CS-340 Introduction to Computer Networking

Lecture 3: Application-layer protocols, HTTP

Steve Tarzia

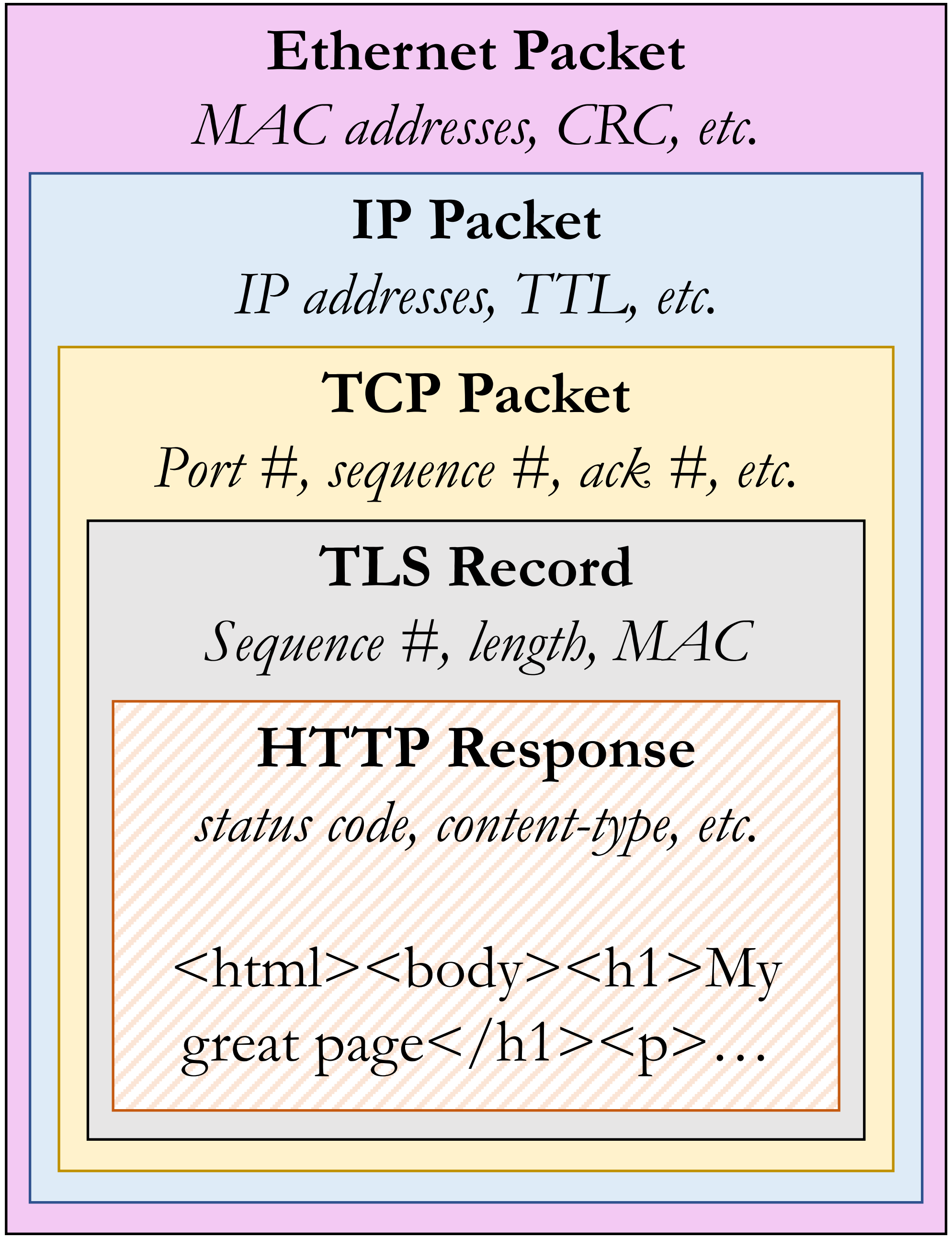
Network diagrams adapted from those by J.F Kurose and K.W. Ross
HTTP slides adapted from website by Chua Hock-Chuan:
https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

Last Lecture

- Packets travel along many *hops* to reach the intended destination
  - Each router has a fixed-size queue; packets are dropped if full
  - Packet is also dropped if a bit-flip error is detected
- Showed four different sources of *packet delay* at each hop:
  - Nodal processing, queueing (associated with the router)
  - Transmission, propagation (associated with the link)
- Internet is a "network of networks"
  - Tier 1 ISPs and big content providers build high-speed *backbone* links.
  - *Peering* is when networks connect to each other without any payment.
- Networks use layered protocols, eg.: Ethernet, IP, TCP, TLS, HTTP
- Socket is a software abstraction of a network connection (TCP or UDP)
  - It's one end of a pipe: you can send data in or get data out
  - Each socket is bound to a particular *port* number. Port number determines which process on a host is responsible for handling a given packet.

