CS 305: Computer Networks

Fall 2022

Lecture 5: Application Layer

Ming Tang

Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

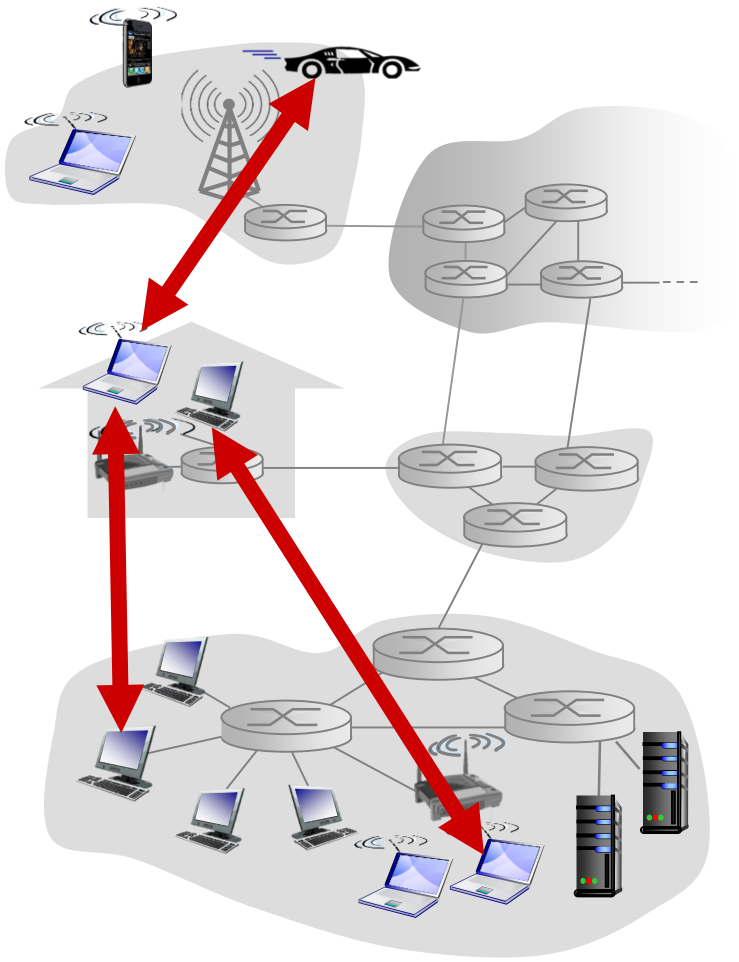
2.5 P2P applications

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



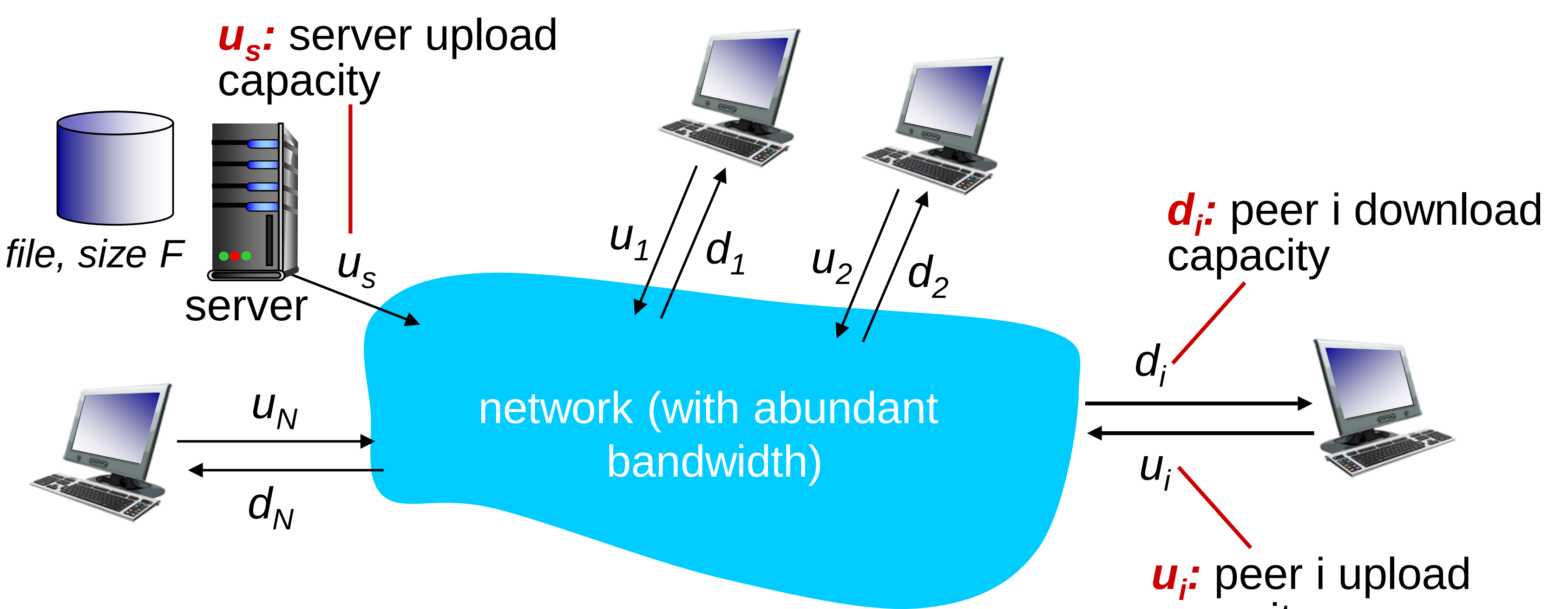
Examples:

- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)

File distribution: client-server vs P2P

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

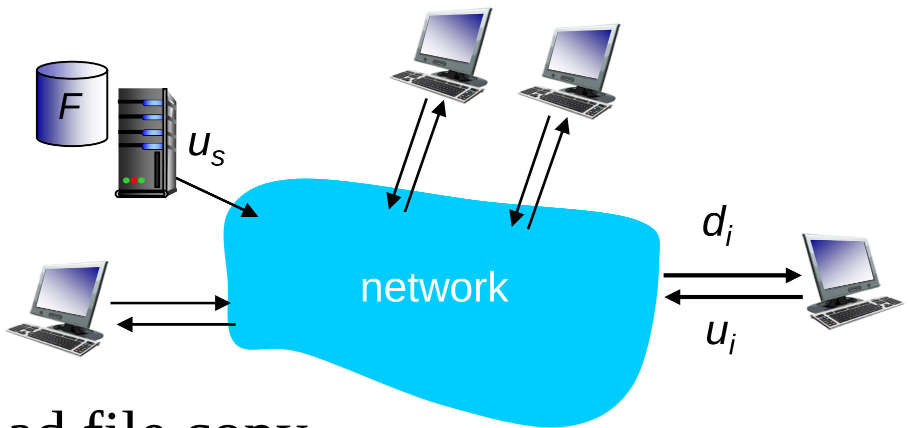
- peer upload/download capacity is limited resource



The **distribution time** is the time it takes to get a copy of the file to all N peers.

