Τεχνολογίες Διαδικτύου 2025-26 (DIT 315)

Δρ. Ειρήνη Λιώτου

eliotou@hua.gr

18/11/2025

Chapter 2: 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

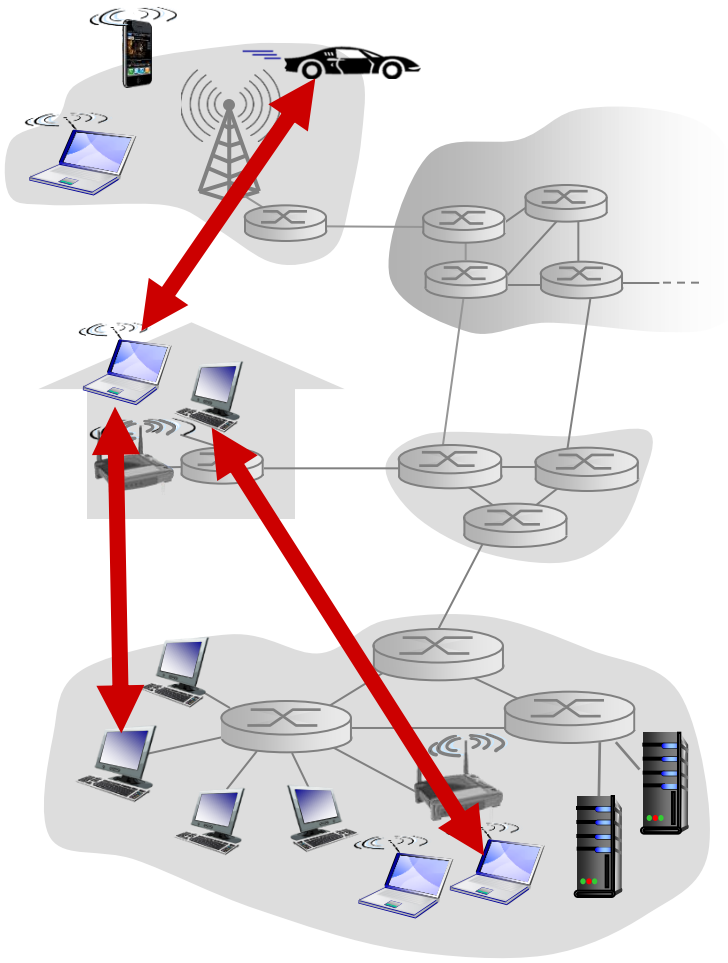
2.7 socket programming with UDP and TCP

Pure P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

examples:

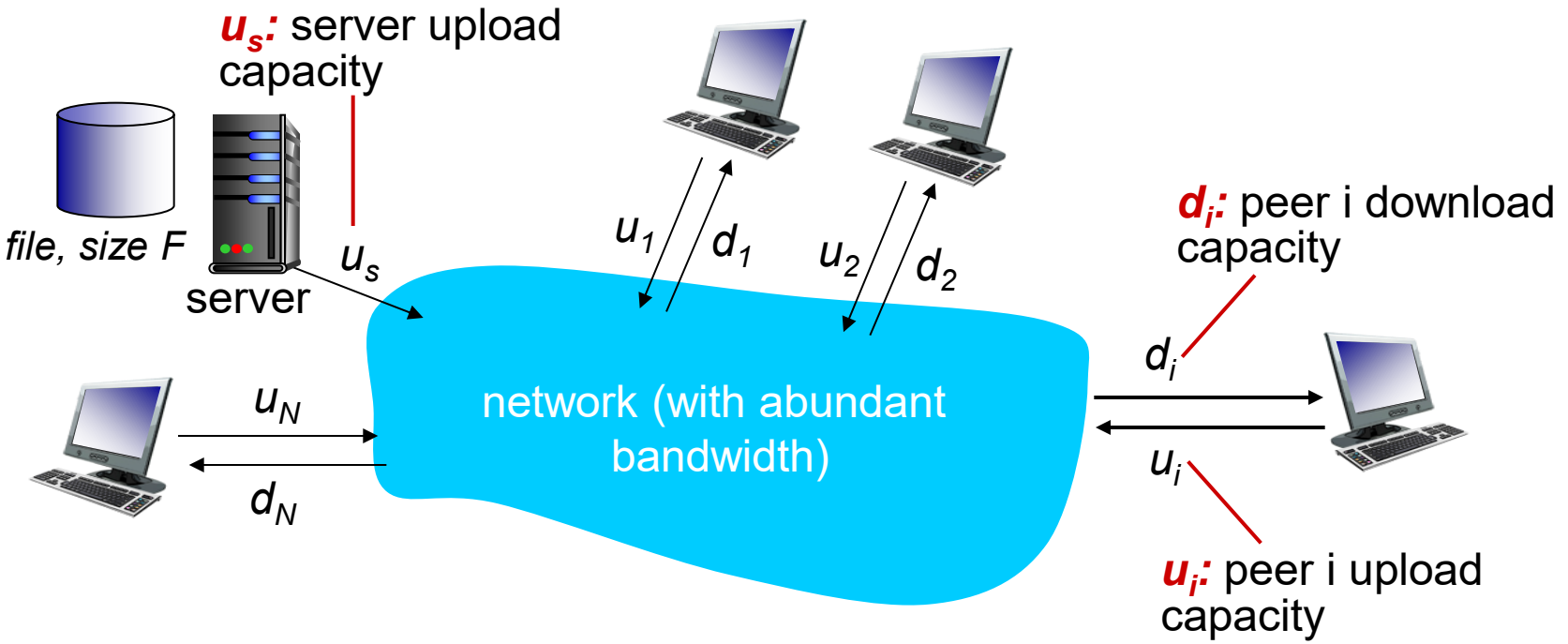
- file distribution (BitTorrent)
- Streaming (KanKan)



File distribution: client-server vs P2P

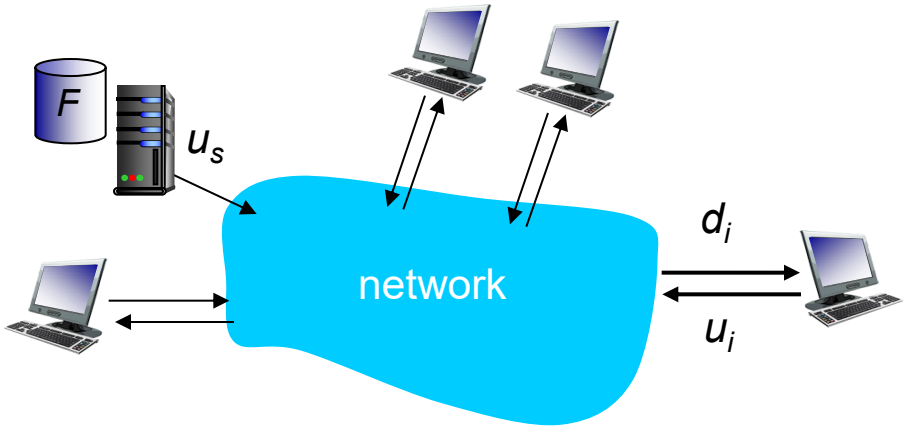
Question: how much time to distribute file (size *F*) from one server to *N* peers?

- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must sequentially send (upload) *N* file copies:
  - 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
  - client download time: F/d_{min}



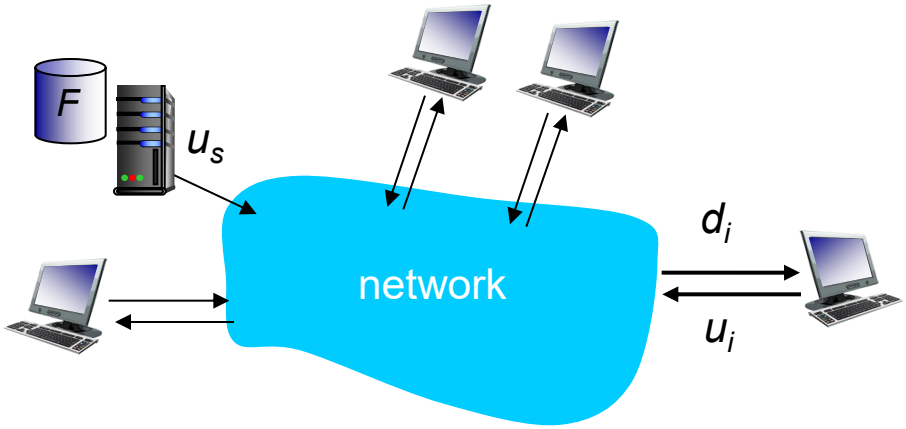
time to distribute F to N clients using client-server approach

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

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
  - client download time: F/d_{min}



- *clients:* as aggregate must download *NF* bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

