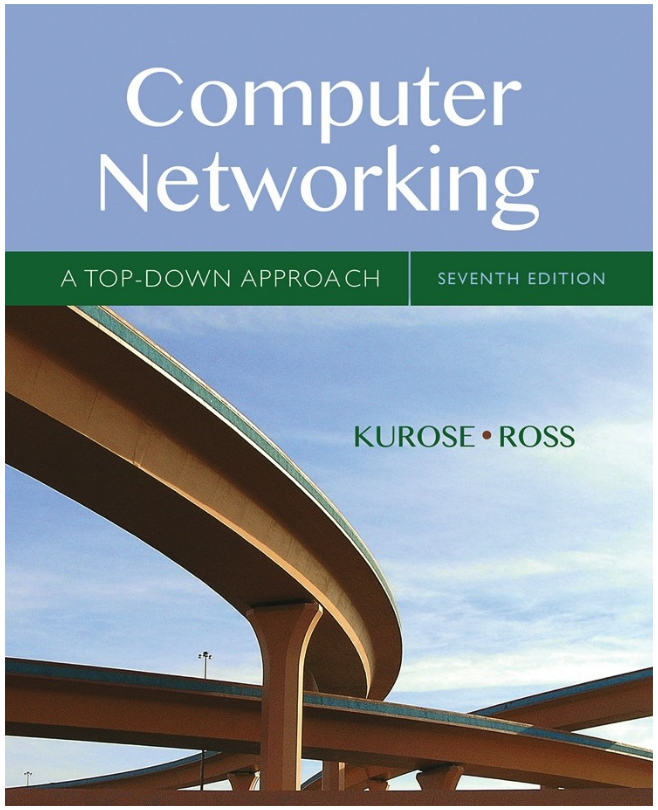
Lecture 03 Part 1 Application Layer



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

User-server state: cookies

many Web sites use cookies

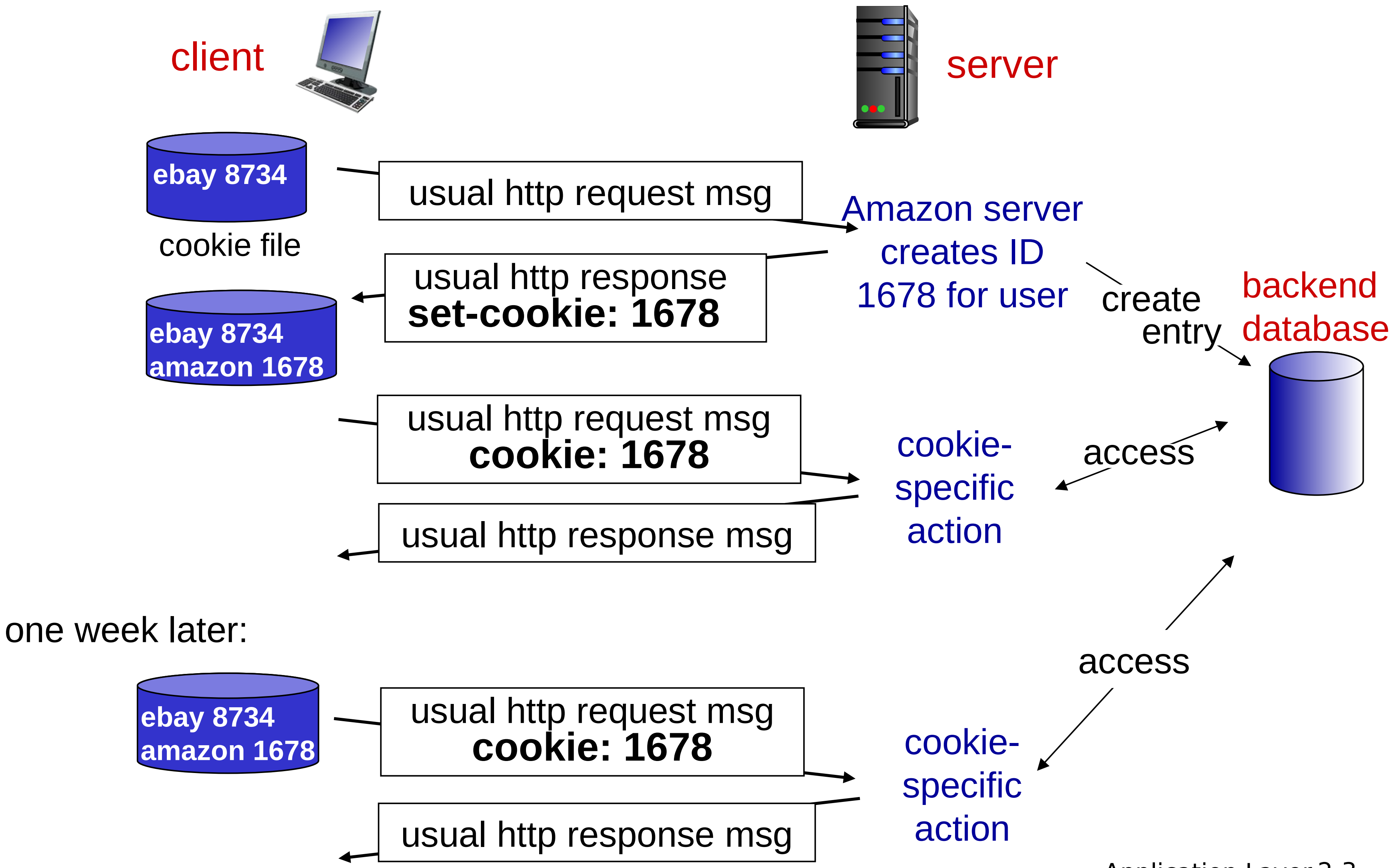
four components:

1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

example:

- Susan always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping "state" (cont.)



Cookies: HTTP Headers

- **Set-Cookie** header is used in the response header to create and give cookies directives, and it has the following general format,
  Set-Cookie: <**cookie value**>**; Optional directive 1; Optional directive 2; . . . etc**
- <cookie value> directive is the cookie user identifier. It is commonly written in the form of **name=value**, e.g. username = Will Smith or sessionID = 98765 . . . etc. More than one cookie identifier may be set at once, e.g.
  Set-Cookie: username = Will Smith; sessionID = 98765.
- Some of the optional directives will be described soon.
- The client side uses **Cookie** request header contains stored HTTP cookies previously have been set by the server with the **Set-Cookie** header. Note that the Cookie header is optional and may be omitted if, for example, the browser's privacy settings block cookies (rejection).

Set-Cookie Header Optional Directives (1)

- **Expires=**<**date**>, which indicates when the cookie should no longer be sent to the server and therefore may be deleted by the browser, e.g. **Set-Cookie: name=Nicholas; expires=Sat, 02 May 2018 23:38:25 GMT**.
- If it is not specified, the cookie will have the lifetime of a **session cookie**. A session is finished when the client is shut down meaning that session cookies will get removed at that point. However, many web browsers have a feature called session restore that will save all user tabs and have them come back next time the user uses the browser. Cookies will also be present and it's like user had never actually closed the browser.
- **Max-Age=**<**number**> can be used also for the same purpose and if both **Expires** and **Max-Age** are set, **Max-Age** directive has the **priority**.

Set-Cookie Header Optional Directives (2)

- **Domain=<domain-value>** , which specifies those hosts to which the cookie will be sent. If not specified, defaults to the host portion of the current document location (but not including sub-domains), e.g.
  Set-Cookie: name=Nicholas; domain=yahoo.com
  means include this cookie in all yahoo sub-domains such as email.yahoo.com.
- **Path=<path-value>**, can be used with (and without) Domain directive to specify the URL path that must exist in the requested resource before sending the Cookie header, e.g. **Set-Cookie: name=Nicholas; domain=yahoo.com; path=/blog**, means only paths start with /blog in yahoo.com domain (and sub-domains) are valid to use Cookie header in the request message.
- **Secure**, which is a directive flag (without values) and indicates that a secure cookie will only be sent to the server when a request is made using SSL and the HTTPS protocol, e.g. **Set-Cookie: name=Nicholas; secure**.

Cookies Types

- There are several classification of cookies depending on their directives, life cycle and usage. The most important two based on their life cycle are **session** cookies and **persistent/permanent** cookies.
- **Session cookies** will get removed when the client is shut down. They don't specify the **Expires** or **Max-Age** directives. Note that web browser have often enabled session restoring.
- Instead of expiring when the client is closed, permanent cookies expire at a specific date (**Expires**) or after a specific length of time (**Max-Age**).

Cookies Rejection

- Cookies may be rejected in certain conditions or simply by the user choice.
- Cookies will be rejected if the domain directive doesn't include the origin server. This is called **Invalid domain** case. For example, the following cookie will be rejected if it was set by a server hosted on **company.com**,
  Set-Cookie: qname=219ffwe; Domain=somecompany.co.uk; Path=/; Expires=Wed, 30 Aug 2019 00:00:00 GMT
- Some cookies with specific prefixes have certain requirements. In particular, Cookies names with the prefixes __Secure- and __Host- can be used only if they are,

  1. set with the secure directive
  2. from a secure (HTTPS) origin

  In addition, cookies with the __Host- prefix must have,

  A a path of "/" (the entire host)
  B must **not** have a domain attribute.