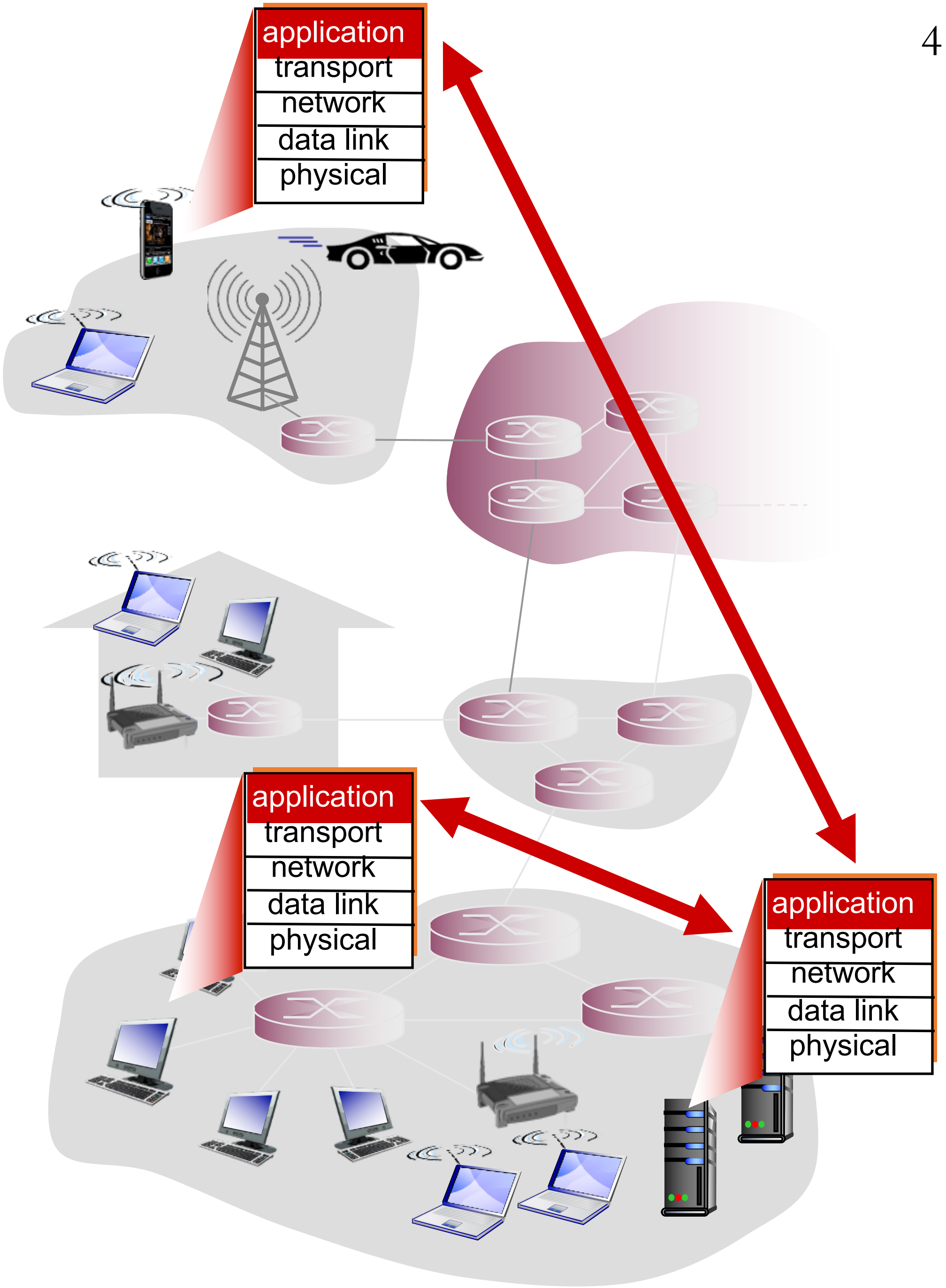
Separation of concerns

- **Link layer**: shares a physical channel among several transmitters/receivers
- **Network layer**: routes from source to destination, along many hops.
- **Transport layer**:
  - Multiplexing >1 connection per machine
  - Ordering, • Acknowledgement, • Pacing
- **Session Security layer**:
  - Encryption, • Authentication.
- **HTTP Application layer**:
  - Resource urls, • Response codes,
  - Caching, • Content-types, • Compression