time to distribute F to N clients using P2P approach

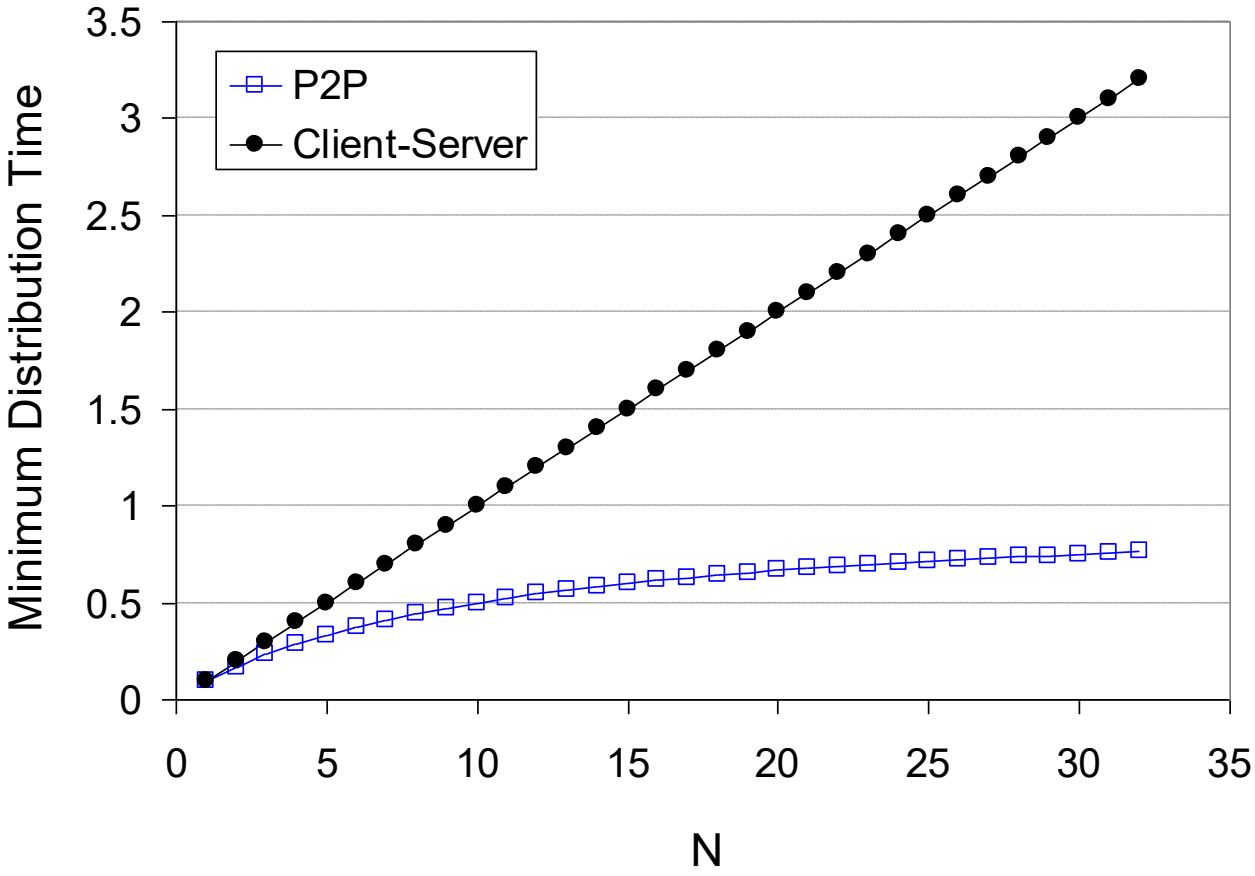
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

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

Client-server vs. P2P: example

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

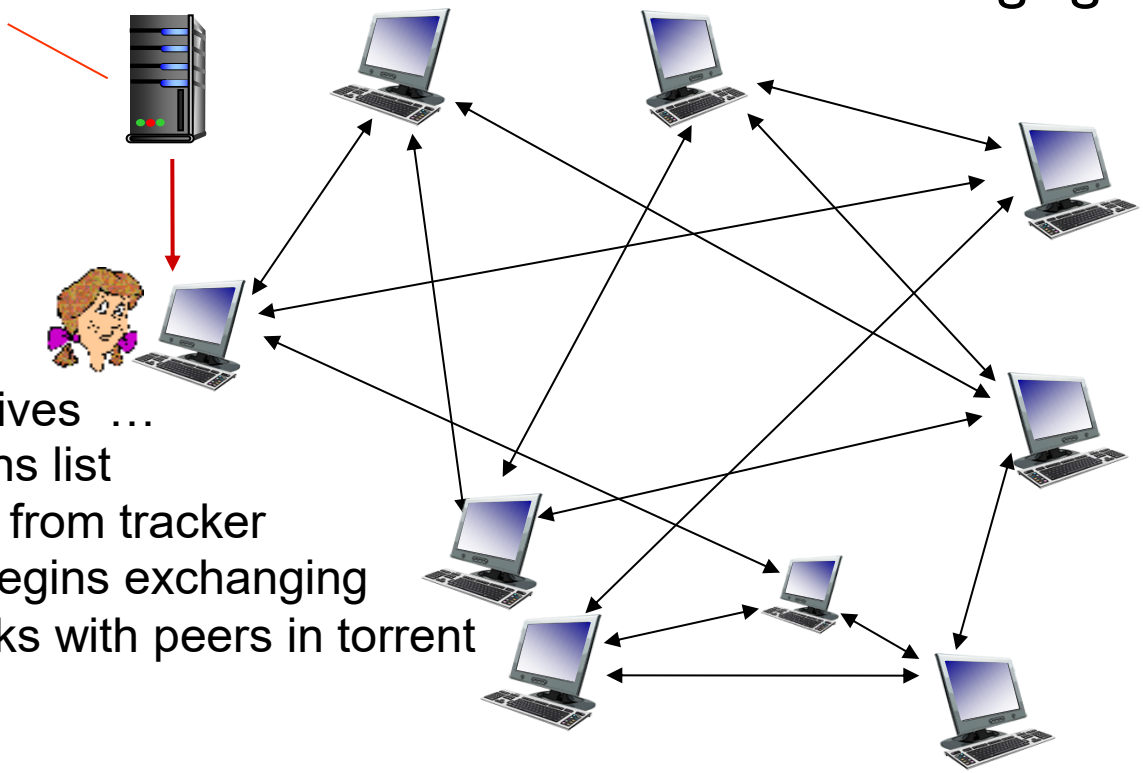


P2P file distribution: BitTorrent

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

tracker: tracks peers participating in torrent

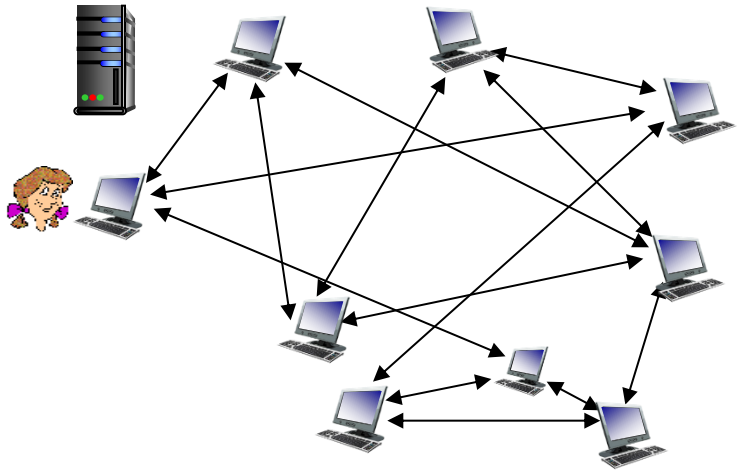
torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

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

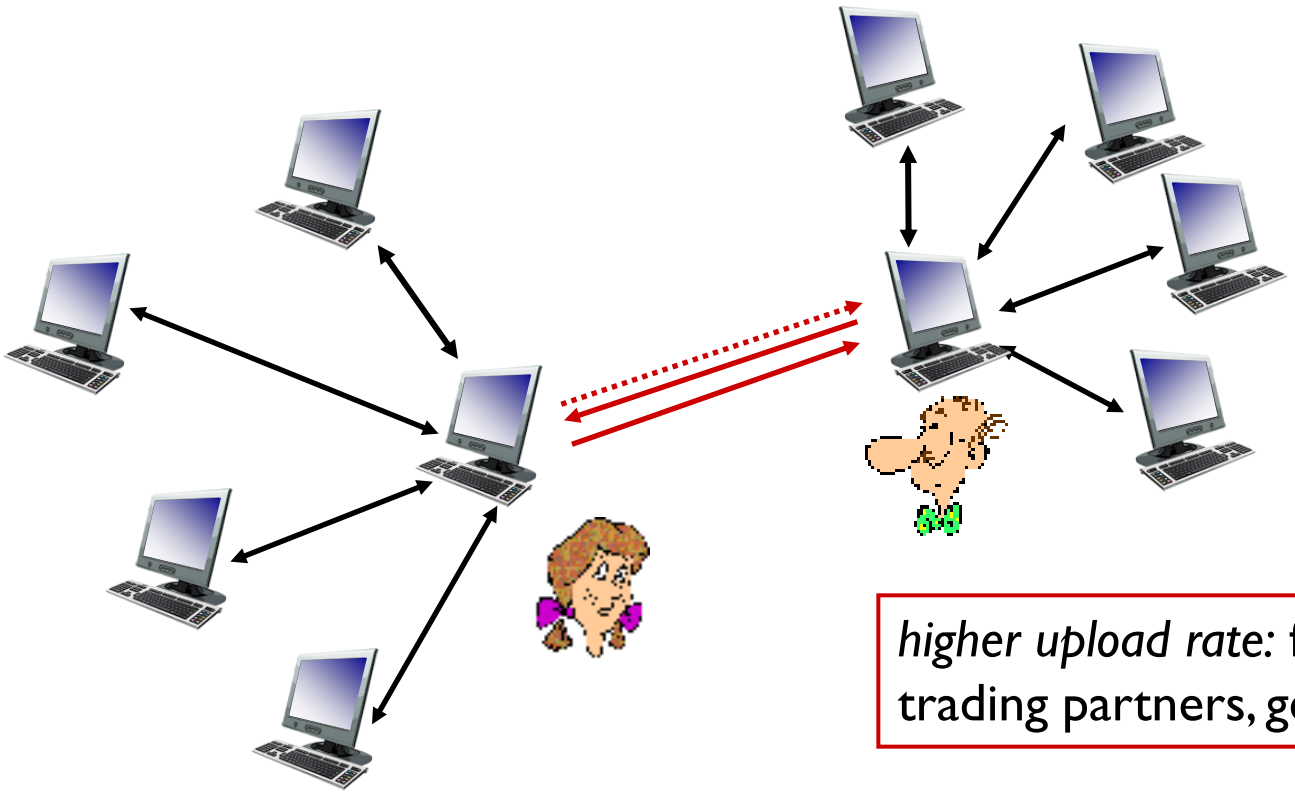
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - 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 !

