

COMP 6461

Computer Networks & Protocols



Lecture 4b

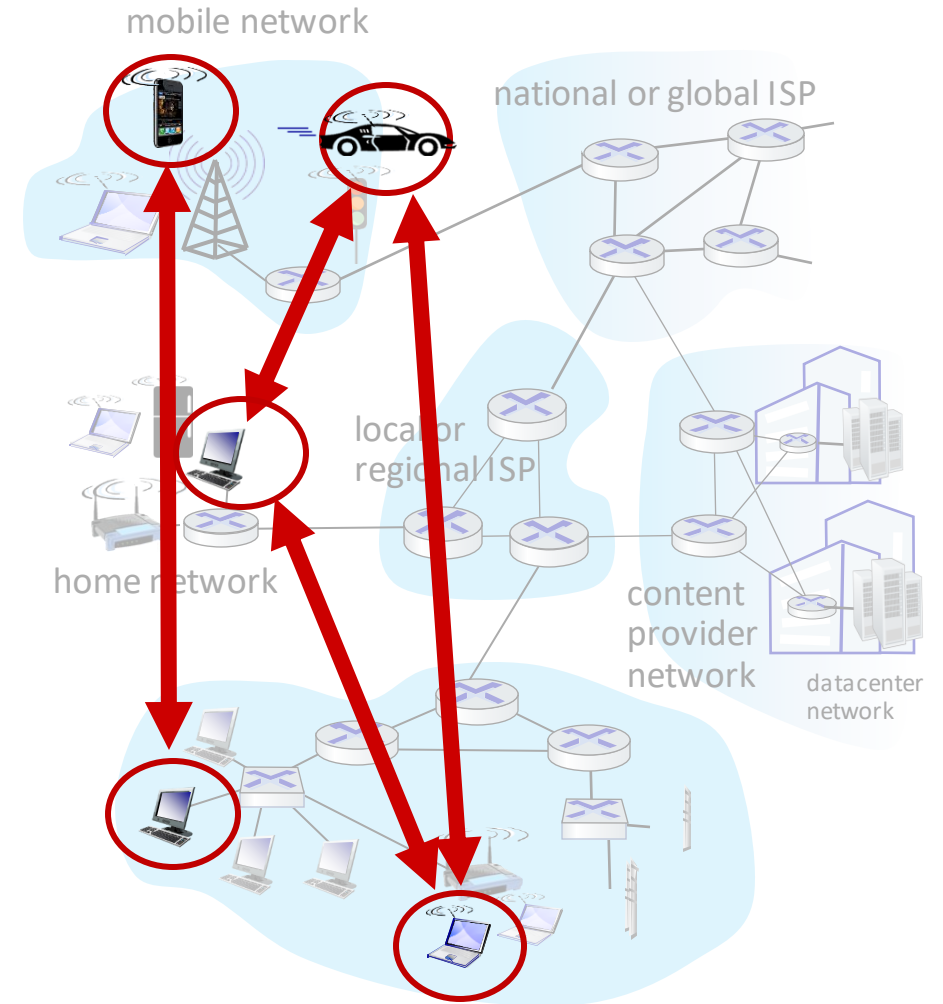
Application Layer (Part 4)

Application Layer: Overview

- Principles of network applications
- video streaming and content distribution networks
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- socket programming with UDP and TCP

Peer-to-peer (P2P) architecture

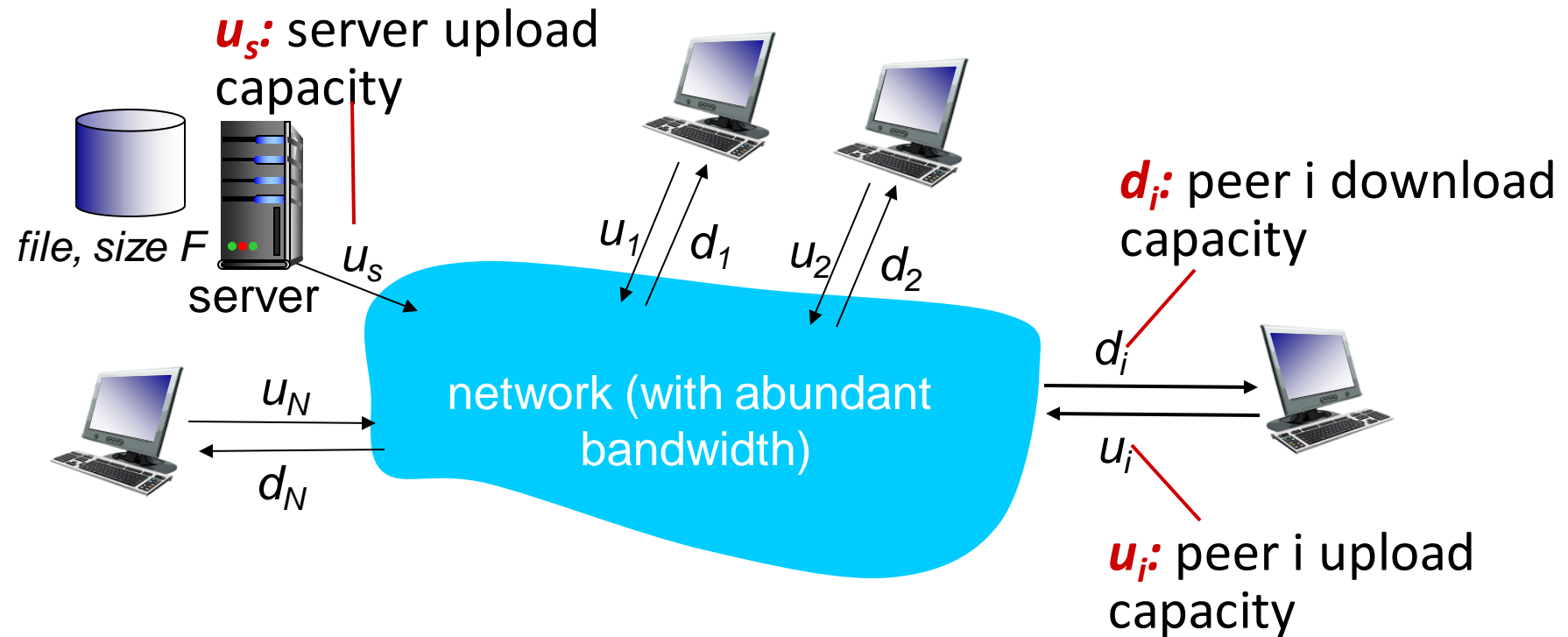
- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
 - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
 -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



File distribution: client-server vs P2P

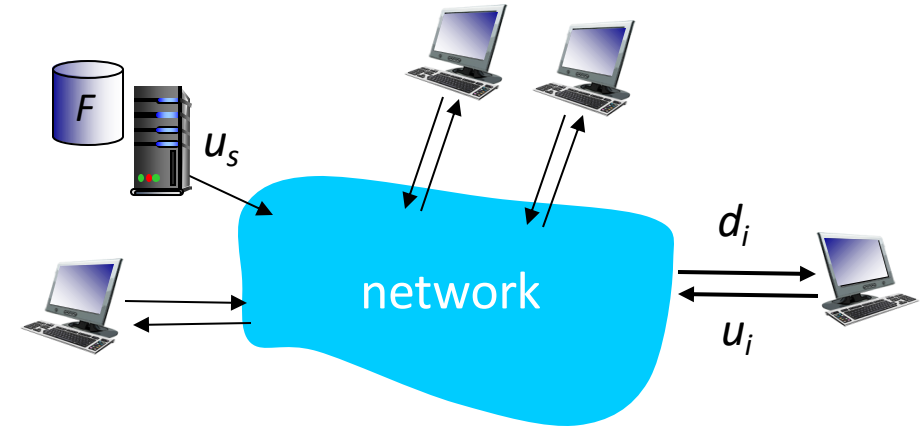
Q: how much time to distribute file (size F) from one server to N peers?

- peer upload/download capacity is limited resource



File distribution time: client-server

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



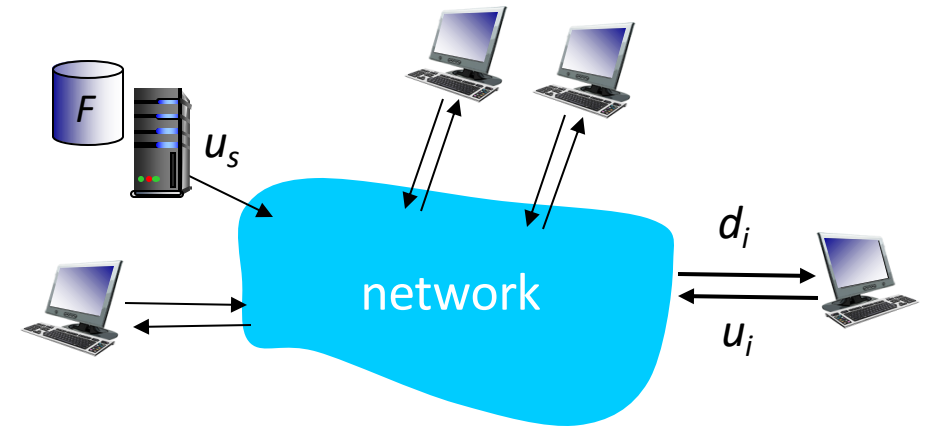
*time to distribute F
to N clients using
client-server approach*

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

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
 - Max client download time: F/d_{min}
- **clients:** as aggregate must download NF bits
 - max upload rate (limiting max download rate) is $u_s + \sum u_i$



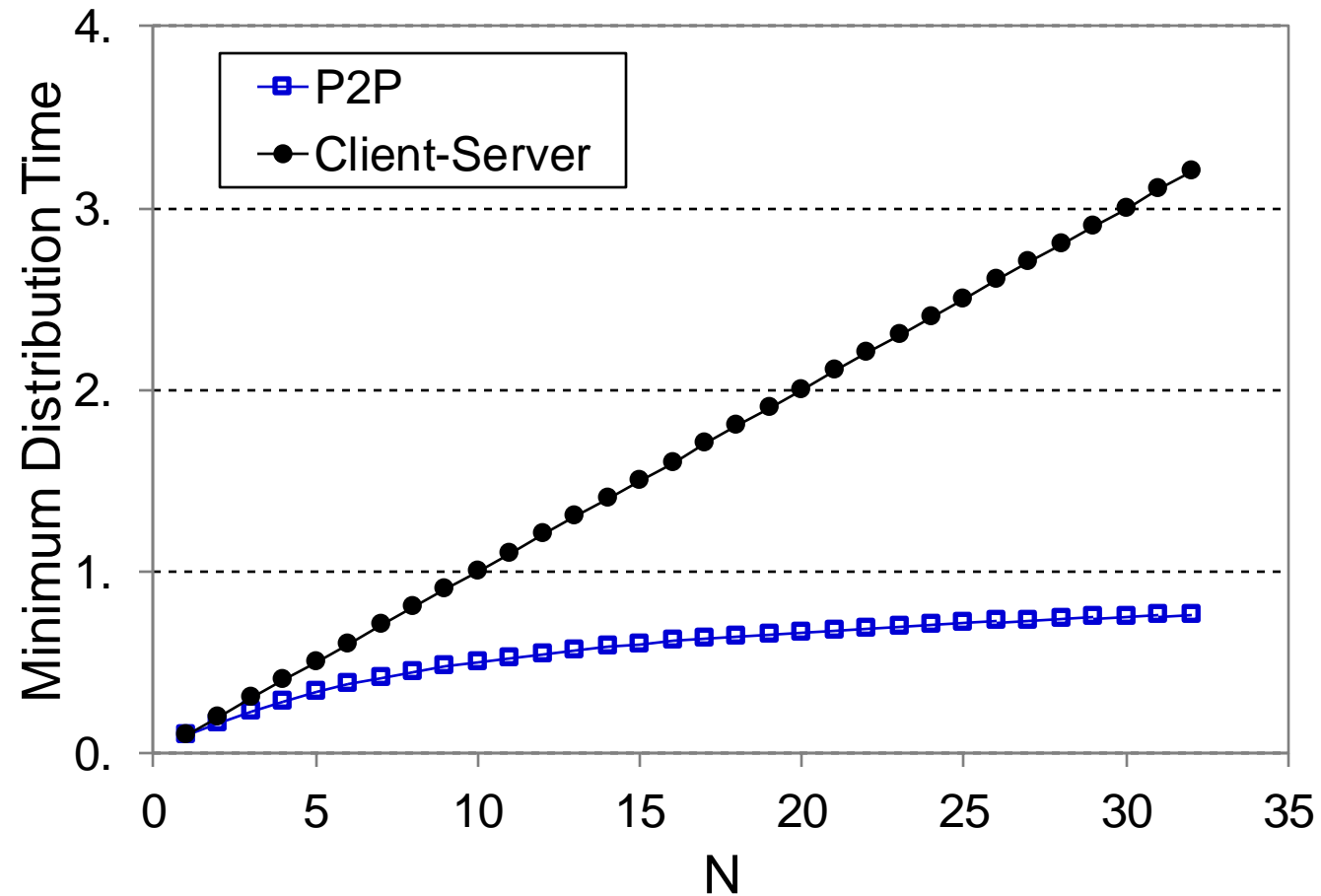
time to distribute F
to N clients using
P2P approach

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

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, $u_s = 10u$, $d_{min} \geq u_s$

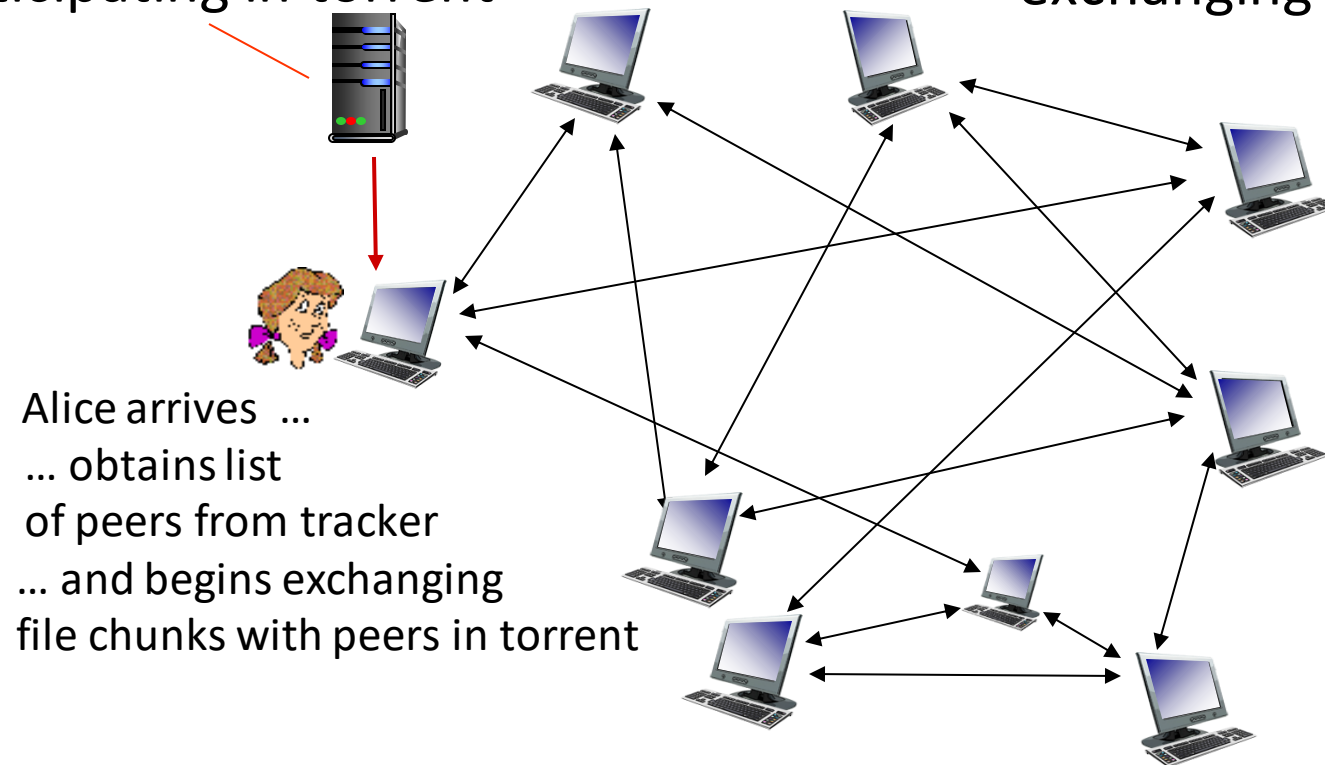


P2P file distribution: BitTorrent

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

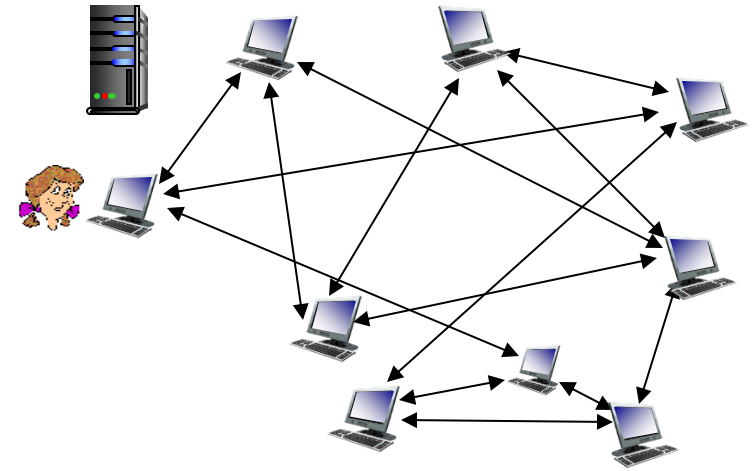
tracker: tracks peers
participating in torrent

torrent: group of peers
exchanging chunks of a file



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:

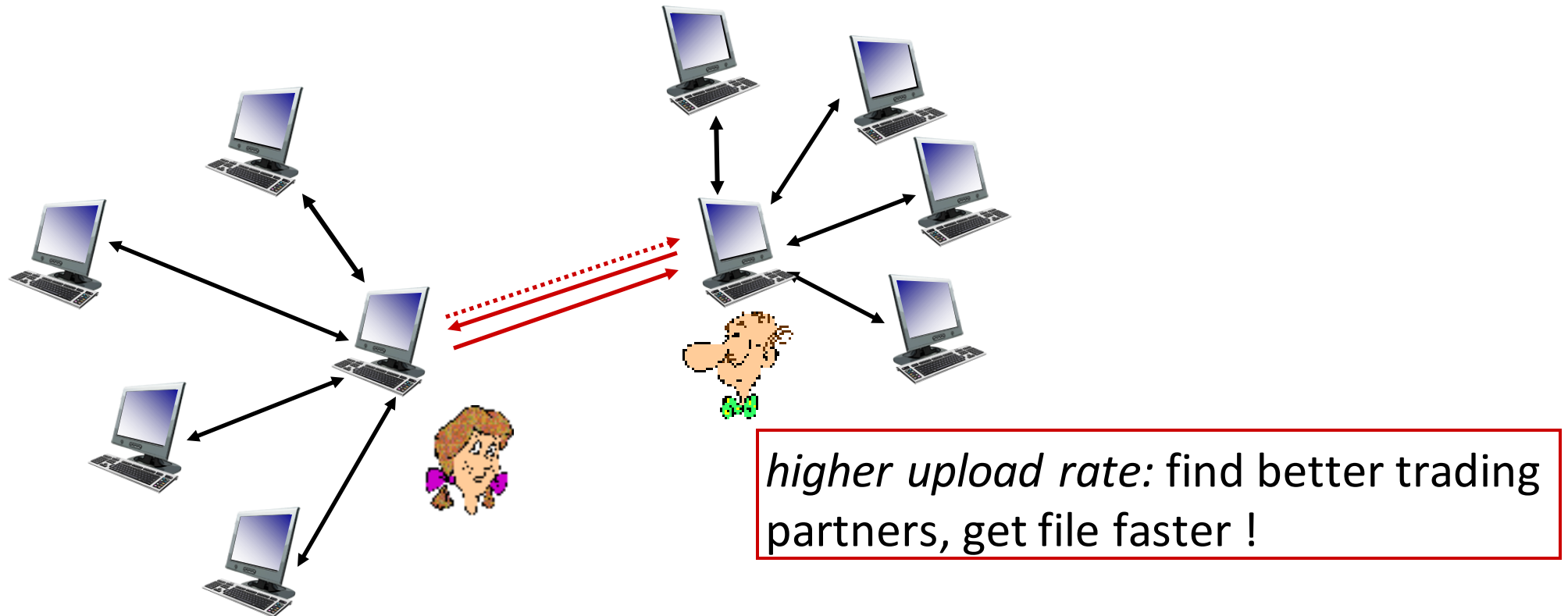
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