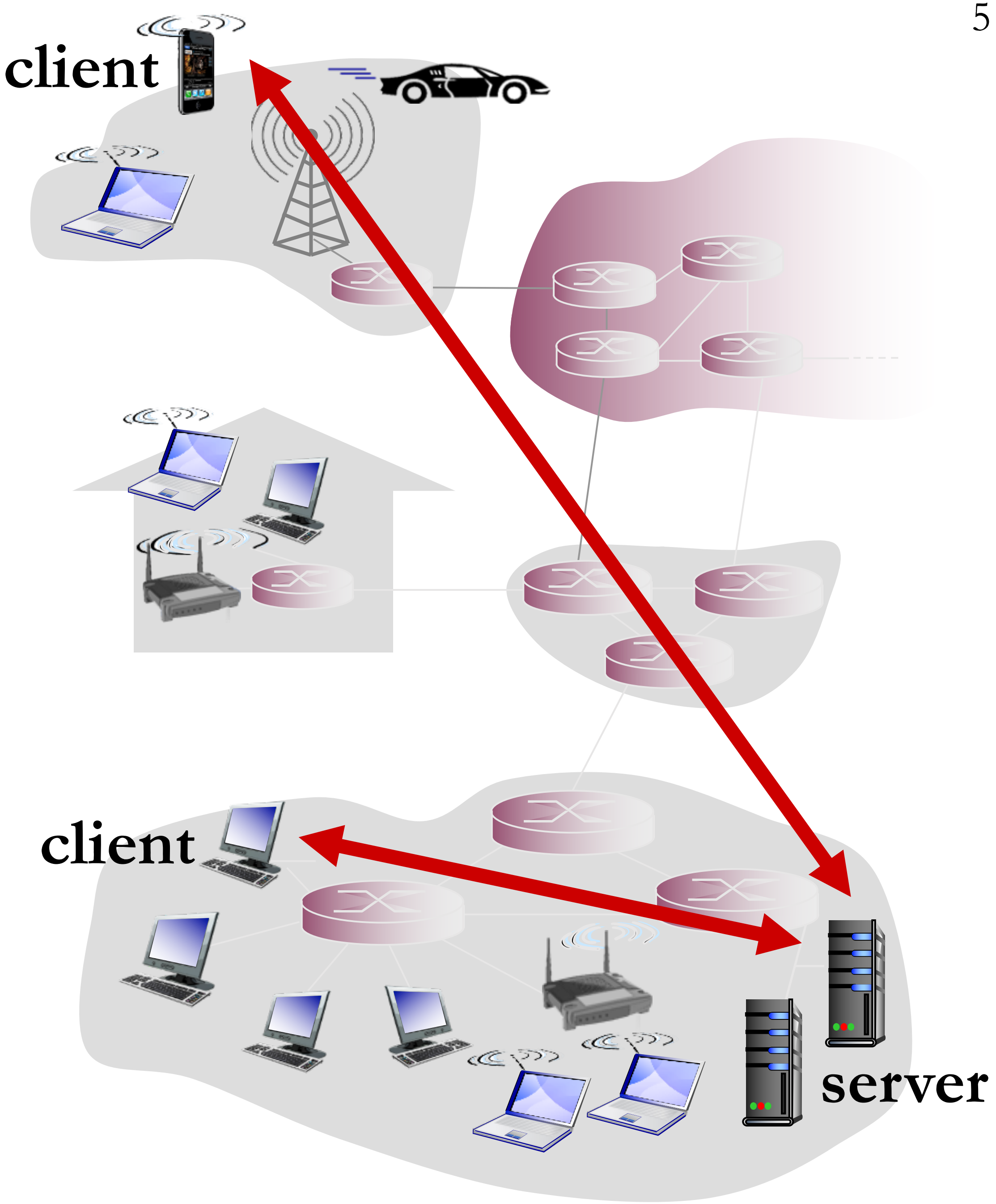
Application-layer protocols

- Purpose is to allow apps running on different computers to communicate.
  - System is called *client-server* or *peer-to-peer* depending on whether it relies on central control (at a server).
- Apps don't worry about low-level details of the network.
- Assume that we can send messages (of arbitrary size) to any host on the network if we know it's address.
  - Every computer has a unique IP address like 34.200.20.23 and domain names like "cs.northwestern.edu" somehow map to IP addresses (using DNS, discussed in next lecture)



Client-server architecture

- ***Servers:*** handle requests
  - Always powered on
  - Permanent IP addresses
  - Usually have a DNS hostname
  - Usually reside in data centers
  - No display, keyboard, or mouse
  - **Listen** for requests from clients.
- ***Clients:*** make requests
  - Opposite of above, in every way.
  - **Do not listen** for *unsolicited* messages
    - Only accept responses to their requests.
  - Do not communicate directly with other clients. Server must *relay* messages.



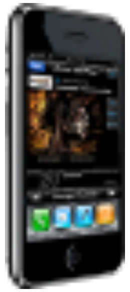
Key difference between client and servers

Servers:

- Do not move!
- Location/address in the network is constant.
- Can listen for requests.