Chapter 2: 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

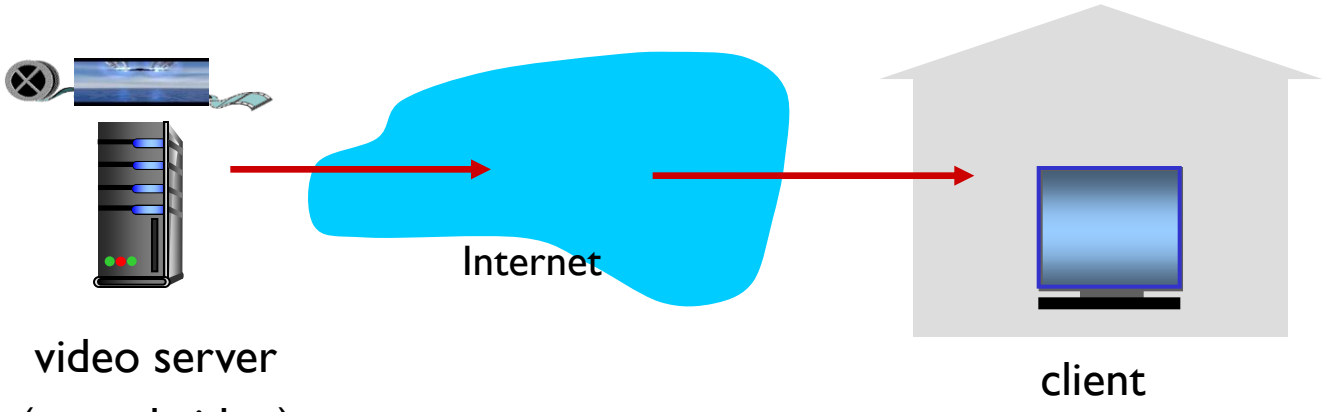
2.7 socket programming with UDP and TCP

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

Streaming stored video:

simple scenario:

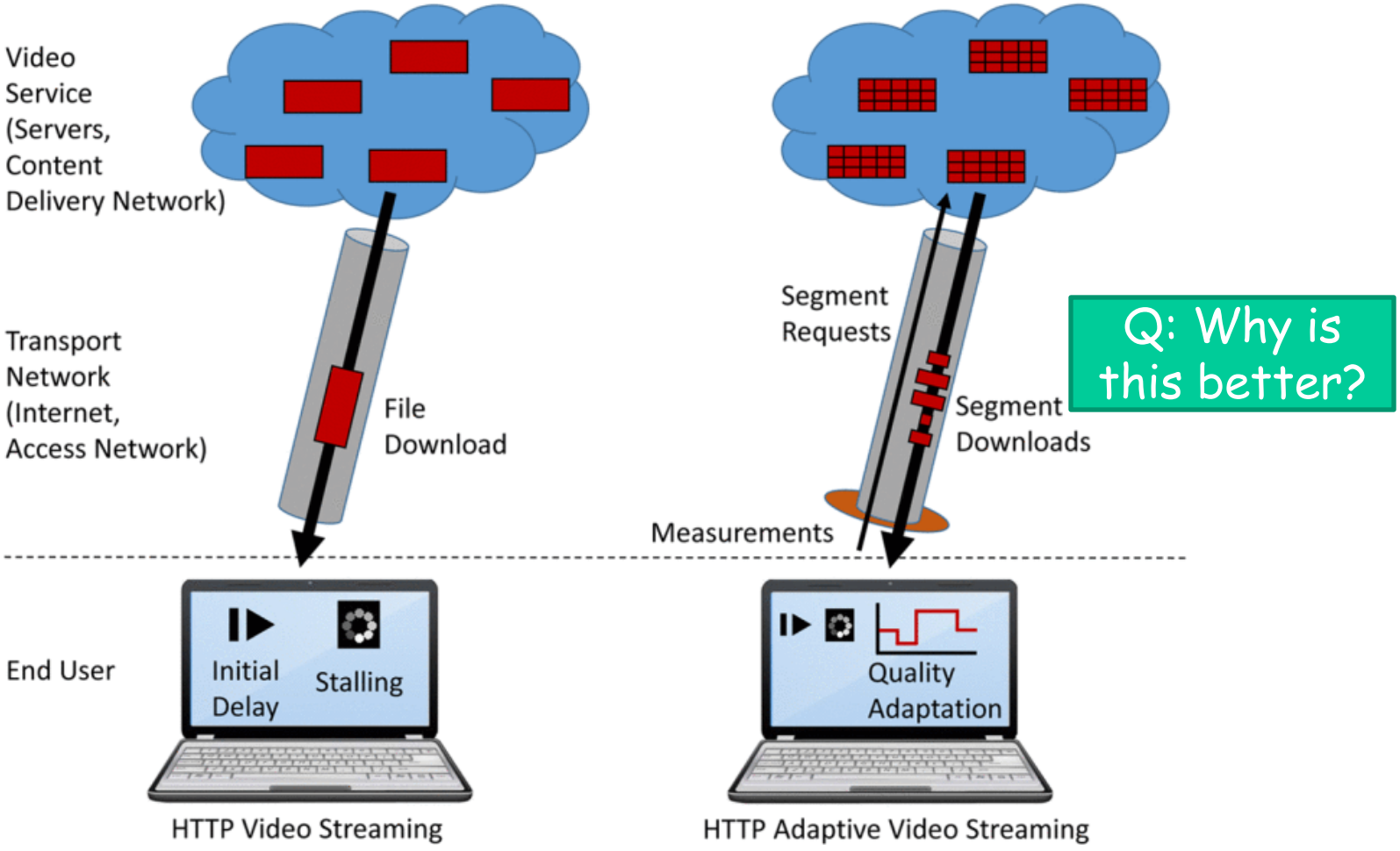


Main challenges:

- 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

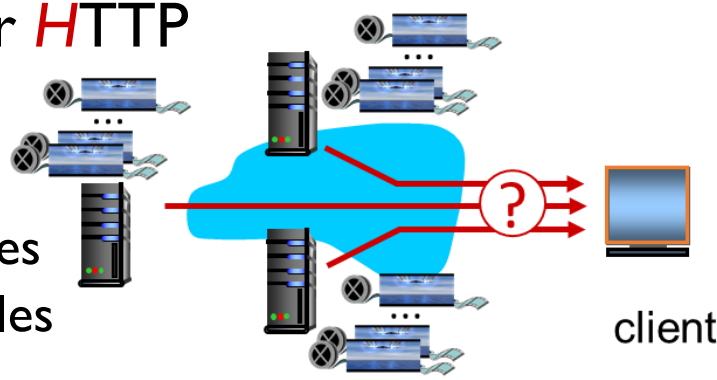
HTTP Adaptive Streaming (HAS)

Comparison of HTTP video streaming and HTTP adaptive video streaming

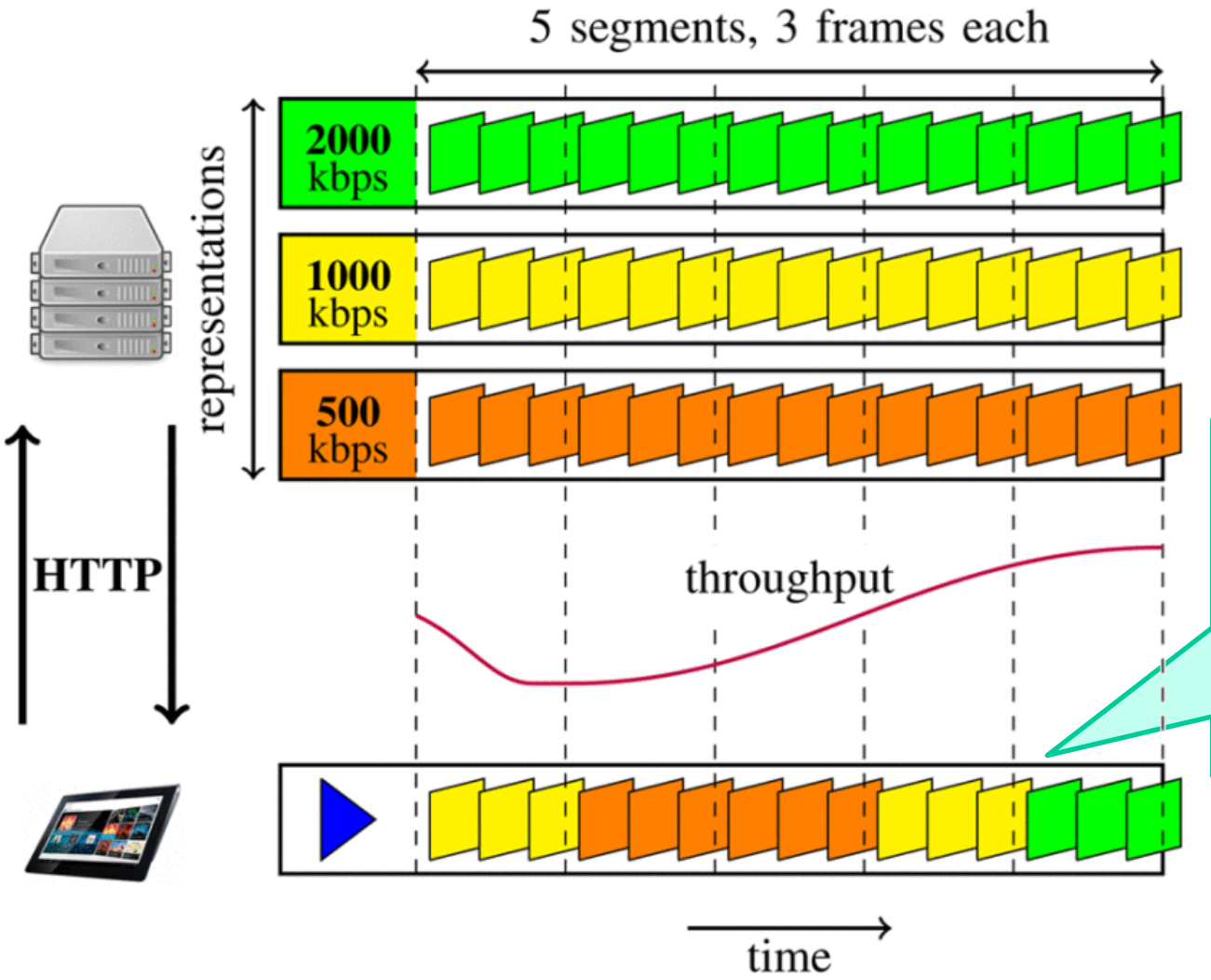


Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *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 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)



DASH example

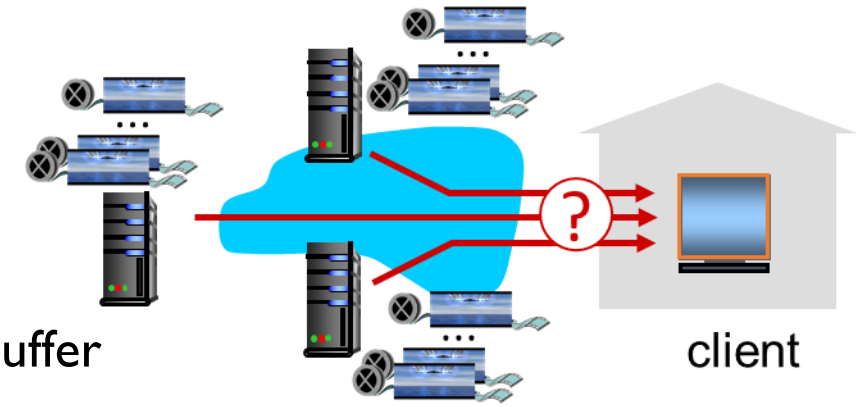


Based on how fast the current (and previous) segments are downloaded, the bit rate of the next segment is selected

* M. Seufert, S. Egger, M.Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, “Survey on Quality of Experience of HTTP Adaptive Streaming”, IEEE Communication Surveys & Tutorials, Vol. 17, No. 1, 2015.

