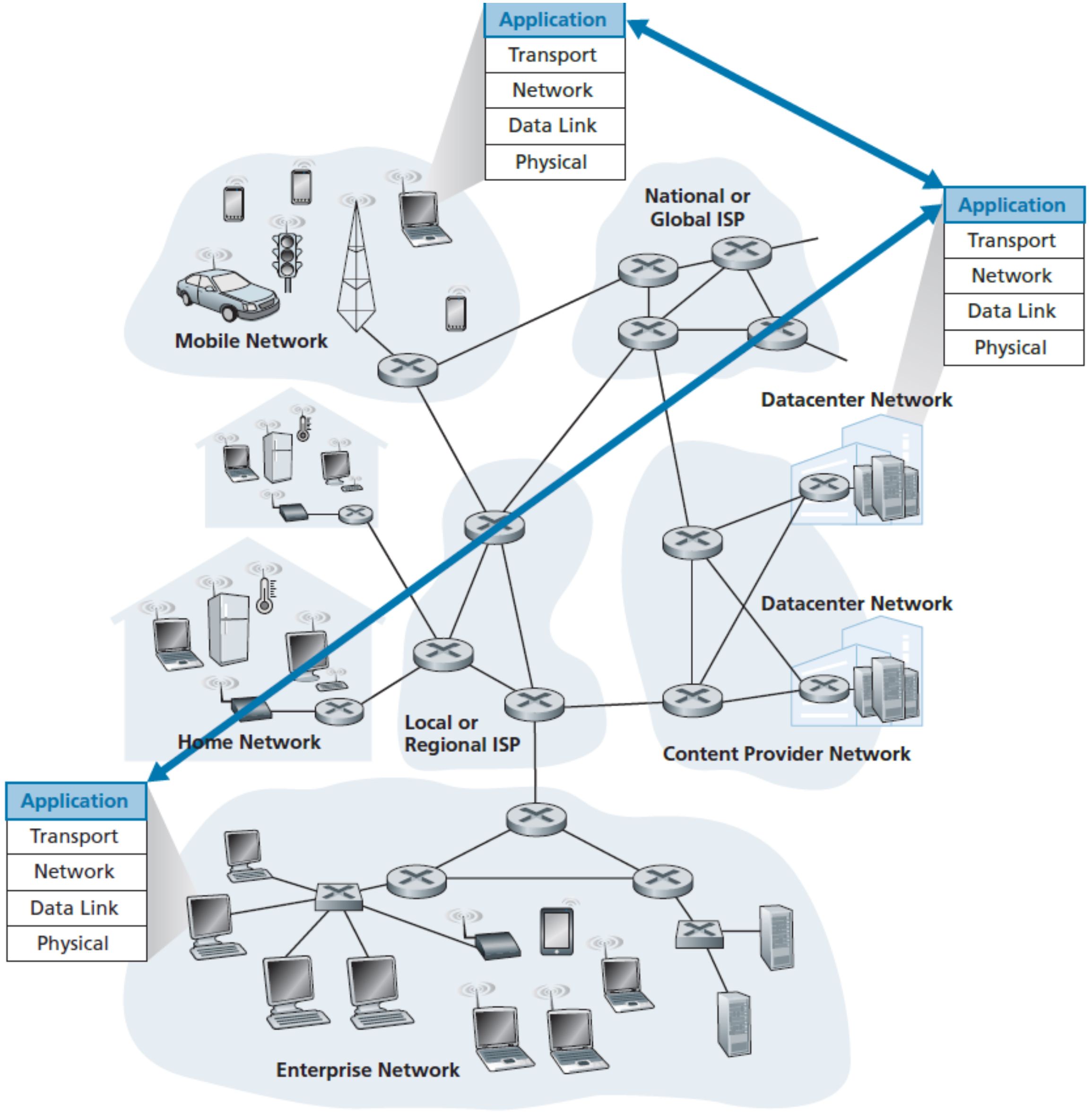
Application layer: overview

- 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

Some network apps

need networking infrastructure and protocols *(since we use network applications), which way to explain?*

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video (YouTube, Netflix)
- P2P file sharing
- voice over IP (Skype)
- real-time video conferencing (Zoom)
- Internet search
- remote login
- …



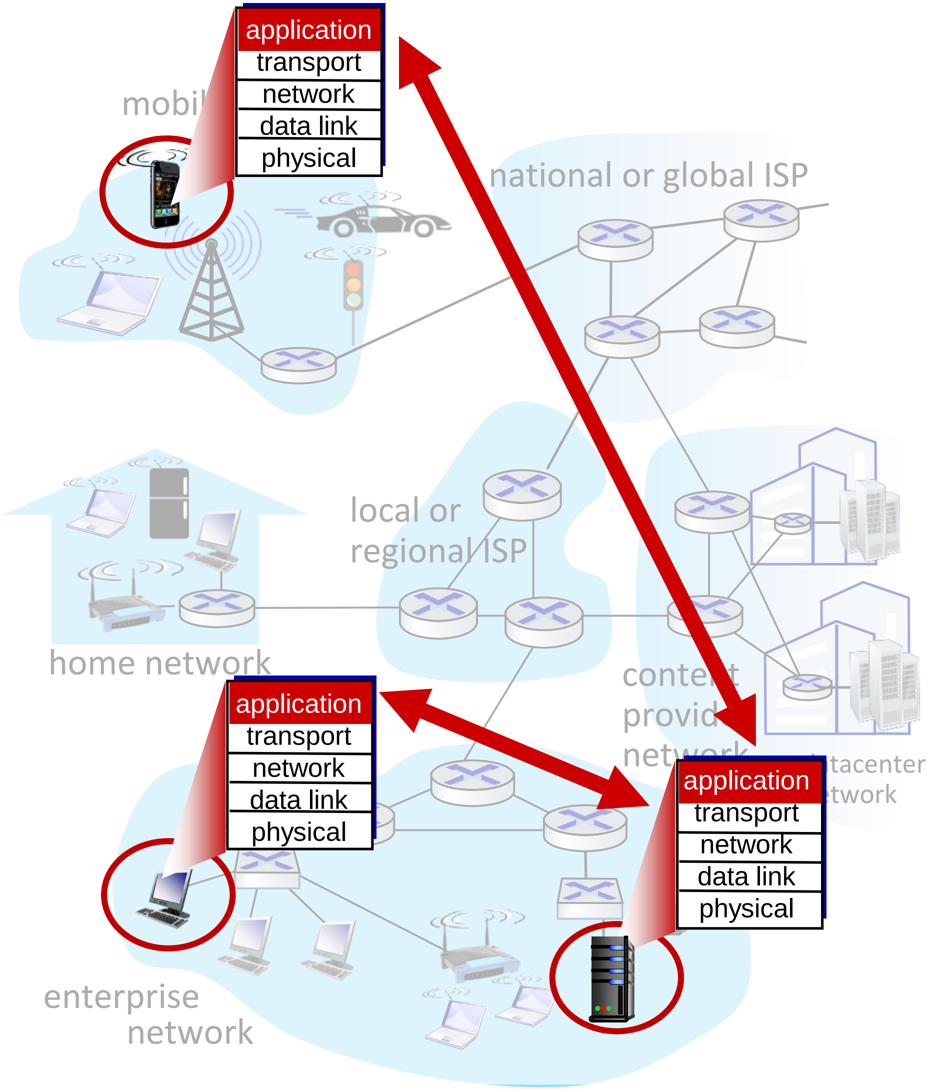
Creating a network app

writing programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software, Netflix-provided client program and and a Netflix server program