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



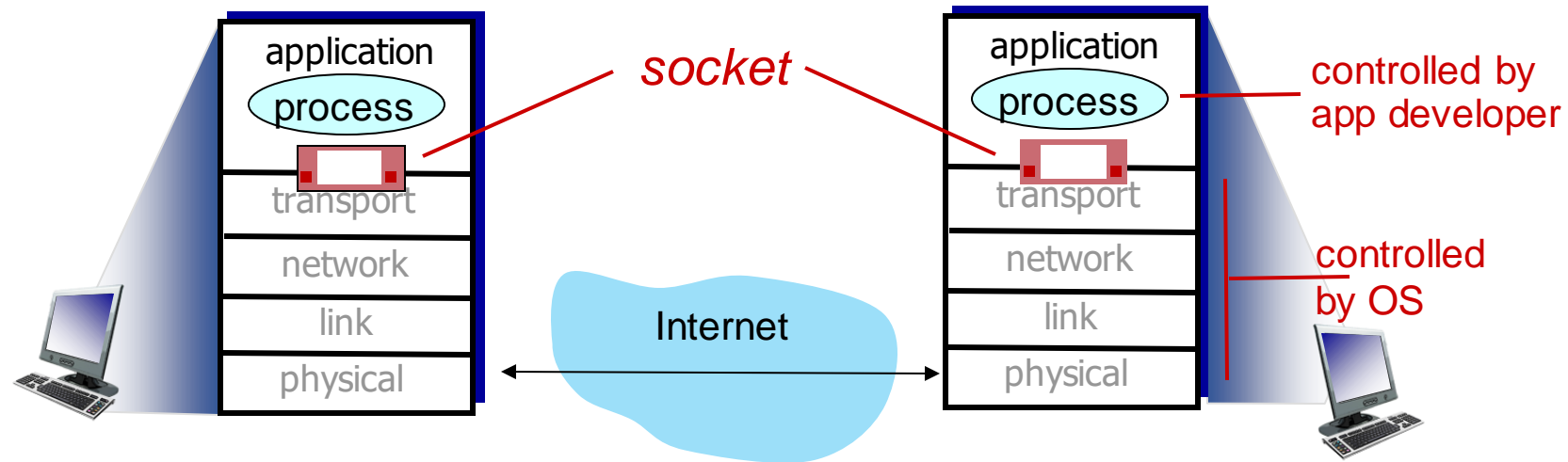
Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- **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



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 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 processes

Client/server socket interaction: UDP



server (running on serverIP)

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 for server	→	clientSocket = socket(AF_INET, SOCK_DGRAM)
get user keyboard input	→	message = raw_input('Input lowercase sentence:')
attach server name, port to message; send into socket	→	clientSocket.sendto(message.encode(), (serverName, serverPort))
read reply characters from socket into string	→	modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print out received string and close socket	→	print modifiedMessage.decode() clientSocket.close()

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
loop forever → while True:
    Read from UDP socket into message, getting → message, clientAddress = serverSocket.recvfrom(2048)
    client's address (client IP and port)
    send upper case string back to this client → 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
- *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 processes

Client/server socket interaction: TCP



server (running on `hostid`)

client



create socket,
port=`x`, for incoming
request:
`serverSocket = socket()`

wait for incoming
connection request
`connectionSocket =`
`serverSocket.accept()`

read request from
`connectionSocket`

write reply to
`connectionSocket`

close
`connectionSocket`

TCP
connection setup

create socket,
connect to `hostid`, port=`x`
`clientSocket = socket()`

send request using
`clientSocket`

read reply from
`clientSocket`

close
`clientSocket`

Example app: TCP client

Python TCPClient

create TCP socket for server,
remote port 12000

```
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()
```

No need to attach server name, port

Example app: TCP server

Python TCPServer

	from socket import *
	serverPort = 12000
create TCP welcoming socket →	serverSocket = socket(AF_INET, SOCK_STREAM)
	serverSocket.bind(('', serverPort))
server begins listening for incoming TCP requests →	serverSocket.listen(1)
	print 'The server is ready to receive'
loop forever →	while True:
server waits on accept() for incoming requests, new socket created on return →	connectionSocket, addr = serverSocket.accept()
read bytes from socket (but not address as in UDP) →	sentence = connectionSocket.recv(1024).decode()
	capitalizedSentence = sentence.upper()
	connectionSocket.send(capitalizedSentence.encode())
close connection to this client (but <i>not</i> welcoming socket) →	connectionSocket.close()

Chapter 2: Summary

our study of network application layer is now complete!

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