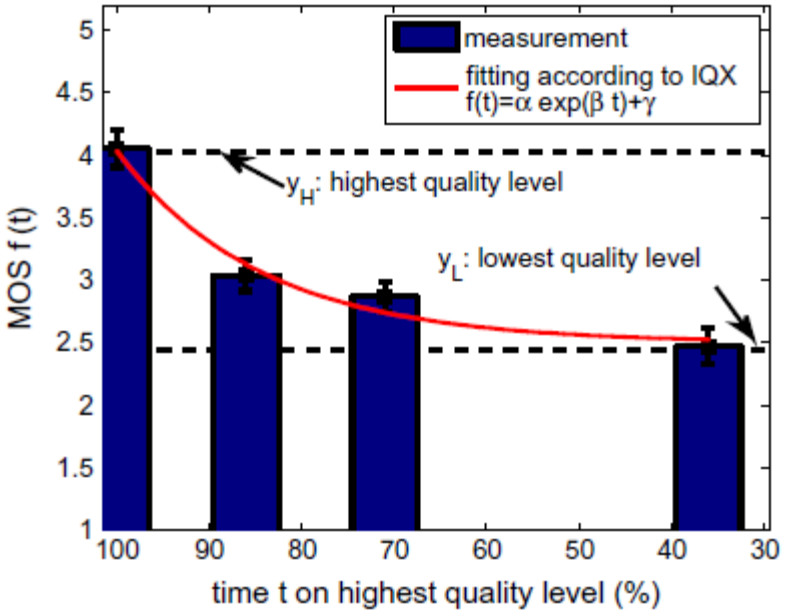
Streaming multimedia: DASH

- *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)



Streaming video = encoding + DASH + playout buffering

HTTP Adaptive Streaming (HAS)

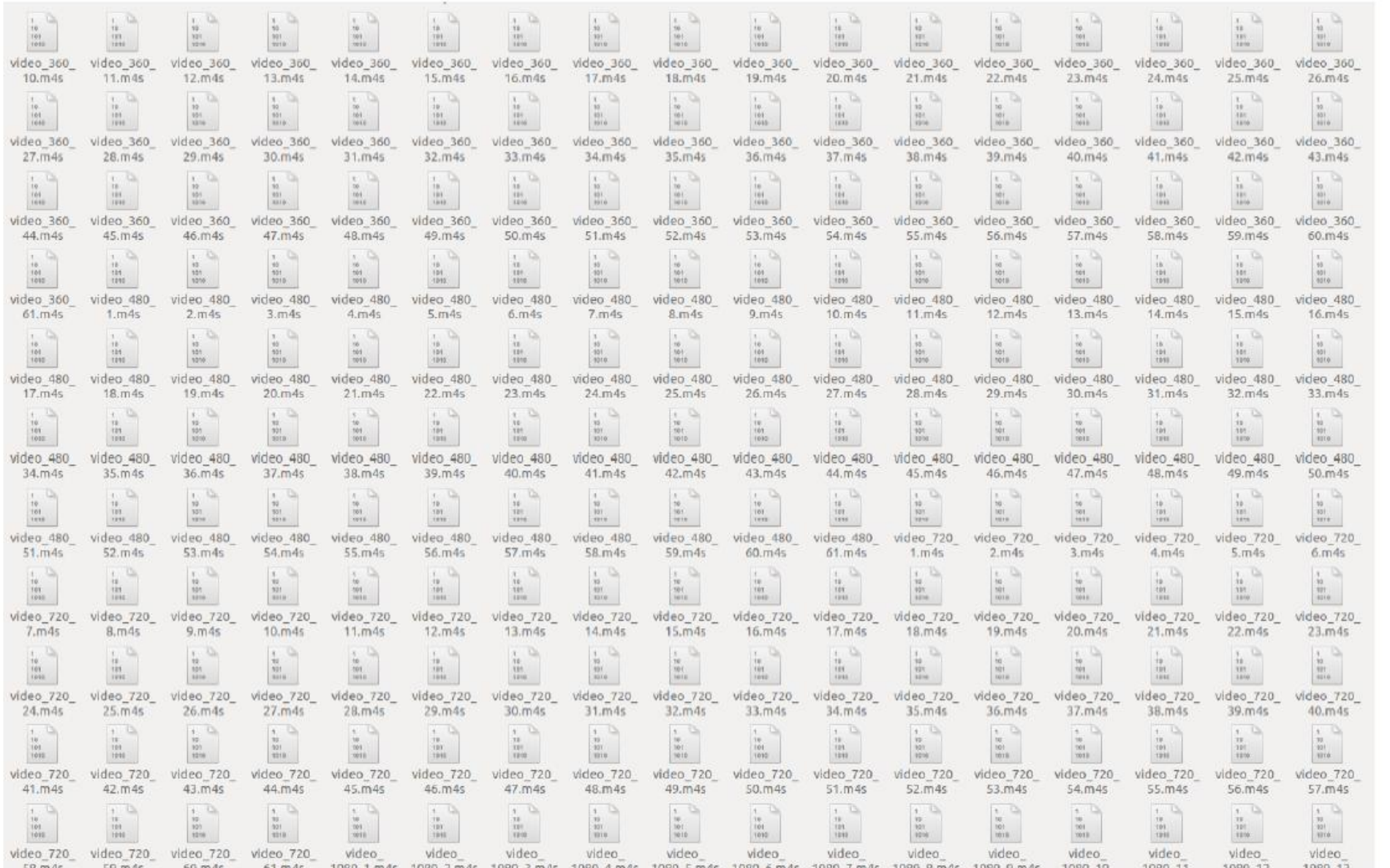


$$QoE = 0.003 * e^{0.064*t} + 2.498$$

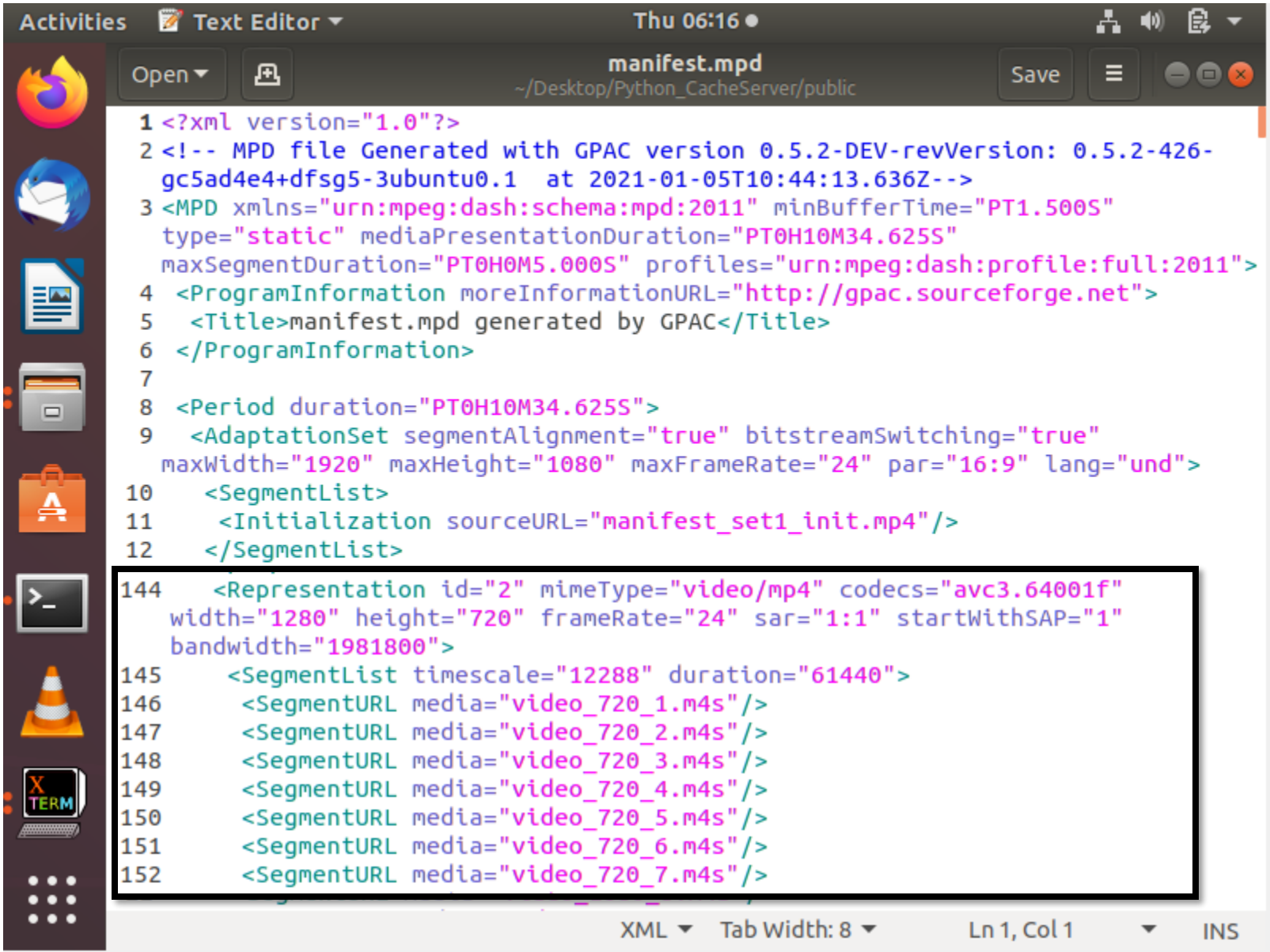
t = time on highest layer

- Adaptation frequency (number of switches), adaptation amplitude, adaptation direction, segment length, buffer size, etc.