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



Creating a network app

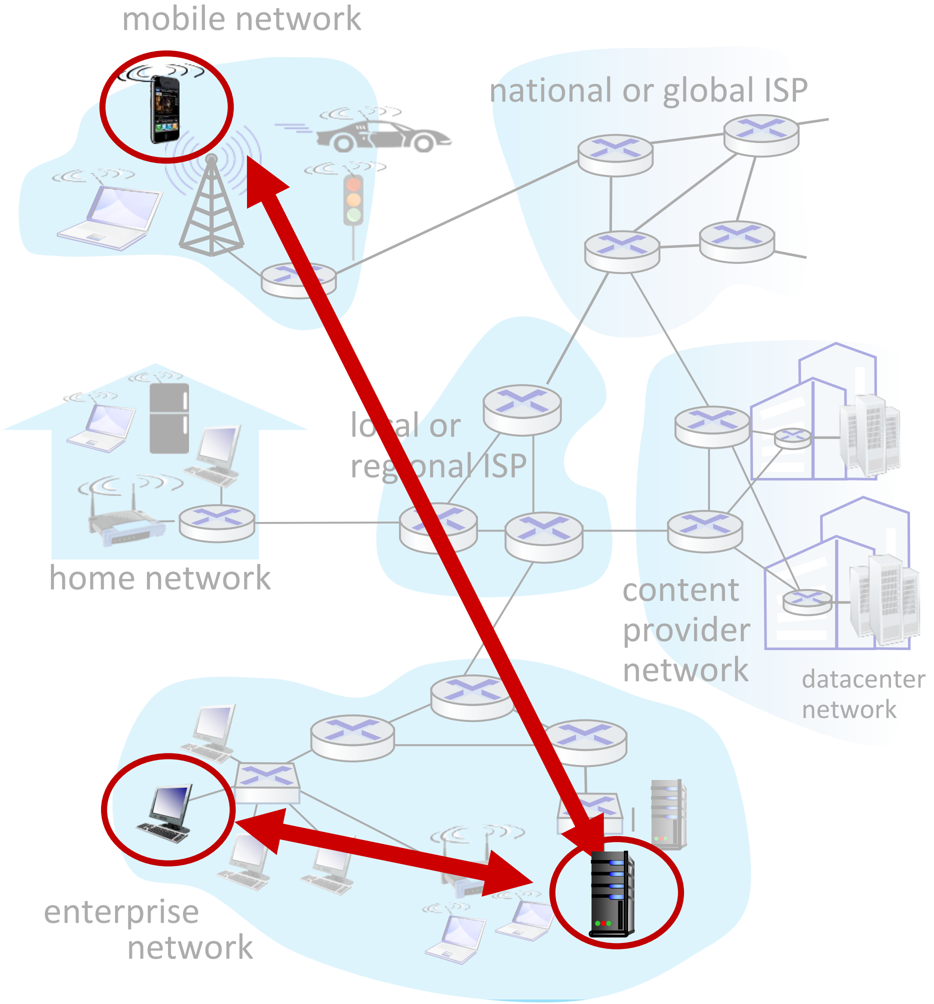
Before starting coding, have an architectural plan for the application.

Application's architecture is different from the network architecture.

For an application developer, the network architecture is fixed *(e.g. five-layered model and the provided services via the protocols)*.

Application architecture: designed by the developer, about how the application is structured over the end systems.

Two common types: client-server and peer-to-peer (P2P).



Client-server paradigm

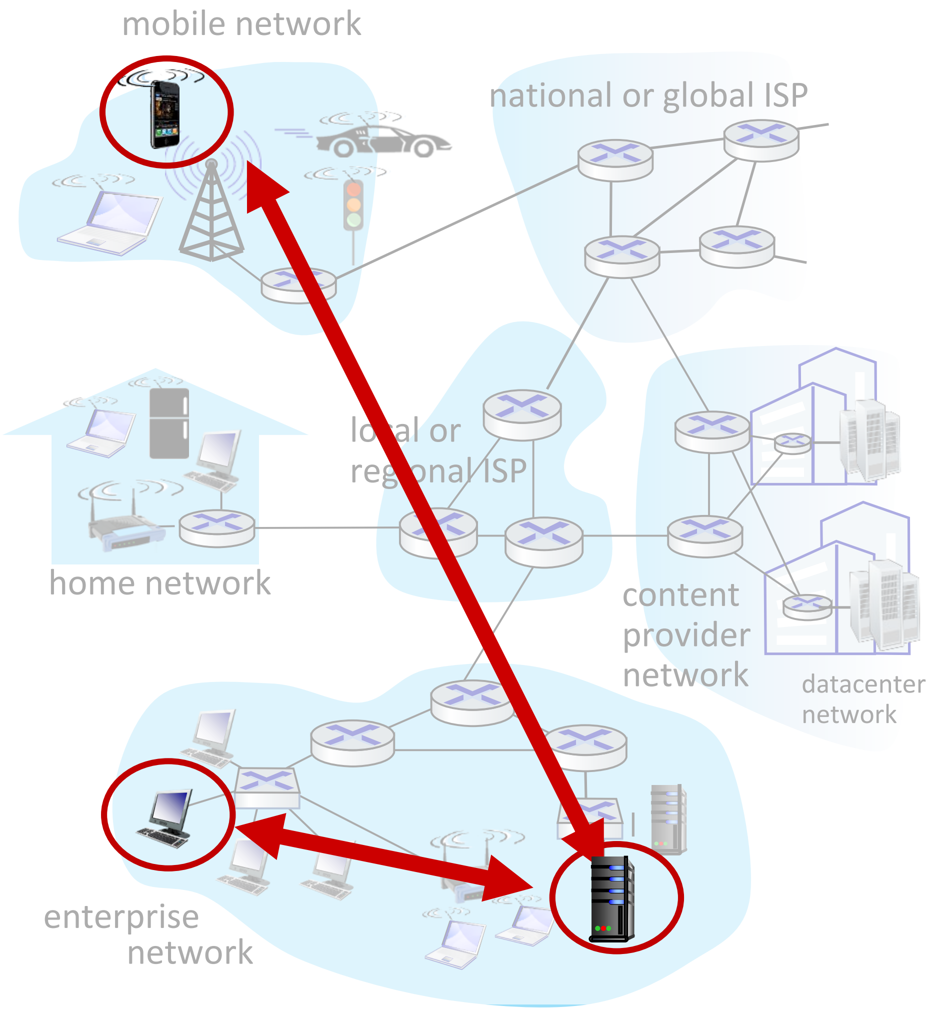
server:

- always-on host
- permanent IP address
- often in data centers, for scaling
- need for multiple services, NLB

clients:

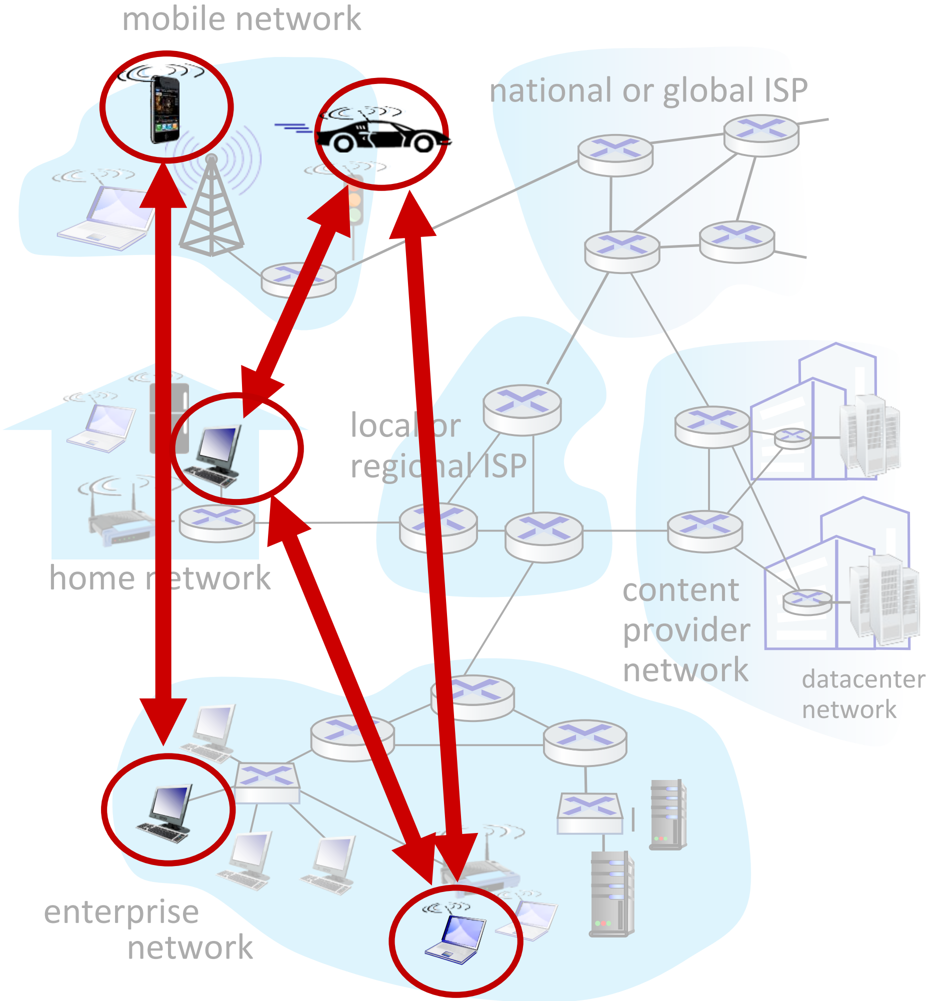
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



Peer-peer 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, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management, security, performance, and reliability issues
  - scalability, cost-effectiveness
- example: P2P file sharing *(each peer adds service capacity to the overall system by distributing files to other peers)*



Processes communicating

Firstly, understand of how the programs, running in multiple end systems, communicate with each other.