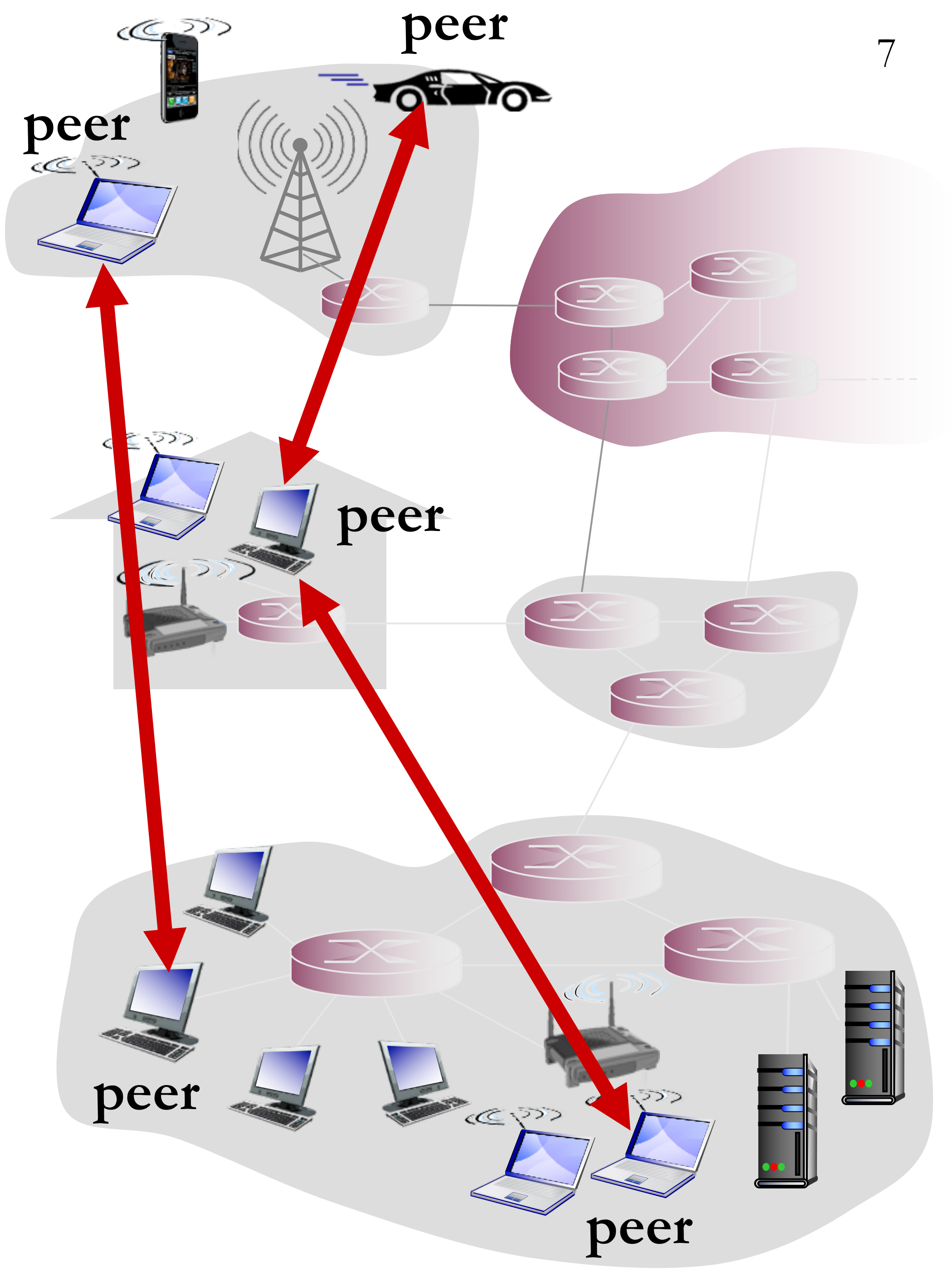
Clients:

- Can move around with users.
- Are difficult to find.
    - Thus, do **not** listen for requests coming from unknown machines.
- Send requests on behalf of user's apps, and listen *briefly* for a response from the one server that was contacted.