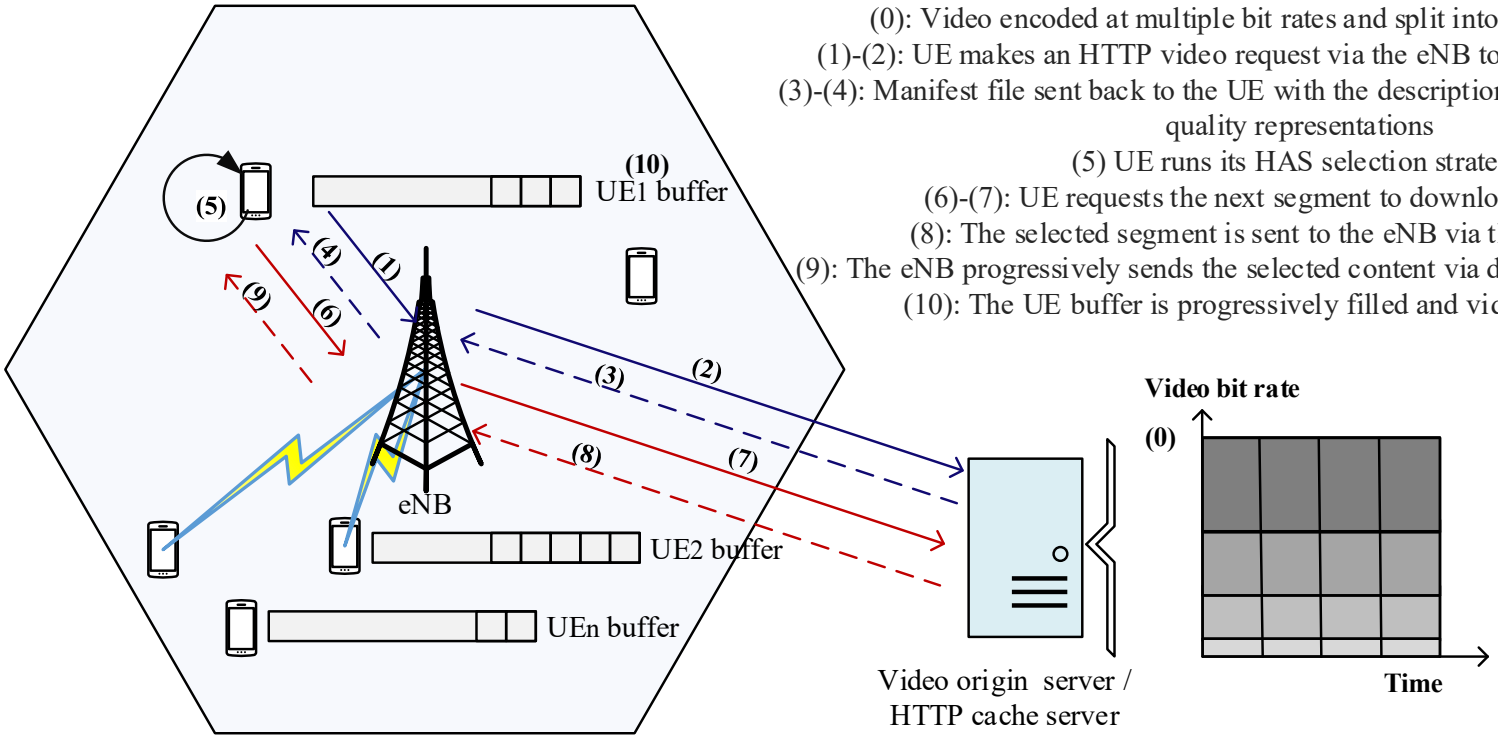
Chunks


video_360_10.m4s
video_360_11.m4s
video_360_12.m4s
video_360_13.m4s
video_360_14.m4s
video_360_15.m4s
video_360_16.m4s
video_360_17.m4s
video_360_18.m4s
video_360_19.m4s
video_360_20.m4s
video_360_21.m4s
video_360_22.m4s
video_360_23.m4s
video_360_24.m4s
video_360_25.m4s
video_360_26.m4s
video_360_27.m4s
video_360_28.m4s
video_360_29.m4s
video_360_30.m4s
video_360_31.m4s
video_360_32.m4s
video_360_33.m4s
video_360_34.m4s
video_360_35.m4s
video_360_36.m4s
video_360_37.m4s
video_360_38.m4s
video_360_39.m4s
video_360_40.m4s
video_360_41.m4s
video_360_42.m4s
video_360_43.m4s
video_360_44.m4s
video_360_45.m4s
video_360_46.m4s
video_360_47.m4s
video_360_48.m4s
video_360_49.m4s
video_360_50.m4s
video_360_51.m4s
video_360_52.m4s
video_360_53.m4s
video_360_54.m4s
video_360_55.m4s
video_360_56.m4s
video_360_57.m4s
video_360_58.m4s
video_360_59.m4s
video_360_60.m4s
video_360_61.m4s
video_480_1.m4s
video_480_2.m4s
video_480_3.m4s
video_480_4.m4s
video_480_5.m4s
video_480_6.m4s
video_480_7.m4s
video_480_8.m4s
video_480_9.m4s
video_480_10.m4s
video_480_11.m4s
video_480_12.m4s
video_480_13.m4s
video_480_14.m4s
video_480_15.m4s
video_480_16.m4s
video_480_17.m4s
video_480_18.m4s
video_480_19.m4s
video_480_20.m4s
video_480_21.m4s
video_480_22.m4s
video_480_23.m4s
video_480_24.m4s
video_480_25.m4s
video_480_26.m4s
video_480_27.m4s
video_480_28.m4s
video_480_29.m4s
video_480_30.m4s
video_480_31.m4s
video_480_32.m4s
video_480_33.m4s
video_480_34.m4s
video_480_35.m4s
video_480_36.m4s
video_480_37.m4s
video_480_38.m4s
video_480_39.m4s
video_480_40.m4s
video_480_41.m4s
video_480_42.m4s
video_480_43.m4s
video_480_44.m4s
video_480_45.m4s
video_480_46.m4s
video_480_47.m4s
video_480_48.m4s
video_480_49.m4s
video_480_50.m4s
video_480_51.m4s
video_480_52.m4s
video_480_53.m4s
video_480_54.m4s
video_480_55.m4s
video_480_56.m4s
video_480_57.m4s
video_480_58.m4s
video_480_59.m4s
video_480_60.m4s
video_480_61.m4s
video_720_1.m4s
video_720_2.m4s
video_720_3.m4s
video_720_4.m4s
video_720_5.m4s
video_720_6.m4s
video_720_7.m4s
video_720_8.m4s
video_720_9.m4s
video_720_10.m4s
video_720_11.m4s
video_720_12.m4s
video_720_13.m4s
video_720_14.m4s
video_720_15.m4s
video_720_16.m4s
video_720_17.m4s
video_720_18.m4s
video_720_19.m4s
video_720_20.m4s
video_720_21.m4s
video_720_22.m4s
video_720_23.m4s
video_720_24.m4s
video_720_25.m4s
video_720_26.m4s
video_720_27.m4s
video_720_28.m4s
video_720_29.m4s
video_720_30.m4s
video_720_31.m4s
video_720_32.m4s
video_720_33.m4s
video_720_34.m4s
video_720_35.m4s
video_720_36.m4s
video_720_37.m4s
video_720_38.m4s
video_720_39.m4s
video_720_40.m4s
video_720_41.m4s
video_720_42.m4s
video_720_43.m4s
video_720_44.m4s
video_720_45.m4s
video_720_46.m4s
video_720_47.m4s
video_720_48.m4s
video_720_49.m4s
video_720_50.m4s
video_720_51.m4s
video_720_52.m4s
video_720_53.m4s
video_720_54.m4s
video_720_55.m4s
video_720_56.m4s
video_720_57.m4s

Manifest file



DASH example: LTE



(0): Video encoded at multiple bit rates and split into temporal segments
(1)-(2): UE makes an HTTP video request via the eNB to the video server/ MEC
(3)-(4): Manifest file sent back to the UE with the description and URLs of all available quality representations
(5) UE runs its HAS selection strategy
(6)-(7): UE requests the next segment to download via the eNB
(8): The selected segment is sent to the eNB via the backhaul link
(9): The eNB progressively sends the selected content via downlink radio scheduling
(10): The UE buffer is progressively filled and video playout starts

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*

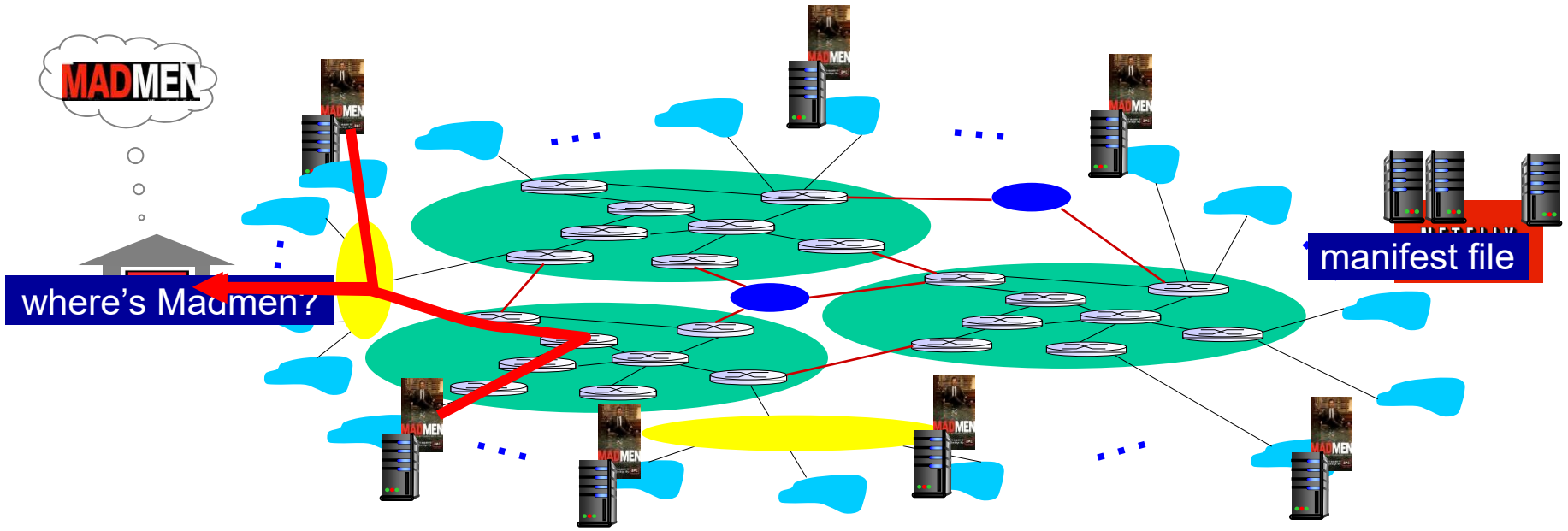
Content distribution networks

- *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 (edge)
    - close to users
    - used by Akamai, 1700 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)