For OS's, processes communicate.

Process: a program running in a host.

Different processes on a single host communicate via interprocess communication, rules are governed by OS.

Processes on different end systems communicate by exchanging messages across the network.

Processes communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

clients, servers

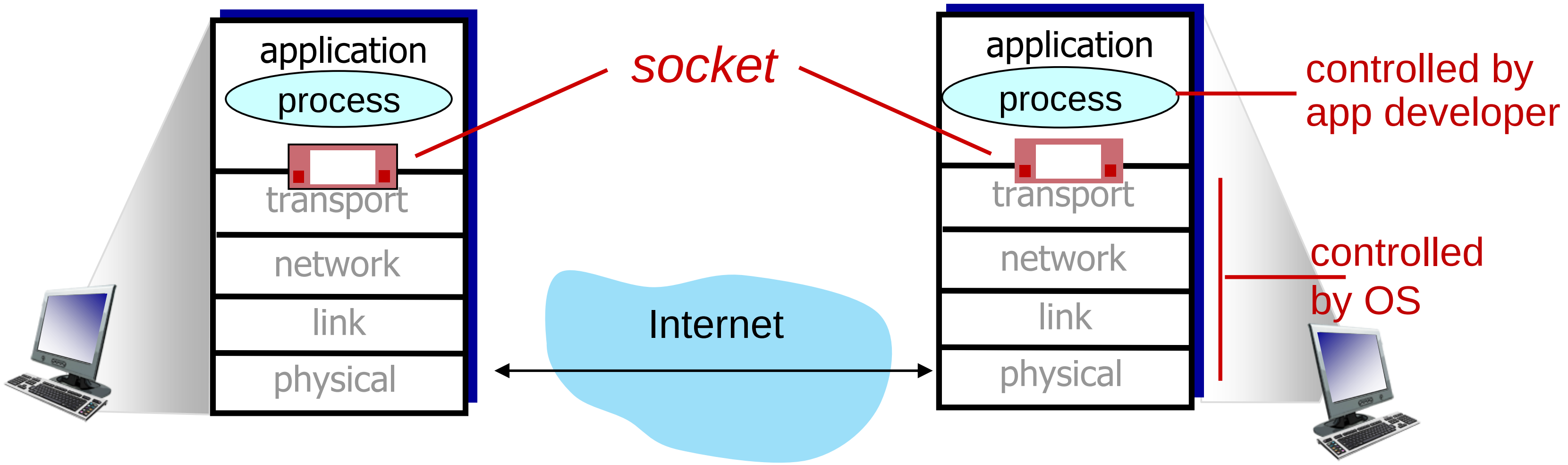
client process: process that initiates communication

server process: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes *(peers downloading and uoloading)*

Sockets

- process sends/receives messages to/from through its socket *(a software interface)*
- interface between the application layer and the transport layer within a host
- API
- developer controls application side, OS controls transport side
- process analogous to house, socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure
  - two sockets involved: one on each side



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80

An application-layer protocol defines:

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype, Zoom

What transport service does an app need?

More than one protocols, how to evaluate and pick one?

reliability (correctly and completely delivery)

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio, multimedia) can tolerate some loss – loss tolerant

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

throughput: a guaranteed available throughput, fixed rate

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective" – bandwidth sensitive
- other apps ("elastic apps") make use of whatever throughput they get

security

- encryption, data integrity, …

Transport service requirements: common apps

TCP or UDP?

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process *(without error and in the proper order)*
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *connection-oriented:* setup required between client and server processes
- *does not provide:* timing, minimum throughput guarantee, security

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Internet applications, and transport protocols

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP – session initiation [RFC 3261], RTP – real time transport [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Securing TCP

TCP does not provide security by itself, can be enhanced at the app. layer

Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in clear text

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn
- cleartext sent into "socket" traverse Internet *encrypted*

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

Web and HTTP

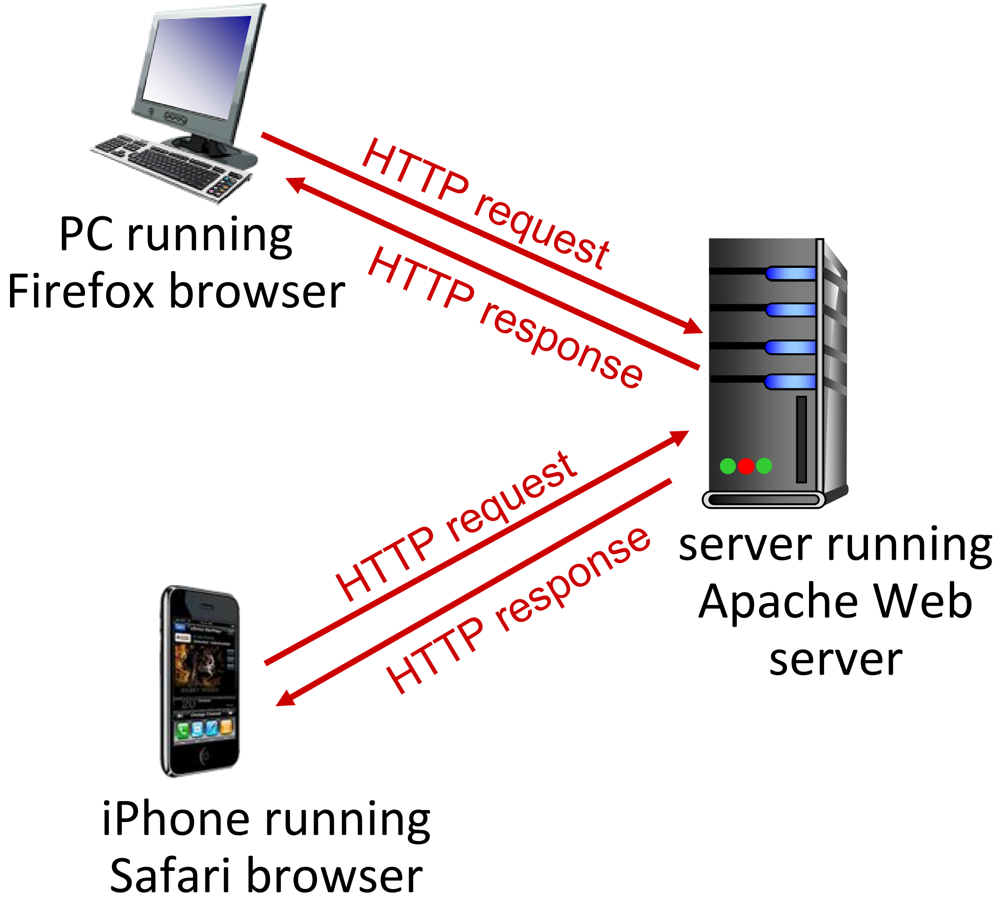
- an application-layer protocol defines ‘how an application’s processes, running on different end systems, pass messages to each other’
- HTTP is an application layer protocol, web is a client-server application (network app.)
- defines the format and sequence of messages (between browser and Web server)
- implemented in two programs: client and a server
- web page (document) consists of *objects,* each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL,* e.g.,

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\texttt{/}\underbrace{\texttt{someDept/pic.gif}}_{\text{path name}}$$

HTTP overview

HTTP: hypertext transfer protocol

- Web's application-layer protocol
- client/server model:
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



HTTP overview (continued)

HTTP uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains *no* information about past client requests (if client asks for the same object twice, server resends the object)

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections: two types

Non-persistent HTTP

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

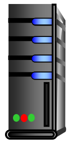
Persistent HTTP

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

Non-persistent HTTP: example

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80



1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80 "accepts" connection, notifying client

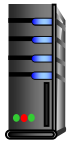
2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Non-persistent HTTP: example (cont.)