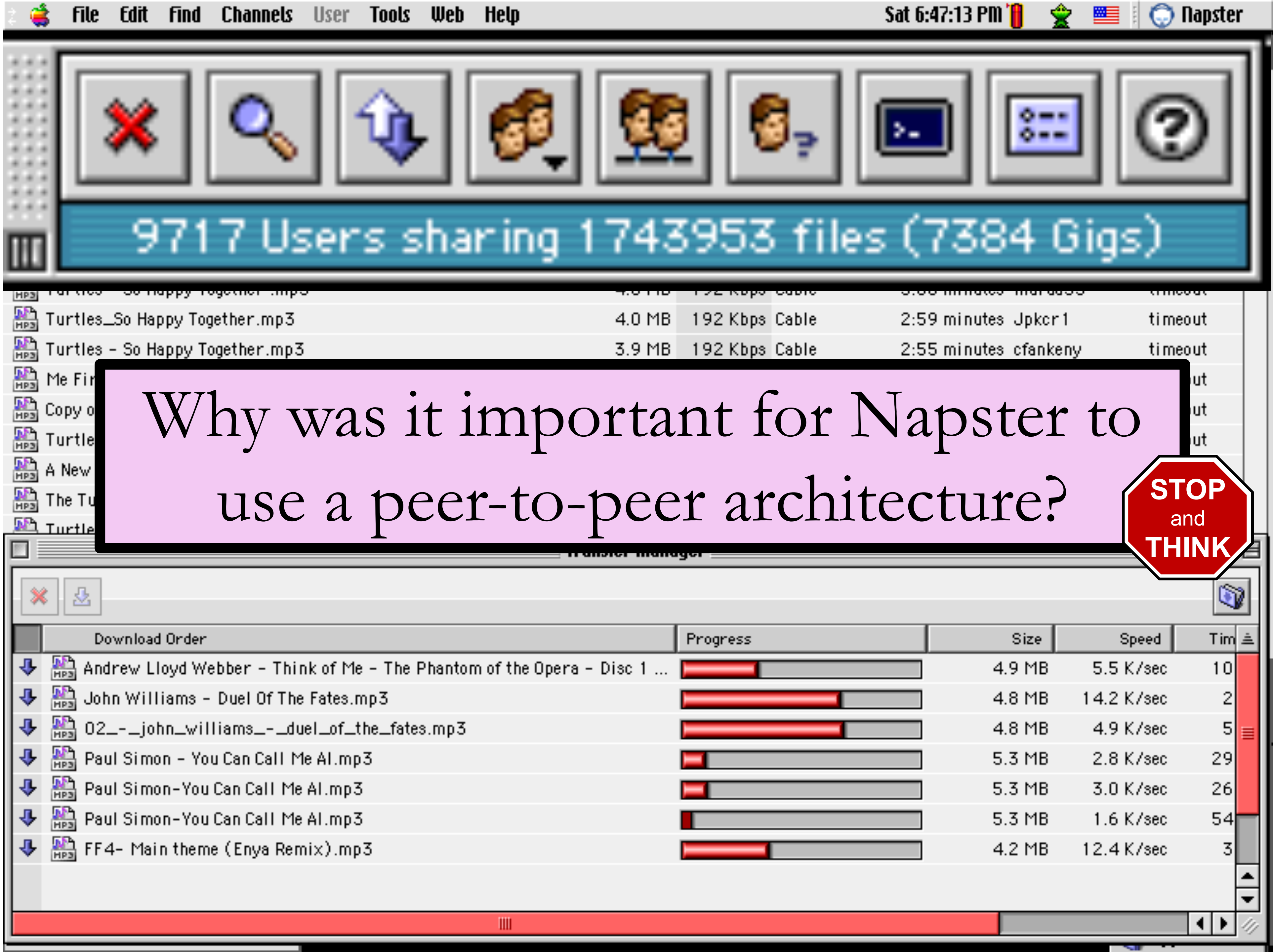
Peer-to-peer architecture (P2P)

- All participants have equal responsibilities, thus are *peers*.
  - Do not rely on powerful, central servers
- A very *scalable* design.
  - Each new participant brings new capacity
- But there are many difficulties:
  - Hosts join and leave the network (*churn*).
  - IP addresses change.
  - Firewalls may block access to peers.
  - Edge networks have limited upload speed.
- Uses kind of centralized directory/tracker.
- Examples: BitTorrent, Skype
  - Might also think of SMTP as P2P



Napster

- A technically innovative P2P app.
- Allowed music *piracy* on massive scale in 2000-2001.
- Shut down after several copyright lawsuits.
- Inspired BitTorrent.



Why was it important for Napster to use a peer-to-peer architecture?

STOP and THINK

Hyper Text Transport Protocol (HTTP)

- HTTP is a client-server data exchange protocol built on top of TCP
  - TCP provides a reliable, bi-directional data stream between two machines.
- HTTP was invented for browsers to fetch pages from webservers
- **Request** specifies:
  - A human-readable header with: *URL*, *method*, (plus some optional headers)
  - An optional *body*, storing raw data (bytes).
- **Response** includes:
  - A human-readable header with *response code*, (plus some optional headers)
  - An optional *body*
- HTTP is a **stateless** protocol:
  - Each request is self-contained – contains all info needed to give a response.
  - Meaning of requests are independent; servers need not remember past requests.

Request:

```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

Request Line
Request Headers
Request Message Header
A blank line separates header & body
Request Message Body
(optional for GET)

Response:

```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

Status Line
Response Headers
Response Message Header
A blank line separates header & body
Response Message Body

From https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

HTTP transaction steps

Notice that “Content-Length” header tells length of message body.

```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

TCP socket.
A bi-directional pipe/stream of bytes

Client

Server


```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```


```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```


```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```


1. Client creates TCP socket, and server accepts the socket.
2. Client writes HTTP request to socket; starts listening for response.
3. Server notices new data on socket and starts reading request data.
4. Server eventually notices that it has received a full HTTP request.
5. Server does some work to generate an appropriate response.
6. Server writes HTTP response to socket.
7. Client reads and parses response data; stops reading after calculating that the response is complete.