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
  - min client download time: F/d_{min}

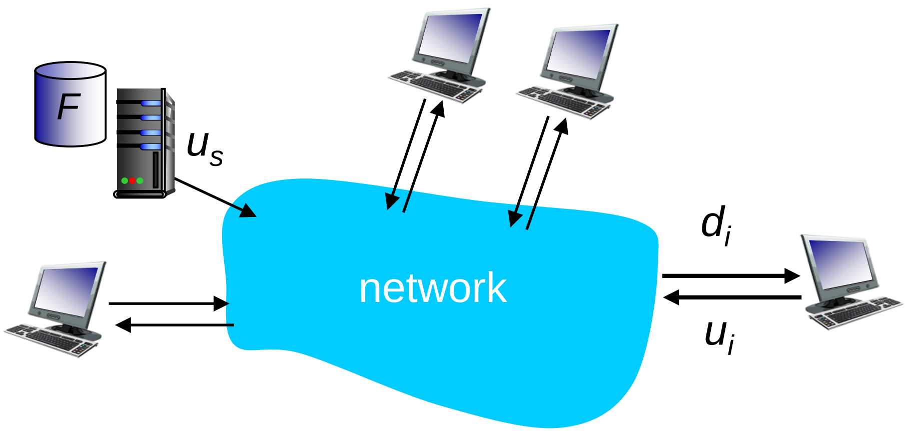
time to distribute F to N clients using client-server approach

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

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 downloading: each client must download file copy
  - min client download time: F/d_{min}
- Clients and server: delivering a total of 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

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

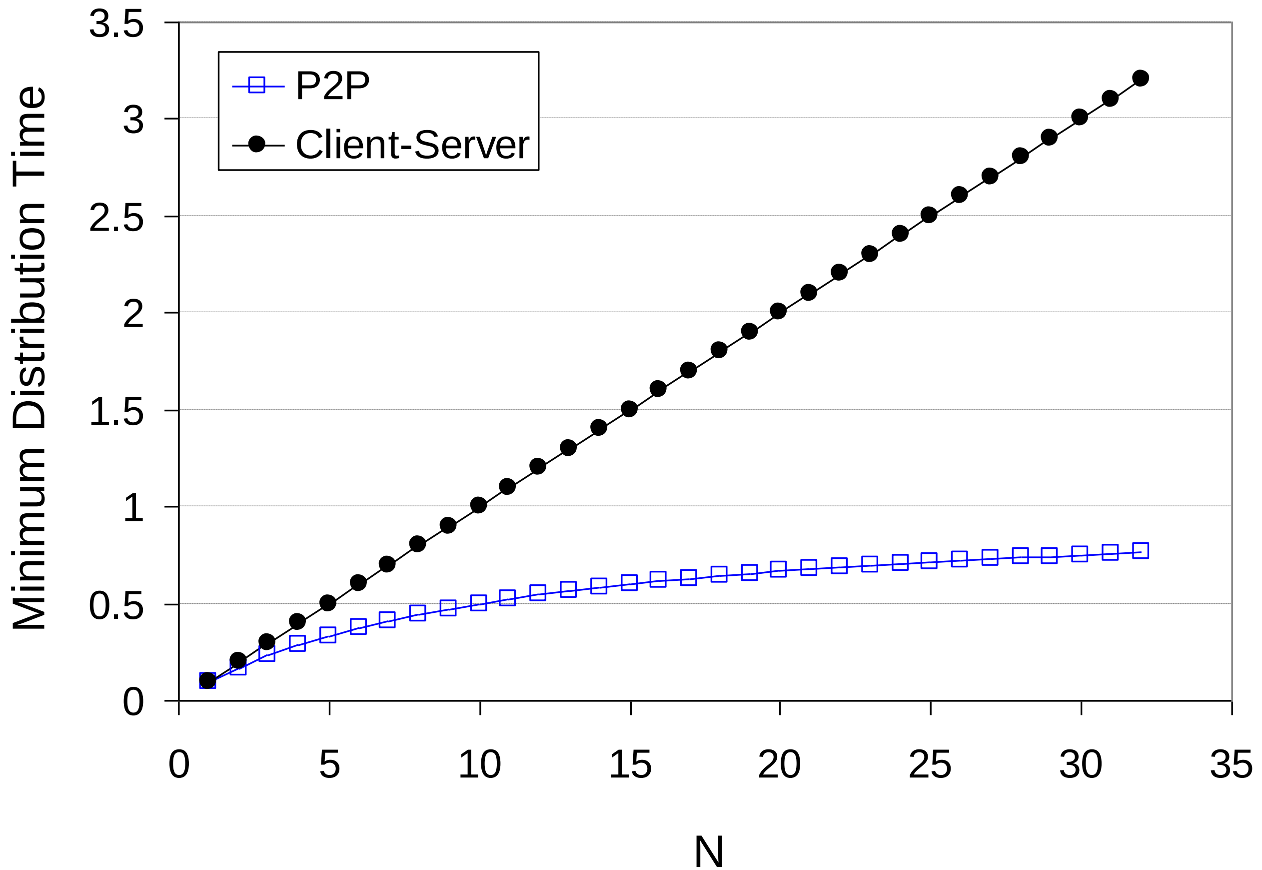
If each peer can redistribute a bit as soon as it receives the bit, then there is a scheme that actually achieves this lower bound