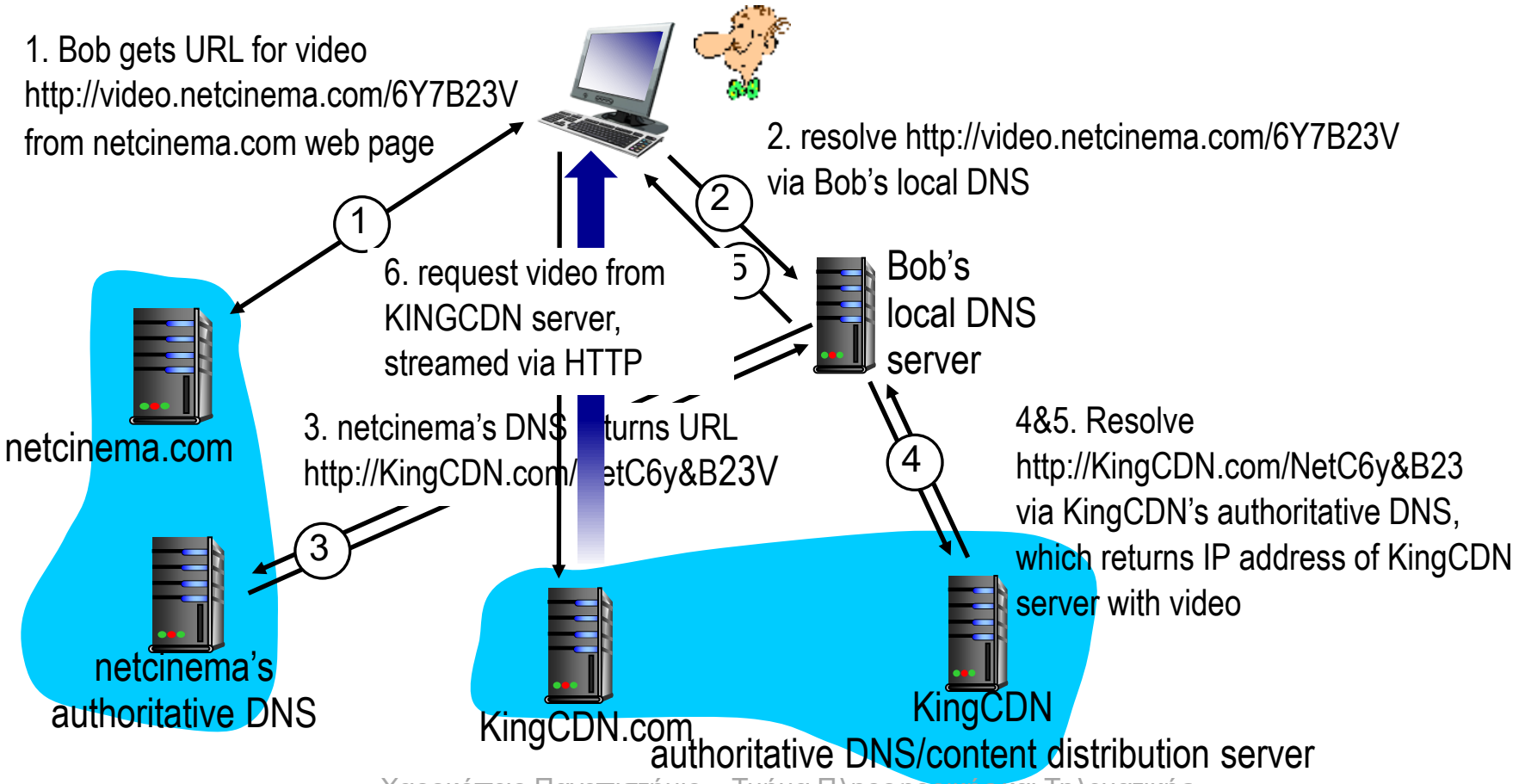
- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested



CDN content access: a closer look

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

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



Content Distribution Networks (CDNs)

Internet host-host communication as a service

OTT challenges: coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

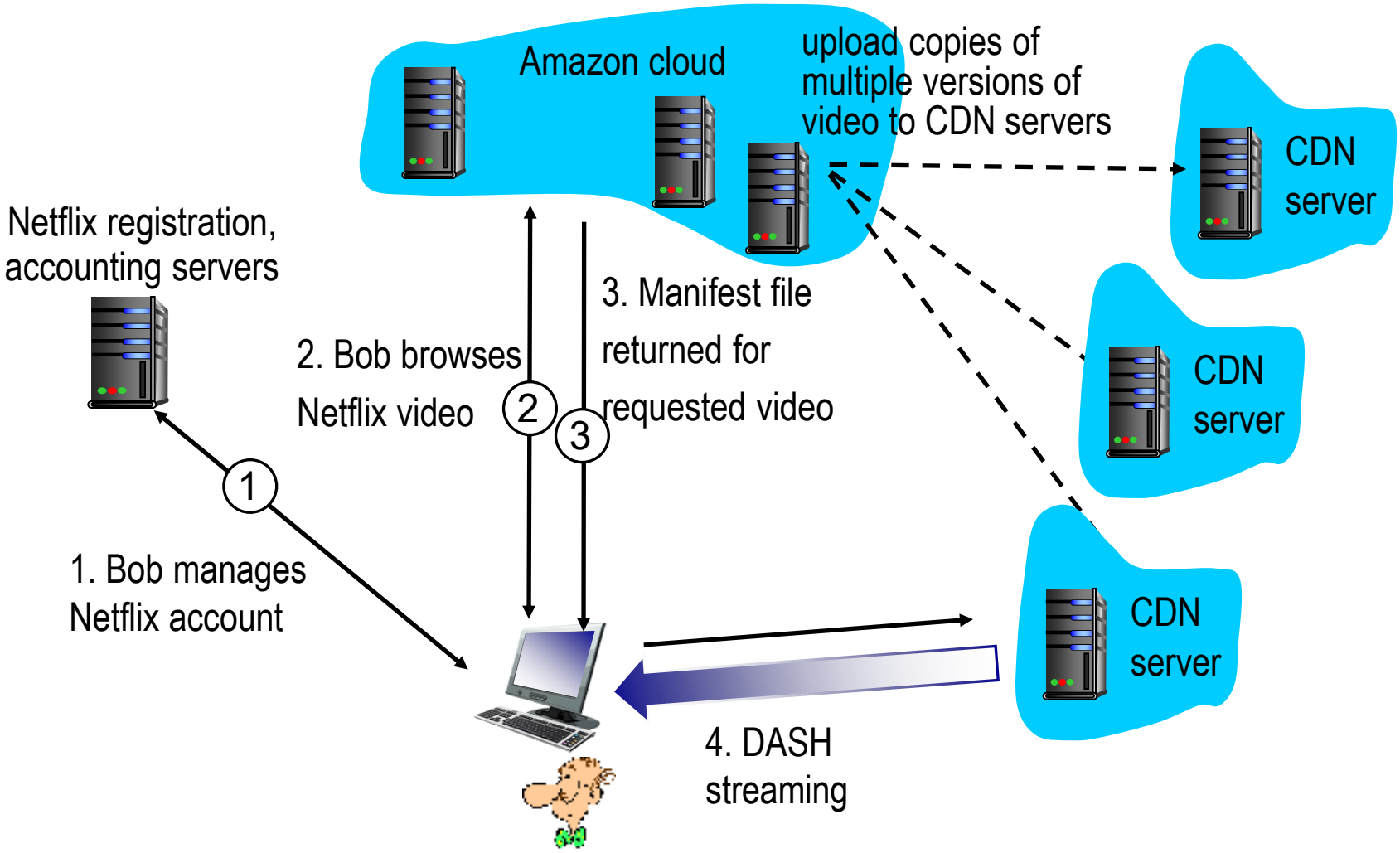
Content distribution networks

- The CDN must intercept the request so that it can:
  - Determine a suitable CDN server cluster for that client at that time, and
  - Redirect the client's request to a server in that cluster
- Cluster selection strategy is a mechanism for dynamically directing clients to a server cluster / data center within the CDN
  - CDN learns the IP address of the client's LDNS server via the client's DNS lookup. After learning this IP address, the CDN needs to select an appropriate cluster based on this IP address
  - CDNs generally employ proprietary cluster selection strategies (geographically closest / real-time measurements-based / etc.)

Case study: Netflix

- Netflix video distribution has two major components: the Amazon cloud and its own private CDN infrastructure.
- **Content ingestion.** Before Netflix can distribute a movie to its customers, it must first ingest and process the movie
- **Content processing.** The machines in the Amazon cloud create many different formats for each movie, suitable for a diverse array of client video players running on desktop computers, smartphones, and game consoles connected to televisions
- **Uploading versions to its CDN.** Once all of the versions of a movie have been created, the hosts in the Amazon cloud upload the versions to its CDN.