HTTP methods and responses

Methods

- **GET**: to request a data
- **POST**: to post data to the server, and perhaps get data back, too.

Less commonly:

- **PUT**: to create a new document on the server.
- **DELETE**: to delete a document.
- **HEAD**: like GET, but just return headers

Response codes

- **200 OK**: success
- **301 Moved Permanently**: redirects to another URL

Client errors (400–499):

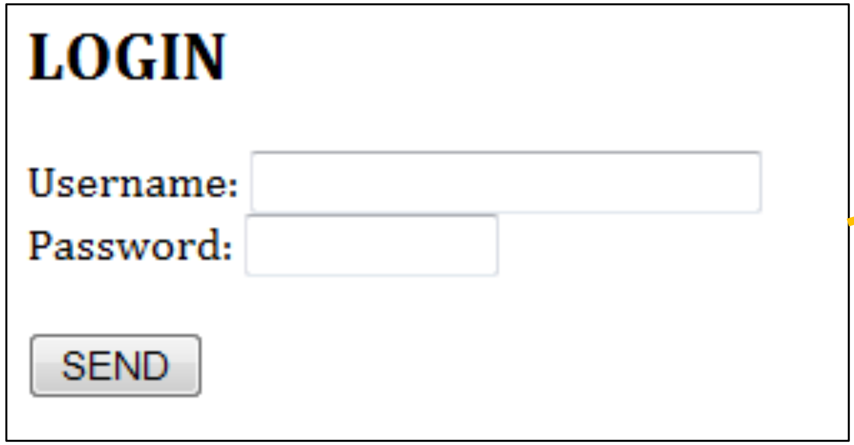
- **403 Forbidden**: lack permission
- **404 Not Found**: URL is bad

Server errors (500-599):

- **500 Internal Server Error**

… and many more

POST method is *often* used when client supplies data




```html
<html><body>
  <h2>LOGIN</h2>
  <form method="post" action="/api/login">

    Username:
    <input type="text" name="user"/><br/>

    Password:
    <input type="password" name="pw"/>
    <br/><br/>

    <input type="hidden"
     name="action" value="login" />

    <input type="submit" value="SEND" />
  </form>
</body></html>
```

Send HTTP POST request when click button

```
POST /api/login HTTP/1.1
Host: somewebsite.com
Accept: image/gif, image/jpeg, */*
Referer: http://somewebsite.com/login.html
Accept-Language: en-us
Content-Type:
  application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (…)
Content-Length: 37
Connection: Keep-Alive
Cache-Control: no-cache

User=Peter+Pan&pw=123456&action=login
```

Response to login request gives user a **cookie**

- Cookies are how web applications track **state,** often to track user identity.
- If username and password were correct, server will return a cookie in the response:

```
HTTP/1.1 302 Found
Location: http://somewebsite.com/account
Set-Cookie: someweb-id=kfj203d14t9s
```

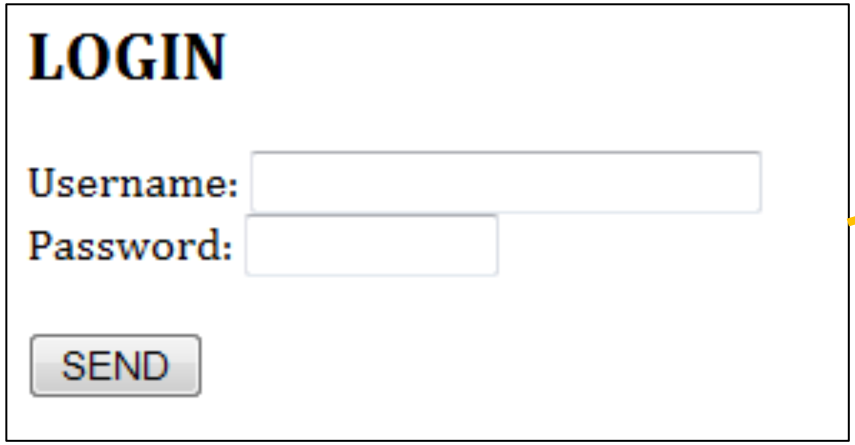
- Response tells the client browser to redirect to `http://somewebsite.com/account`, but it also gives the browser a cookie to remember.
- Browser will include the cookie in all future HTTP requests to `somewebsite.com`:

Is HTTP with cookies still stateless?


```
GET /account HTTP/1.1
Host: somewebsite.com
Referer: http://somewebsite.com/api/login
Cookie: someweb-id=kfj203d14t9s
…
```

- Server getting this request can use the cookie to determine which user it came from!