increases linearly in N …

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

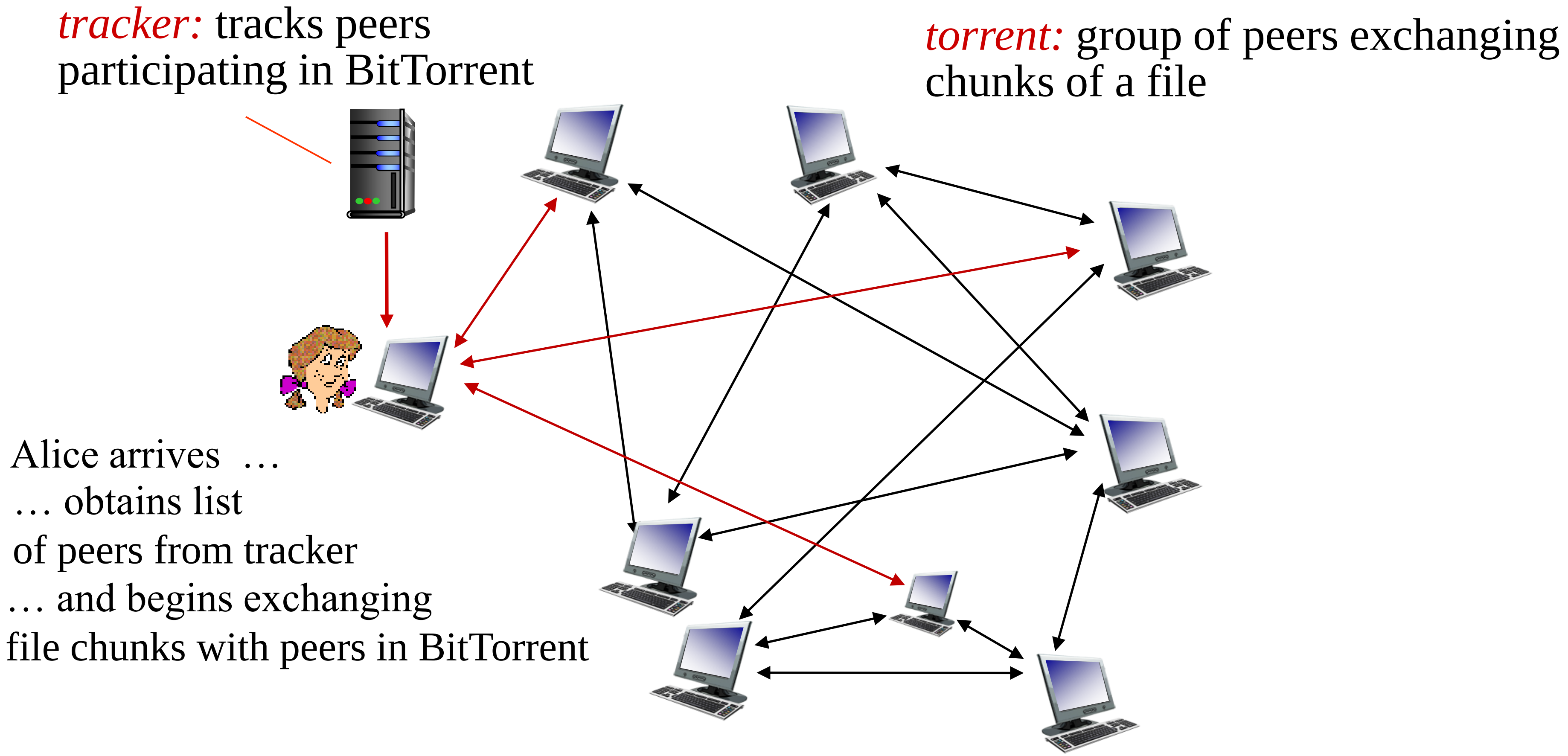
Client-server vs. P2P: example

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



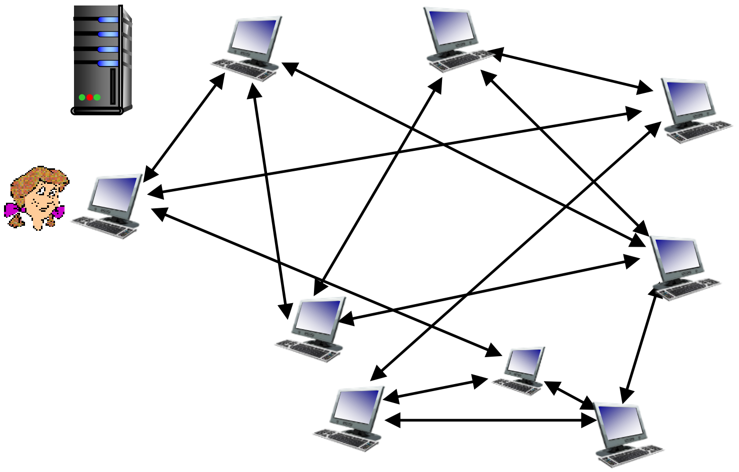
P2P file distribution: BitTorrent

- File divided into 256Kb chunks
- Peers in BitTorrent send/receive file chunks



P2P file distribution: BitTorrent

- Peer joining BitTorrent:
  - 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
  - Peers may come and go
  - The neighbors may change
- Once peer has entire file, it may (selfishly) leave or (altruistically) remain in BitTorrent

BitTorrent: requesting, sending file chunks

Q1: which chunks should she request first from her neighbors?

1	2	3	4	5	6

Q2: to which of her neighbors should she send requested 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

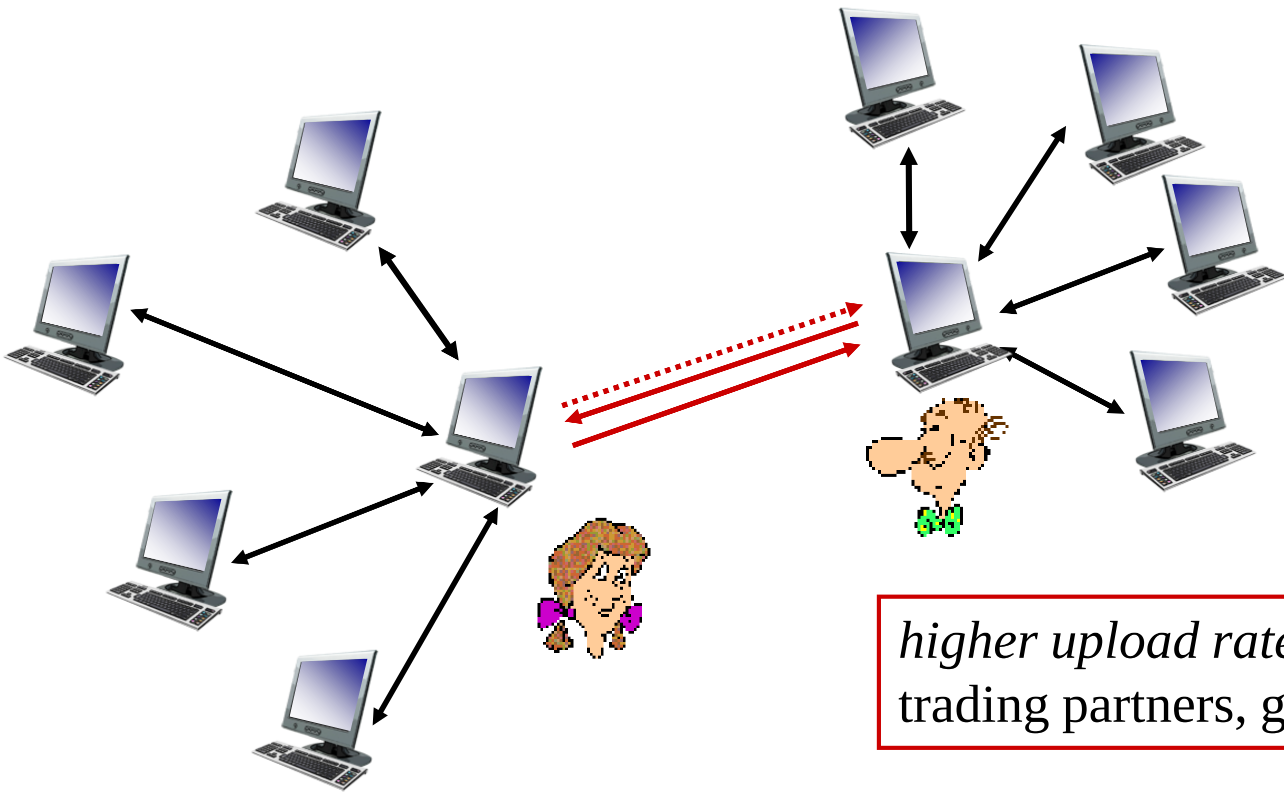
- Alice sends chunks to those four peers currently sending her chunks at highest rate
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly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

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!

