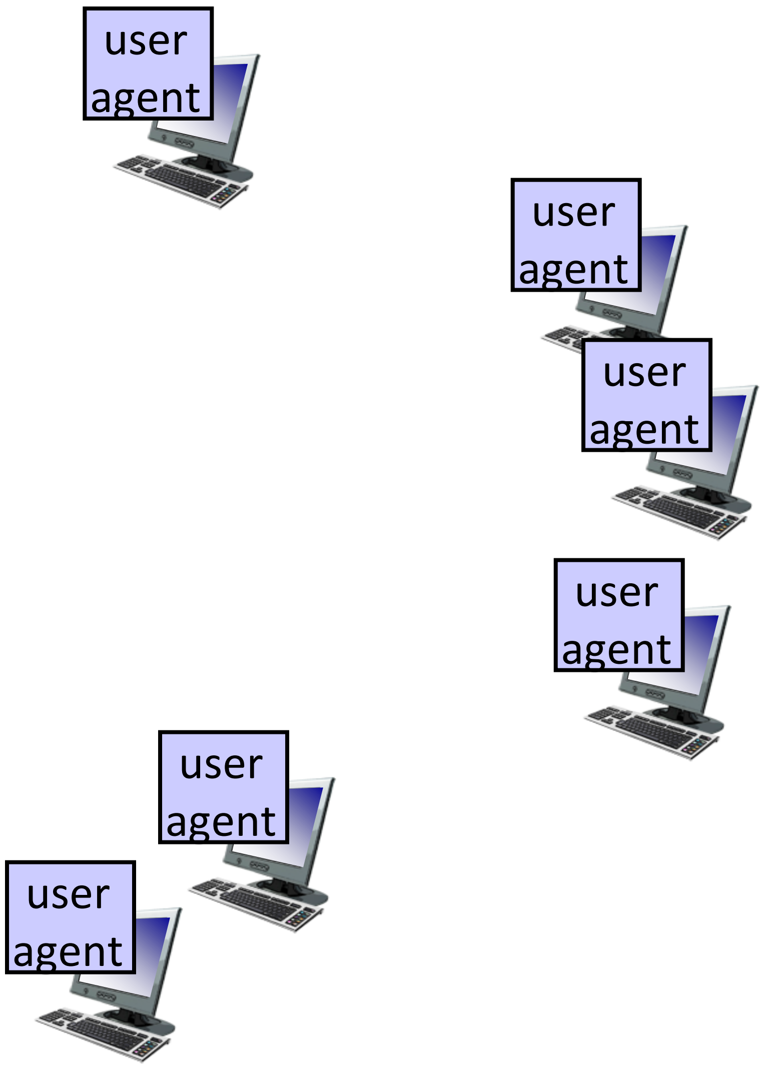
Three major components:

- user agents
- mail servers
- simple mail transfer protocol: SMTP



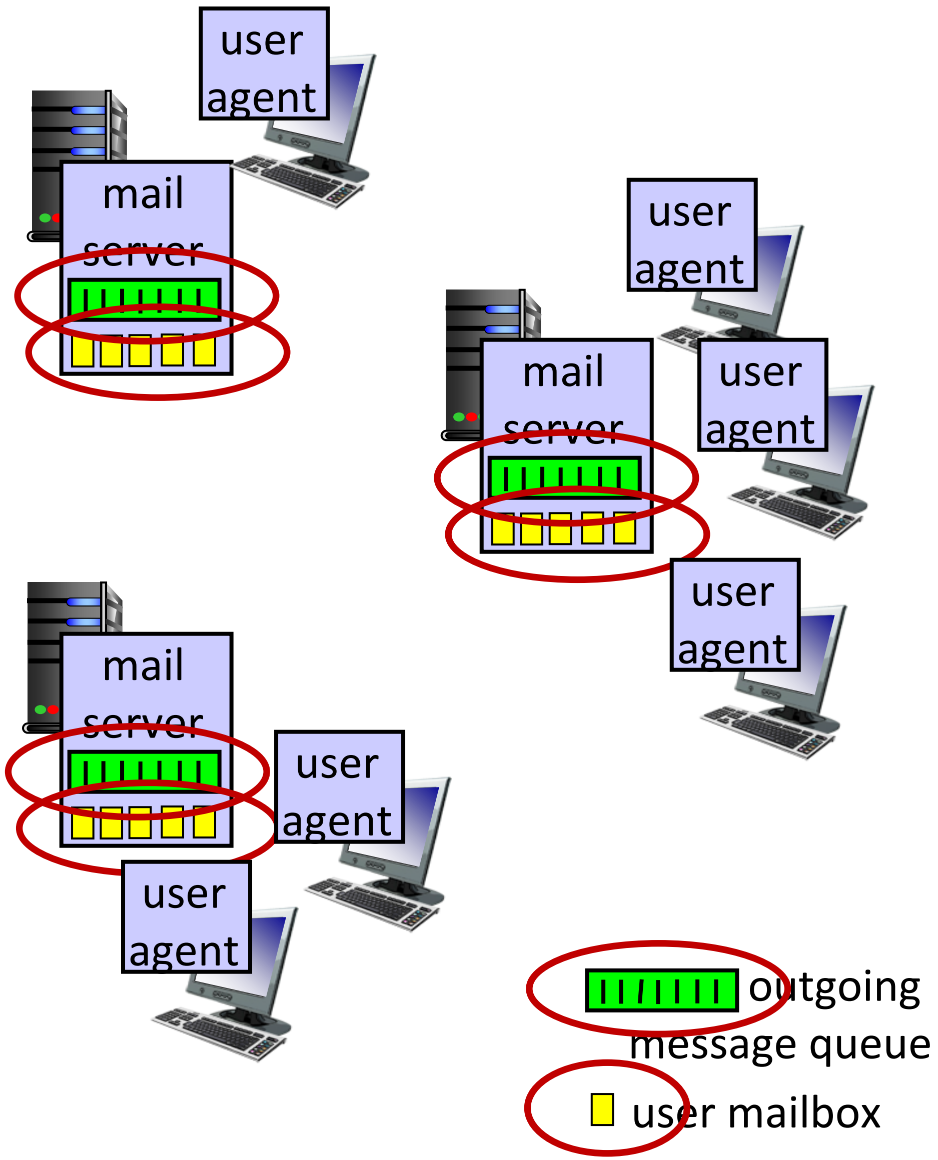
E-mail: user agents

- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- Outgoing and incoming messages stored on server
- Messages can be read/copied on local devices through the user agents.



E-mail: mail servers

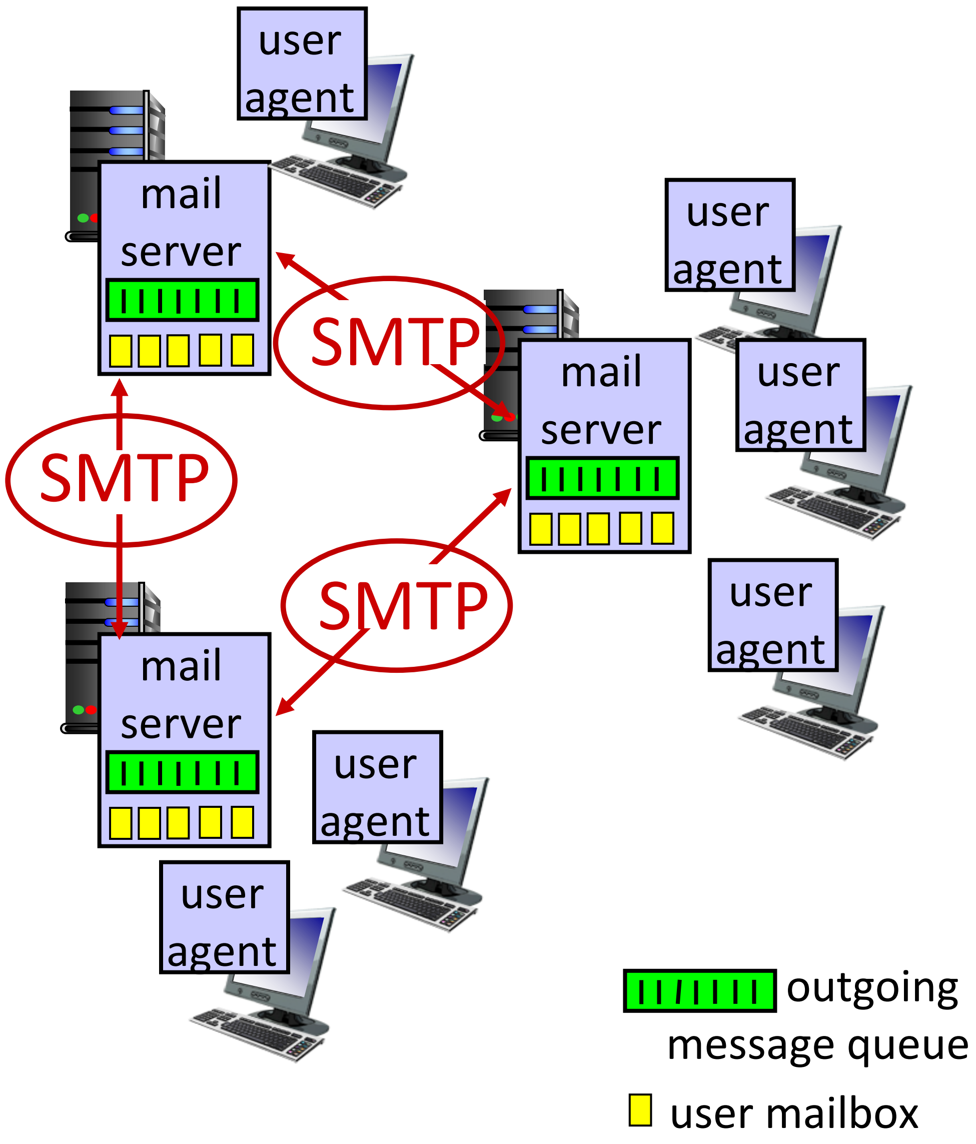
- Store outgoing and incoming messages
- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages



E-mail: SMTP protocol

SMTP protocol between mail servers to send email messages

- client: sending mail server
- "server": receiving mail server



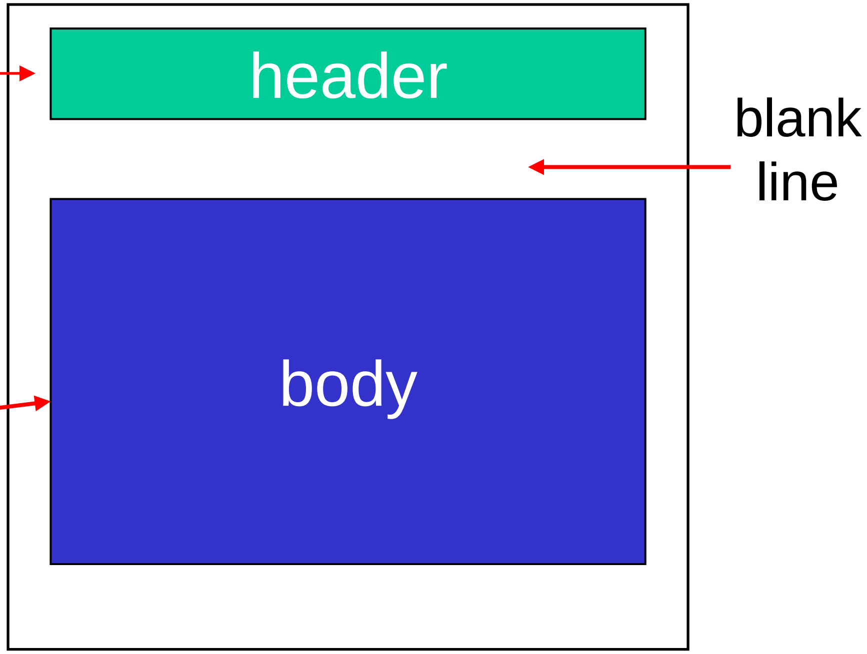
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 5321 (like RFC 7231 defines HTTP)

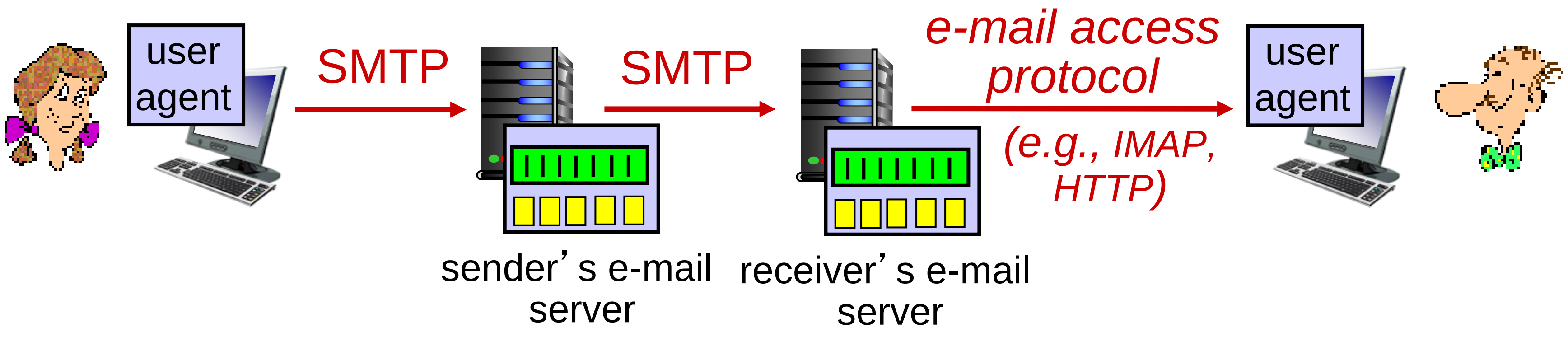
RFC 2822 defines *syntax* for e-mail message itself (like HTML defines syntax for web documents)

- header lines, e.g.,
  - To:
  - From:
  - Subject:

  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the "message" , ASCII characters only



Retrieving email: mail access protocols



- SMTP: delivery/storage of e-mail messages to receiver’s server
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides functions like retrieval and deletion of folders of stored messages on server
- HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

Examples applications we have discussed!

- Web applications: client-server
- Video streaming: client-server
- P2P file distribution: peer-to-peer
- E-Mail: client-server

Application Layer: Summary

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - P2P: BitTorrent
  - SMTP, IMAP
- video streaming, CDNs
- socket programming: TCP, UDP sockets

Application Layer: Summary

Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data:* info(payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- “complexity at network edge”