Case study: Netflix



Chapter 2: 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

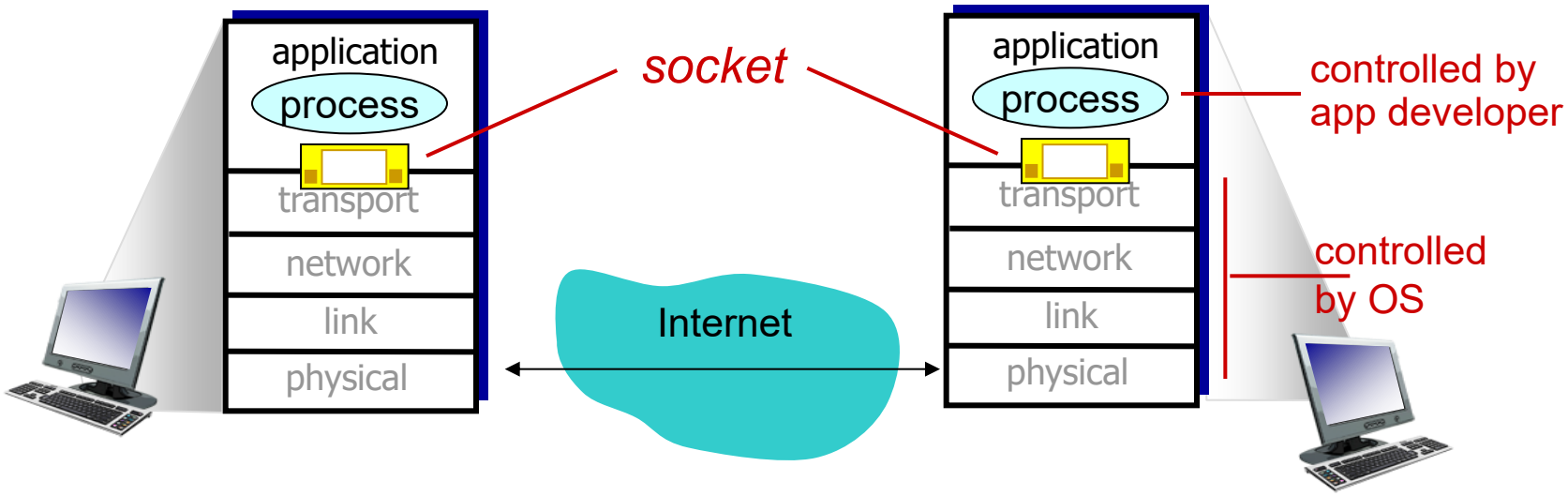
2.6 video streaming and content distribution networks

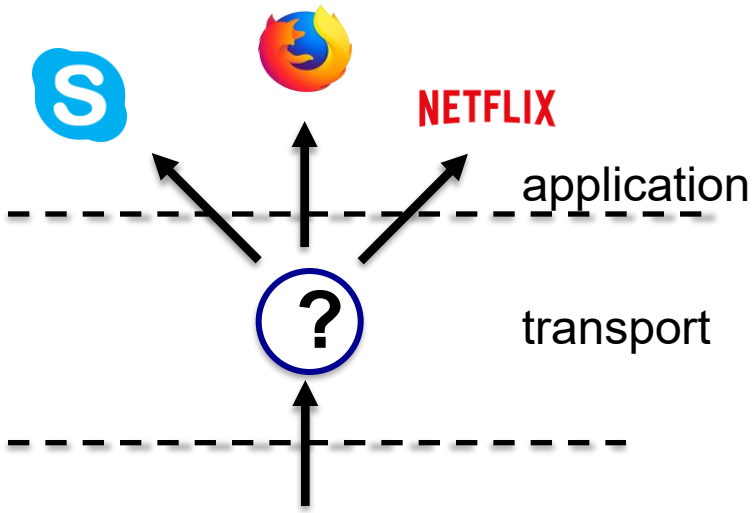
2.7 socket programming with UDP and TCP

Socket programming

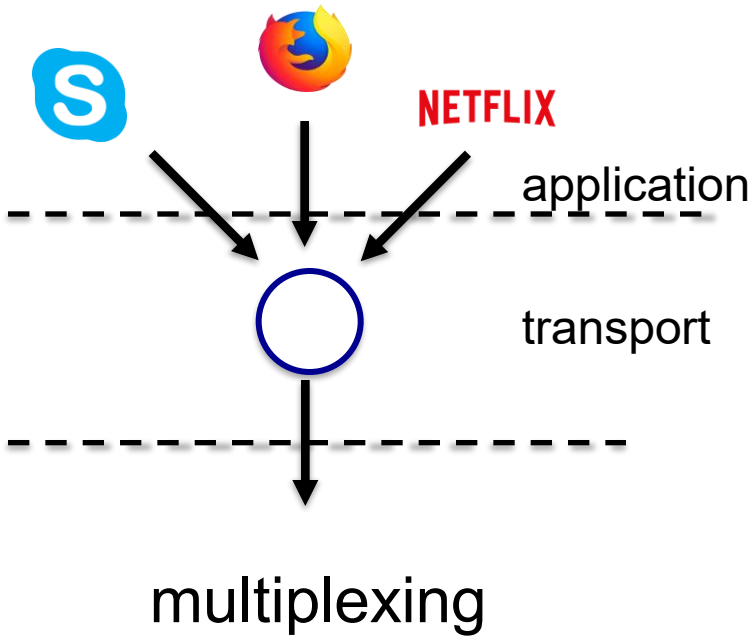
goal: learn how to build client/server applications that communicate using sockets

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





de-multiplexing



multiplexing

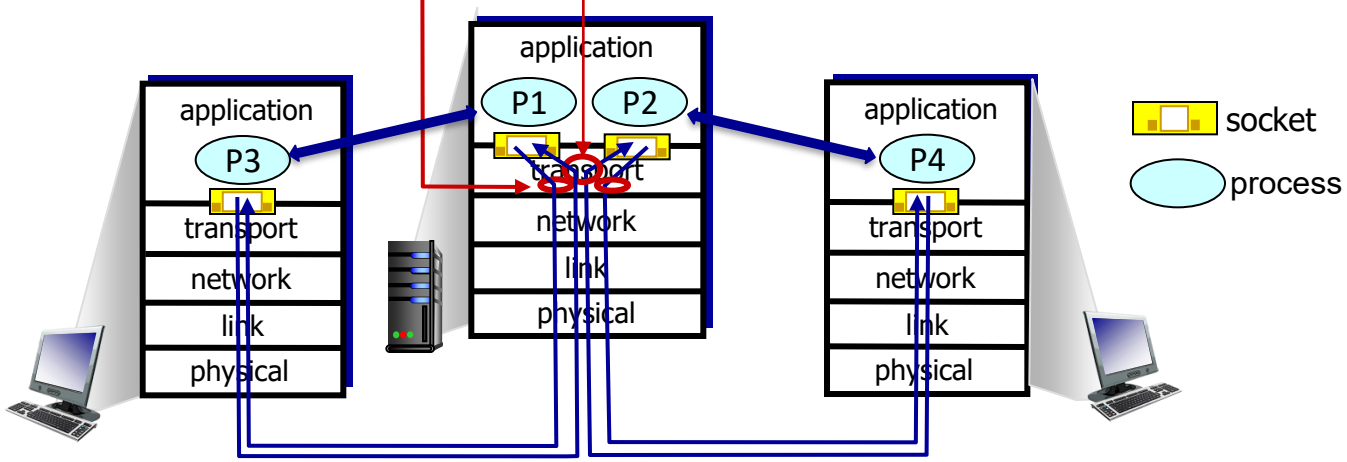
Multiplexing/demultiplexing

multiplexing as sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

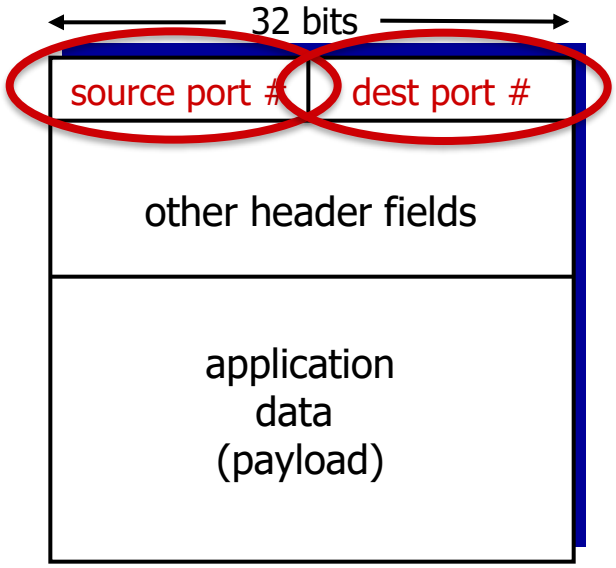
demultiplexing as receiver:

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



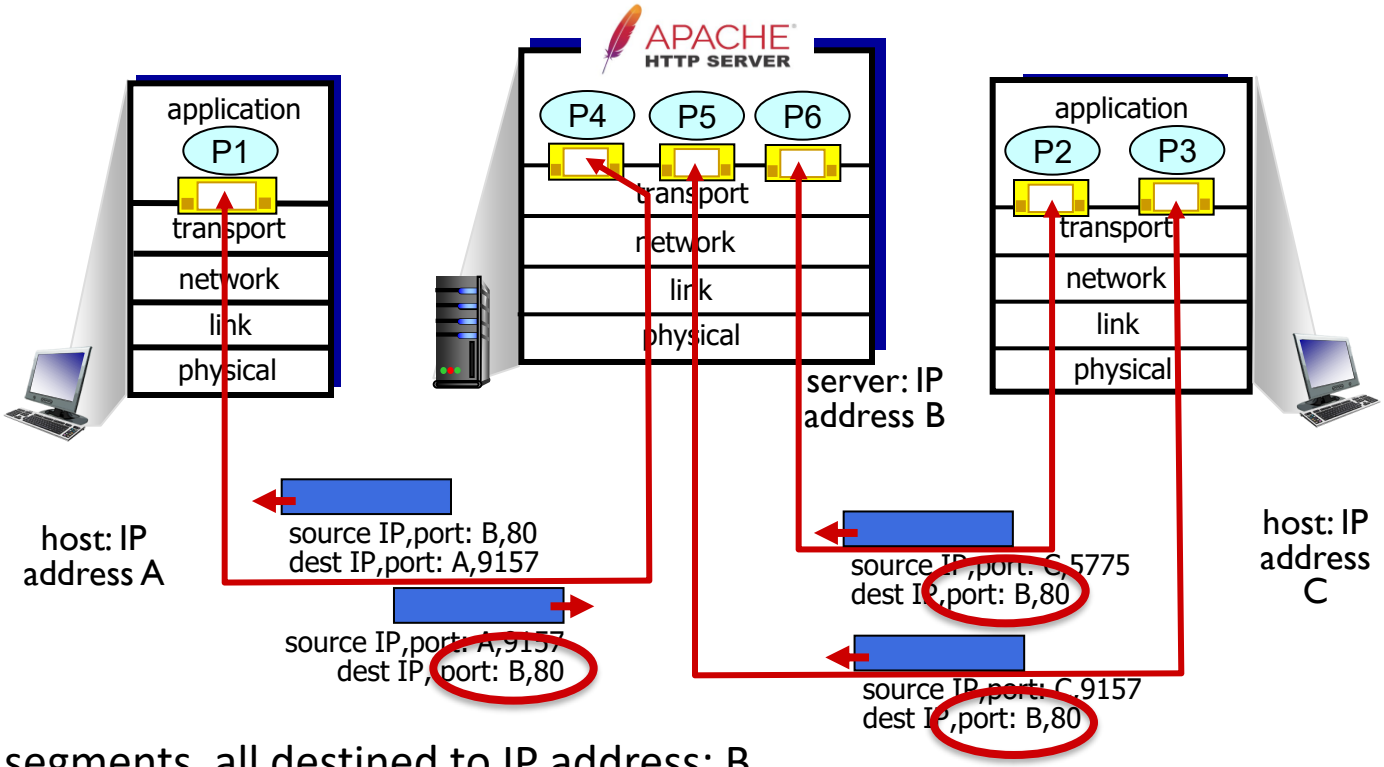
TCP/UDP segment format

Connection-oriented demultiplexing

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

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

Connection-oriented demultiplexing: example



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

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 & server

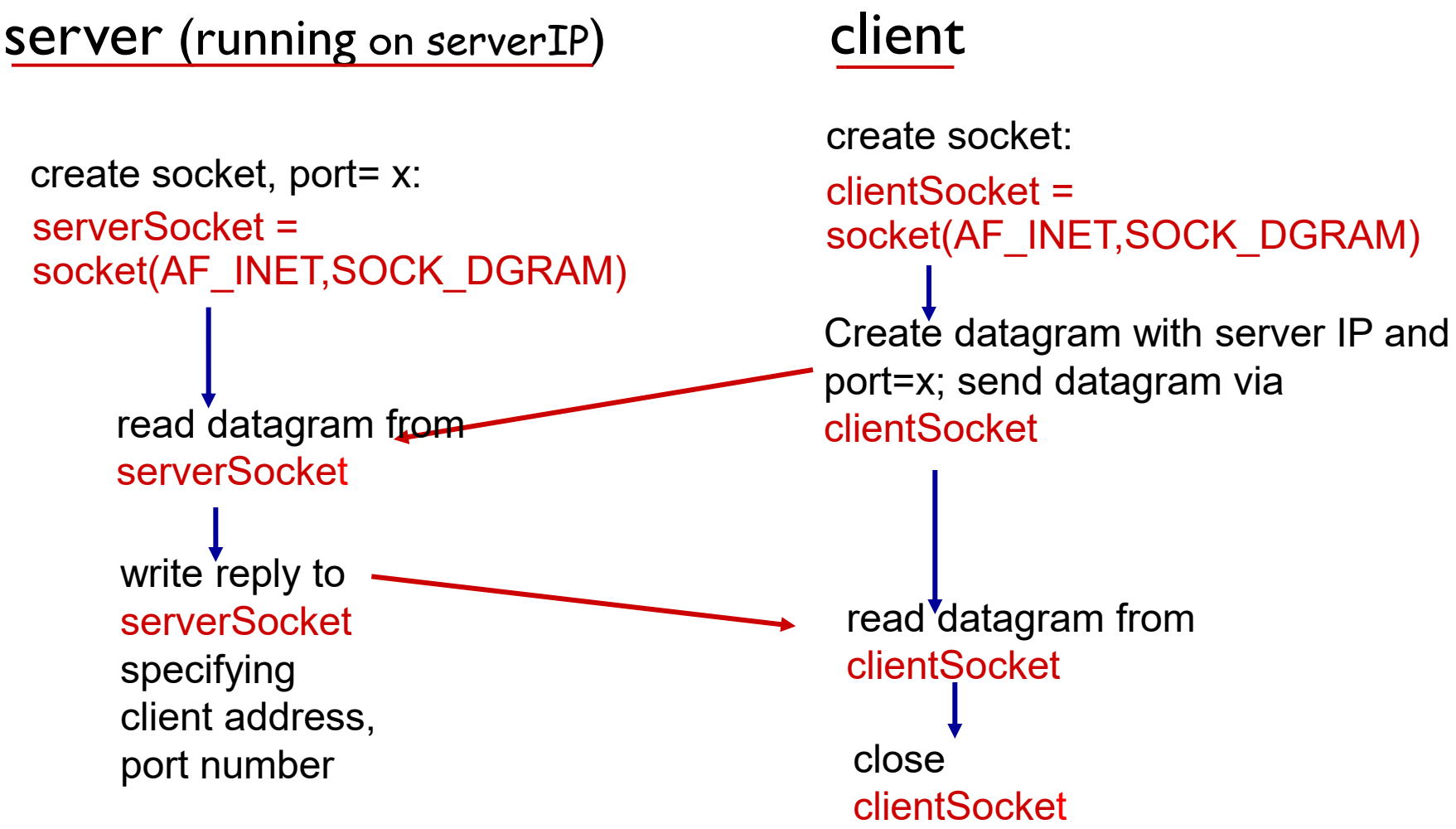
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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

Client/server socket interaction: UDP



Example app: UDP client

Python UDPClient

include Python's socket library

create UDP socket for server

get user keyboard input

Attach server name, port to message; send into socket

read reply characters from socket into string

print out received string and close socket

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
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)
```

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

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

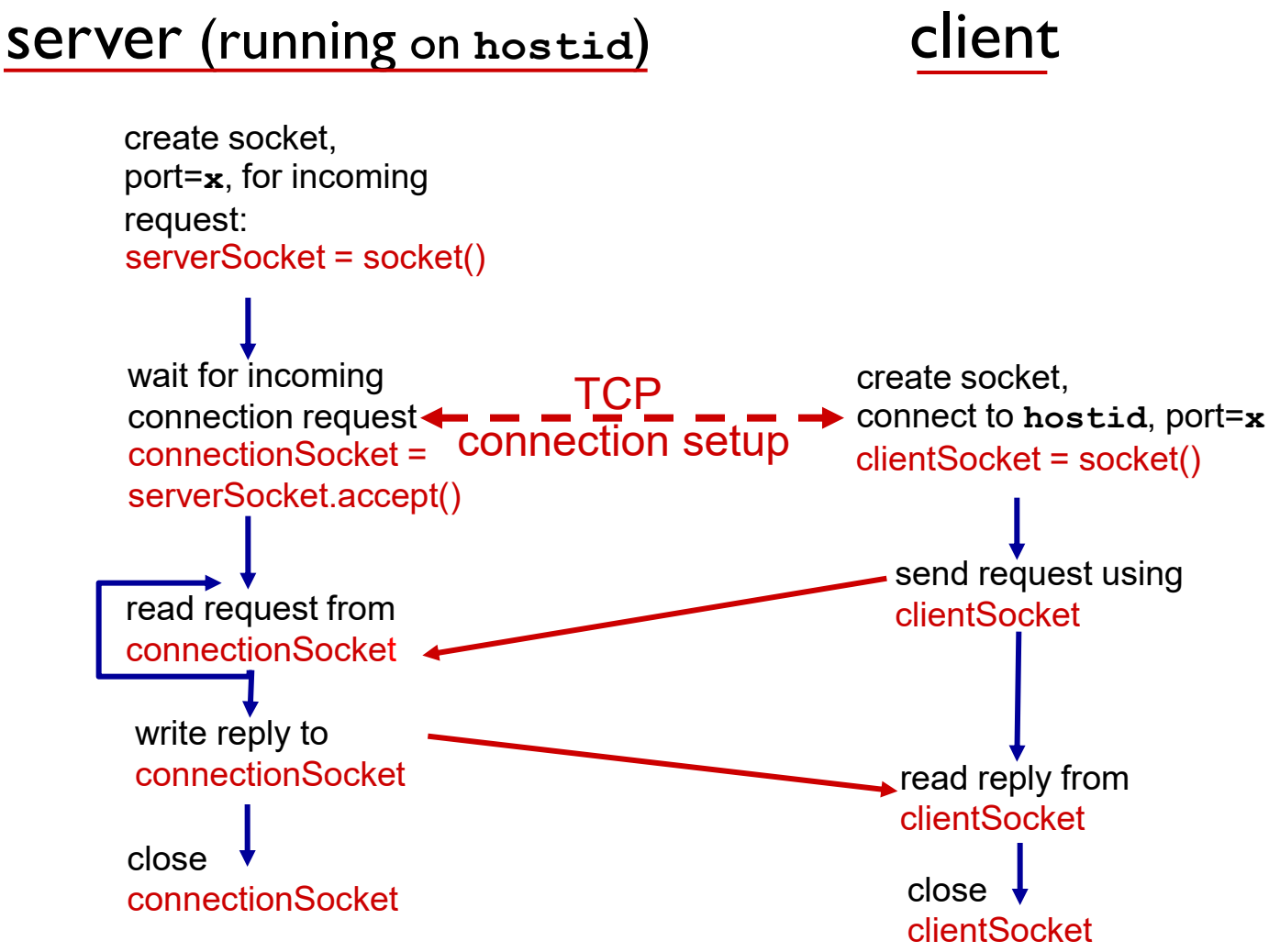
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

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

application viewpoint:

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

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

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

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
 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()
```

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)

Chapter 2: summary

our study of network apps 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
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming: TCP, UDP sockets

Chapter 2: 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:

- control vs. messages
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable message transfer
- "complexity at network edge"