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)



4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects

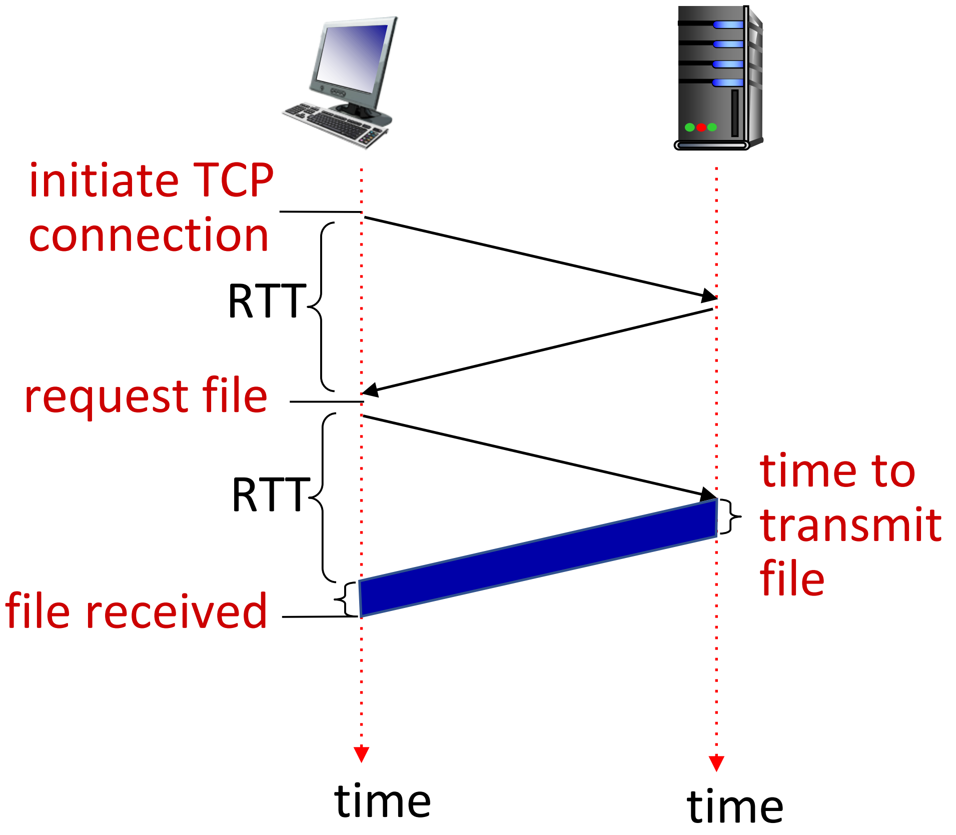
time

The slides are based on the slides by Computer Networking: A Top-Down Approach 8th edition, Jim Kurose, Keith Ross, Pearson, 2020

Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back, includes packet delays

HTTP response time (per object):
- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



Non-persistent HTTP response time = 2RTT+ file transmission time

Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

Persistent HTTP (HTTP1.1): default one

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)

HTTP request message

- two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

- first line: req. line, three fields: method, URL, and HTTP version
- method: GET, POST, HEAD, PUT, or DELETE
- GET: requested object is identified in URL
- subsequent lines: header lines

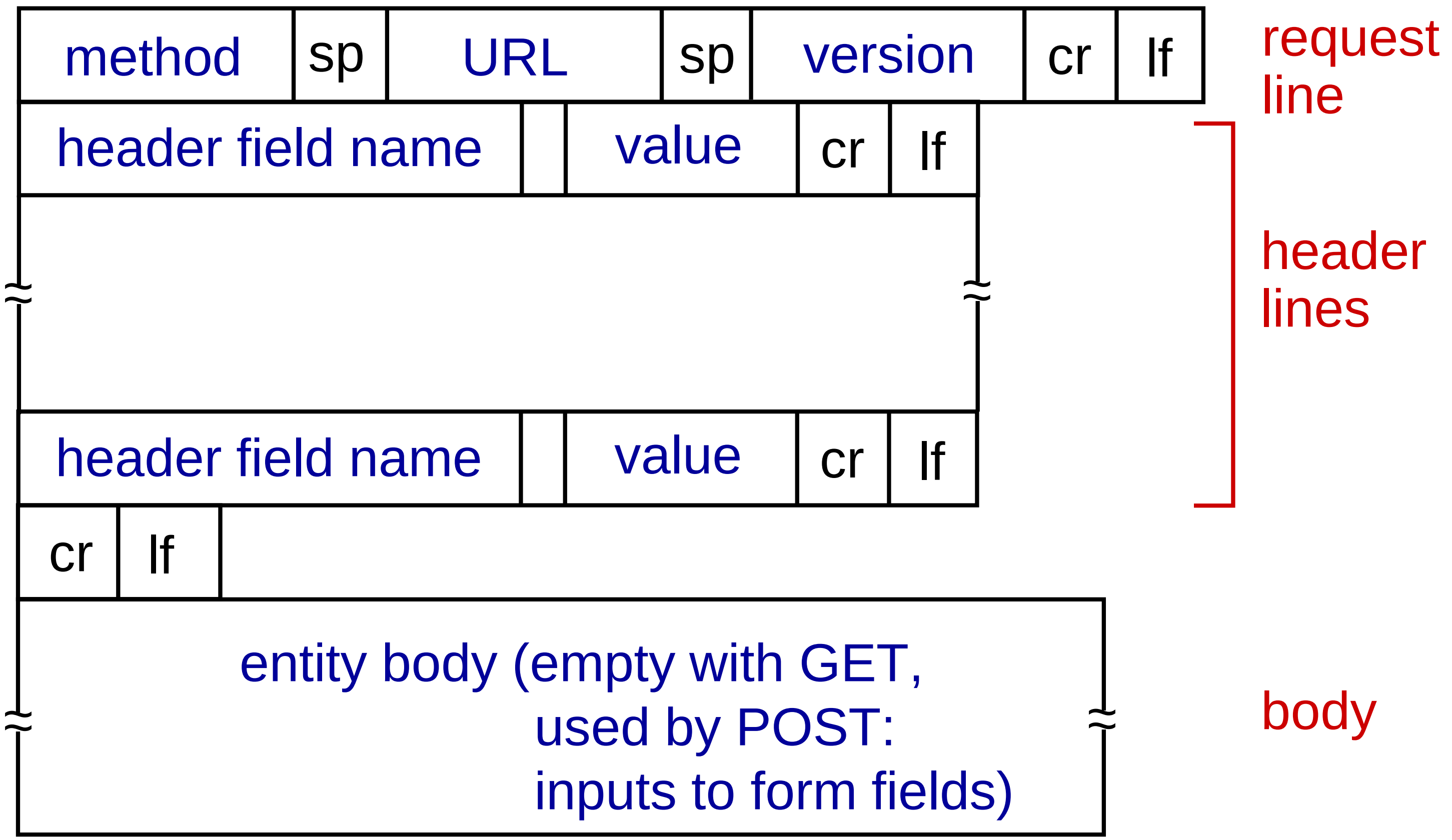
  - host: web server

  - conn. close: 'close of conn. after sending the requested object' claimed from the server

  - UA: browser type

  - Lang.: preferred version of the object requested

HTTP request message: general format



Other HTTP request messages

POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

www.somesite.com/search?name&telephoneNumber

HEAD method:

- for debugging
- server responds with an HTTP message but it leaves out the requested object

PUT method:

- uploads new file (object) to server

DELETE method:

- deletes an object on a server

HTTP response message

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```

- three sections: an initial status line, header lines, and then the entity body
- entity body contains the requested object
- status line has three fields: the protocol version field, a status code, and a corresponding status message
- Connection: close header line: server will close the TCP connection after sending
- Date: header line: the time and date when the HTTP response was created and sent
- Server: header line: the Web server
- Last-Modified: header line: the time and date when the object was created (or last modified)
- Content-Length: header line: the number of bytes in the object
- Content-Type: header line: object in the entity body is HTML text

HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently

- requested object moved, new location specified later in this message (in Location: field), client software will automatically retrieve the new URL

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Maintaining user/server state: cookies

(since HTTP is stateless)

Web sites and client browser use *cookies* to maintain some state between transactions, sites can keep track of users

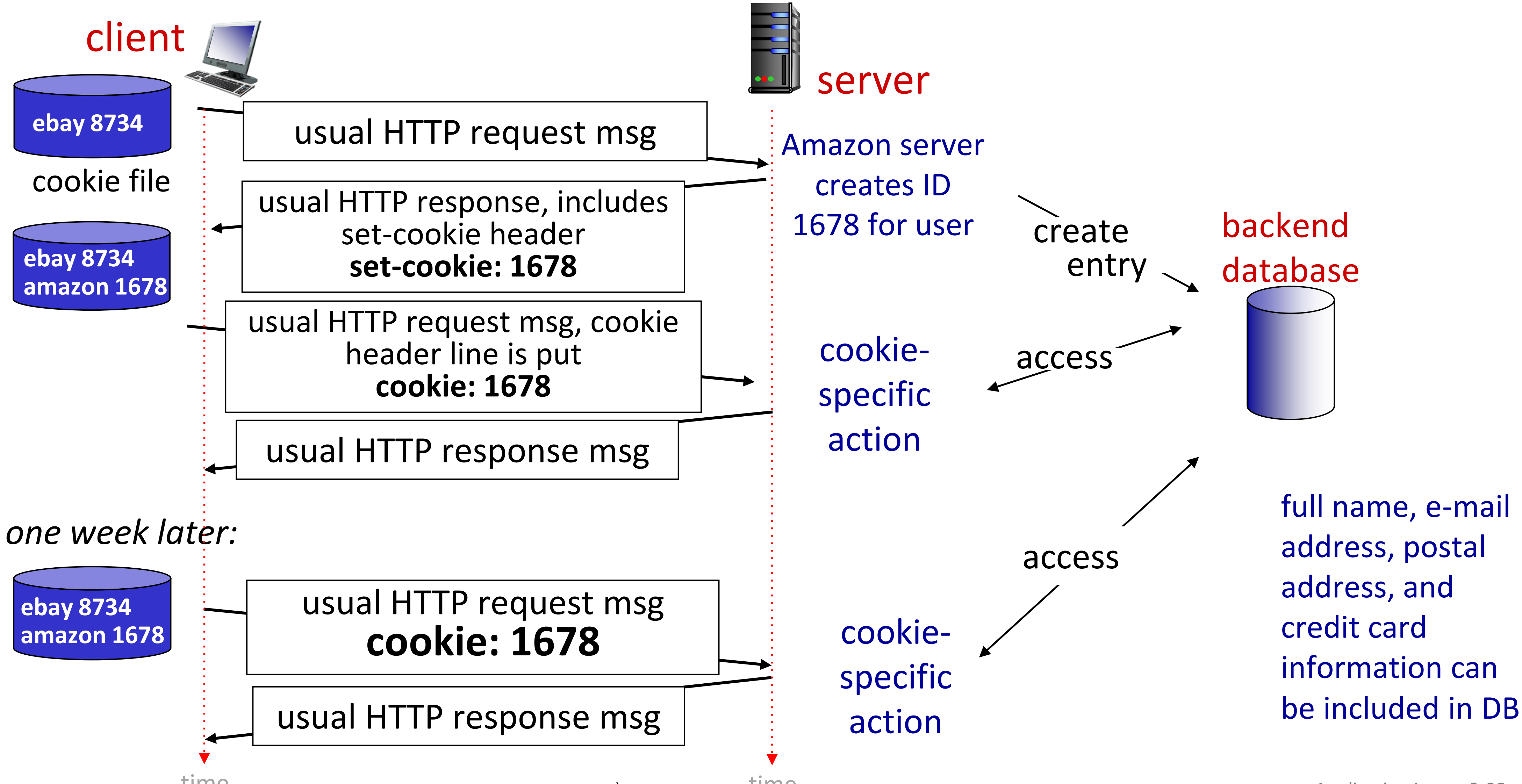
four components:

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 uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka "cookie")
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

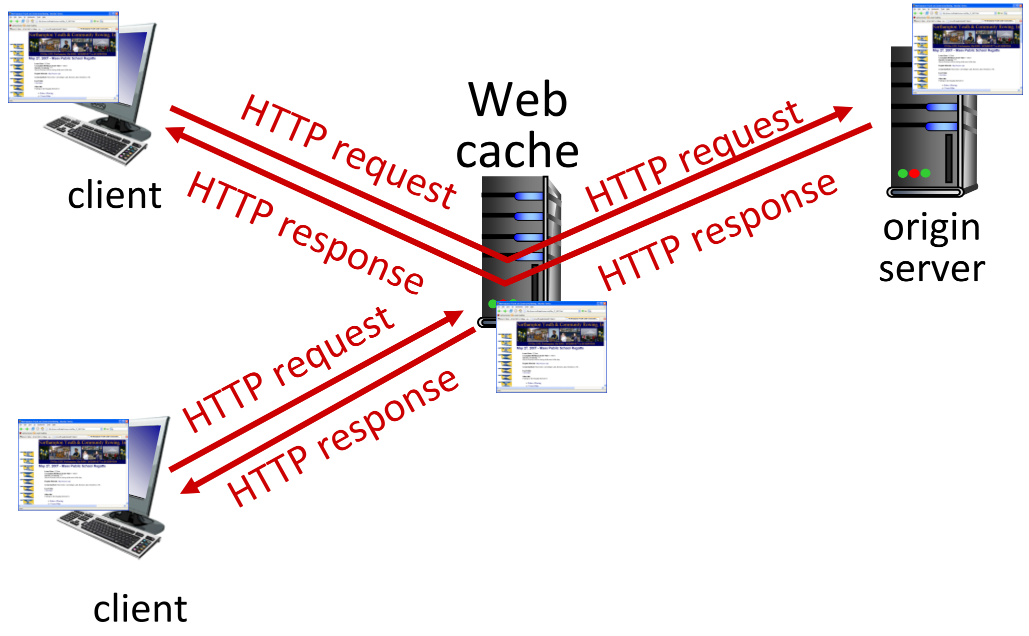
Maintaining user/server state: cookies



Web caches (proxy)

Goal: satisfy client requests without involving origin server, respond to HTTP requests on the behalf of a Web server

- receives the object, stores a copy in its local storage and sends a copy to the client
- user configures browser to point to a (local) *Web cache*
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, then returns object to client