GET requests can send data in a URL's *query string*




```
<html><body>
  <h2>LOGIN</h2>
    <form method="get" action="/api/login">

    Username:
    <input type="text" name="user"/><br/>

    Password:
    <input type="password" name="pw"/>
    <br/><br/>

    <input type="hidden"
     name="action" value="login" />

    <input type="submit" value="SEND" />
  </form>
</body></html>
```

Send HTTP **GET** request when click button

```
GET /api/login?User=Peter+Lee&pw=123456&ac
tion=login HTTP/1.1
Host: somewebsite.com
Accept: image/gif, image/jpeg, */*
Referer: http://somewebsite.com/login.html
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (…)
Connection: Keep-Alive
Cache-Control: no-cache
```

Notice that some characters must be translated to be compatible with a URL, eg., space become "+" or "%20"

The evolution of HTTP & the Web

- Early 1990s: HTTP was just a document-fetching service
  - Web servers would just serve up *static* HTML and image files (~Project 1).
  - `GET /index.html` → refers to an HTML file stored on the server
- Late 1990s: Web servers ran scripts to generate content on-demand
  - `GET /product/1234` → generates a page using information found in a database relevant to "product 1234" as well as user-specific information.
- 2005+: Javascript allows pages to be interactive (Gmail, Google Maps)
  - AJAX: HTTP request that gets more data w/out re-loading entire page
- 2010s: HTTP spreads beyond web applications
  - HTTP infrastructure is robust:
    - libraries, software, caches, proxies, encryption, compression
  - It's convenient base all client-server, request-response interactions on HTTP.
  - Eg., smartphone-app-to-server, server-to-server

A weather information service (REST API)

HTTP Request

```
GET
http://api.wthr.com/[key]/fore
cast?location=San+Francisco
HTTP/1.1

Accept-Encoding: gzip

Cache-Control: no-cache

Connection: keep-alive
```

HTTP Response

```
HTTP/1.1 200 OK
Content-Length: 2102
Content-Type:
application/json
```


```
{ "wind_dir": "NNW",
  "wind_degrees": 346,
  "wind_mph": 22.0,
  "feelslike_f": "66.3",
  "feelslike_c": "19.1",
  "visibility_mi": "10.0",
  "UV": "5", … }
```

REST API example (REpresentational State Transfer)

- https://petstore.swagger.io/
- https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/post-statuses-update

Inputs and outputs for an API built on top of HTTP

Request Inputs

- Method
  - GET/POST/PUT/DELETE
- URL
  - Usually identifies the type of request, but may also supply parameters.
- Query parameters after URL
- ~~Headers~~
  - ~~Cookies, custom headers~~
- Body
  - Usually form-encoded or JSON

Response Outputs

- Status code
  - 200, 404, 403, etc.
- ~~Headers~~
- Body
  - Usually JSON encoded

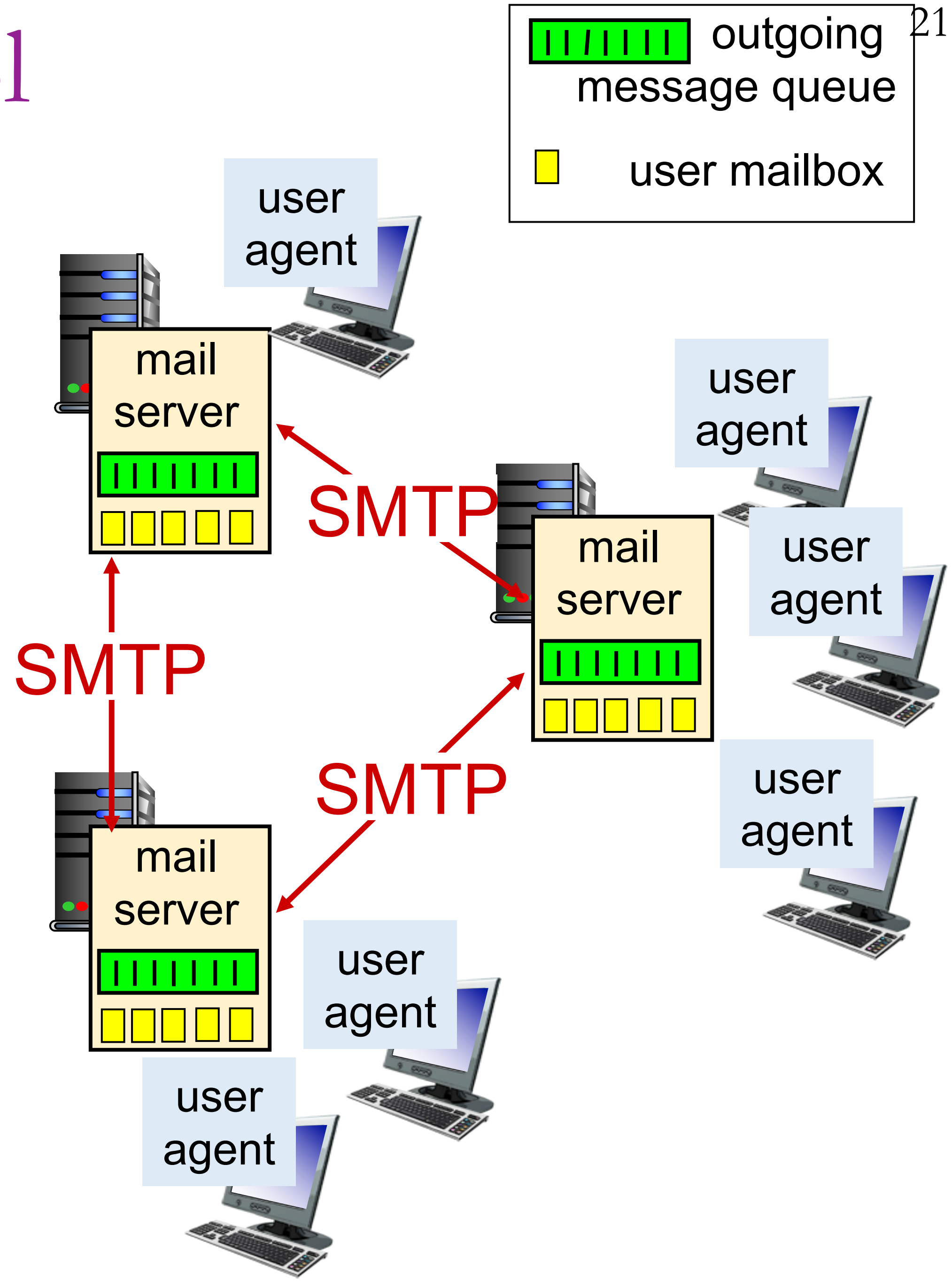
It's bad style to HTTP headers for input/output.
Goal is to build on top of HTTP, not alter it.

Why use HTTP for new applications?

- Web community has already solved the problems you are likely face.
    - Encryption
    - Compression
    - Every programming language already has HTTP client libraries
    - Many different server frameworks to choose from, and these already handle encryption, queueing, database connection pooling:
        - Eg., Apache httpd, Tomcat, Node.js, Django, Flask
    - Web proxies and caches can be reused (Squid, Nginx, Akamai, etc.)
    - HTTP response codes are generic enough to be adapted to other services.
- Disadvantages:
    - Inherit some unneeded complexities, and perhaps unexpected behaviors.
    - Human-readable headers introduce overhead (but compression helps)
    - May have to rethink your API to fit the URL/resource model.

Simple Mail Transport Protocol

- Another protocol built on top of TCP.
- Defined in [RFC 2821](#).
- Uses port 25 by default.
- Developed in 1982, earlier than HTTP: Internet's first popular app.
- SMTP is a P2P protocol used by mail servers to exchange users' messages.
- *Mail servers* act as clients when sending, and as servers when receiving.
  - Each domain has its own mail server(s).
- *User agents* use different protocols to fetch emails (IMAP, POP3, webmail)



Example

S: means server
C: means client

```
S: 220 smtp.example.com ESMTP Postfix
C: HELO relay.example.com
S: 250 smtp.example.com, I am glad to meet you
C: MAIL FROM:<bob@example.com>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: RCPT TO:<theboss@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" <bob@example.com>
C: To: "Alice Example" <alice@example.com>
C: Cc: theboss@example.com
C: Date: Tue, 15 January 2008 16:02:43 -0500
C: Subject: Test message
C:
C: Hello Alice.
C: This is a test message with 5 header fields and 4 lines in the message body.
C: Your friend,
C: Bob
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
{The server closes the connection}
```

How is this different than HTTP?

It's stateful.

SMTP telnet demo

Try SMTP for yourself

It’s one of the simplest protocols

- `$ telnet <servername> 25`
  - *telnet* command is available on murphy.wot.eecs.northwestern.edu.
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands
- This lets you send email without a mail app (or an email account!)

- Few SMTP servers will relay arbitrary messages
- Try connecting to the specific SMTP server for the recipient:
- `$ nslookup -type=MX u.northwestern.edu`
  - Returns: `aspmx.l.google.com`
- However, your message will likely end up in the “junk” folder

Recap

- Application-layer protocols are usually built on top of TCP
  - Don't have to worry about network itself, just create socket connections to other hosts. The socket hides many details from the app.
- Most applications use a *client-server* architecture: request-response.
- A *peer-to-peer* architecture is more scalable, but difficult to organize.
- *HTTP* was invented for fetching documents from web servers.
  - It's now used as the basis for many request-response interactions.
  - URLs, request method, response status, human-readable headers, body
  - REST APIs are built on top of HTTP, so it's a networking layer itself.
- *SMTP* is an earlier application-layer protocol, for sending email.
  - Unlike HTTP, it's *stateful* (server must remember what you previously said).