Skype

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

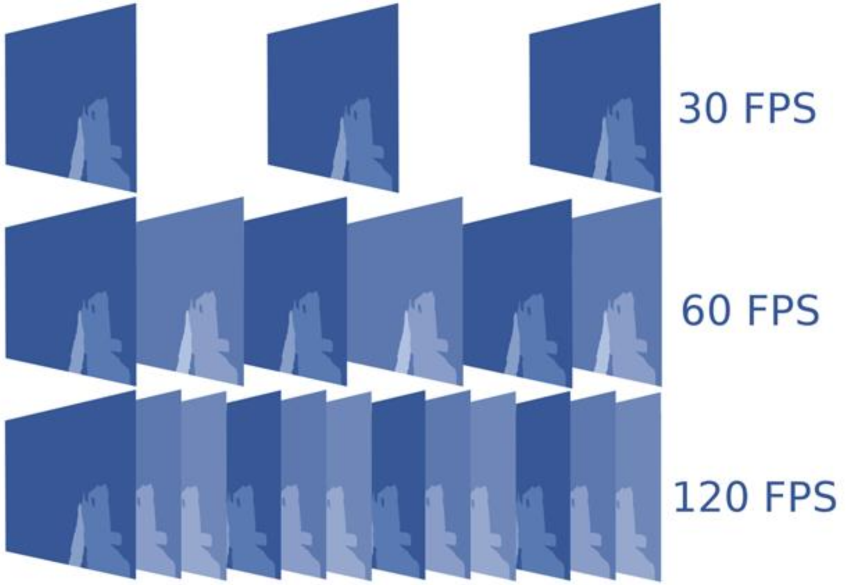
2.7 socket programming with UDP and TCP

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

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



Multimedia: video

Coding (Compression): 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*

frame *i+1*

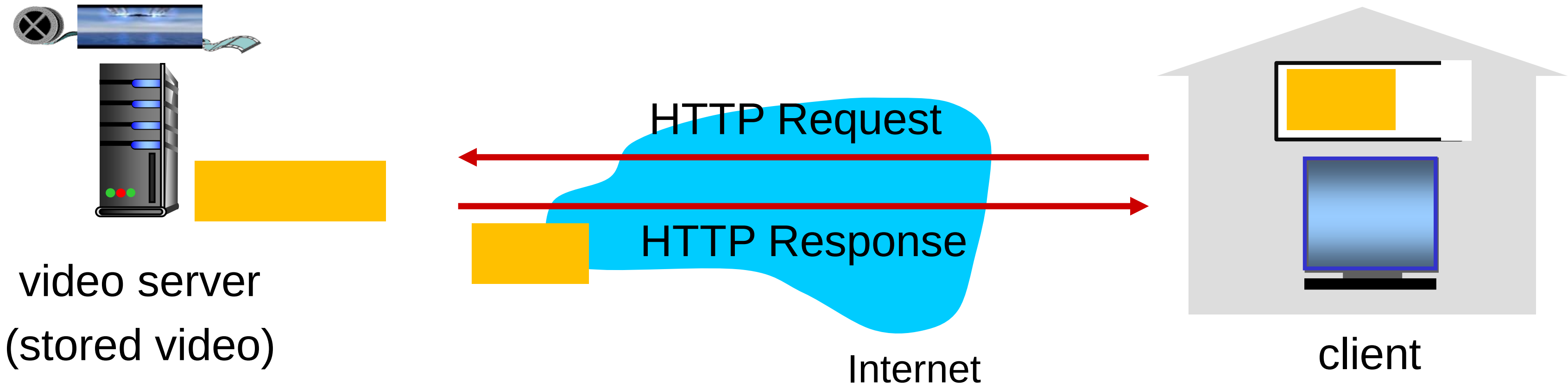
temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

Multimedia: video

Type	Video Bitrate, Standard Frame Rate (24, 25, 30)	Video Bitrate, High Frame Rate (48, 50, 60)
2160p (4k)	35-45 Mbps	53-68 Mbps
1440p (2k)	16 Mbps	24 Mbps
1080p	8 Mbps	12 Mbps
720p	5 Mbps	7.5 Mbps
480p	2.5 Mbps	4 Mbps
360p	1 Mbps	1.5 Mbps

- CBR: (constant bit rate): video encoding rate fixed
- VBR: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes

HTTP Streaming



All clients receive the same encoding of the video:

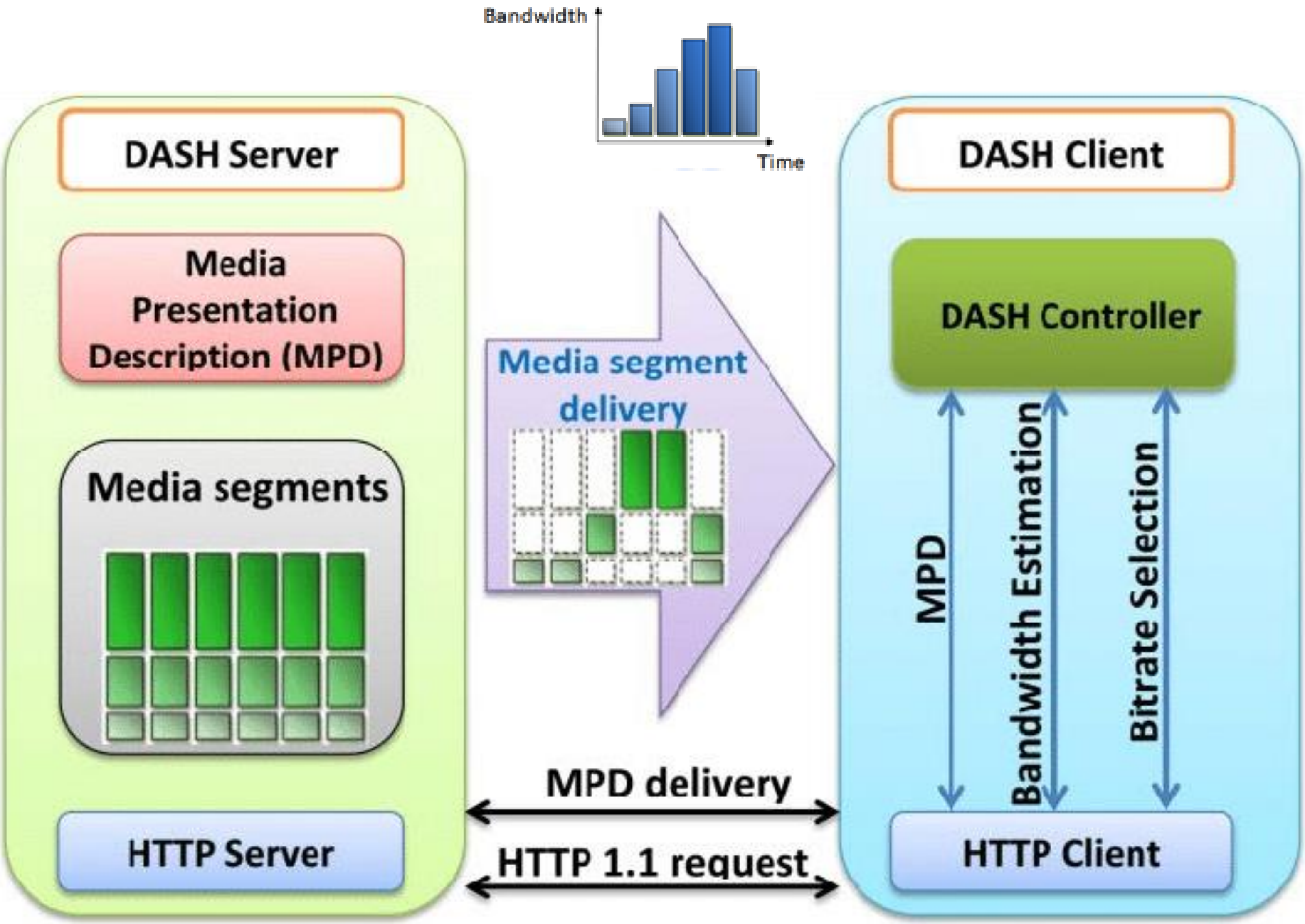
- Human users may have different requirements
- Clients may have different available bandwidth, which may be time-varying

How to deal with this?

Streaming multimedia: DASH

- DASH: Dynamic, Adaptive Streaming over HTTP
- Server:
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - manifest file: provides URLs for different chunks encoded at different rates
- 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