HTTP/2

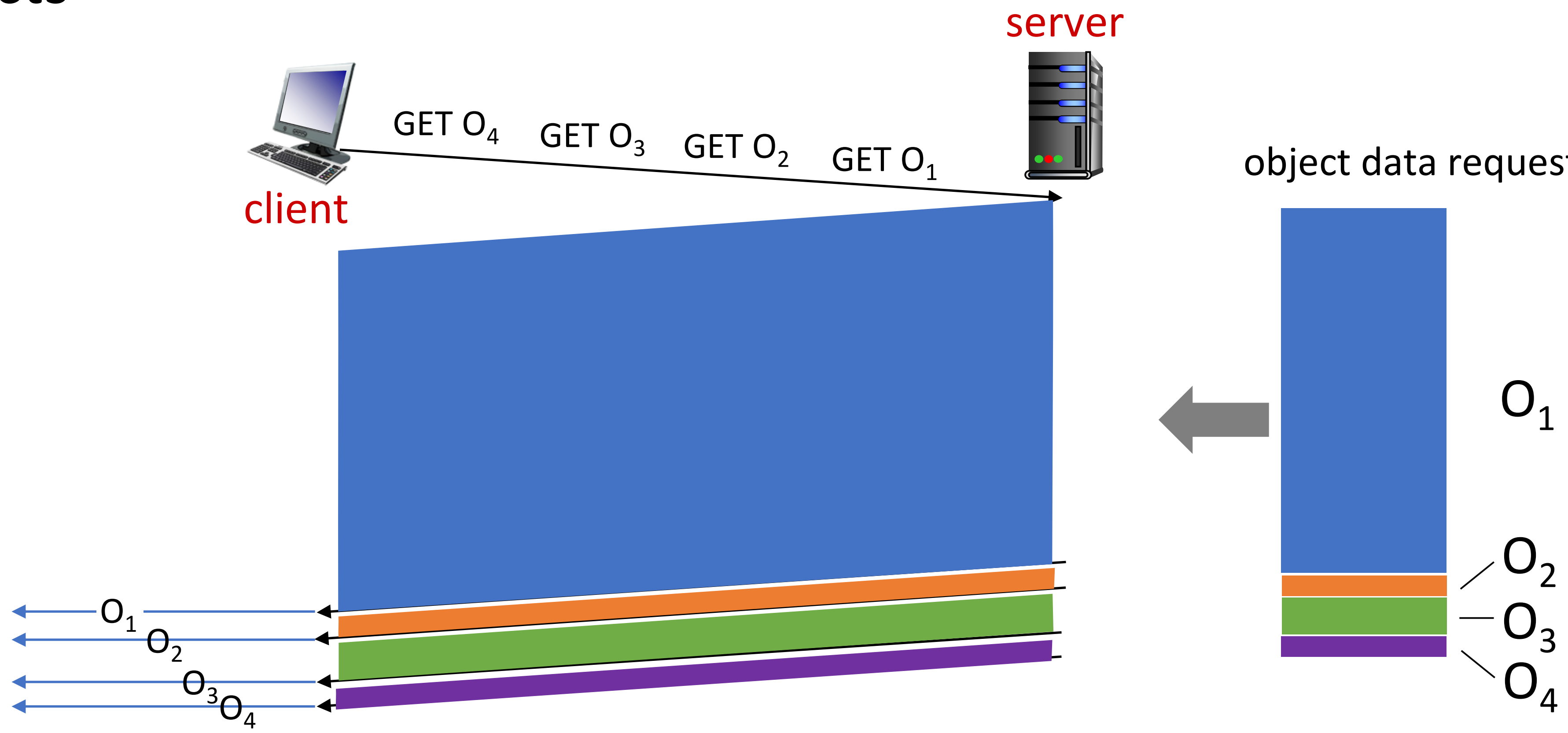
Key goal: decreased delay in multi-object HTTP requests

HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending objects to client.

- methods, status codes, most header fields unchanged from HTTP 1.1
- changes how the data is formatted and transported
- HTTP/1.1 uses persistent TCP connections (by default)
    - a Web page is sent from server to client over a single TCP conn.
    - number of sockets at the server is reduced
    - has a Head of Line (HOL) blocking problem

HTTP/2: mitigating HOL blocking

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

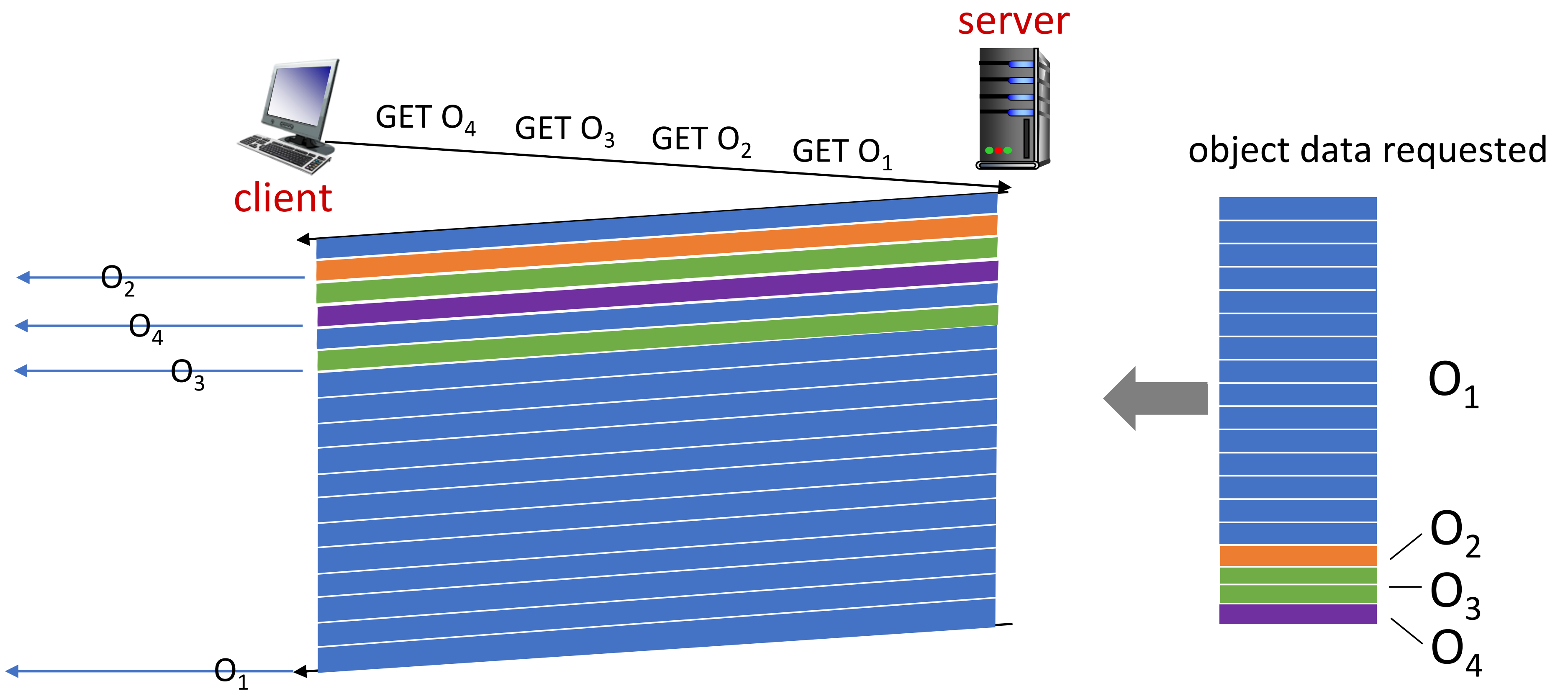


objects delivered in order requested: O_2, O_3, O_4 wait behind O_1; can solve this problem by opening multiple parallel TCP connections – a good idea?

HTTP/2: mitigating HOL blocking

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

after sending one frame from the video clip, the first frames of each of the small objects are sent



