# Networks Lecture 5

Paolo Ciancarini

Innopolis University

January 31, 2022

# Source of the material

- This lecture is based on the following resources
  - Chapter 2 of Computer Networking: A Top Down Approach by J Kurose and K Ross
  - https://www.diffen.com/difference/TCP_vs_UDP
  - The material is aligned and add/deleted according to the need of the students.

# Topic of the lecture

- P2P applications
- Socket Programming with UDP and TCP
- Transport-layer services
- Connectionless transport: UDP
- Principles of reliable data transfer with TCP

# Topic of the tutorial

- Email Protocol Suit
- TCP Socket Programming

# Topic of the lab

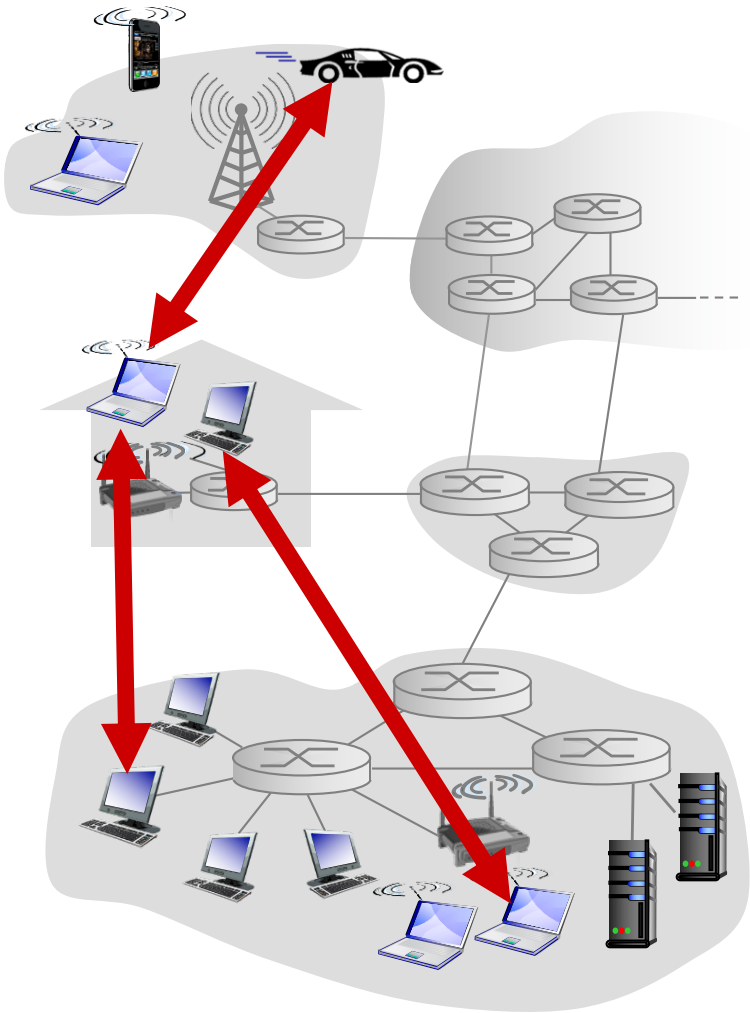
- TCP Socket Programming

# P2P applications

- The applications described so far—including the Web, e-mail, and DNS—all employ client-server architectures with significant reliance on always-on infrastructure servers.
- With a P2P architecture, there is minimal (or no) reliance on always-on infrastructure servers.
- Instead, pairs of intermittently connected hosts, called peers, communicate directly with each other and cooperate to offer some service.
- The peers are not owned by a service provider, but are instead devices controlled by users

# Pure P2P application architecture

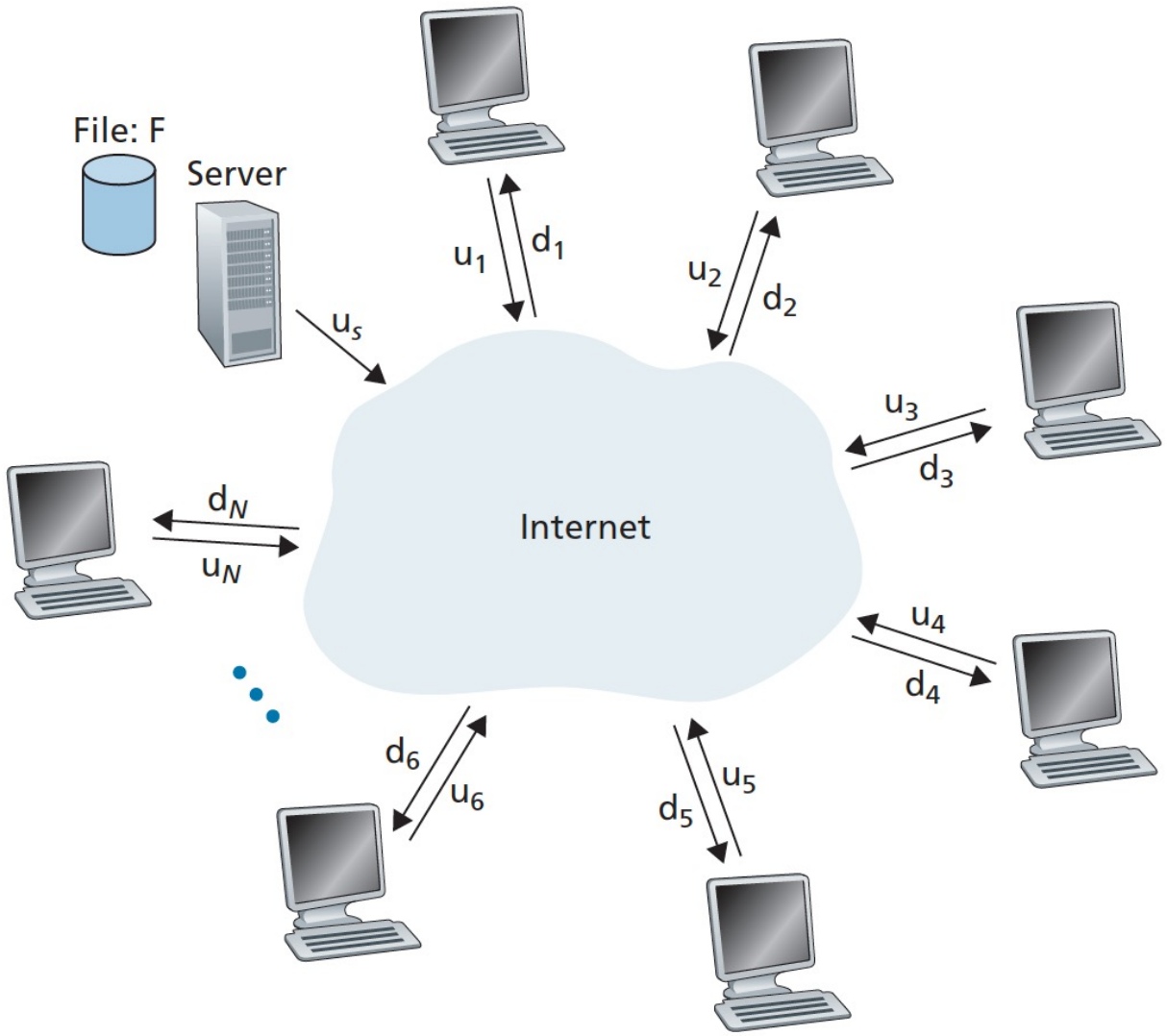
- *No* always-on server
- Arbitrary end systems directly communicate
- Peers are intermittently connected and change IP addresses



*Examples:*

- File distribution (BitTorrent)
- Streaming (KanKan – video on demand)
- VoIP (Skype)

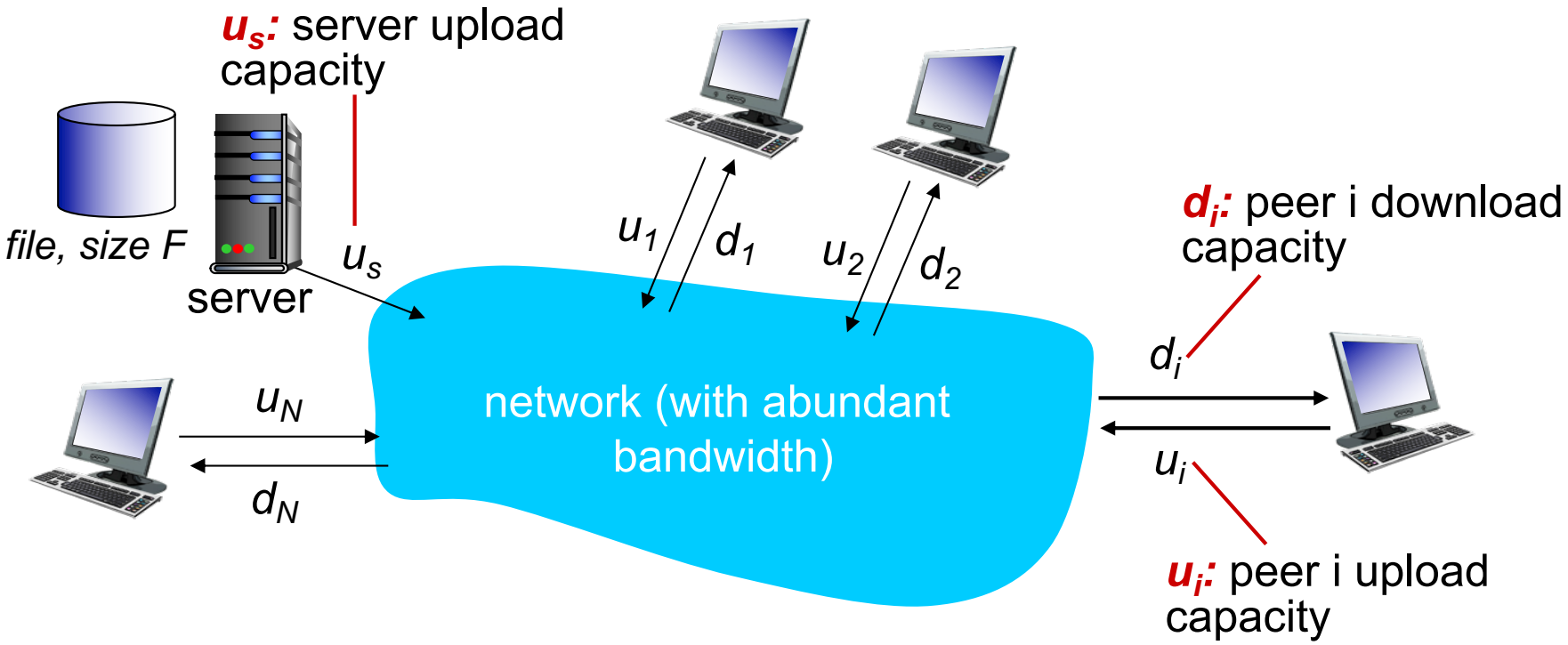
# A file distribution problem



# File Distribution: client-server vs P2P

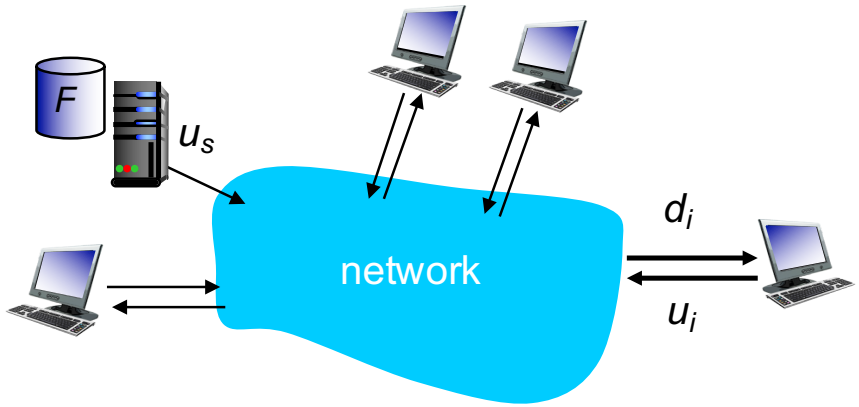
*Question:* how much time to distribute file (size *F*) from one server to *N peers*?

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

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

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

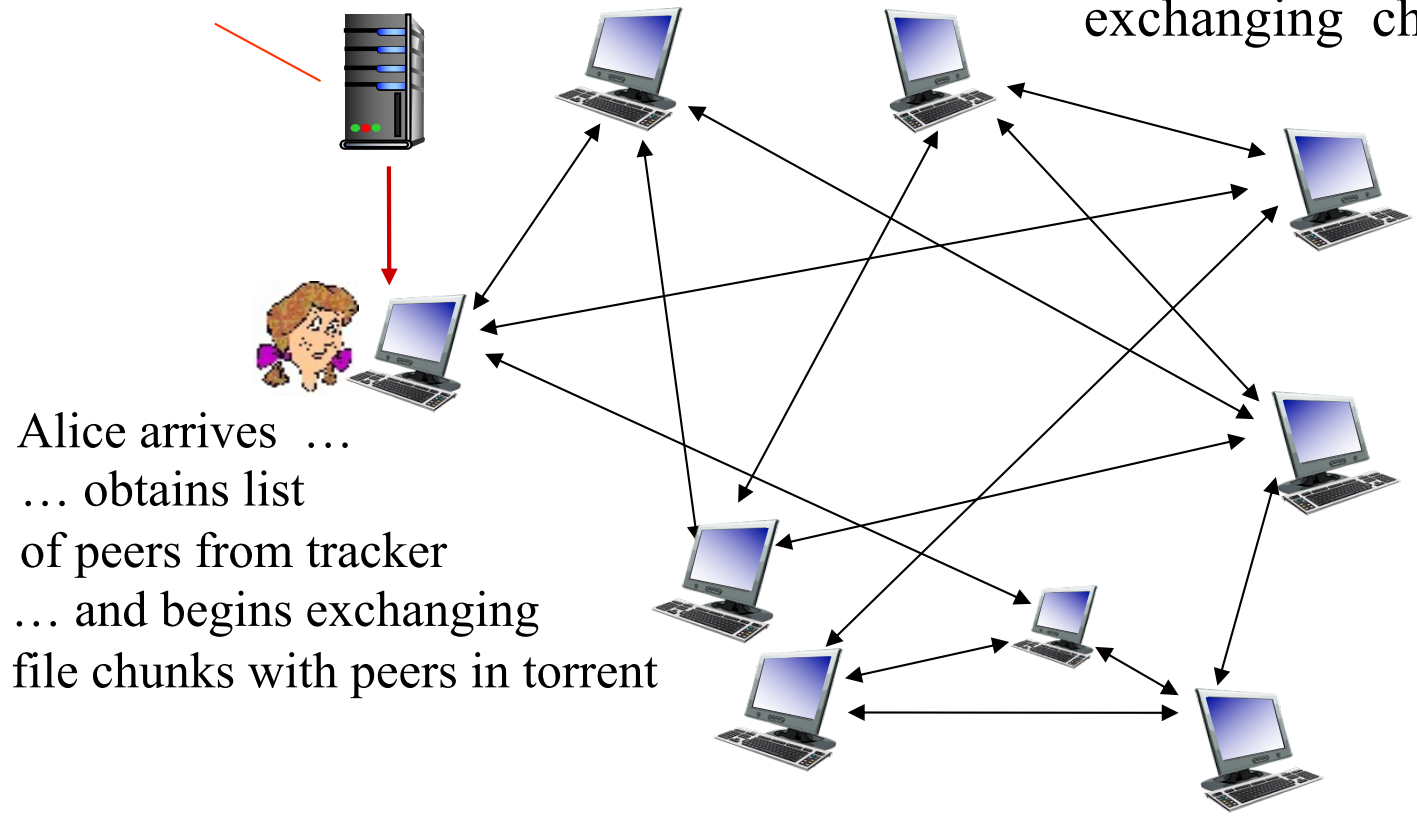
increases linearly in N

# P2P File Distribution: BitTorrent

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

*tracker:* tracks peers participating in torrent

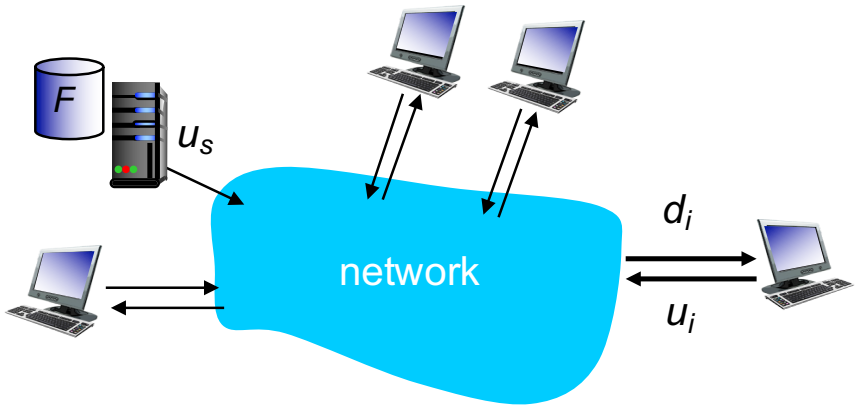
*torrent:* group of peers exchanging chunks of a file



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

# File Distribution Time: P2P

- *Server transmission:* must upload at least one copy
  - time to send one copy: $F/u_s$
- *Clients:* as aggregate must download $NF$ bits
  - max upload rate (limting max download rate) is $u_s + Su_i$




*time to distribute F to N clients using P2P approach*

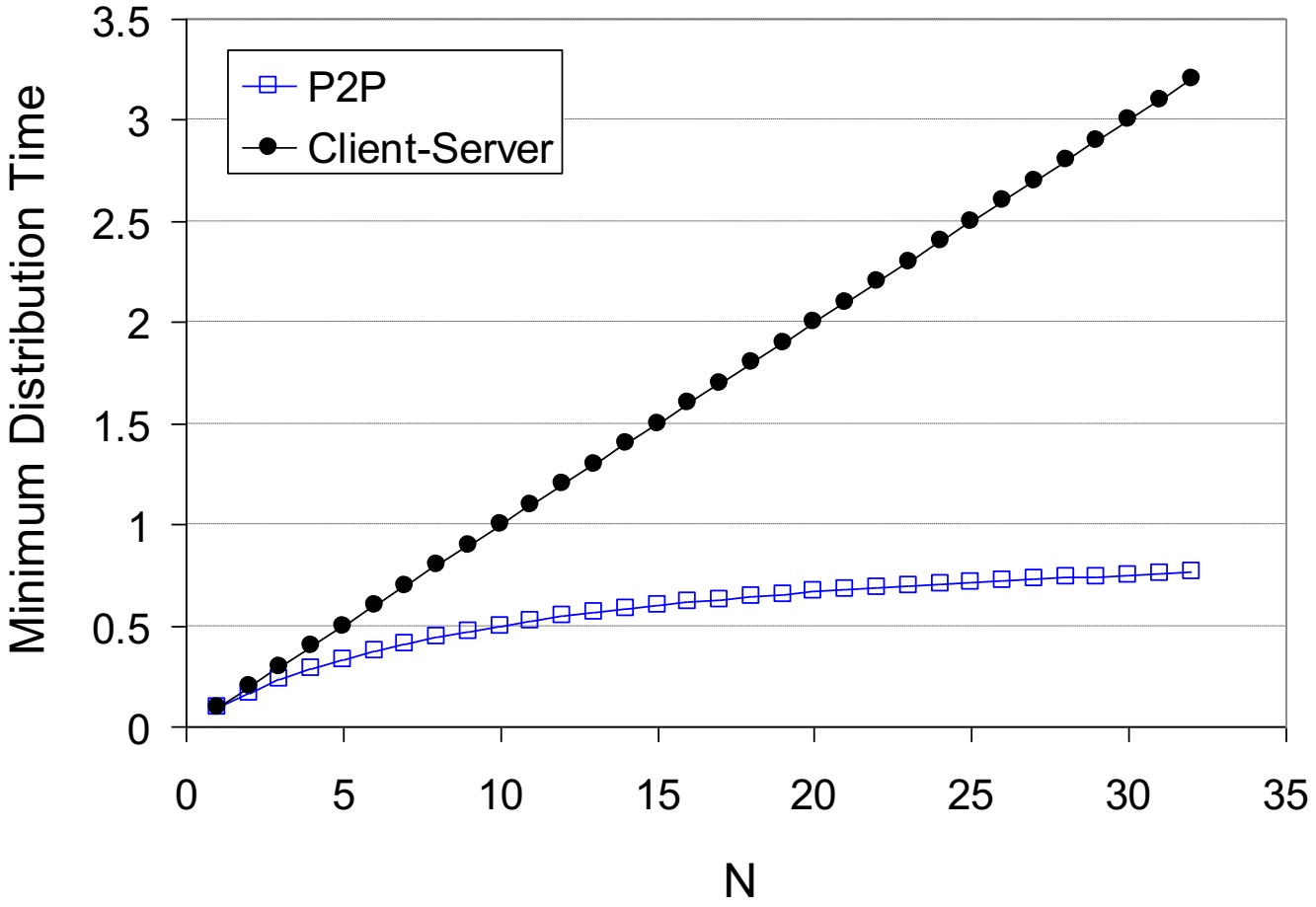
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

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



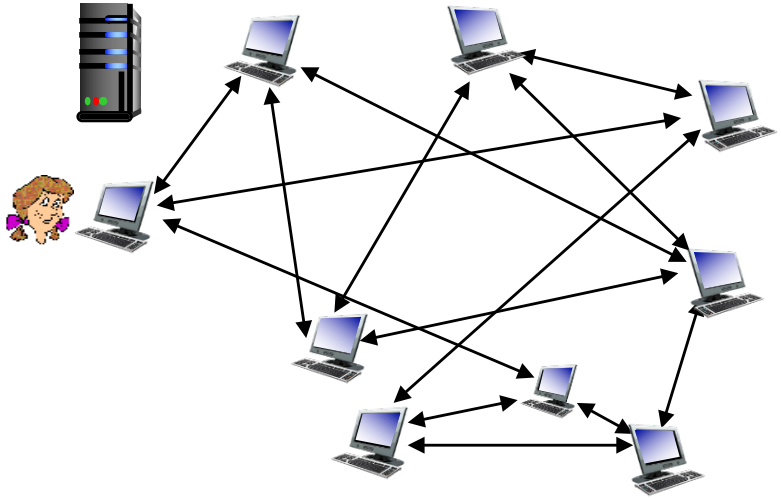

We assume that all peers have the same upload rate u.
a peer can transmit the entire file in one hour, the server transmission rate is 10 times the peer upload rate.
For simplicity the peer download rates are set large so as not to have an effect.

We see that for the client-server architecture, the distribution time increases linearly and without bound as the number of clients increases

We see that P2P is autoscaling

# 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*

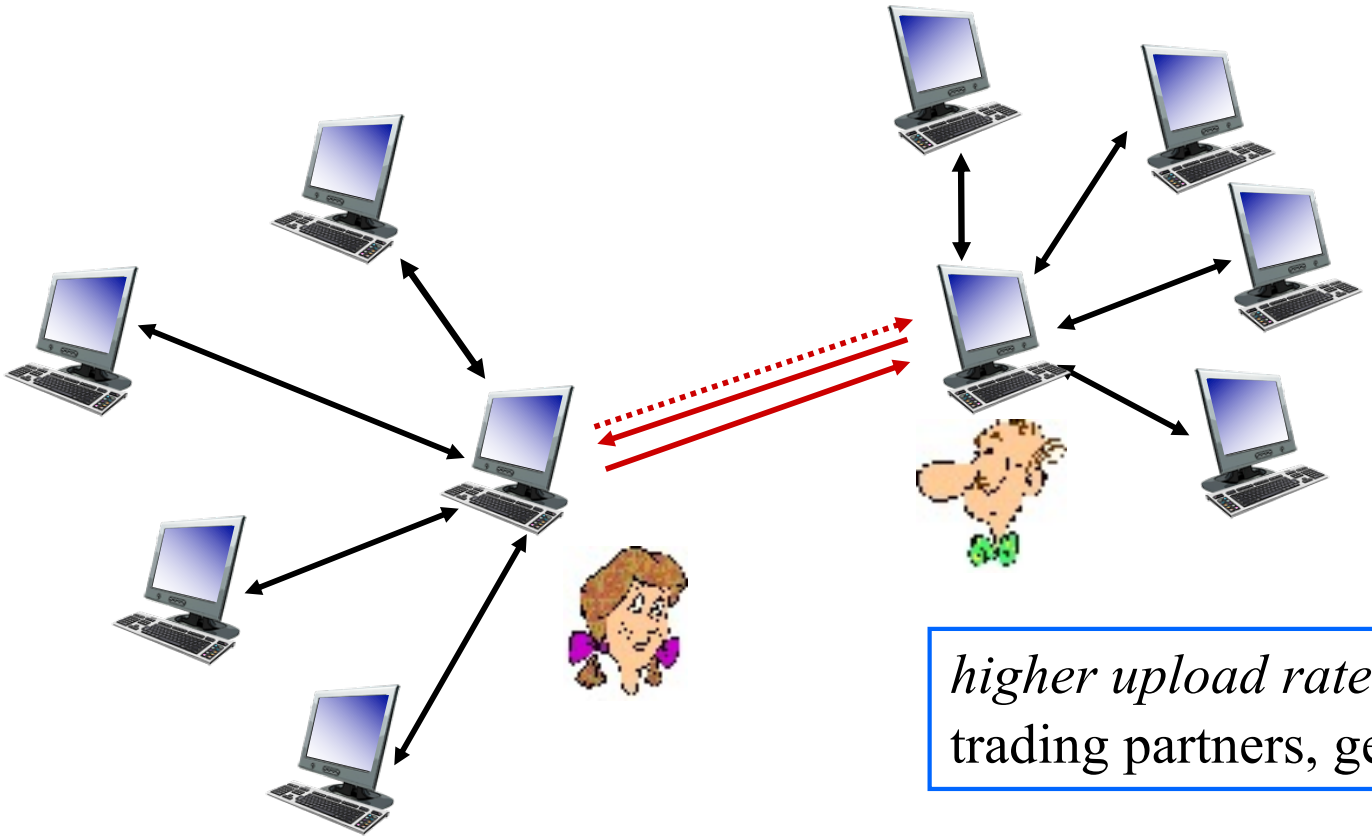
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
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

# Distributed Hash Table (DHT)

- Hash table
- DHT paradigm
- Circular DHT and overlay networks
- Peer churn

https://wps.pearsoned.com/ecs_kurose_compnetw_6/230/58924/15084652.cw/content/index.html

# A simple Database

Simple databases with(key, value) pairs:

1) key: *human name*; value: social security #

| Key | Value |
|---|---|
| John Washington | 132-54-3570 |
| Diana Louise Jones | 761-55-3791 |
| Xiaoming Liu | 385-41-0902 |
| Rakesh Gopal | 441-89-1956 |
| Linda Cohen | 217-66-5609 |
| ……. | …… |
| Lisa Kobayashi | 177-23-0199 |

*2)* key*: movie title; value: IP address*

# Hash Table

- More convenient to store and search on numerical representation of key
- key = hash(original key)

| Original Key | Key | Value |
|---|---|---|
| John Washington | 8962458 | 132-54-3570 |
| Diana Louise Jones | 7800356 | 761-55-3791 |
| Xiaoming Liu | 1567109 | 385-41-0902 |
| Rakesh Gopal | 2360012 | 441-89-1956 |
| Linda Cohen | 5430938 | 217-66-5609 |
| ……. | | …… |
| Lisa Kobayashi | 9290124 | 177-23-0199 |

# Distributed Hash Table (DHT)

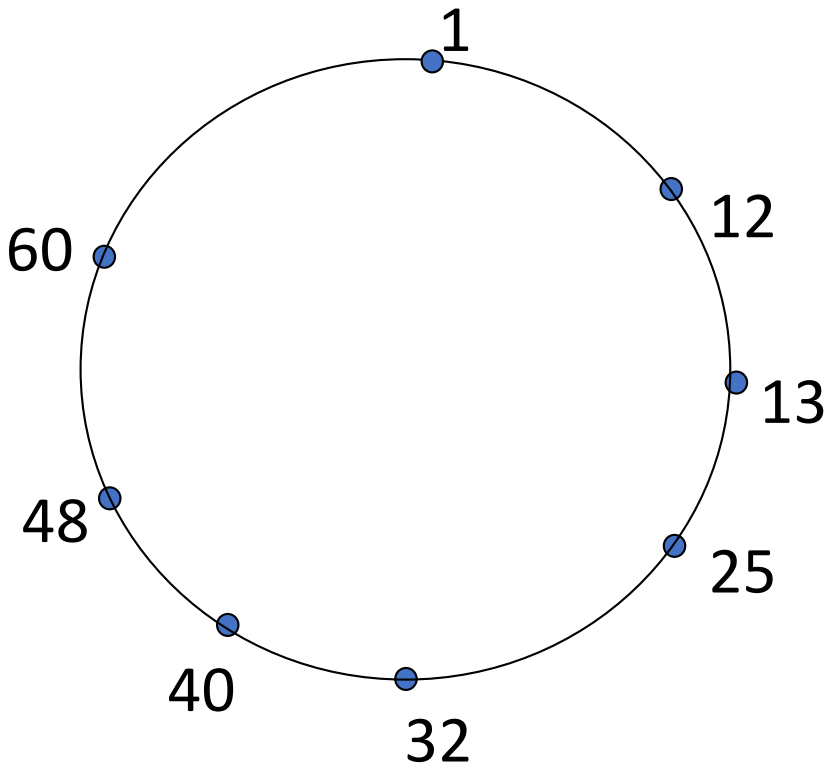
- Distribute (key, value) pairs over millions of peers
  - pairs are evenly distributed over peers
- Any peer can query database with a key
  - database returns value for the key
  - To resolve query, small number of messages exchanged among peers
- Each peer only knows a small number of other peers
- Robust to peers coming and going (churn)

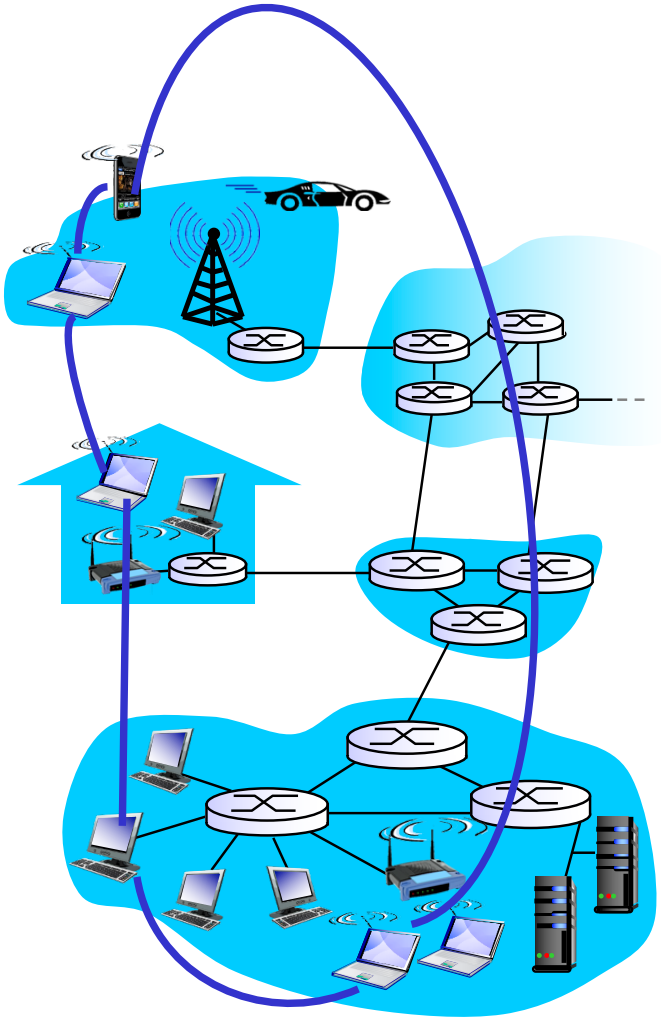
# Assign key-value pairs to peers

- Rule: assign key-value pair to the peer that has the *closest* ID.
- Convention: closest is the *immediate successor* of the key.
  - For Example: ID is related to space {0,1,2,3,…,63}. It can be udpated.
- Suppose 8 peers: 1,12,13,25,32,40,48,60
  - If key = 51, then assigned to peer 60
  - If key = 60, then assigned to peer 60
  - If key = 61, then assigned to peer 1

# Circular DHT

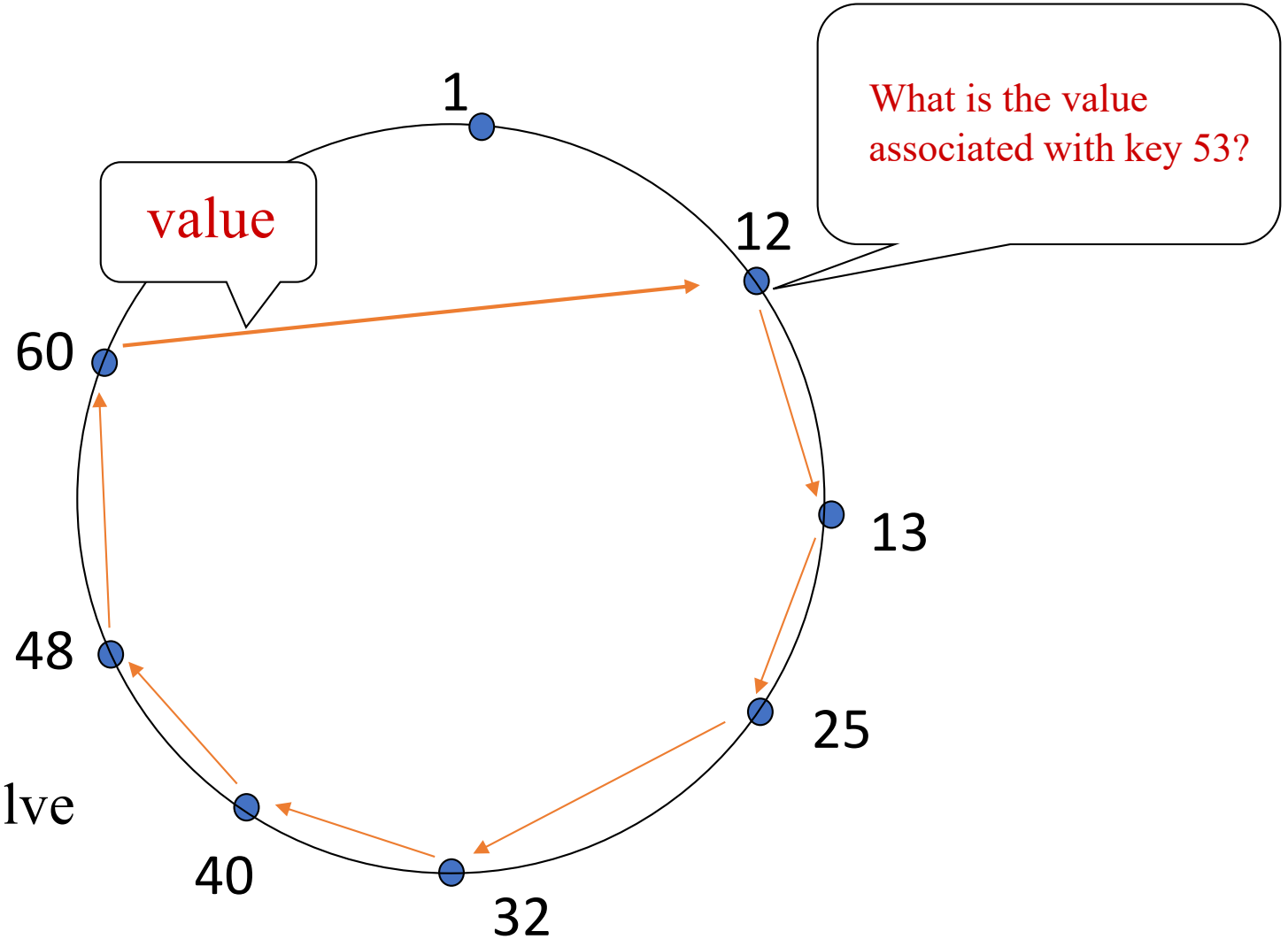
- Each peer *only* aware of immediate successor and predecessor.



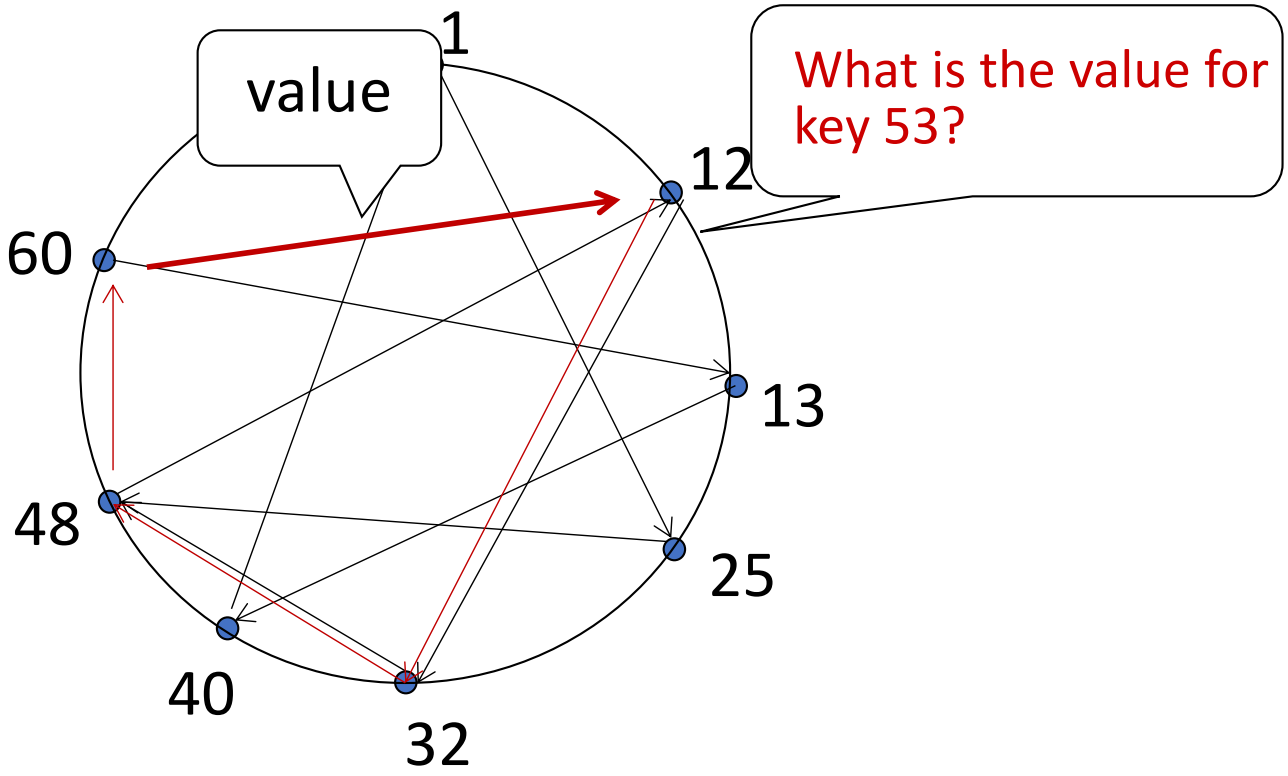
"overlay network"

# Resolving a Query



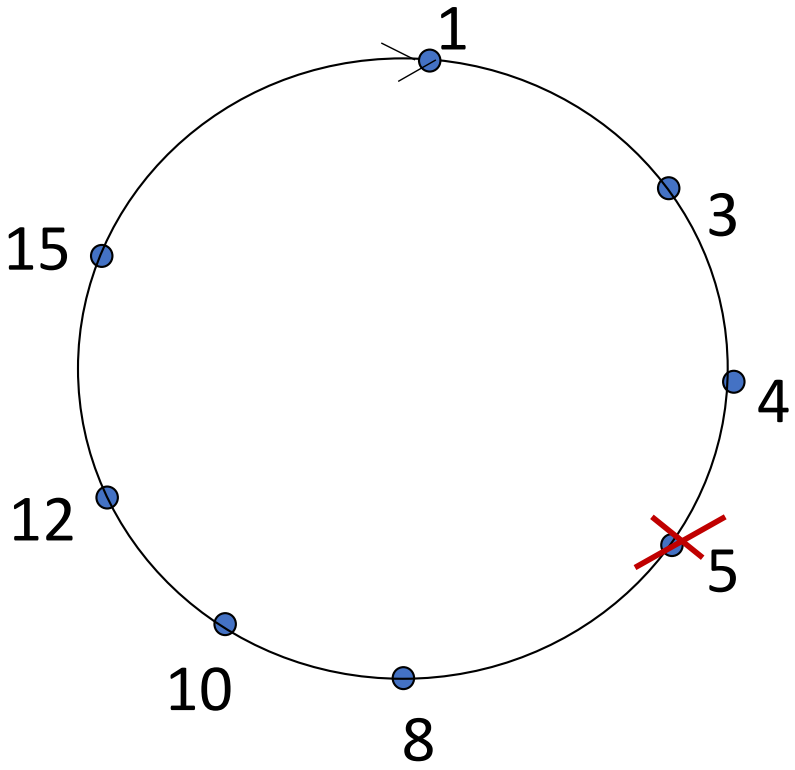

*O(N)* messages on average to resolve query, when there are *N* peers

# Circular DHT with Shortcuts




- Each peer keeps track of IP addresses of predecessor, successor, short cuts.
- Reduced from 6 to 3 messages.
- Possible to design shortcuts with *O(log N)* neighbors, *O(log N)* messages in query

# Peer Churn

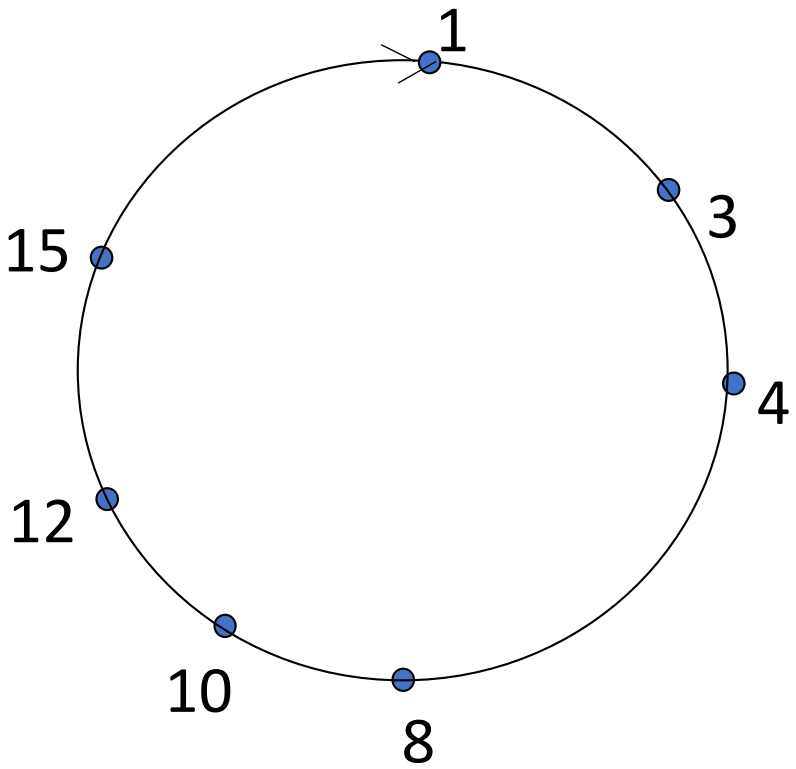



## Handling peer churn:

- Peers may come and go (churn)
- Each peer knows the addresses of its two successors
- Each peer periodically pings its two successors to check if alive
- If the immediate successor leaves, choose next successor as new immediate successor

*Example: peer 5 abruptly leaves*

# Peer Churn




## Handling peer churn:

- Peers may come and go (churn)
- Each peer knows address of its two successors
- Each peer periodically pings its two successors to check aliveness
- If immediate successor leaves, choose next successor as new immediate successor

- *Example: peer 5 abruptly leaves*
  - Peer 4 detects peer 5's departure; makes 8 its immediate successor
  - 4 asks 8 who its immediate successor is; makes 8's immediate successor its second successor.

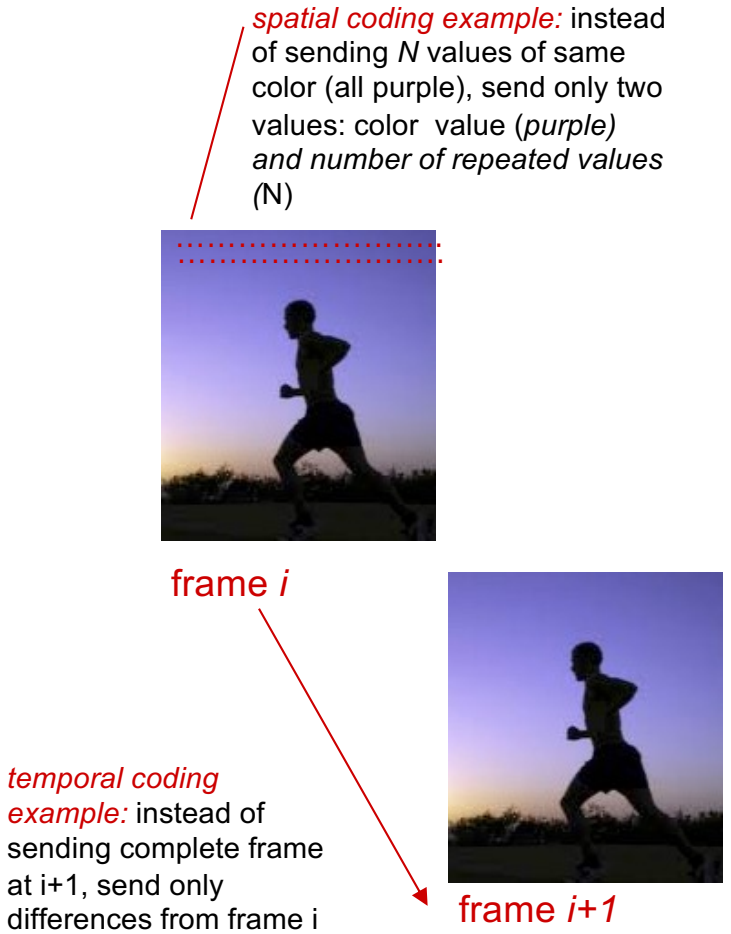
# Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- *challenge:* scale - how to reach ~1B users?
- *challenge:* heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution:* distributed, application-level infrastructure



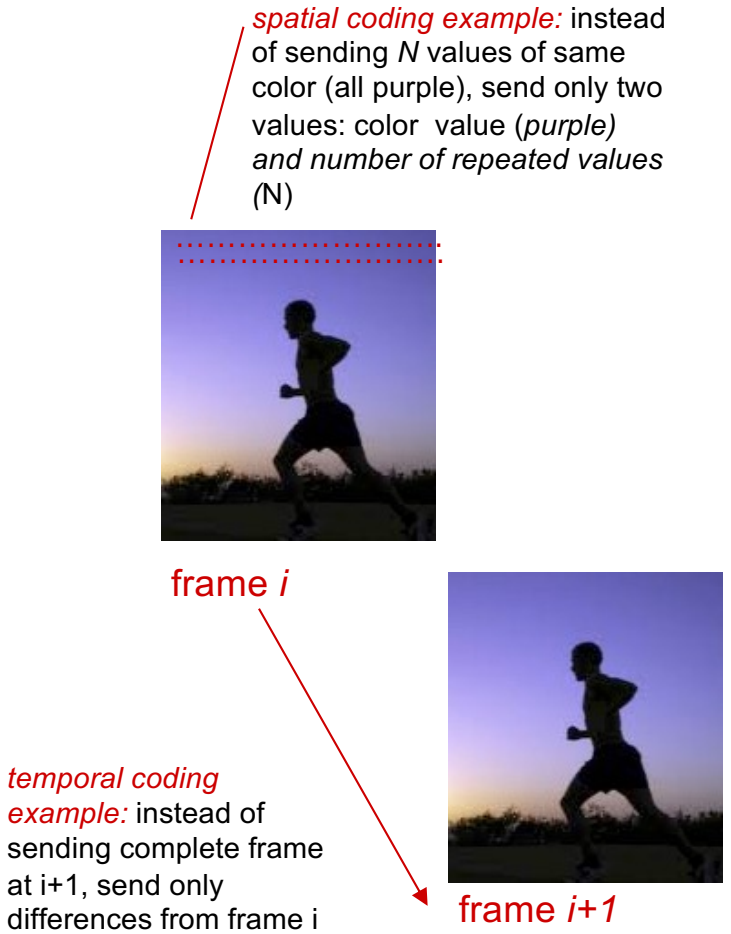

# Multimedia: video

- 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)



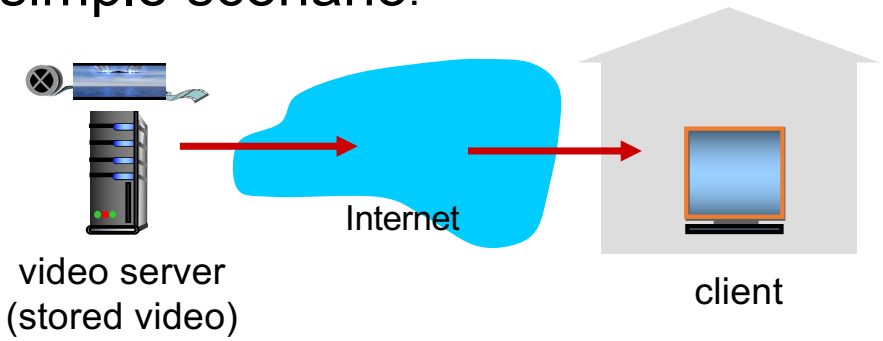
# Multimedia: video

- 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, 64Kbps – 12 Mbps)



# Streaming stored video

simple scenario:




Main challenges:

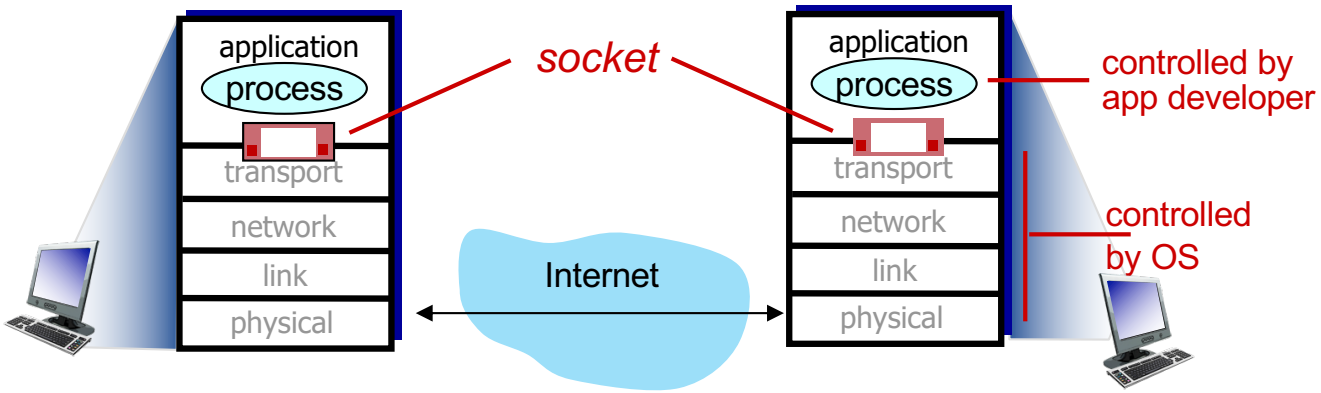
- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, access network, network core, video server)
- packet loss, delay due to congestion will delay playout, or result in poor video quality

See section 2.6 in the *textbook Kurose-Ross 8th edition*

# Socket programming

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

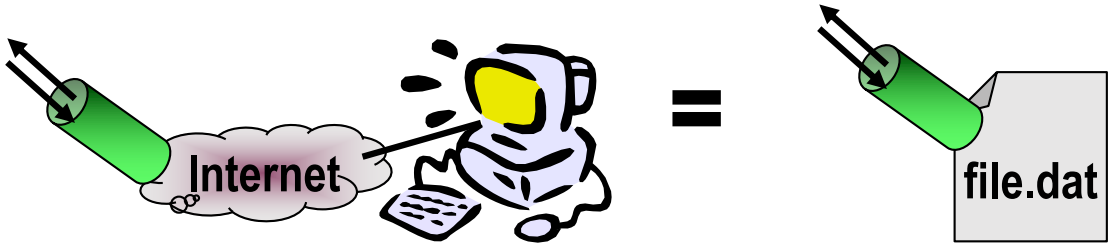
*socket:* door between application process and end-end-transport protocol



# Where is Socket From?

- Originally developed in 4.2BSD (Berkeley Software Distribution), commonly known as Berkeley UNIX.
- In UNIX, a process has a set of I/O descriptors that one reads from and writes to. These descriptors may refer to communication channels (sockets).
- The lifetime of a descriptor is made up of three phases:
  - creation (open socket),
  - reading and writing (receive and send to socket), and
  - destruction (close socket).

# The Socket Interface




- The basic ideas:
  - a ***socket*** is like a file:
    - you can read/write to/from the network just like you would a file
  - For connection-oriented communication (e.g. TCP)
    - servers (passive open) do **listen** and **accept** operations
    - clients (active open) do **connect** operations
    - both sides can then do **read** and/or **write** (or **send** and **recv)**
    - then each side must **close**
    - There are more details, but those are the most important ideas
  - Connectionless (e.g. UDP): uses **sendto** and **recvfrom**

# Sockets and Socket Libraries

- In Unix, socket procedures (e.g. `listen`, `connect`, etc.) are *system calls*
  - Part of the operating system
  - Implemented in the “top half” of the kernel
  - When you call the system call, control moves to the operating system, and you are using “system” CPU time

# Sockets and Socket Libraries

- On some other systems, socket procedures are *not* part of the OS
  - Instead, they are implemented as a library, linked into the application object code (e.g. a DLL under Windows)
  - Typically, this DLL makes calls to similar procedures that ***are*** part of the native operating system.
  - Unfortunately, different set of commands for different OS and some are semi compatible (Compatibility problem)
  - This is what the Comer text calls a ***socket library***
    - A socket library simulates Berkeley sockets on OS's where the underlying OS networking calls are different from Berkeley sockets

# Connection Oriented and Connection Less Protocols

- Connection Oriented
  - A connection is established before starting the communication.
  - When connection is established, we send the message or the information and then we release the connection.
    - Example: TCP
- Connection Less
  - It is similar to the postal services, as it carries the full address where the message (letter) is to be carried. Each message is routed independently from source to destination.
  - The order of message sent can be different from the order received.
  - In connectionless the data is transferred in one direction from source to destination without checking that destination is still there or not or if it prepared to accept the message.
  - Authentication is not required.
    - Example: UDP

# Types of Socket

*Two socket types for two transport services:*

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

***Application Example:***

- Client reads a line of characters (data) from its keyboard and sends the data to the server.
- The server receives the data and converts characters to uppercase.
- The server sends the modified data to the client.
- The client receives the modified data and displays the line on its screen.

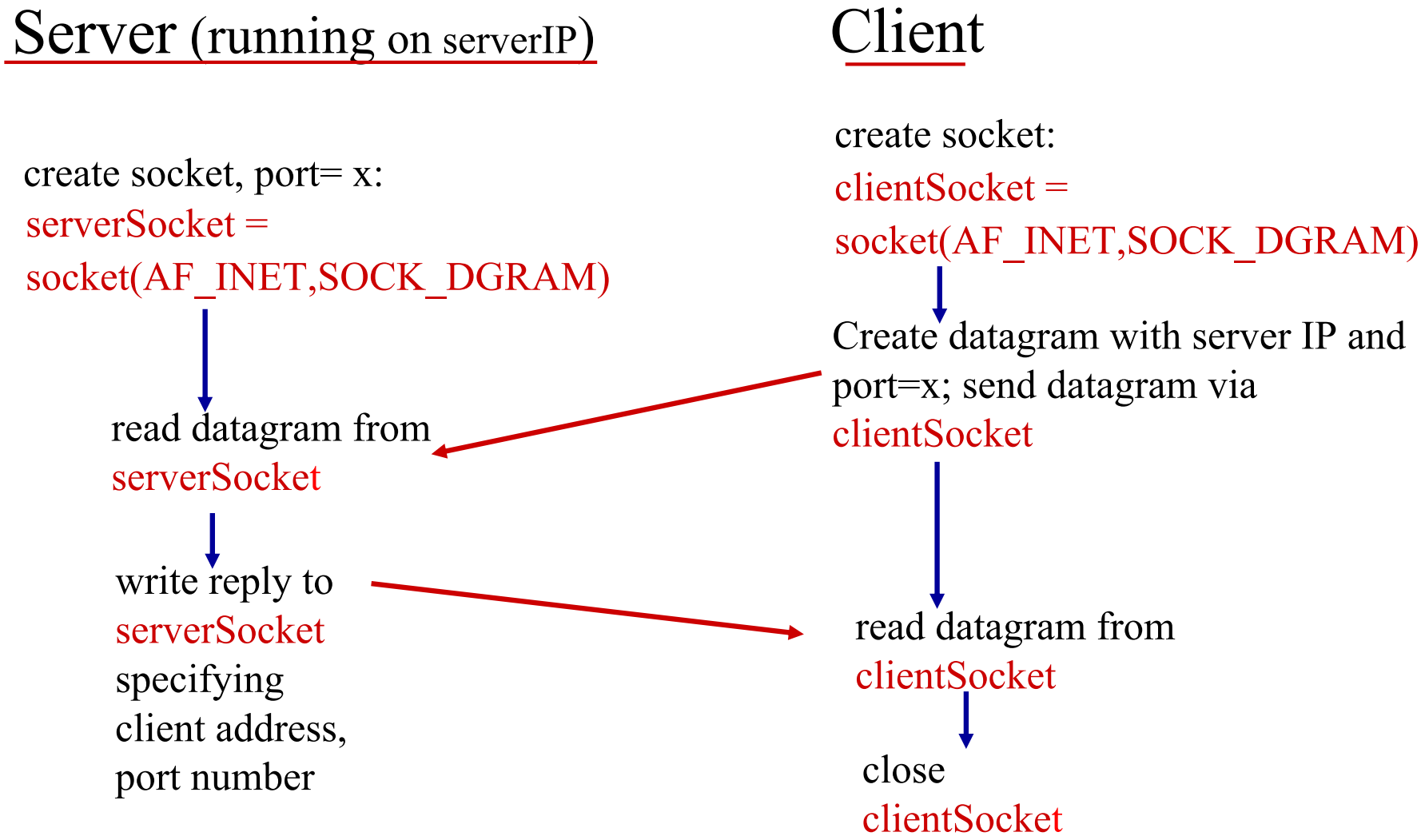
# Socket Programming with UDP

- UDP: no “connection” between client and server
  - no handshaking before sending data
  - sender explicitly attaches IP destination address and port # to each packet
  - receiver need to listen
  - 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



# 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:

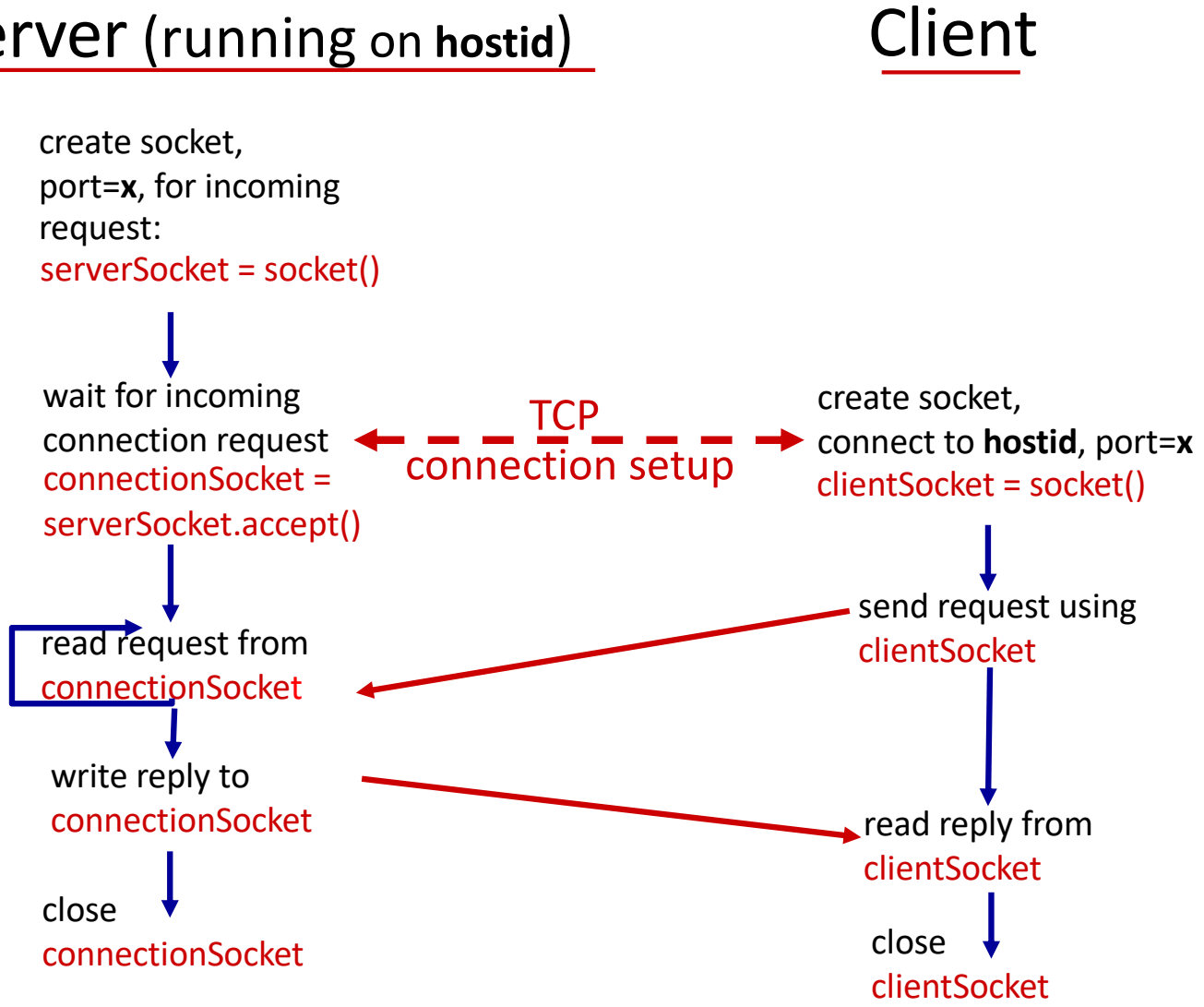
- Creating TCP socket, specifying IP address, port number of server process
- *When client creates socket:* Client TCP establishes connection to server TCP

- 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) →

# Layer 7: Summary

- 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, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming: TCP, UDP sockets

our study of network application layer is now complete!