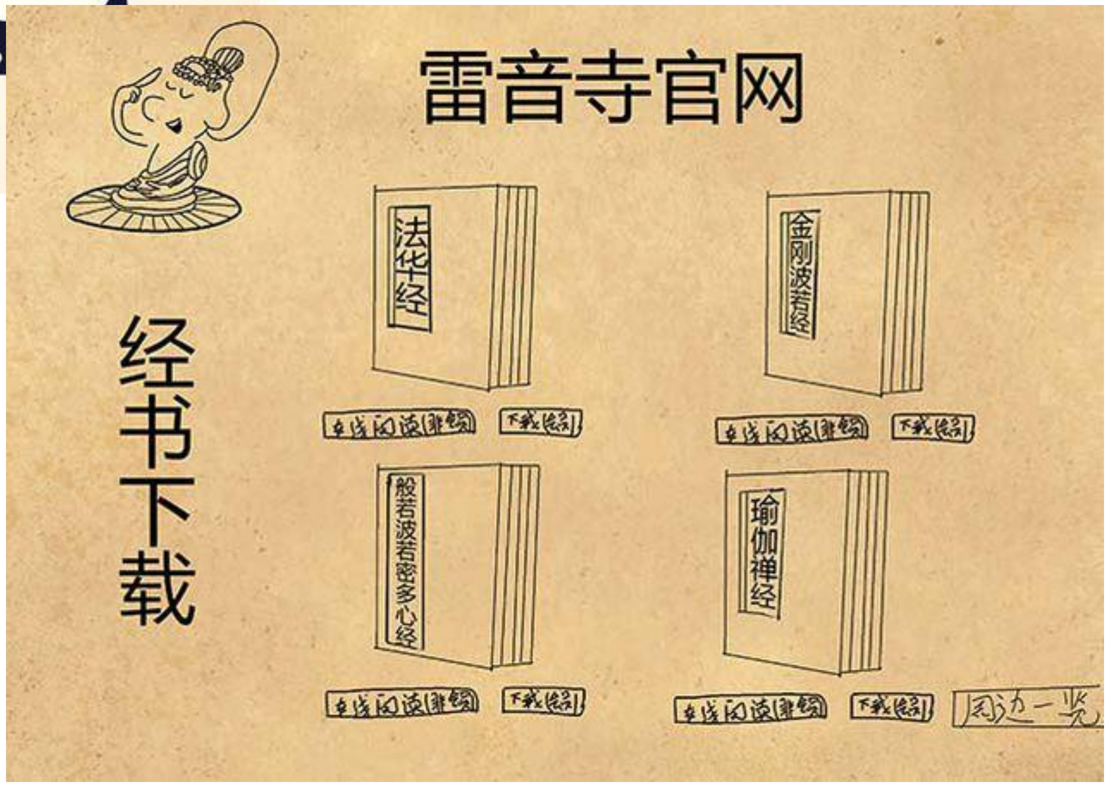


Streaming multimedia: DASH

"intelligence" at client: client determines

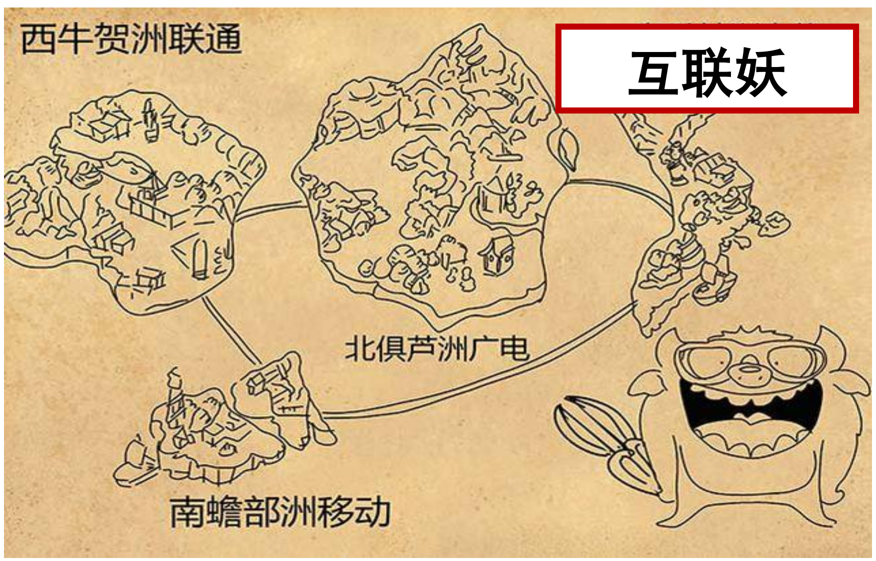
- when to request chunk (so that buffer starvation, or overflow does not occur)
- what encoding rate to request (higher quality when more bandwidth available)
- where to request chunk (can request from URL server that is "close" to client or has high available bandwidth)

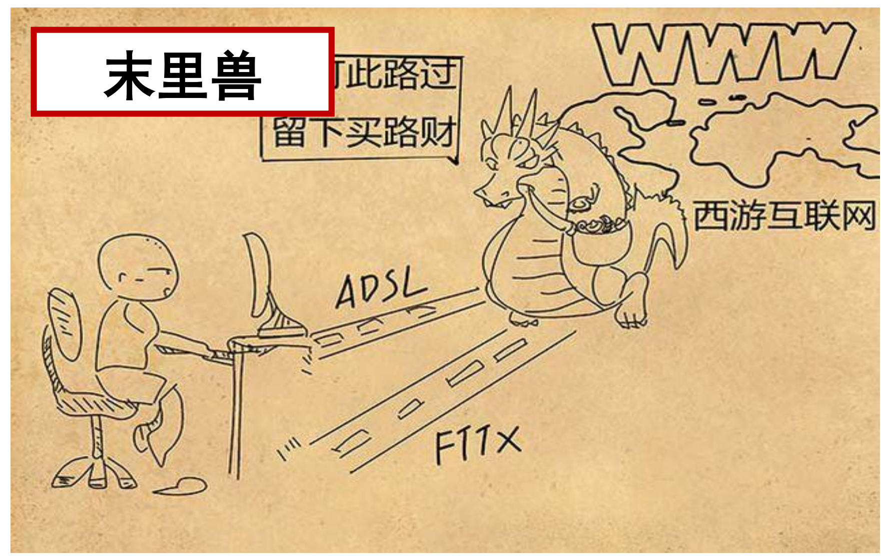
Content distribution networks



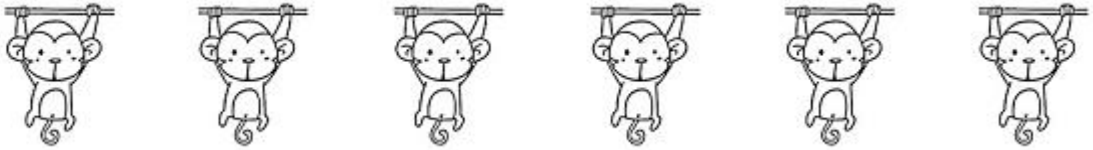
来源：小黑羊JoinWings
https://baijia.baidu.com/s?old_id=126615&wfr=pc&fr=app_list

Content distribution networks





Content distribution networks



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
  - huge traffic
  - long path to distant clients
  - multiple copies of video sent over outgoing link

….quite simply: this solution doesn't scale

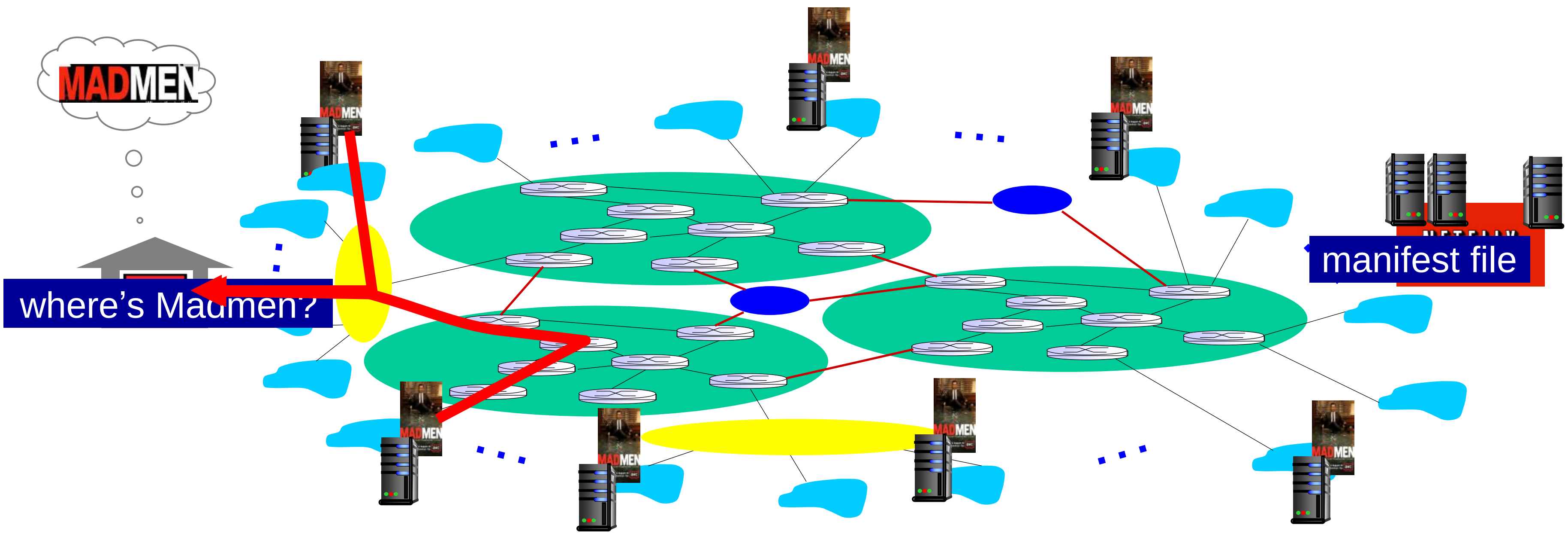
Content distribution networks

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

- Option 2: Content distribution networks (CDN) store/serve multiple copies of videos at multiple geographically distributed sites
  - Enter deep: push CDN servers deep into many access networks; inside ISPs
    - close to users
    - used by Akamai, 1700 locations
  - Bring home: smaller number (10's) of larger clusters in Internet Exchange Point (IXP); outside ISPs
    - 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