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

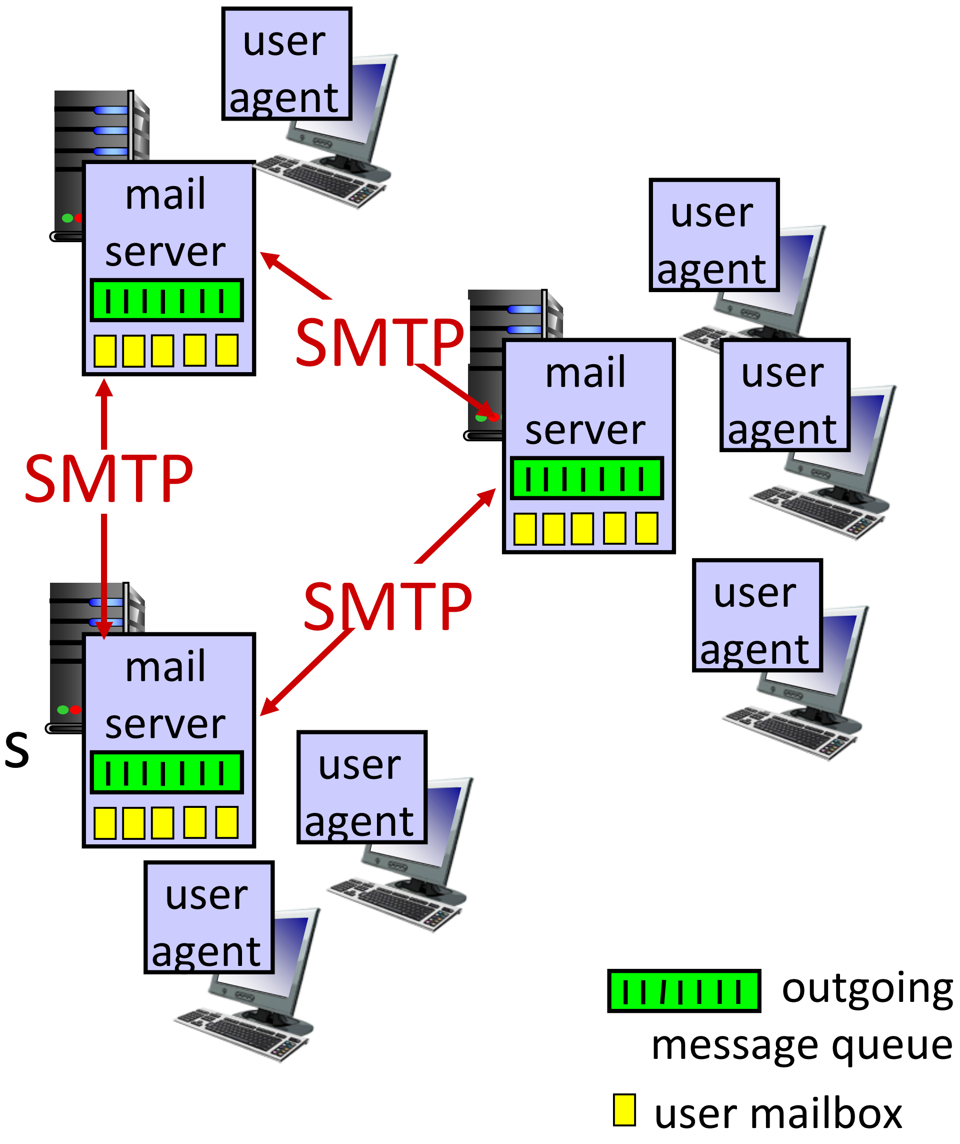
Application layer: overview

- 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

E-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, iPhone mail client
- outgoing, incoming messages stored on server, user agent retrieves the message from mailboxes in a server

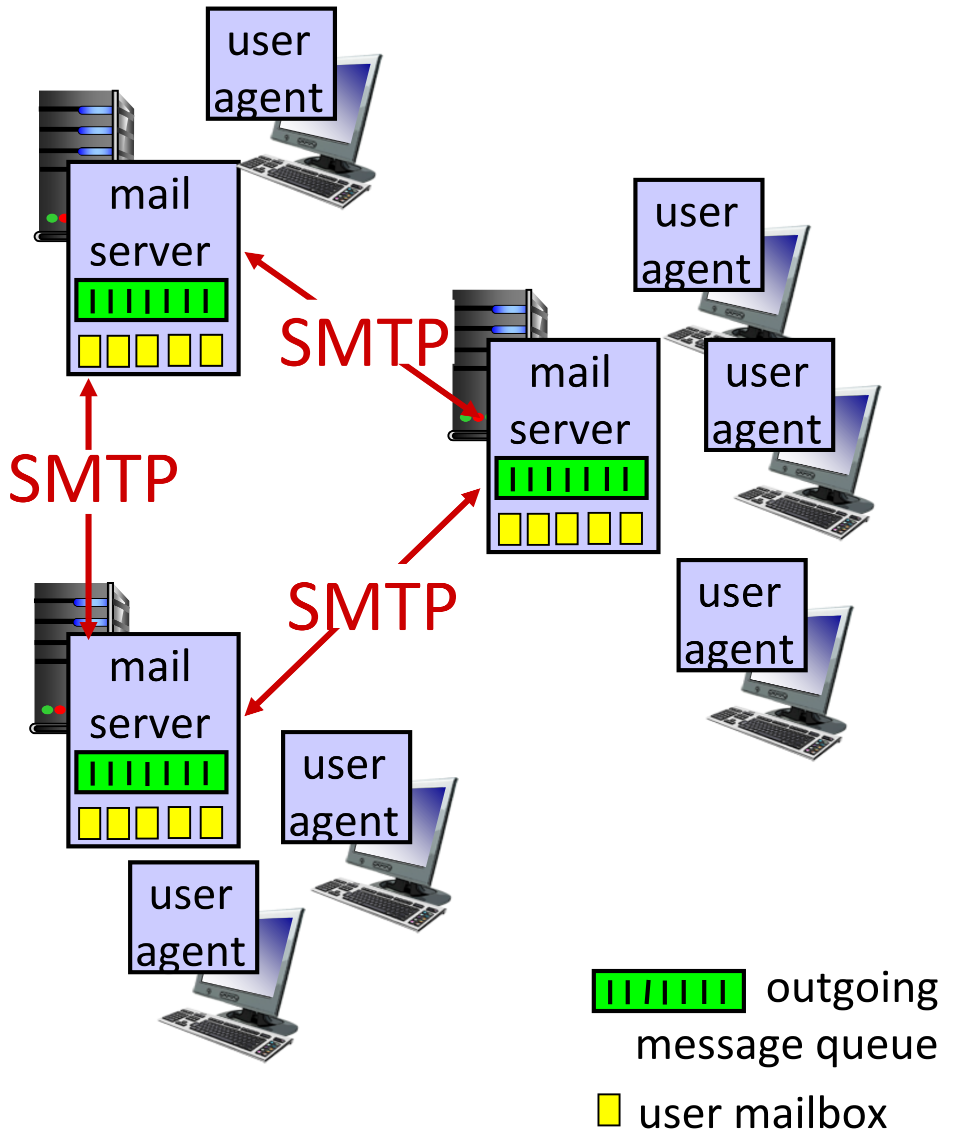
E-mail: mail servers

mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages
- Senders mail server also deal with failures in recipients mail server (If the message cannot be delivered, senders server holds the message in a message queue and attempts to transfer the message later, reattempts are done every 30 mins)

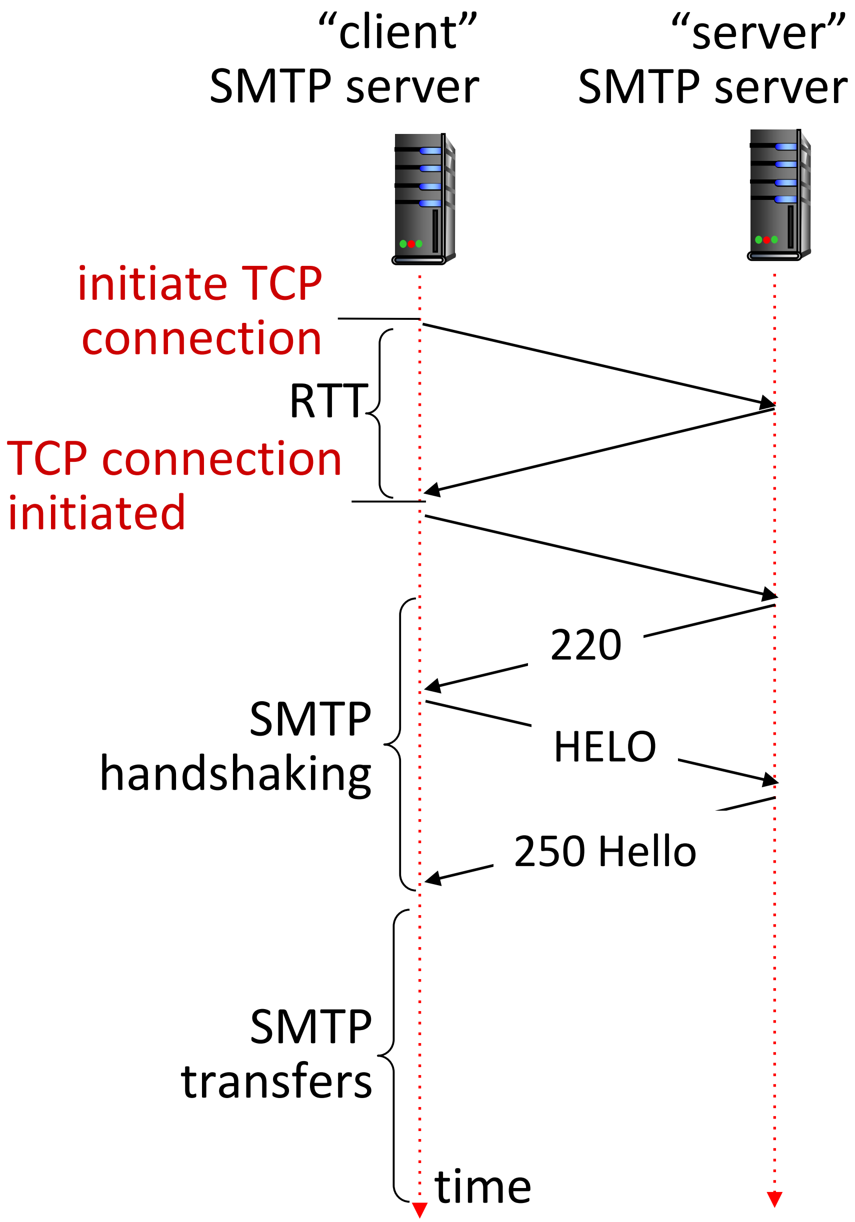
SMTP protocol between mail servers to send email messages