Cookies Rejection Examples

```
// Both accepted when from a secure origin (HTTPS)
Set-Cookie: __Secure-ID=123; Secure; Domain=example.com
Set-Cookie: __Host-ID=123; Secure; Path=/


// Rejected due to missing Secure directive
Set-Cookie: __Secure-id=1


// Rejected due to the missing Path=/ directive
Set-Cookie: __Host-id=1; Secure


// Rejected due to setting a domain
Set-Cookie: __Host-id=1; Secure; Path=/; domain=example.com
```

Cookies (continued)

what cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

how to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

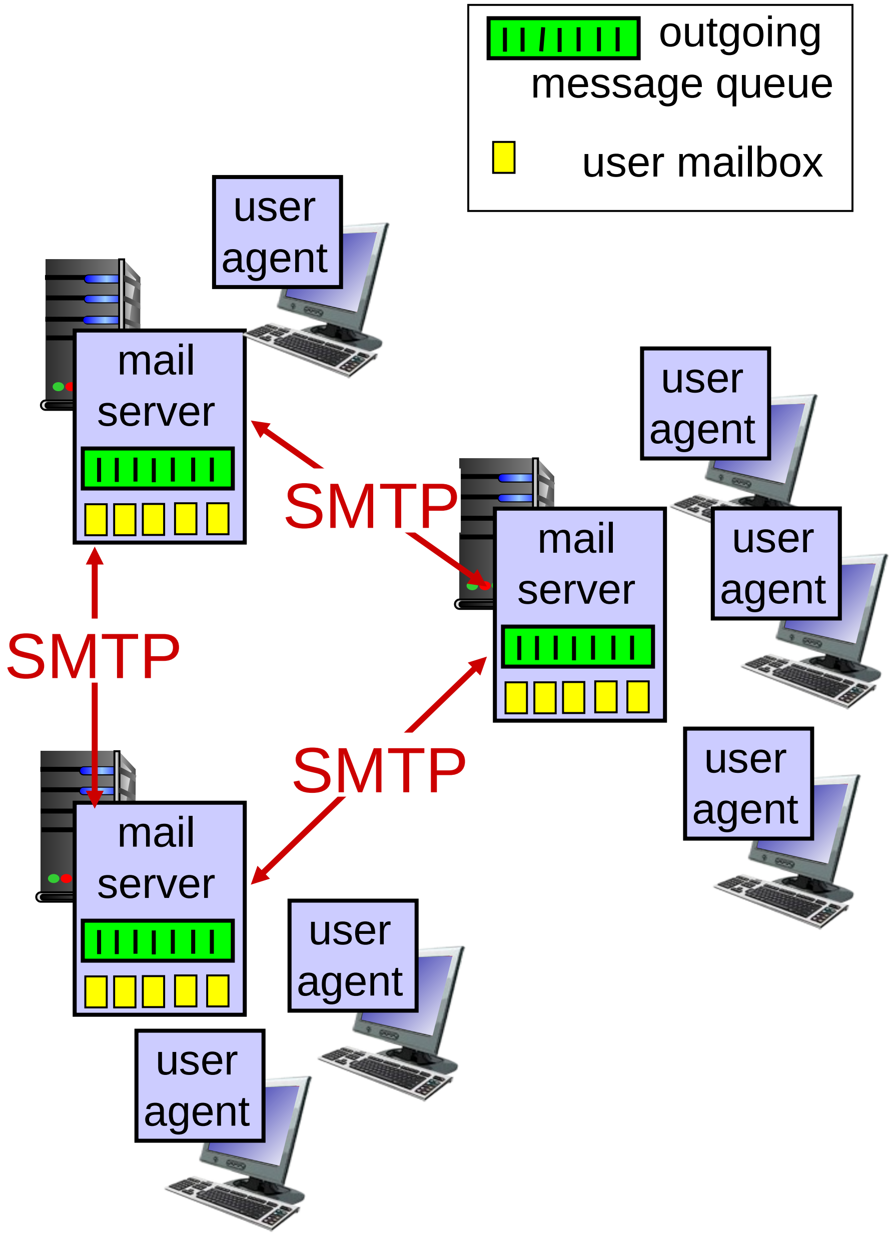
- SMTP, POP3, IMAP

2.5 P2P applications

Electronic mail

Three major components:
- user agents
- mail servers
- simple mail transfer protocol: SMTP

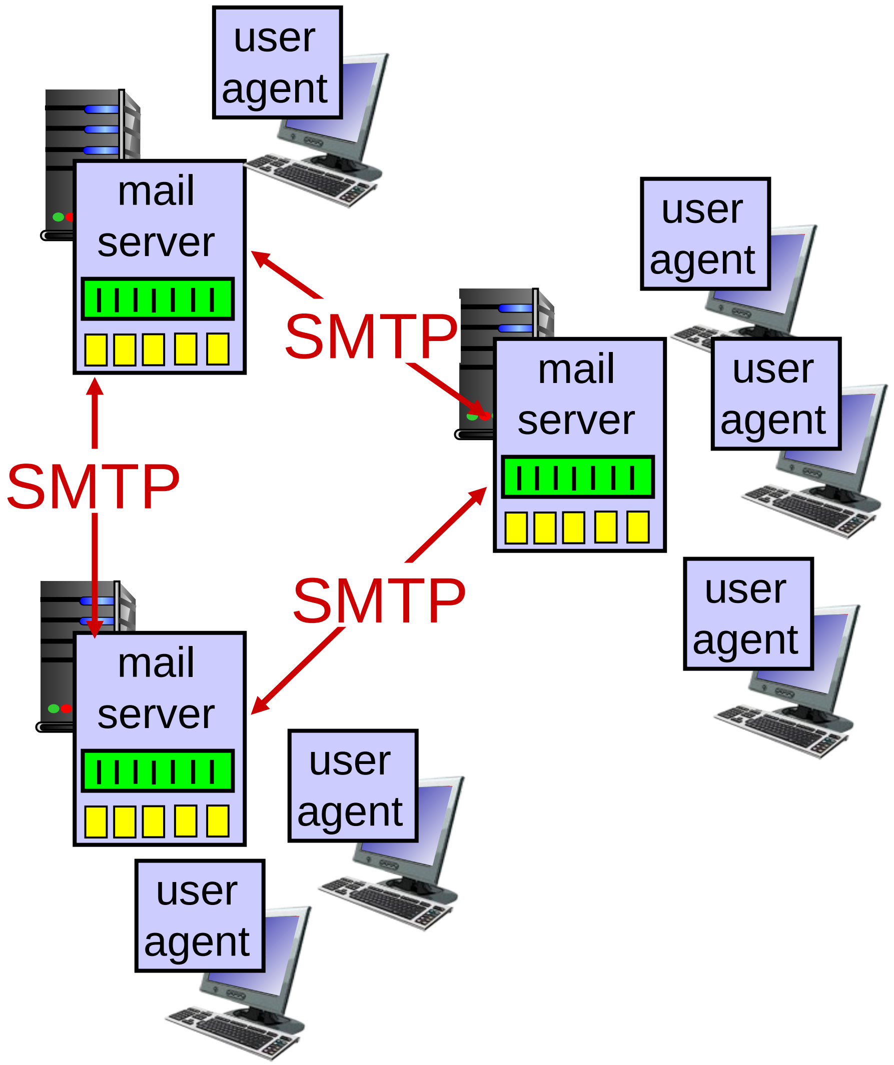
User Agent
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



Electronic mail: mail servers

mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages
- *SMTP protocol* between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server

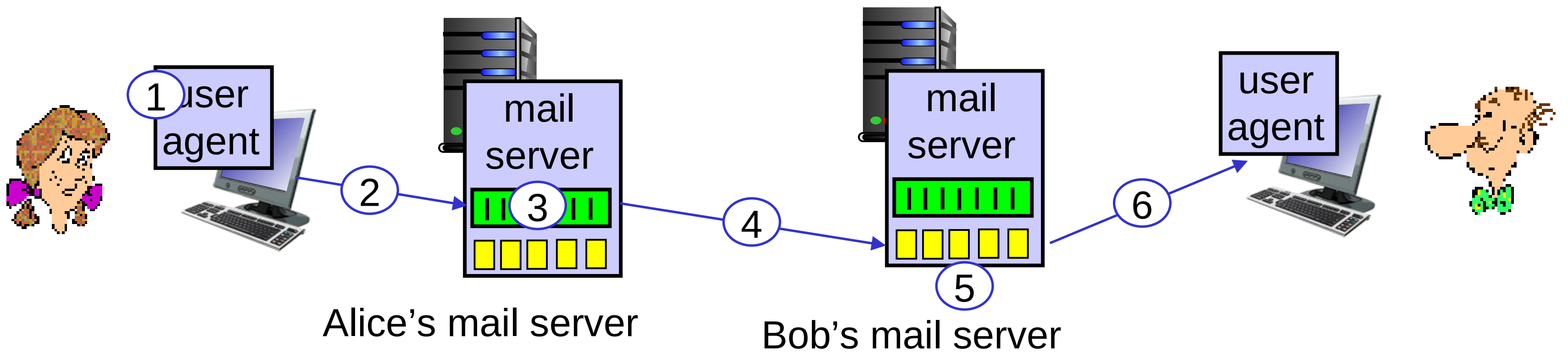


Electronic Mail: SMTP [RFC ~~2821]~~

- 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 HTTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCI

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 queue
3) client side of SMTP opens 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 message in Bob's mailbox
6) Bob invokes his user agent to read message



SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message

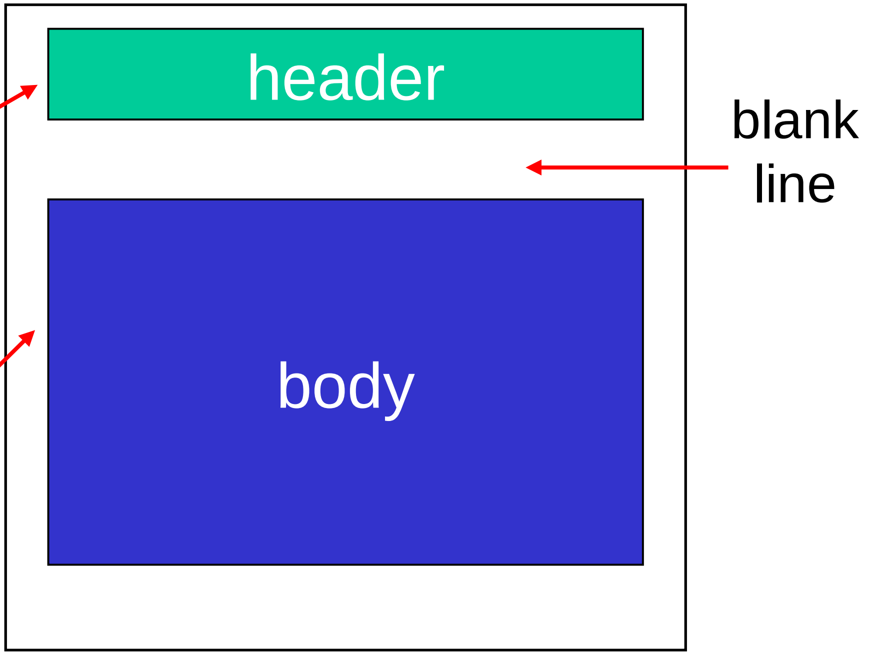
Mail message format

SMTP: protocol for exchanging email messages

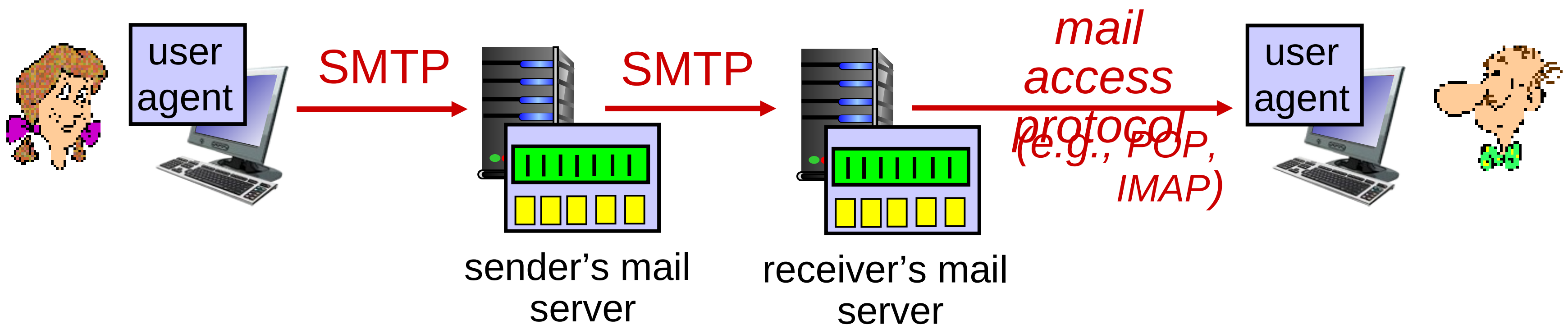
RFC 822: standard for text message format:

- 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/storage to receiver's server
- mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]: authorization, download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored messages on server
  - HTTP: gmail, Hotmail, Yahoo! Mail, etc.

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
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 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-read e-mail if he changes client
- 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 messages in folders
- keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

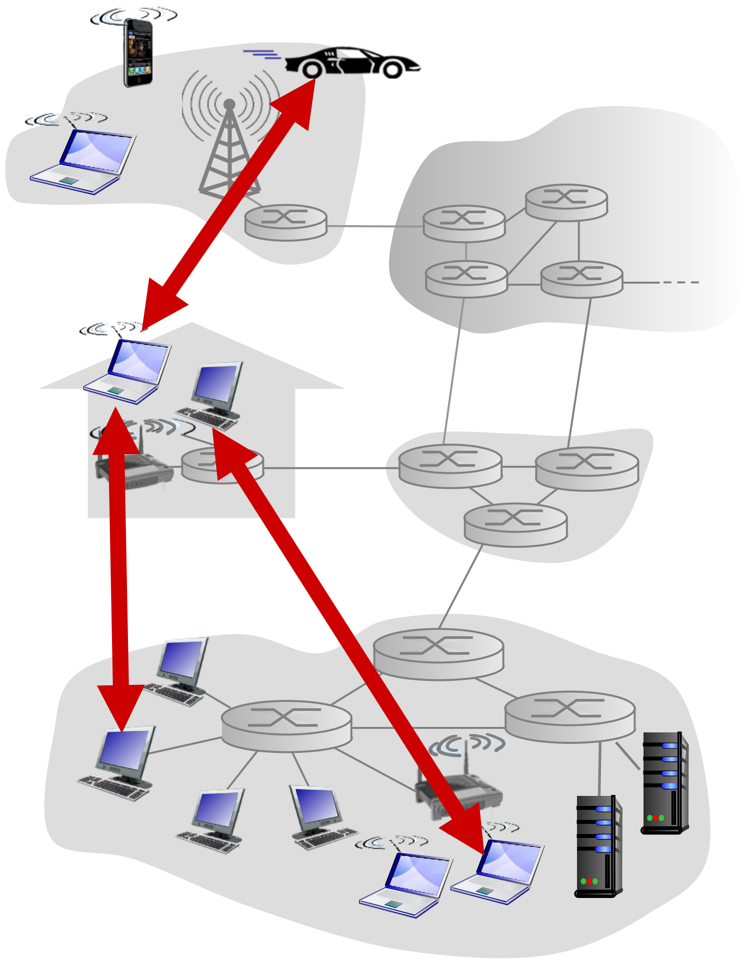
2.5 P2P applications

Pure P2P architecture

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

examples:

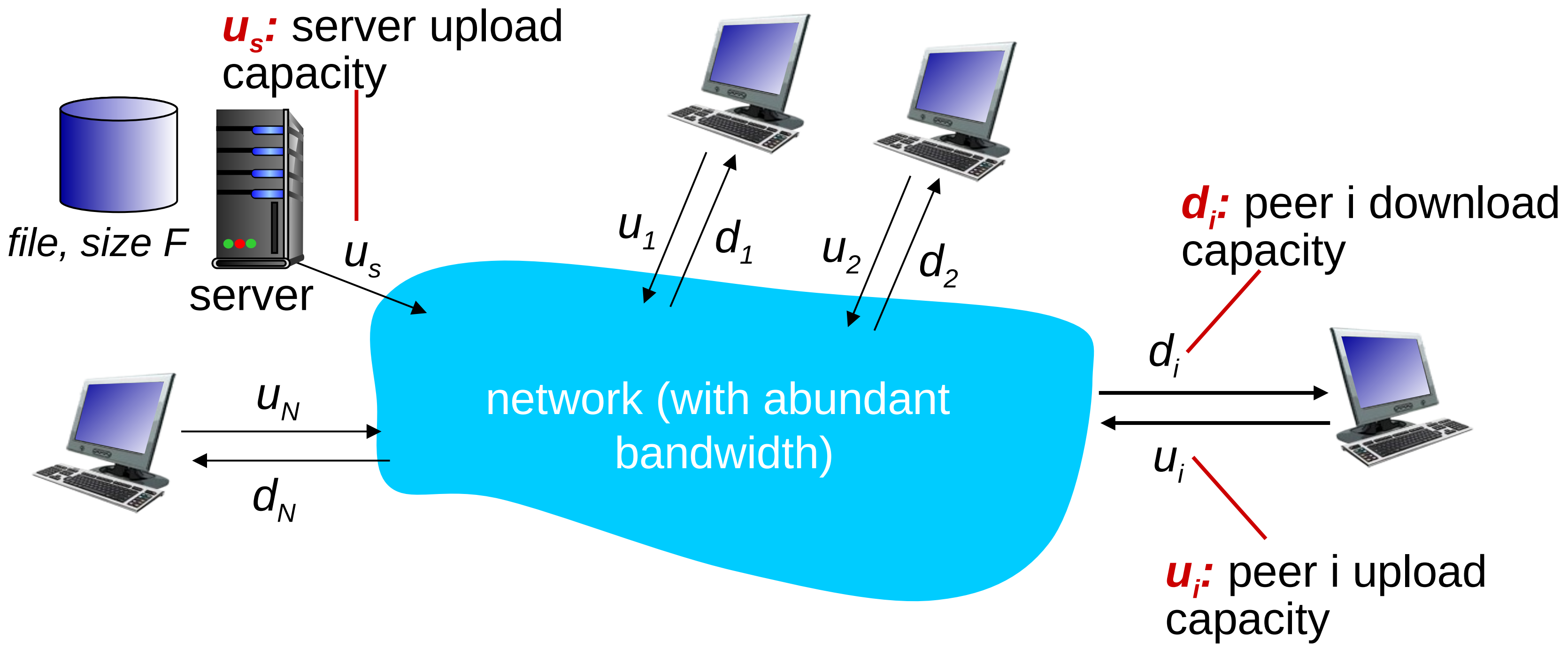
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)



File distribution: client-server vs P2P

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

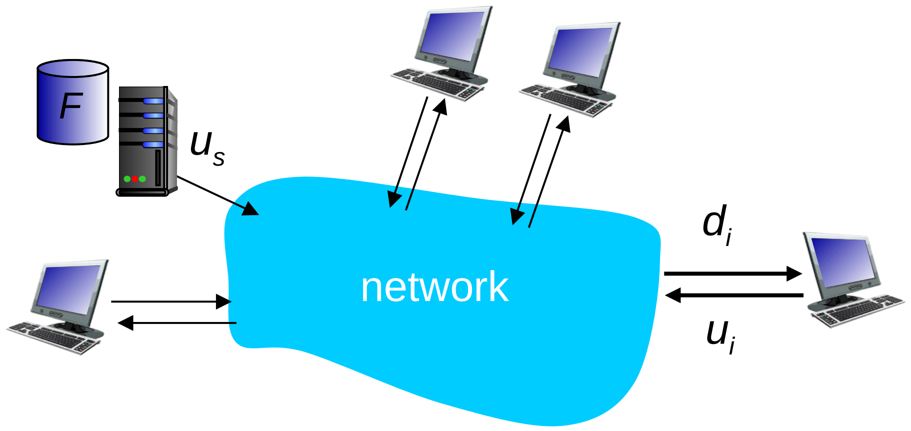
- peer upload/download capacity is limited resource



u_s: server upload capacity
file, size F
server
u_s
u_1
d_1
u_2
d_2
d_i: peer i download capacity
d_i
u_N
d_N
network (with abundant bandwidth)
u_i
u_i: peer i upload capacity

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}

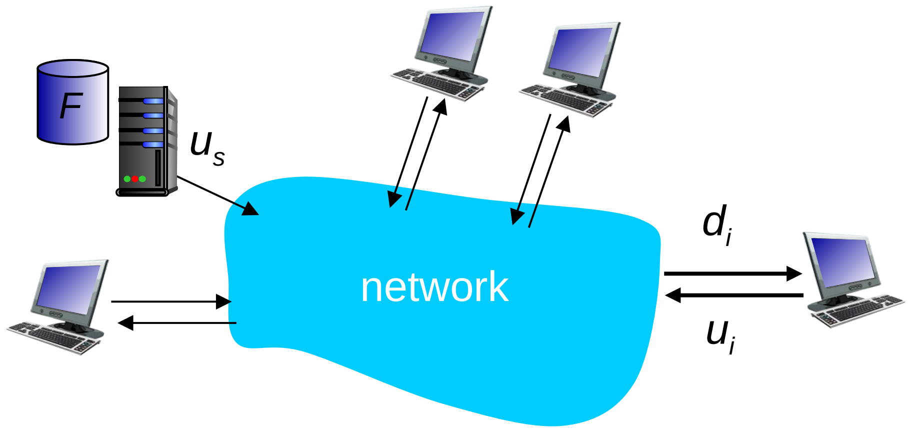
time to distribute F to N clients using client-server approach

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

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:* each client must download file copy
  - min 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

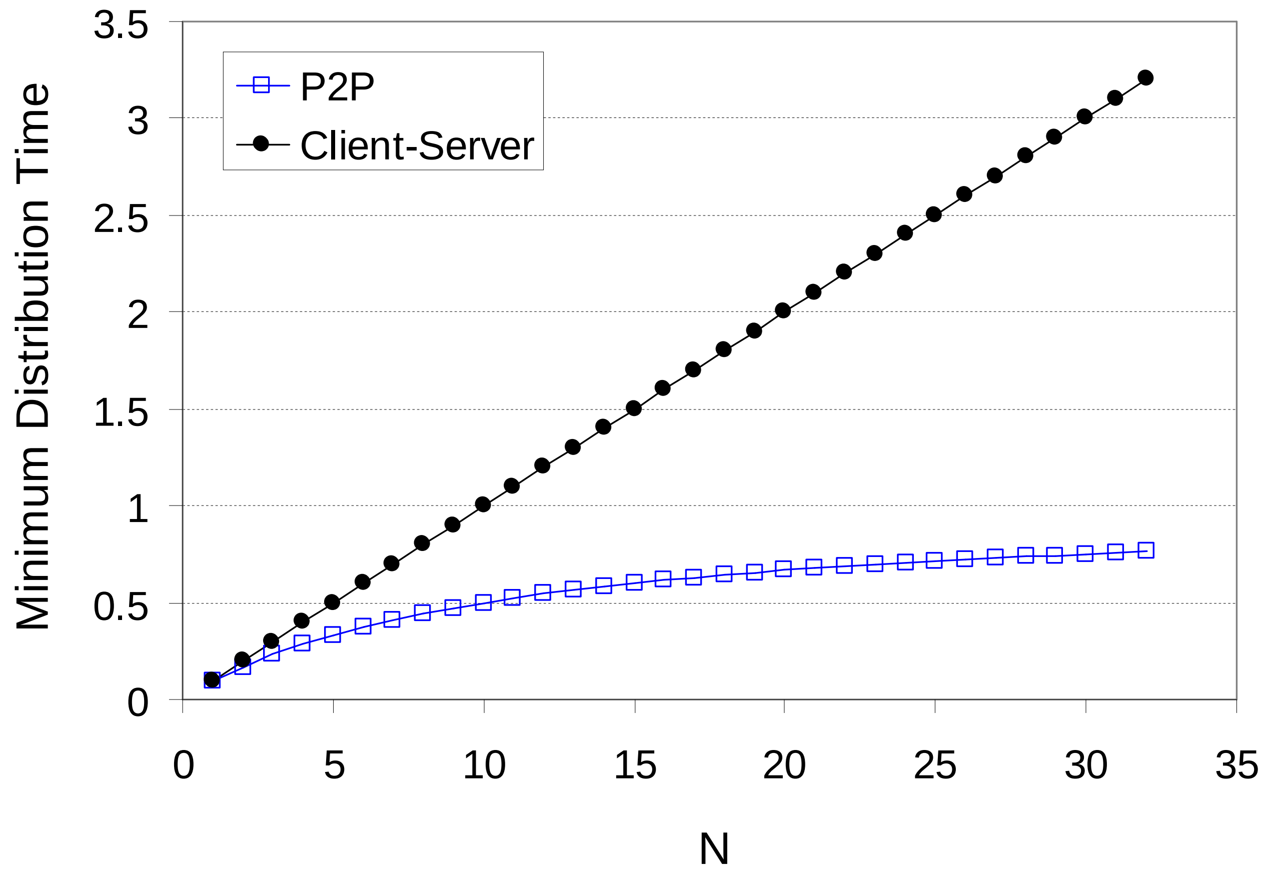
$$D_{P2P} \geq max\{F/u_{s,},F/d_{min,},NF/(u_s + \Sigma u_i)\}$$

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

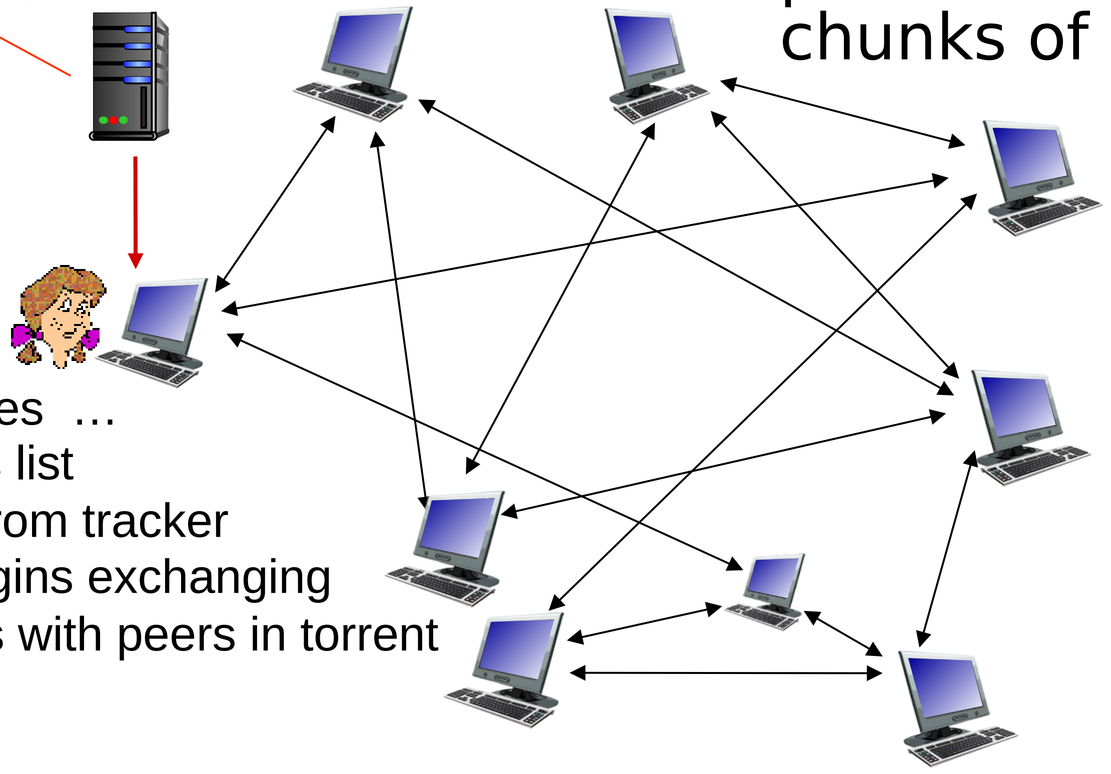


P2P file distribution: ~~BitTorrent~~

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

tracker: tracks peers participating in torrent

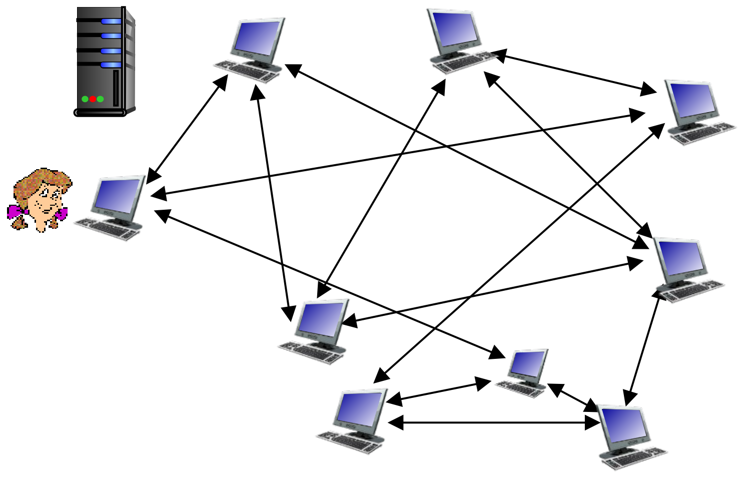
torrent: group of peers exchanging chunks of a file



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 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

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

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:

- stateless vs. stateful
- reliable vs. unreliable message transfer