Content Distribution Networks (CDNs)

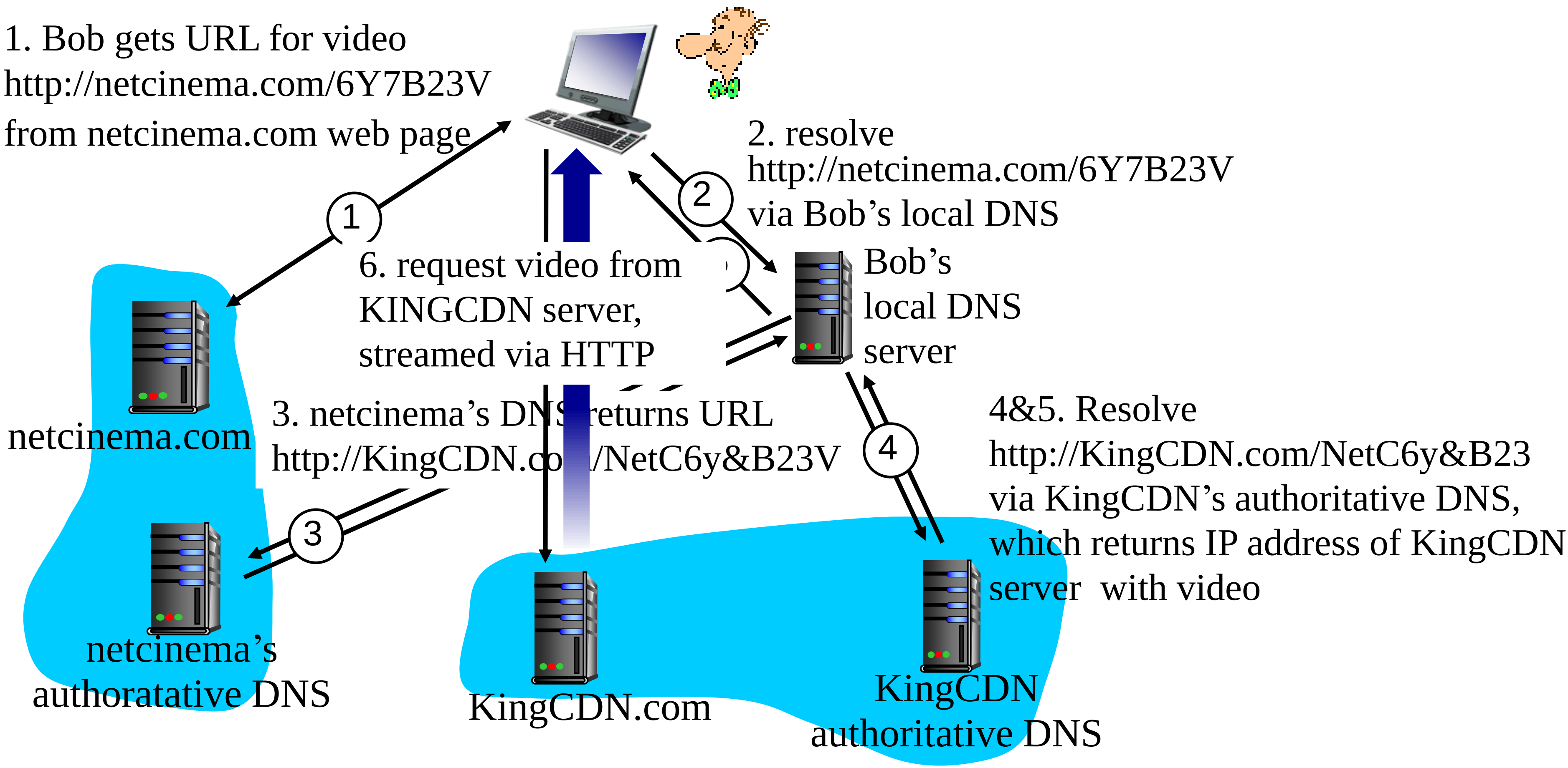
Challenges: Coping with a congested Internet

- what content to place in CDN node?
  - Simple pull strategy: request, then store
- from which CDN node to retrieve content?
  - Cluster selection strategy
- the operation for retrieving content?
  - CDN operation

CDN Operation

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

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



CDN: Cluster Selection Strategy

One simple strategy is to assign the client to the cluster that is **geographically closest**:

- When a DNS request is received from a particular LDNS, the CDN chooses the geographically closest cluster
- may not be the closest cluster in terms of the length or number of hops
- ignore the variation in delay and available bandwidth over time

Periodic **real-time measurements** of delay and loss performance between their clusters and clients:

- a CDN can have each of its clusters periodically send probes to all of the LDNSs around the world.
- many LDNSs are configured to not respond to such probes.