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



SMTP RFC (5321)

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
  - direct transfer: sending server (acting like client) to receiving server
- three phases of transfer
  - SMTP handshaking (greeting)
  - SMTP transfer of messages
  - SMTP closure
- command/response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase



Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message "to" bob@someschool.edu

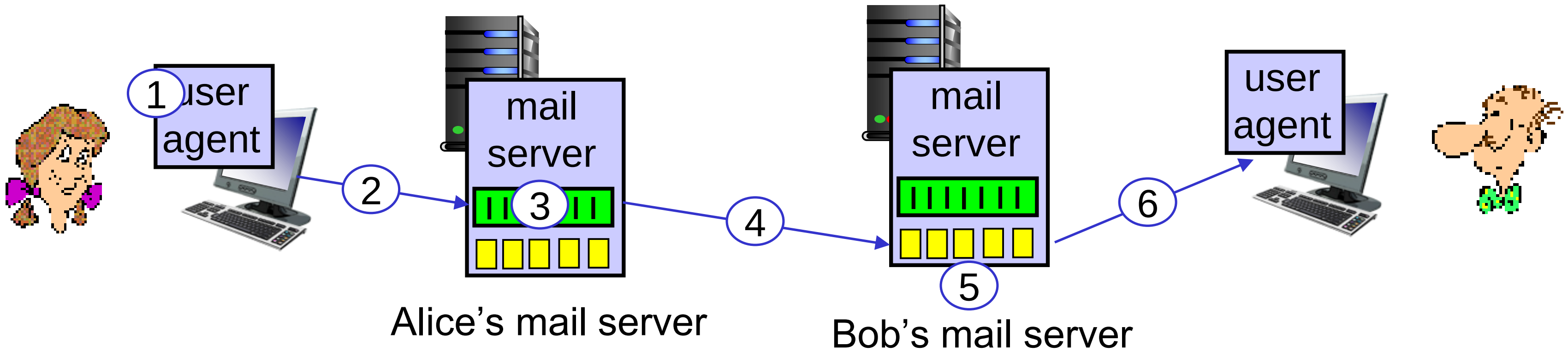
2) Alice's UA sends message to her mail server using SMTP; message placed in message queue

3) client side of SMTP at mail server 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



Sample SMTP interaction

```
      S: 220 hamburger.edu          (service ready)
      C: HELO crepes.fr
      S: 250  Hello crepes.fr, pleased to meet you
 (requested mail action okay)
      C: MAIL FROM: <alice@crepes.fr>
      S: 250 alice@crepes.fr... Sender ok
      C: RCPT TO: <bob@hamburger.edu>
      S: 250 bob@hamburger.edu ... Recipient ok
      C: DATA
      S: 354 Enter mail, end with "." on a line by itself
 (start mail input)
      C: Do you like ketchup?
      C: How about pickles?
      C: .
      S: 250 Message accepted for delivery
      C: QUIT
```

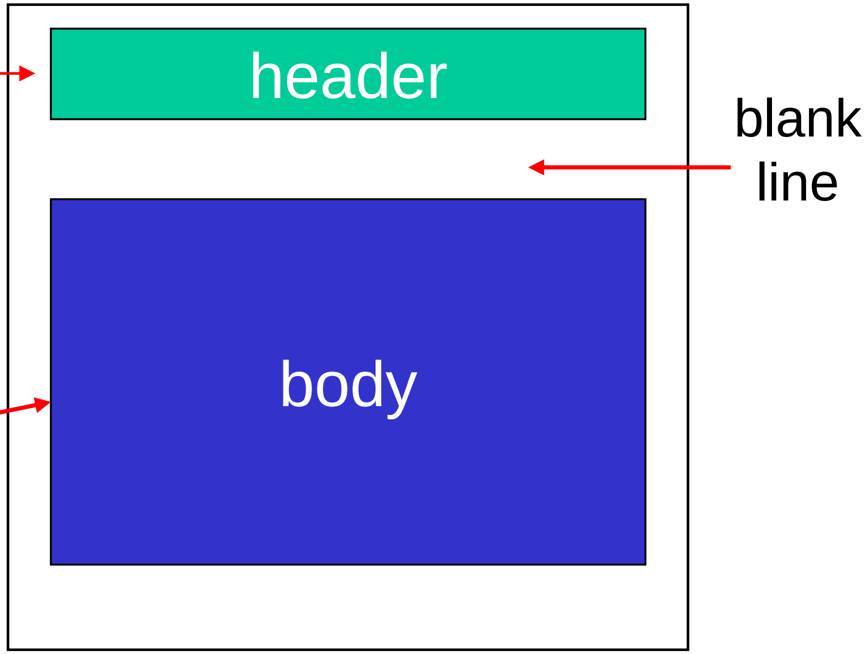
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 5321 (like RFC 7231 defines HTTP)

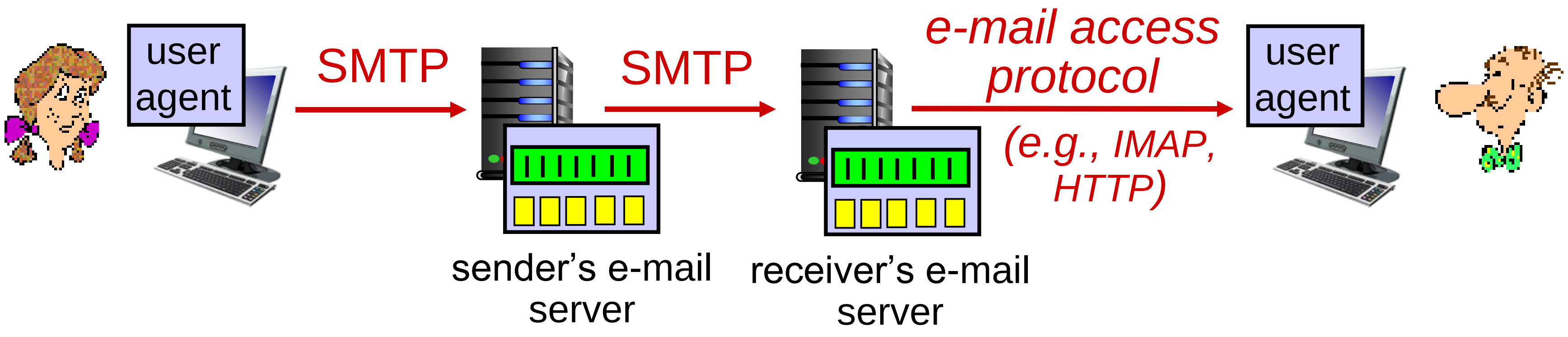
RFC 2822 defines *syntax* for e-mail message itself (like HTML defines syntax for web documents)

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

  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the “message” , ASCII characters only



Retrieving email: mail access protocols



- SMTP: delivery/storage of e-mail messages to receiver's server
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages