Advanced Network Technologies

Applications

Dr. Wei Bao | Lecturer
School of Computer Science

THE UNIVERSITY OF SYDNEY

HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined GETs over single TCP connection

- server respond e-first-served scheduling) to

- with FCFS, small object may ha transmission (head-of-line (HOL) blocking) b bject(s)

- loss recovery (retransmitting lost TCP segments) stalls object transmission

HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending object

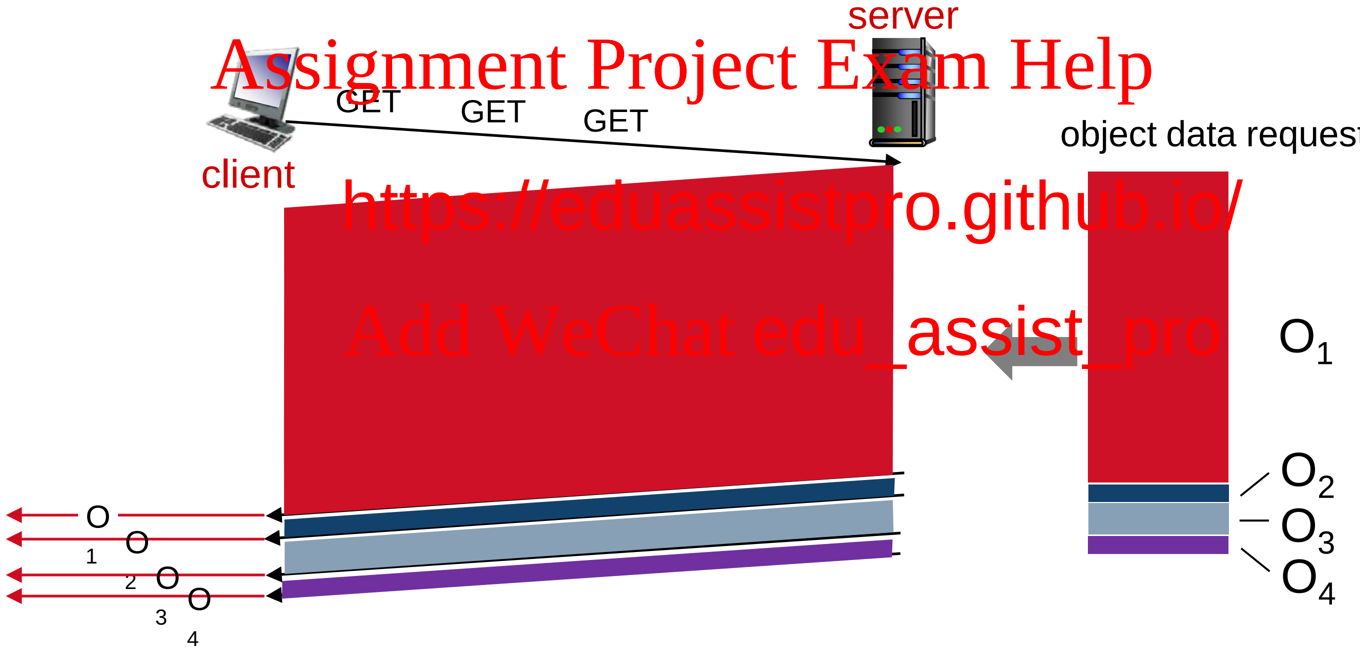
- methods, statu unchanged from HTTP 1.1

- transmission order of requested d on client-specified object priority (not necessarily FCFS)

- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate Head-of-line (HOL) blocking

HTTP/2: mitigating HOL blocking

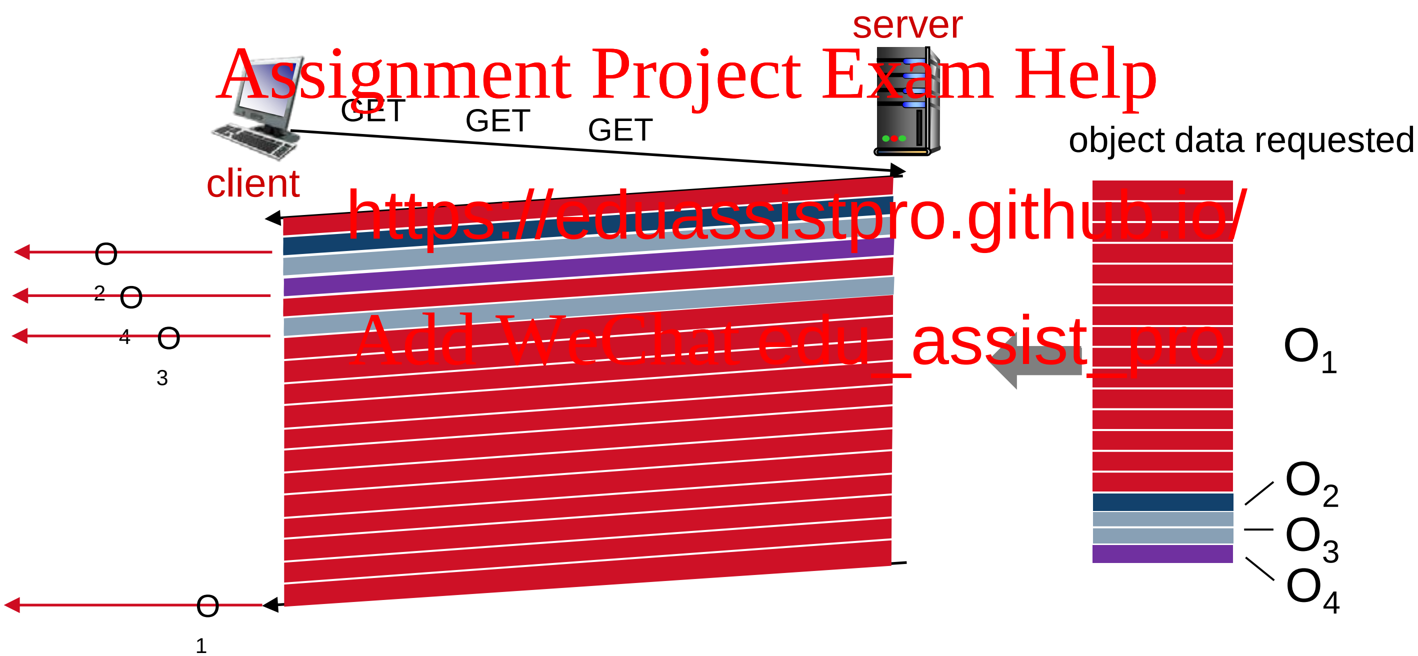
HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)



objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2: mitigating HOL blocking

HTTP/2: objects divided into frames, frame transmission interleaved



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

Client

Server

HTTP/2 Streams and frames

Frames:

- Basic HTTP/2 data unit, replacing HTTP/1.1 header and body format.
- HTTP/2 frame g (more efficient).
- Header frames, Data frames

Streams

- Bidirectional channel where frames are transmitted
- Replacing HTTP/1.1 Request-Response mode

A single TCP connection to carry multiple streams

HTTP/2 push

The HTTP/2 Server Push mechanism allows the server to send resources proactively without waiting for a request, when it believes the client will need them.