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

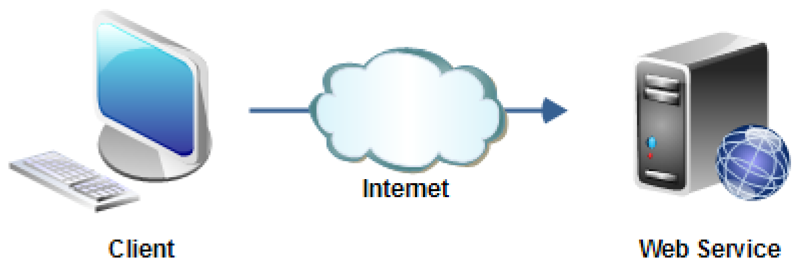
2.4 DNS

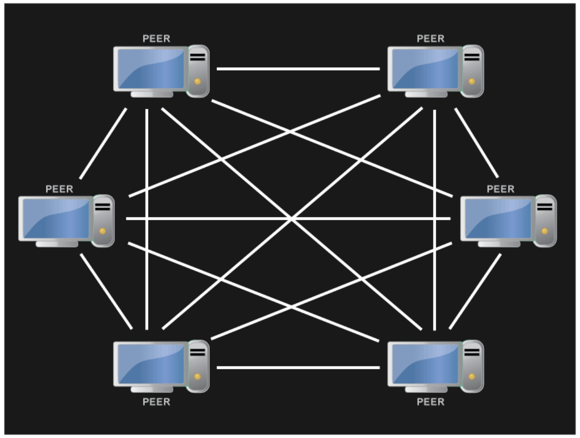
2.5 P2P applications

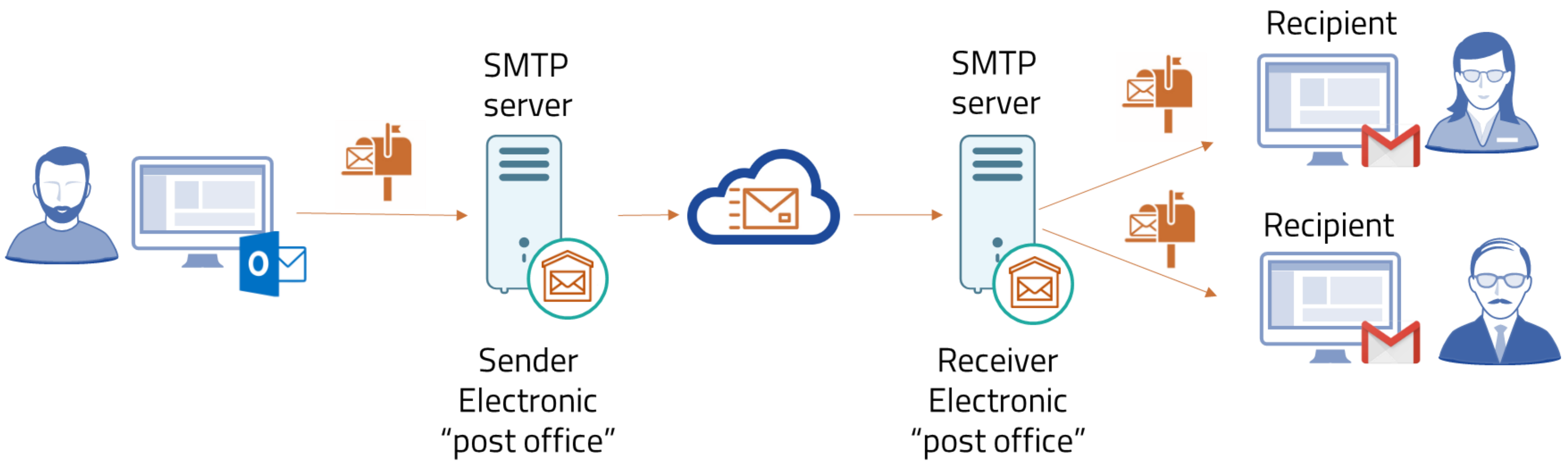
2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

Network Applications







A client program and a server program

- Client process and server process

Create Network Applications

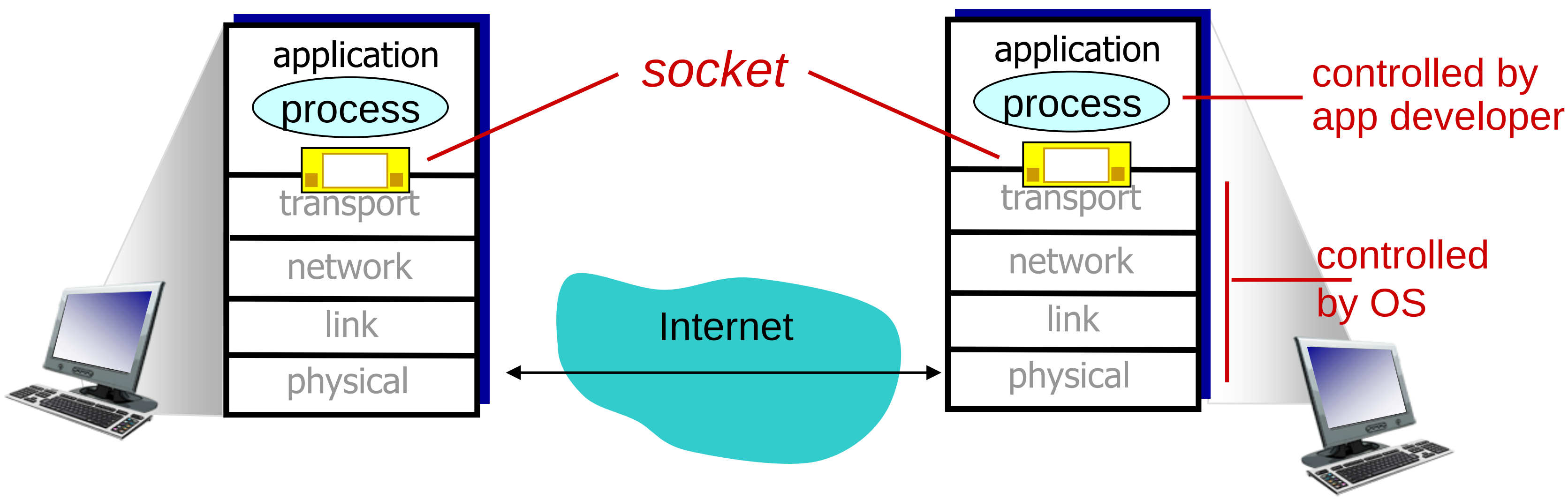
- Network application whose operation is specified in a protocol standard, e.g., RFC
  - Open source; fully follows the rules of the RFC
  - Client and server programs can be developed by different companies
  - Use the well-known port number associated the protocols

- Proprietary network application
  - Not been openly published
  - Both client and server programs should be developed by one company
  - Avoid using well-known port numbers

Socket programming

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

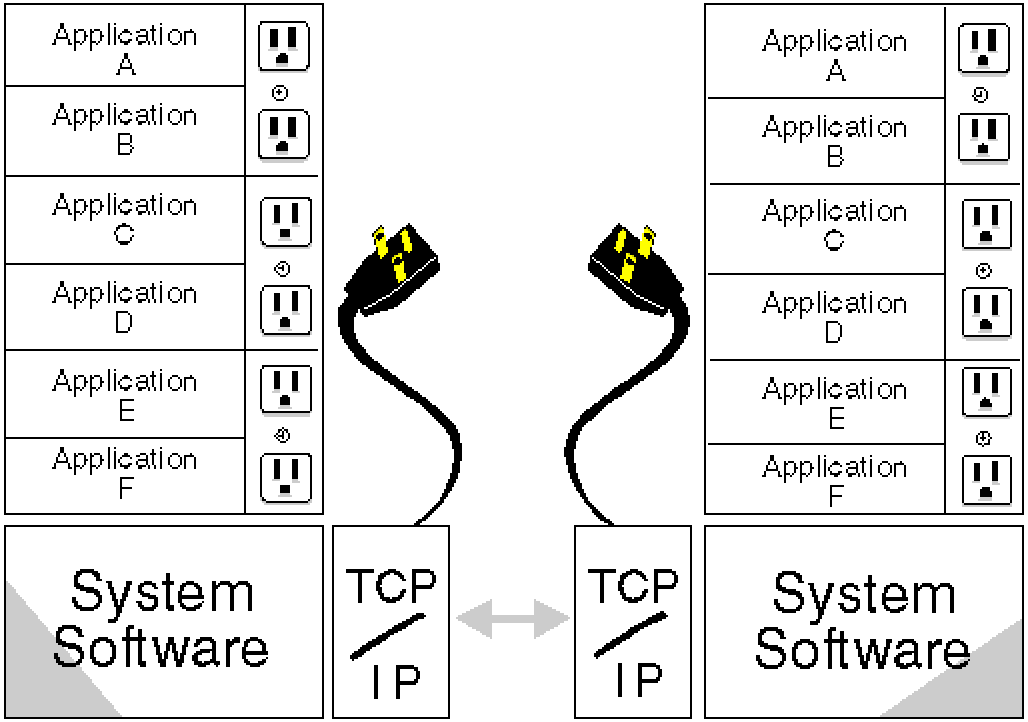
Socket: door between application process and end-end-transport protocol



Socket

A socket is one endpoint of a two-way communication link between two programs running on the network.

- A socket is bound to a port number so that the transport layer can identify the application that data is destined to be sent to.



Socket programming

Socket programming: how we can use socket API for creating communication between client and server processes.

Two socket types for two transport services:

- UDP: unreliable datagram
- TCP: reliable, connection-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 number to each packet
- receiver extracts sender IP address and port number 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

server (running on serverIP)

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

read UPD datagram from
serverSocket

write reply to serverSocket
specifying client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Segment: the transport-layer packet for TCP
Datagram: the packet for UDP

Example app: UDP client

Python UDPClient

include Python's socket library

```
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()
```

either the IP address (e.g., "128.138.32.126") or the hostname (e.g., "cis.poly.edu")

IPv4

UDP socket

Create the client's socket

UDP socket is identified by destination IP address and port number

get user keyboard input

Attach server name, port to message; send into socket

IP + portnumber

read reply characters from socket into string

print out received string and close socket

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)
```

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
- Client TCP establishes connection to server TCP

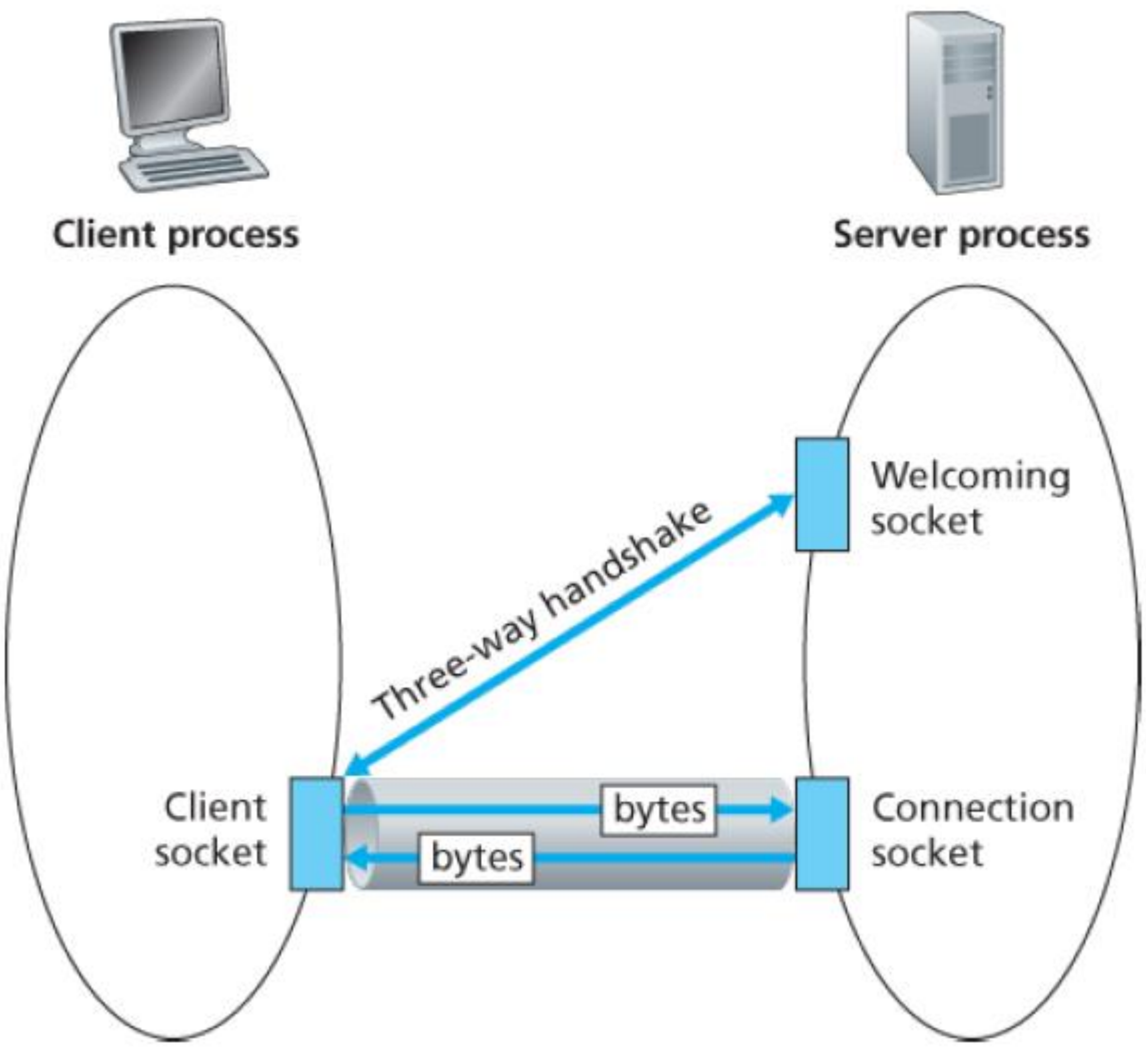
- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

TCP socket is identified by (destination IP address, destination port number, source IP address, source port number)

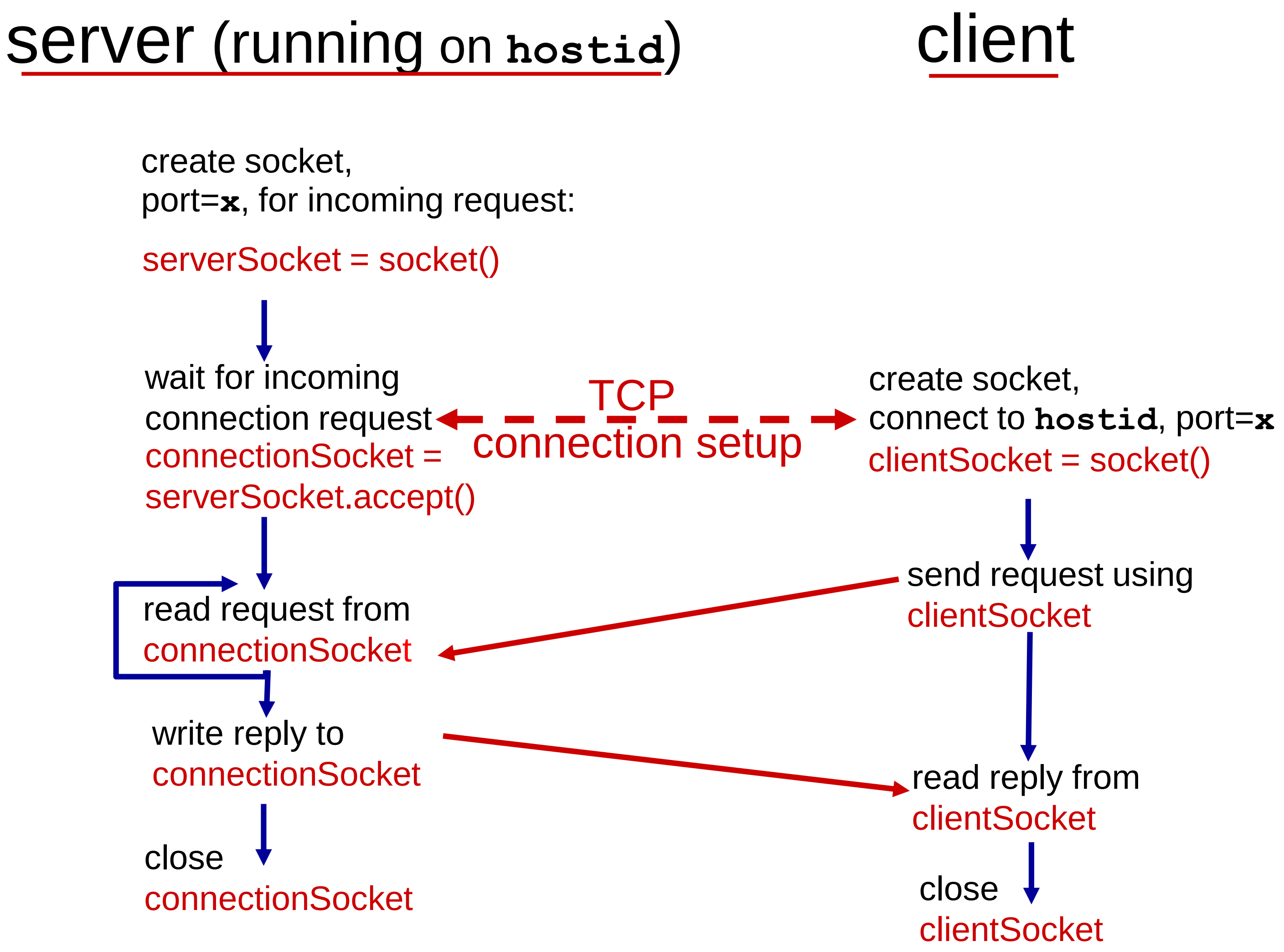
Application viewpoint:

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

Socket programming with TCP



Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

create TCP socket for server

No need to attach server name, port

TCP socket

```
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()
```

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()
```

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 being communicated

important themes:

- control vs. messages
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable message transfer