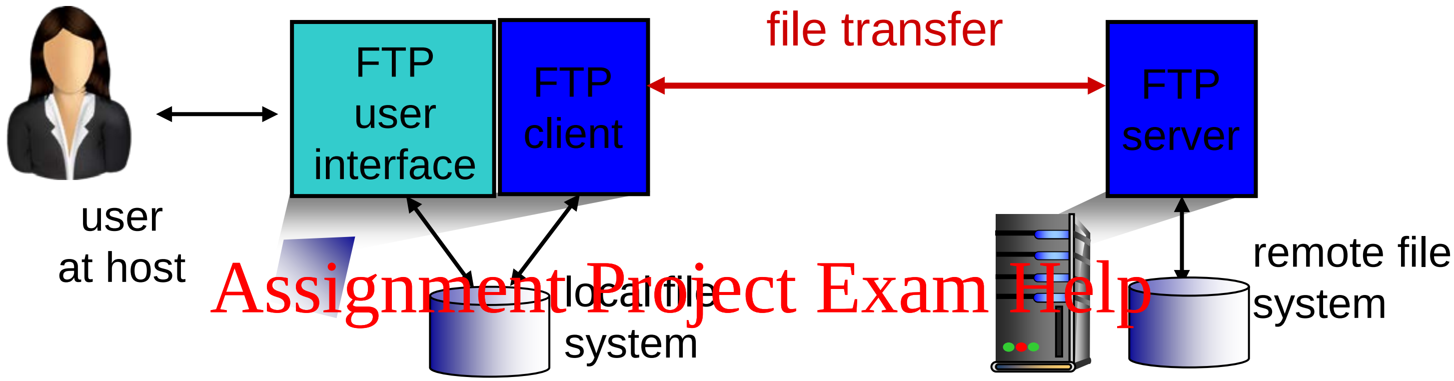
https://blog.golang.org/h2push

Outline

- Web and HTTP (Done)
- FTP
- Email
- DNS
- P2P

FTP

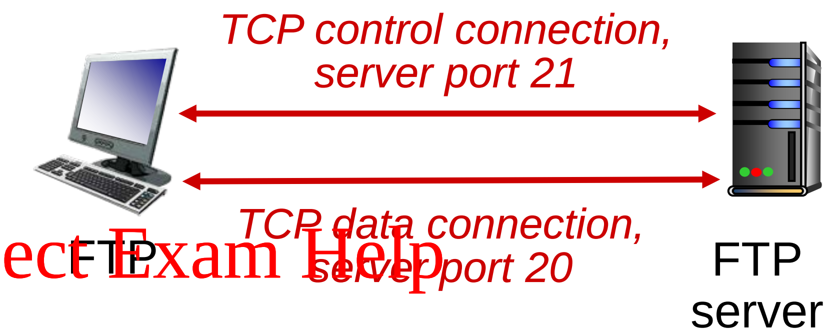
FTP: the file transfer protocol



- transfer file to/from rem
- client/server model
  - *client:* side that initiates transfer (either to/from remote)
  - *server:* remote host
- ftp: RFC 959
- ftp server: port 21, 20

FTP: separate control, data connections

- FTP client contacts FTP server at port 21, using TCP
- client authorized over control connection
- client browses re sends commands o connection
- when server receives file transfer command, *server* opens 2nd TCP data connection (for file) *to* client
- after transferring one file, server closes data connection



pens another TCP ection to transfer le

- control connection: *"out of band"*
- FTP server maintains "state": current directory, earlier authentication

FTP commands, responses

sample commands:

- sent as ASCII text over control channel
- **`USER username`**
- **`PASS password`**
- **`LIST`** return list of current directory
- **`RETR filename`** retrieves (gets) file
- **`STOR filename`** stores (puts) file onto remote host

sample return codes

- status code and phrase (as in HTTP)
- **`331 Username OK, password required`**
- **`125 data connection already open; transfer starting`**
- **`425 Can't open data connection`**
- **`452 Error writing file`**

Email

SMTP: Simple Mail Transfer Protocol

IMAP: Internet Message Access Protocol

POP3: Post Office Protocol 3

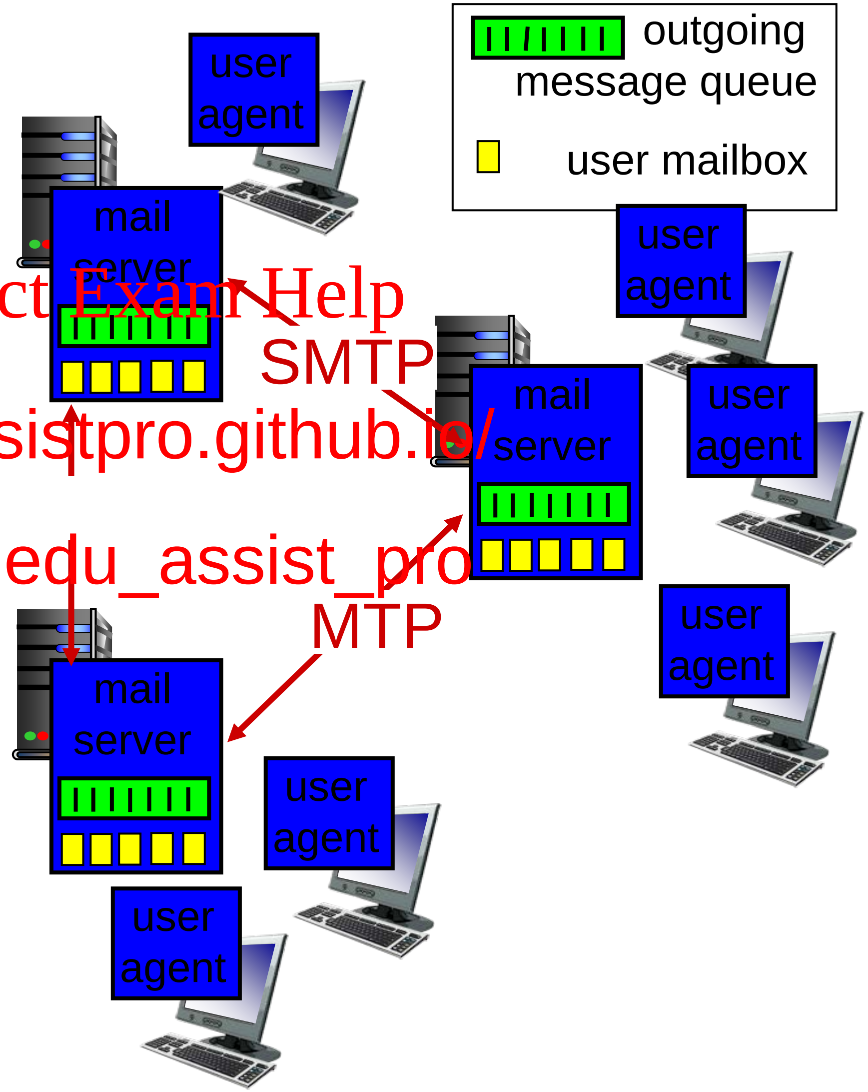
Electronic mail

Three major components:

- user agents (clients)
- mail servers
- simple mail transfer protocol: SMTP

User Agent

- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client



Electronic mail: mail servers

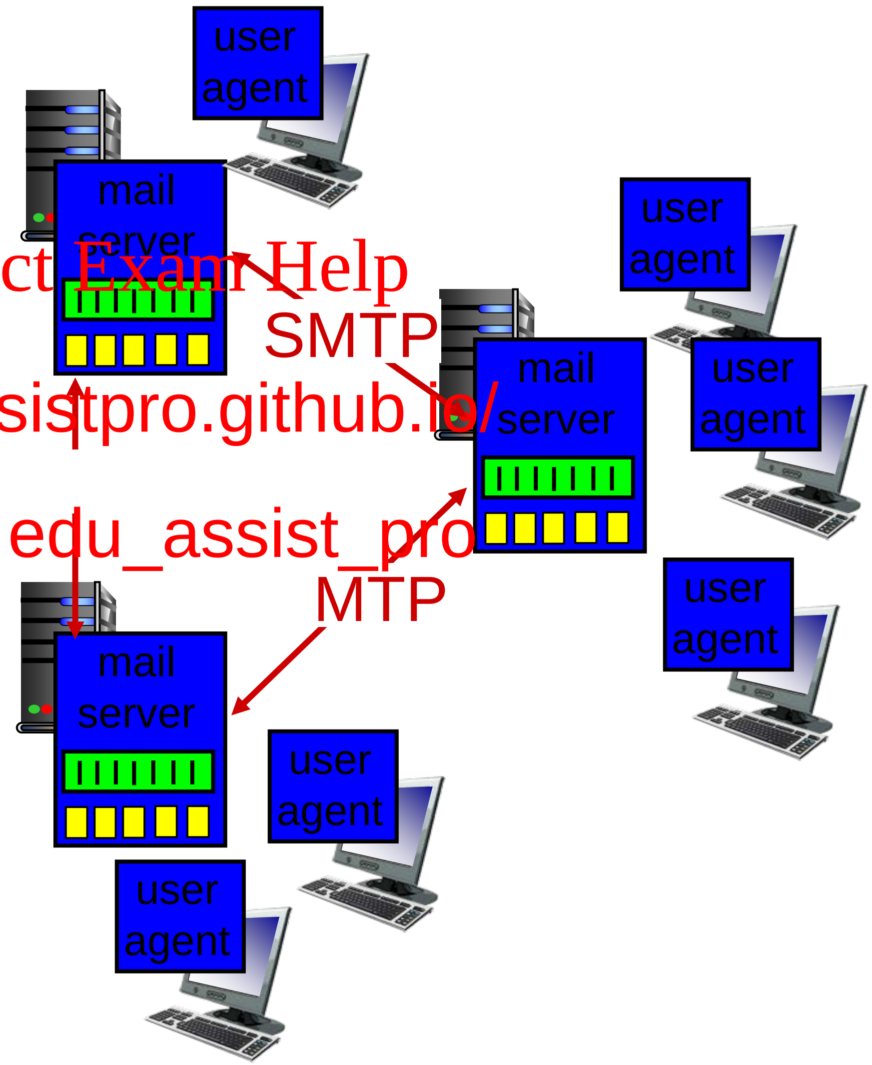
mail servers:

› *mailbox* contains incoming messages for user

› *message queue* of ou be sent) mail messa

› *SMTP protocol* to send email messages between mail servers

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



Electronic Mail: SMTP [RFC 2821]

- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HT
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII
- Q: is SMTP stateful or stateless?
  - Stateful

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message "to" `bob@someschool.edu`

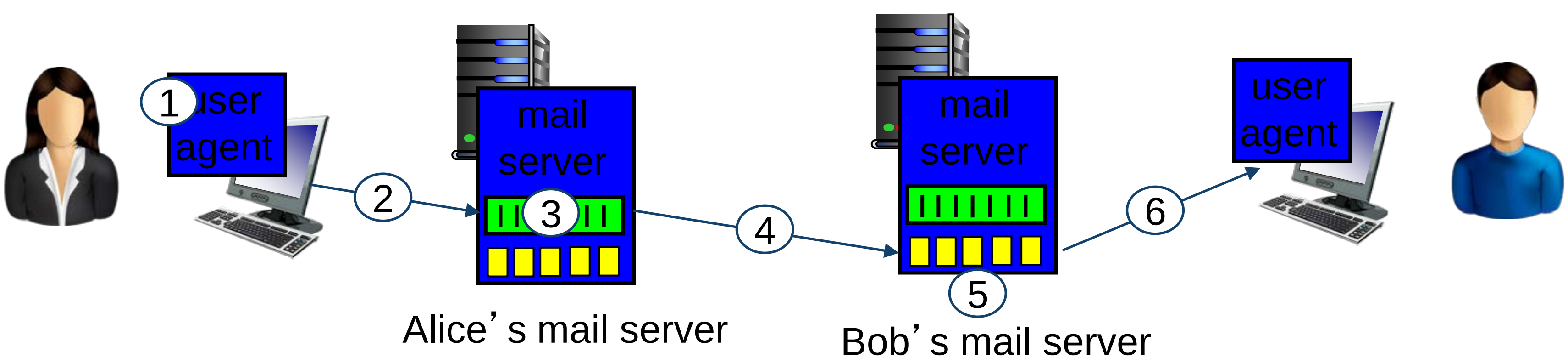
2) Alice's UA sends message to her mail server; message placed in message

3) client side of SMT TCP connection with Bob's mail server

4) SMTP client sends Alice's message over the TCP connection

5) Bob's mail server places the ob's mailbox

s his user agent to r



Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamb                                ok
C: DATA
S: 354 Enter ma                                ne by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) t
bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message
  - Carriage return
  - Line feed

comparison with HTTP:

- HTTP: pull
- SMTP: push

CII
sponse
in tus codes

- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in one msg

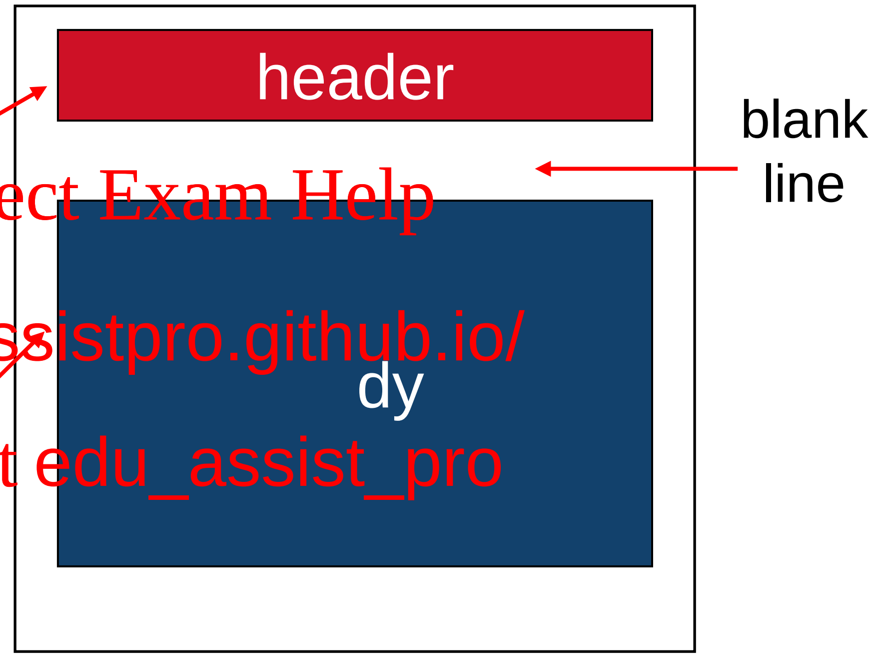
Mail message format

SMTP: protocol for exchanging email msgs

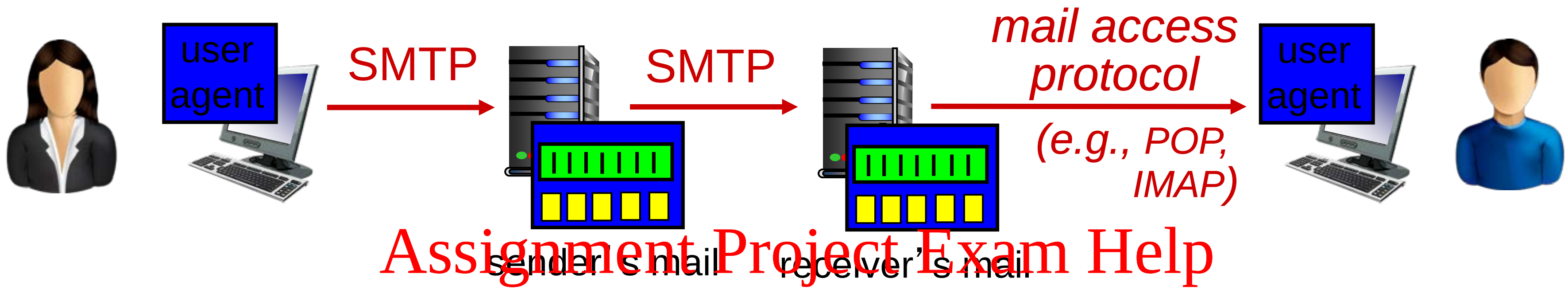
RFC 822: standard for text message format:

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

  different from SMTP MAIL FROM, RCPT TO: commands!
- Body: the "message"
  - ASCII characters only



Mail access protocols



- SMTP: delivery/stora
- mail access protocol: retrieval from ser
  - POP: Post Office Protocol [RFC 1939]: download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  - HTTP: Using a browser to access a webmail `https://webmail.sydney.edu.au`

POP3 Protocol

authorization phase

- client commands:
  - **`user:`** declare username
  - **`pass:`** password
- server responses
  - **`+OK`**
  - **`-ERR`**

transaction phase, client:

- **`list:`** list message numbers
- **`retr:`** retrieve message by number
- **`dele:`** delete
- **`quit`**

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on
C: list
        8
        2

           ge 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 (more) and IMAP

more about POP3

- previous example uses POP3 "download and delete" mode
  - Bob cannot re-r
    if he changes cli
- POP3 "download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

IMAP

- keeps all messages in one place: at server
- allows user to organize
  folders
  state across
  s
  - names of folders and mappings between message IDs and folder name

DNS

DNS: domain name system

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., www.yahoo.co humans

people: many identifiers:

- name, passport #

Q: how to map between IP address and name, and vice versa ?

Domain Name System:

› *distributed database* implemented in hierarchy of many *name*

ayer protocol: hosts, n communicate to *r* (address/name translation)

THE UNIVERSITY OF SYDNEY

DNS: services, structure

DNS services

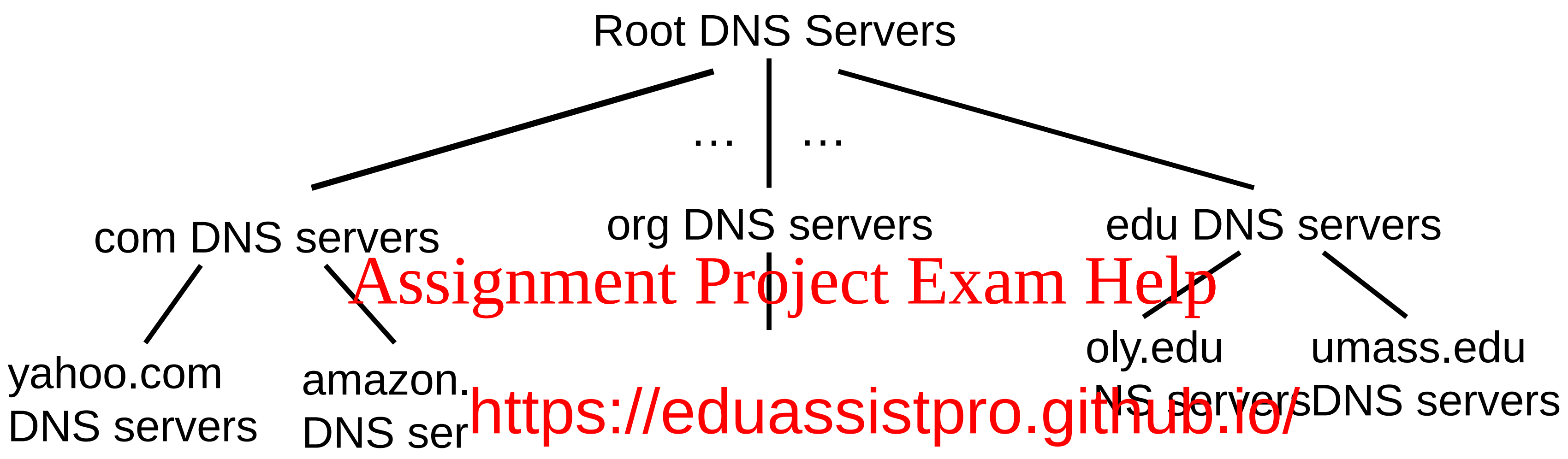
- hostname to IP address translation
- host aliasing
  - canonical, alias na
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

- single point of failure
- distant centralized database

THE UNIVERSITY OF SYDNEY

DNS: a distributed, hierarchical database



client wants IP for www.amazon.com;

- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS: root name servers

c. Cogent, Herndon, VA (5 other sites)
d. U Maryland College Park, MD
h. ARL Aberdeen, MD
j. Verisign, Dulle

k. RIPE London (17 other sites)

37 other sites)

m. WIDE Tokyo
(5 other sites)

e. NASA Mt View, CA
f. Internet Software C.
Palo Alto, CA (and 48 other
sites)

a. Verisign, Los Angeles CA
(5 other sites)
b. USC-ISI Marina del Rey, CA
l. ICANN Los Angeles, CA
(41 other sites)

g. US DoD Columbus,
OH (5 other sites)

13 root name "servers" worldwide

TLD, authoritative servers

top-level domain (TLD) servers:

- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TL

authoritative DNS servers:

- organization's own DNS server(s), providi hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

Local DNS name server

› does not strictly belong to hierarchy

› each ISP (residential ISP, company, university) has one

  - also called "default name server"

› when host makes DNS query, query is sent to its local DNS server

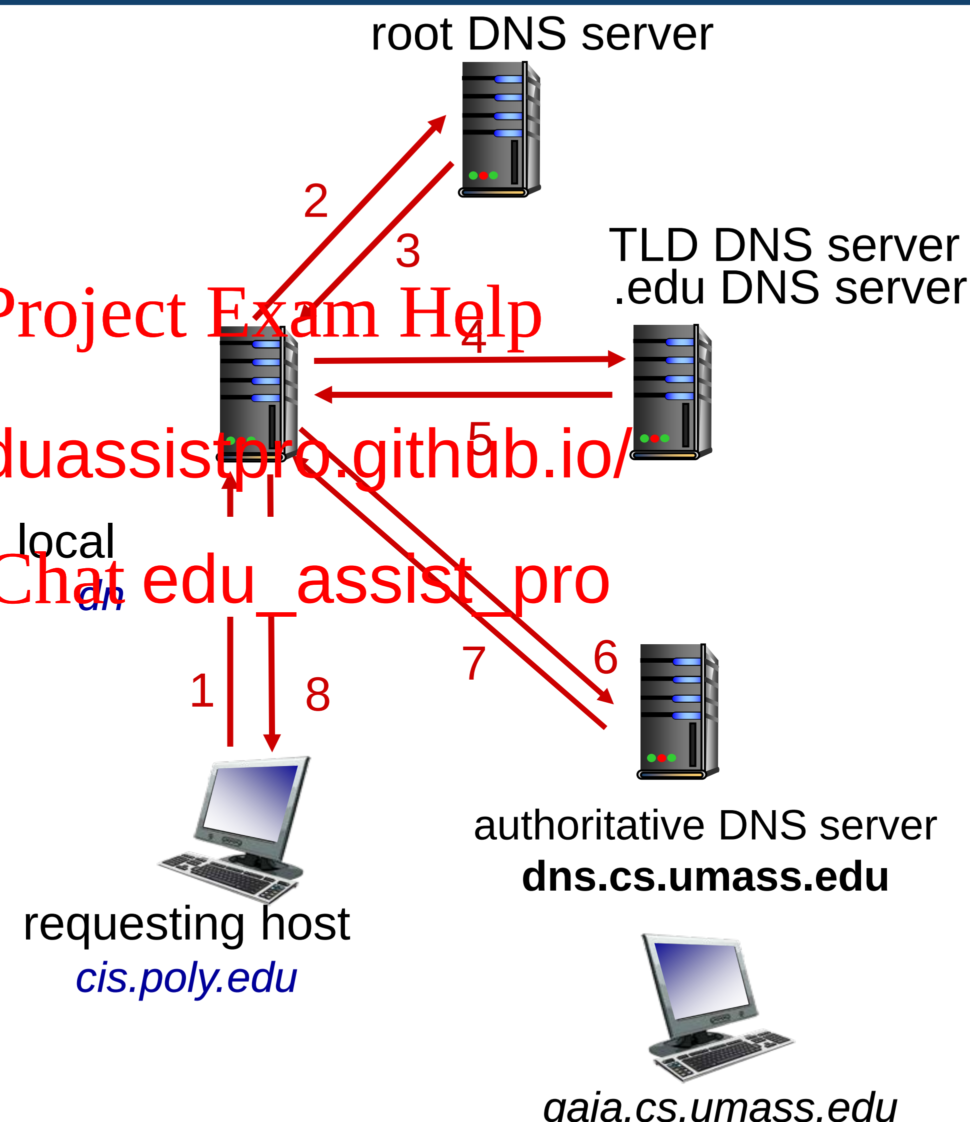
  - has local cache of recent (but may be out of date!)

  - acts as proxy, forwards query into hierarchy

DNS name resolution example

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu

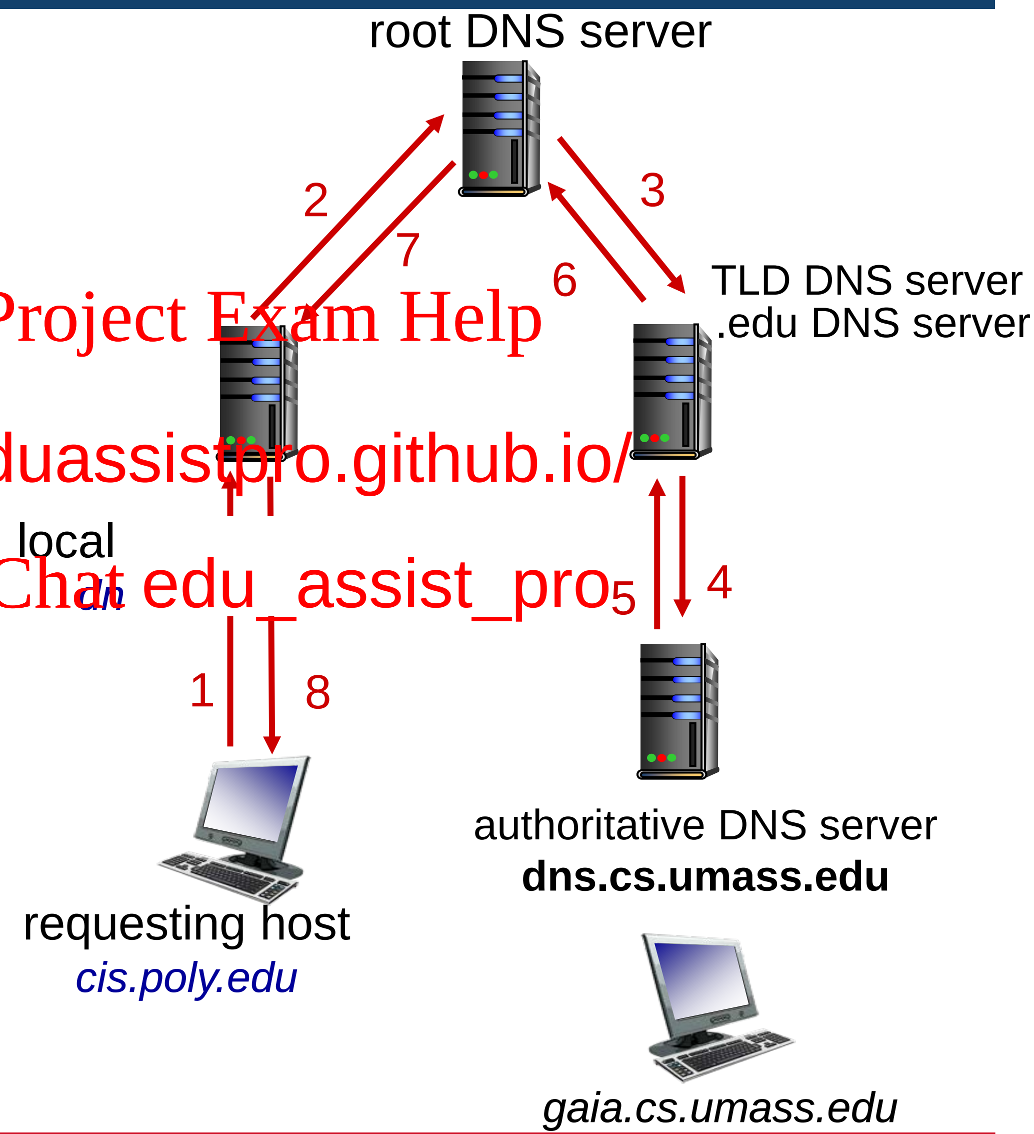
iterated query:
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example (cont'd)

recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



DNS caching, updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
- cached entries st effort name-to-address tra
  - if name host changes IP addres e known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS records

DNS: distributed db storing resource records (RR)

RR format: `(name, value, type, ttl)`

type=A
- `name` is hostna
- `value` is IP add

E

lias name for some
he real) name
- om is really
t.backup2.ibm.com
- `value` is canonical name

type=NS
- `name` is domain (e.g., foo.com)
- `value` is hostname of authoritative name server for this domain

type=MX
- `value` is name of mailserver associated with `name`

Inserting records into DNS

- example: new startup "Network Utopia"
- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server
  - registrar inserts two RRs into .com TLD server:


```
(networkutopia.co                                m, NS)
(dns1.networkutop                               A)
```


- create at authoritative server

type A record for www.networkuptopia.com;

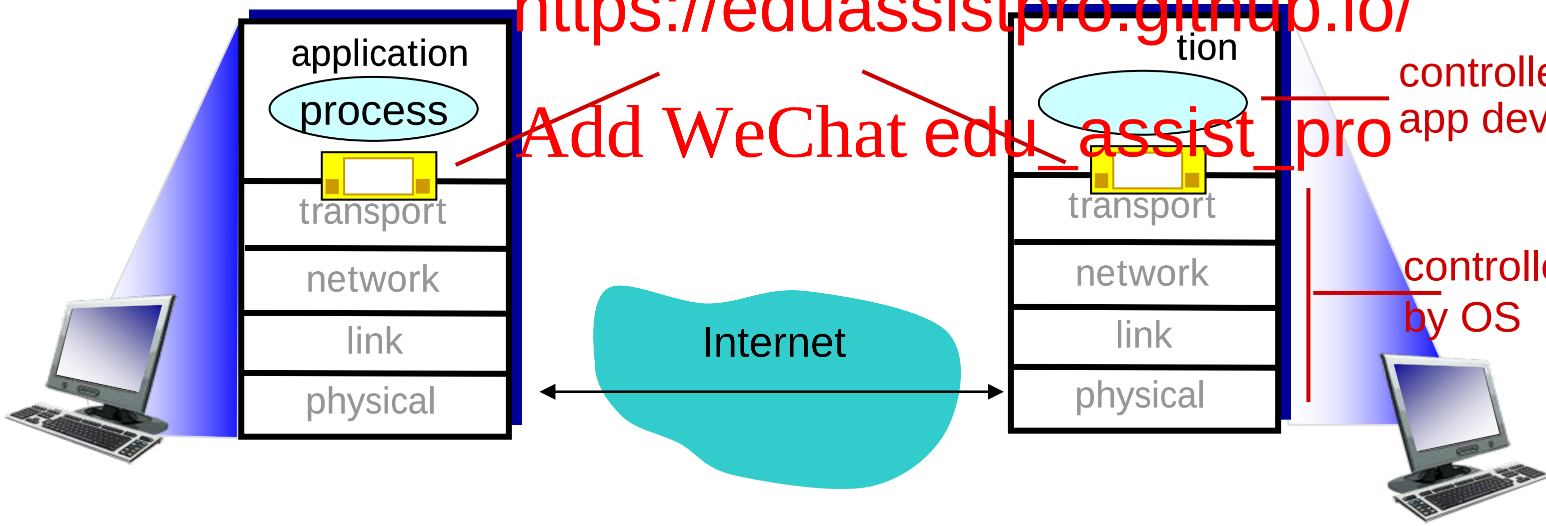
```
(www.networkutopia.com, 212.212.212.22, A)
(www.home.networkutopia.com, www.networkutopia.com, CNAME)
```

Socket Programming

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 Examp

1. Client reads a ta) from its keyboard and sends the da erver.

2. The server receives the dat erts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the 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 se from received packet

UDP: transmitted data may be l ived 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 datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:

clientSocket =
socket(AF_INET,SOCK_DGRAM)

e datagram with server IP and
; send datagram via
cket

read datagram from
clientSocket

close
clientSocket

Python UDPClient

include Python's socket library

create UDP 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
c                                   .AF_INET,
                                    SOCK_DGRAM)
                                    case sentence:')
message = messa                     tf-8')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress =
                         clientSocket.recvfrom(2048)
print (modifiedMessage.decode('utf-8'))
clientSocket.close()
```


convert from string to bytes
convert from bytes to string
New feature in Python 3

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)
                                       ort))
                                       receive'
while 1:
  message, clientA                     rSocket.recvfrom(2048)
   message=message.decode('utf-8')
  modifiedMessage = message.upper()
  serverSocket.sendto(modifiedMessage.encode('utf-8'),
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, connecting server by specifying IP address, port number of server process
- *client connects:* 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
  - numbers used to
  - nts

application viewpoint:

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

Client-server socket interaction TCP

server (running on **hostid**)

create socket,
port=**x**,
serverSocket = socket()

wait for incoming
connection request
serverSocket.listen

Accept client
connectionSocket =
serverSocket.accept()

connection setup

read request from
connectionSocket

write reply to
connectionSocket

close
connectionSocket

client

et,
hostid, port=**x**
socket()
nnect((hostid, x))

send request using
clientSocket

read reply from
clientSocket

close
clientSocket

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
c                                  T, SOCK_STREAM)
c                                  Name,serverPort))
s                                  case sentence:’)
clientSocket.send(                  code('utf-8’))
                                   ## Do not specify serverName,serverPort
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode('utf-8’))
clientSocket.close()
```

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)

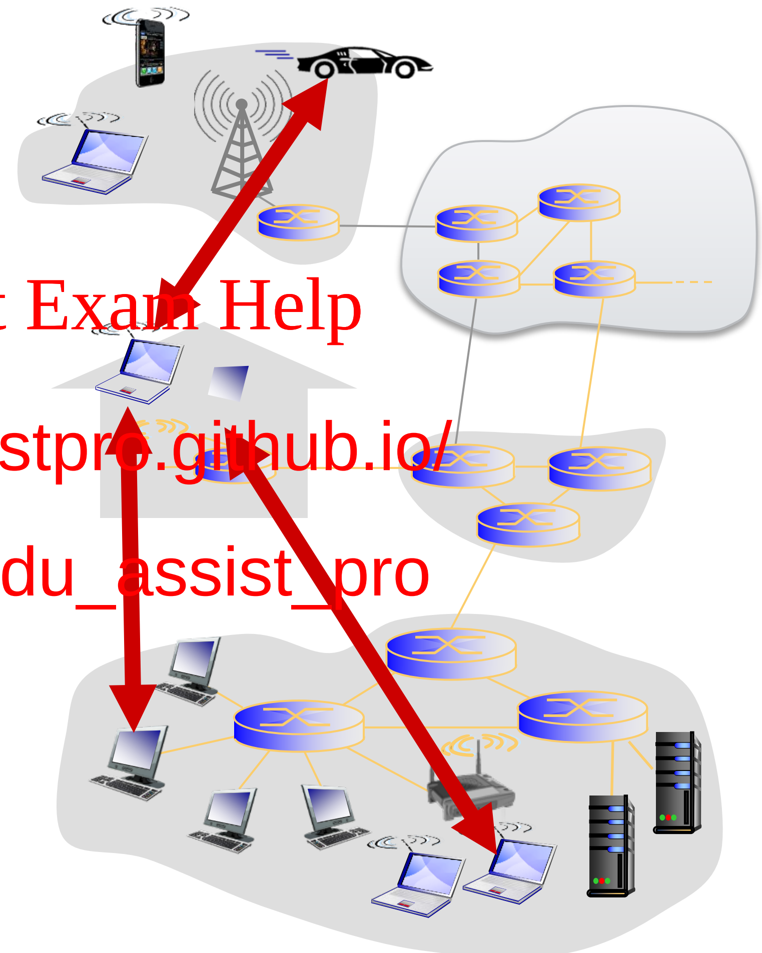
close connection to this client (but *not* welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))

receive')
while 1:
    connectionSoc    rverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.decode('utf-8').upper().encode('utf-8')
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

Peer-to-Peer

Pure peer-to-peer model architecture

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



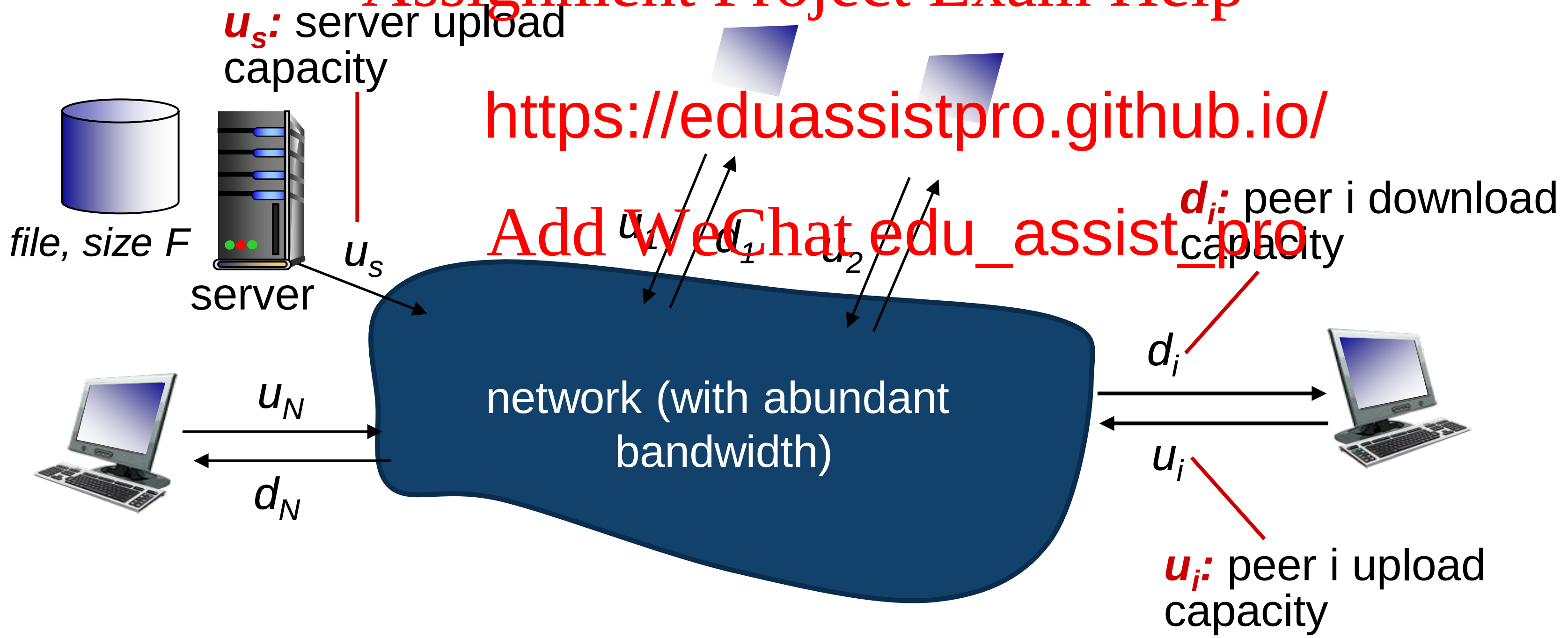
examples:

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

File distribution: client-server vs. p2p

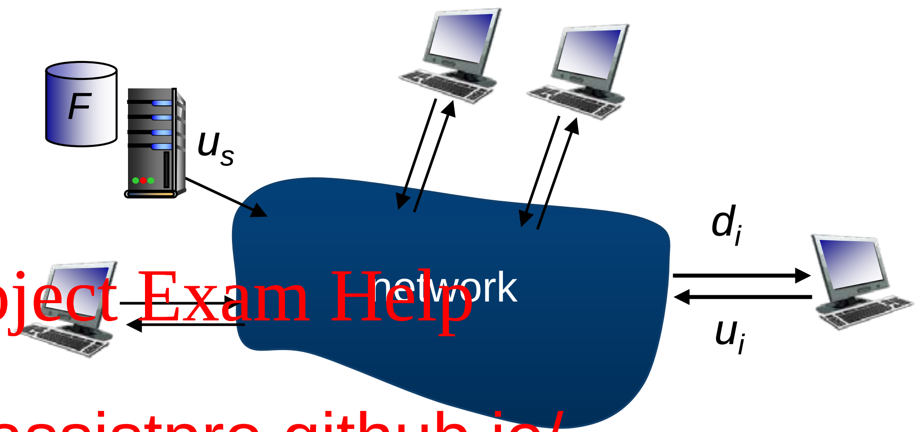
Question: how much time to distribute file (size *F*) from one server to *N* peers?

- peer upload/download capacity is limited resource



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 copie
- *client:* each client download file co
  - d_{min} = min client download rate
  - (worst case) client download time: F/d_{min}

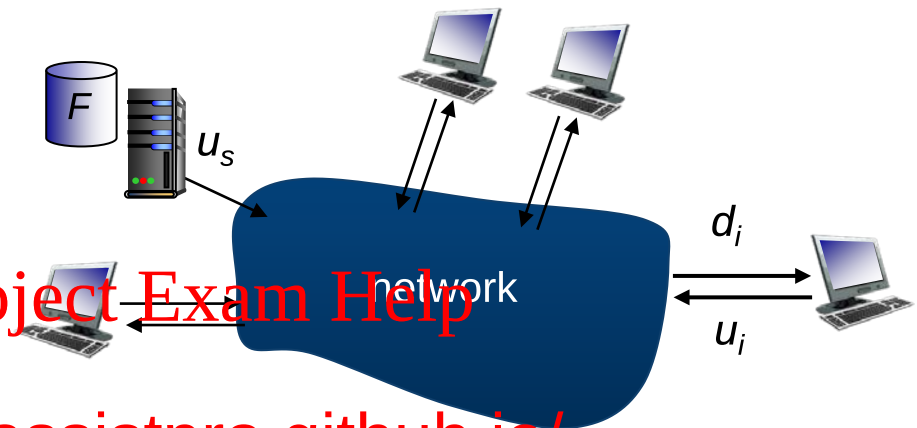


time to distribute F to N clients using client-server approach

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

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 ti
    - *clients:* as aggregat = upload *NF* bits
      - Max upload rate $u_s + \Sigma u_i$
      - $NF/(u_s + \Sigma u_i)$

time to distribute F to N clients using P2P approach

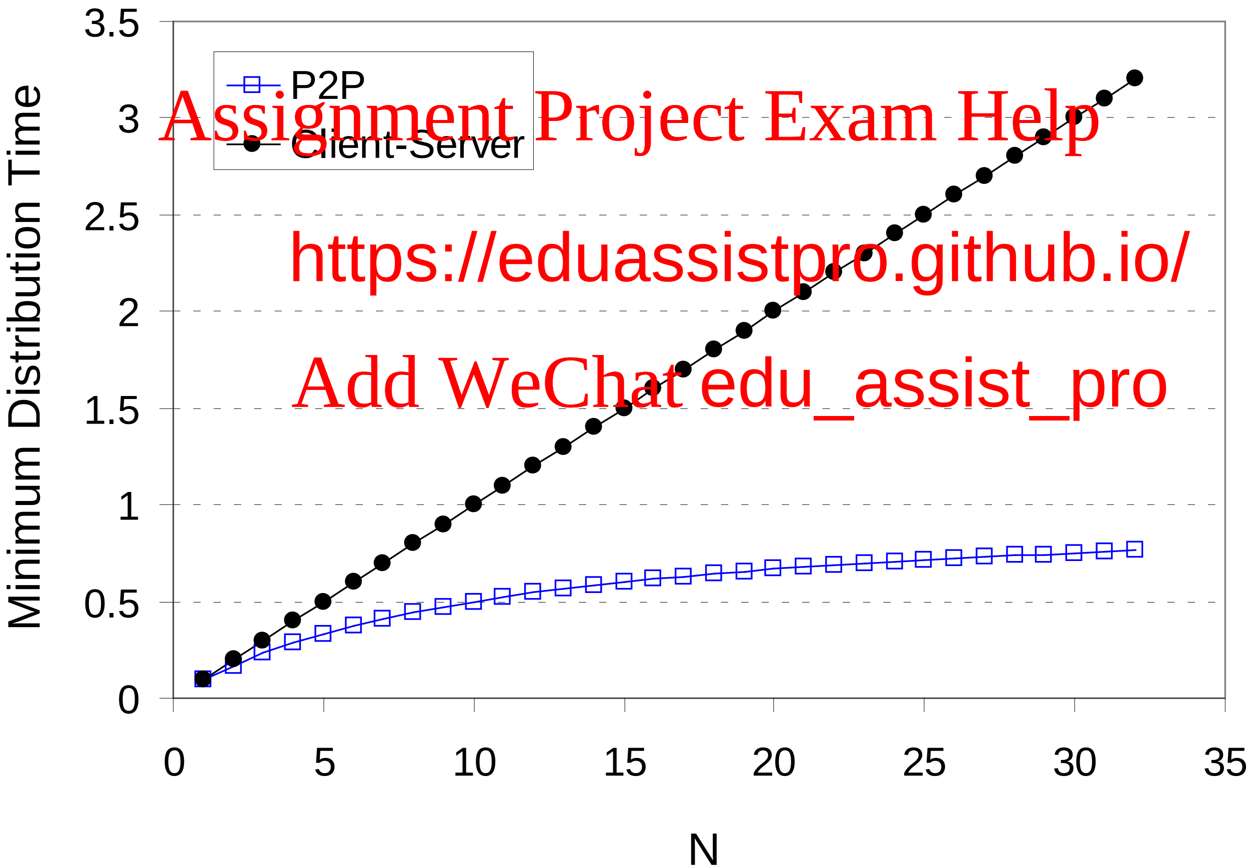
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

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

Client-server vs. p2p

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



THE UNIVERSITY OF SYDNEY

P2P file distribution: BitTorrent

<u>BitTorrent, a file sharing application</u>

- 20% of European internet traffic in 2012.
- Used for Linux distribution, software patches, distributing movies
- <u>Goal</u>: quickly replicate large files to large number of clients

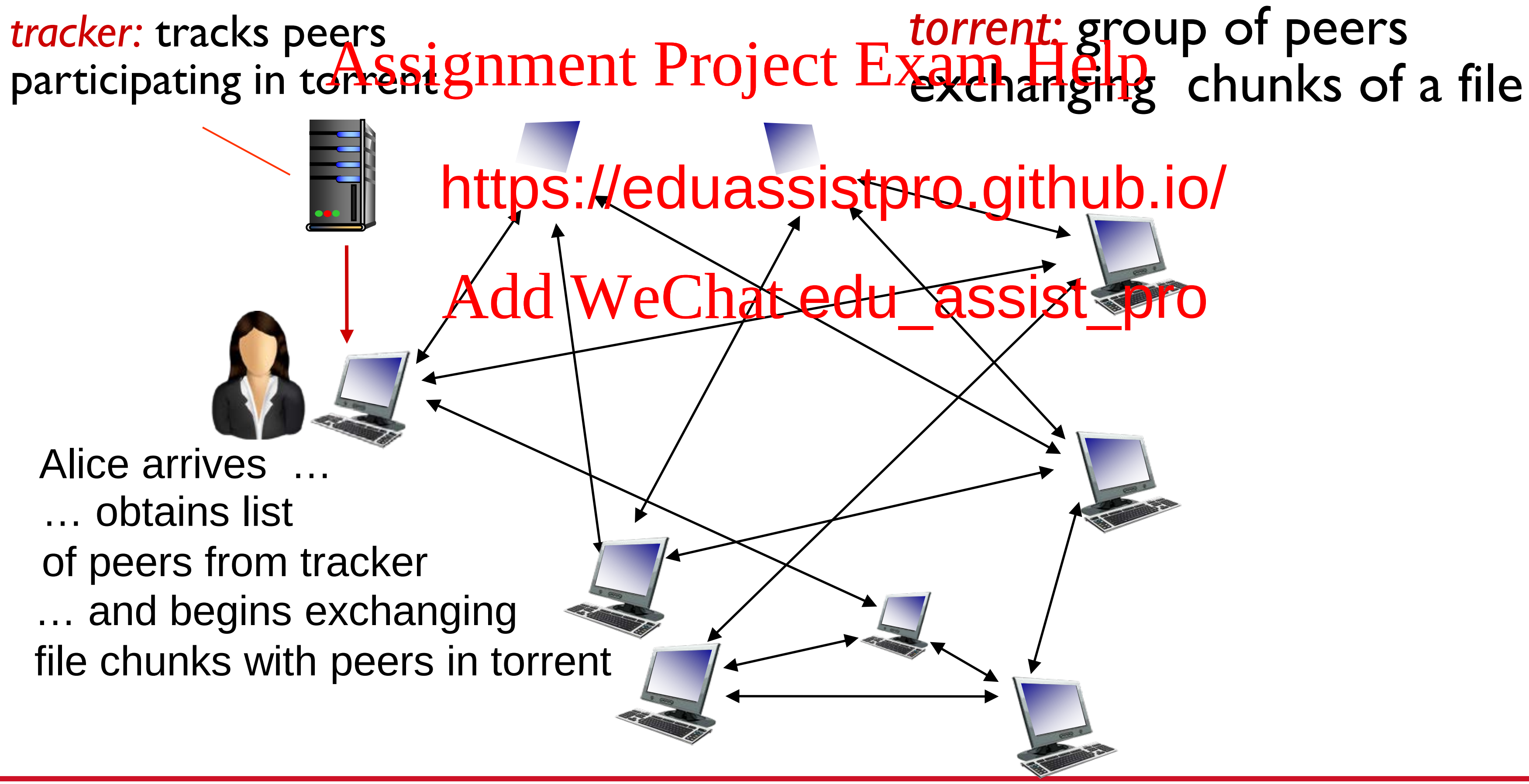
- Web server hosts a .torrent file (w/ file l racker's URL…)

- A tracker tracks downloaders/owners of
- Files are divided into chunks (256kB-1MB)
- Downloaders download chunks from themselves (and owners)
- <u>Tit-for-tat</u>: the more one shares (server), the faster it can download (client)

P2P file distribution: BitTorrent

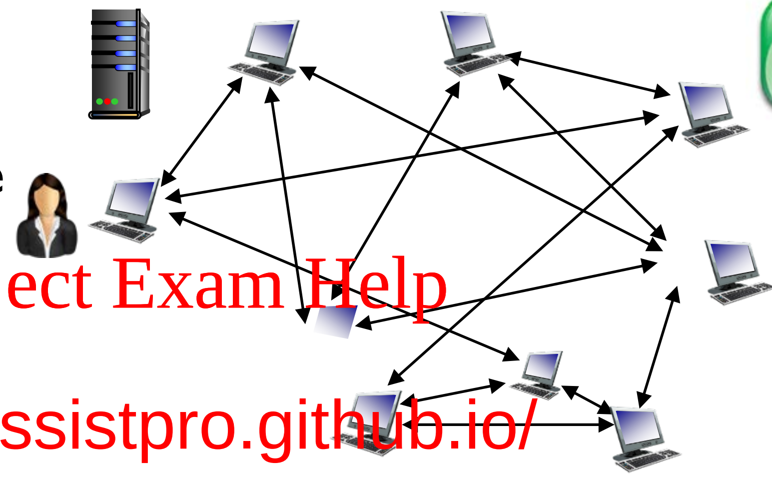
› file divided into 256KB chunks

› 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 trac
    peers, connects to
    ("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 peer for list of chun they have
- Alice requests missing chunks from peers, rarest first

sending chunks: tit-for-tat

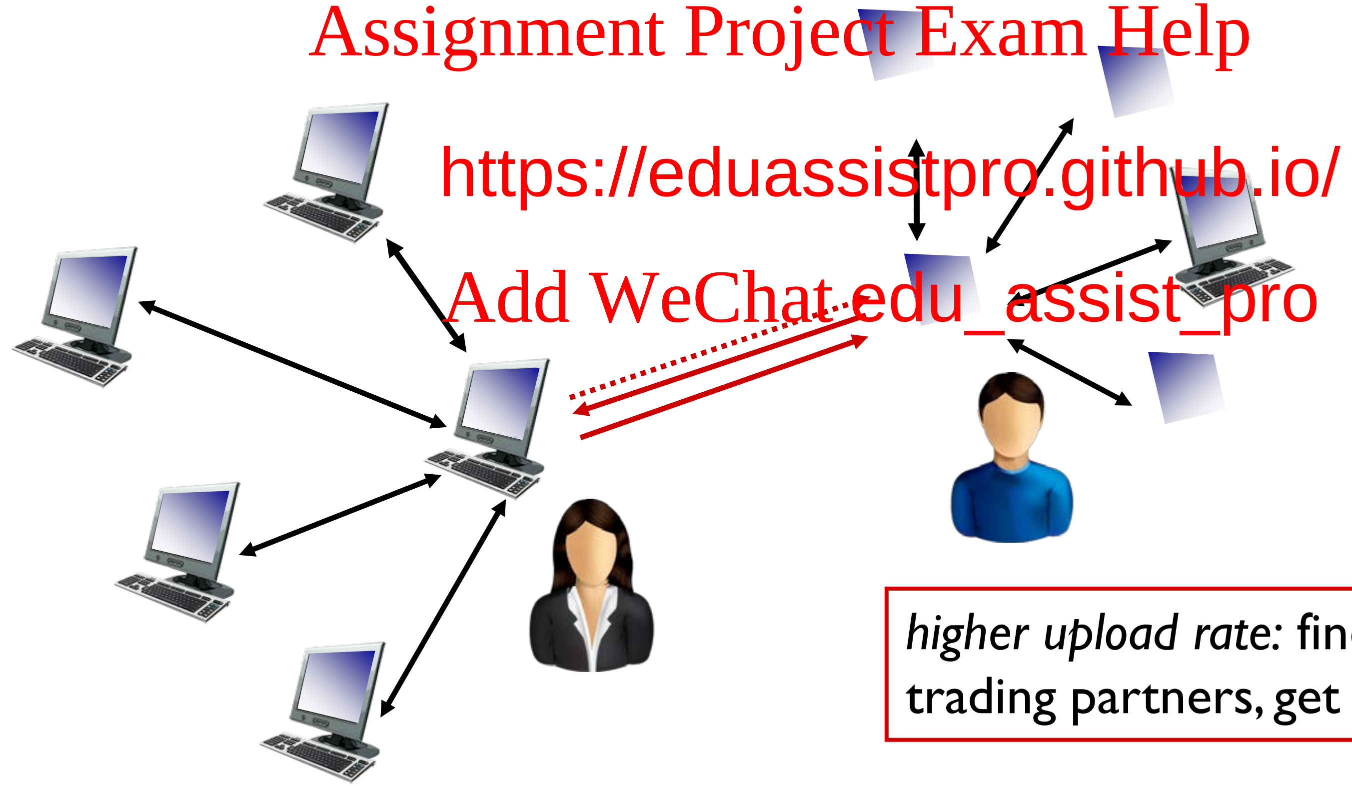
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - rs are choked by Alice (do e chunk
  - p 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

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 !