SC250
Computer Networking I

Application Layer Protocols

Prof. Matthias Grossglauser

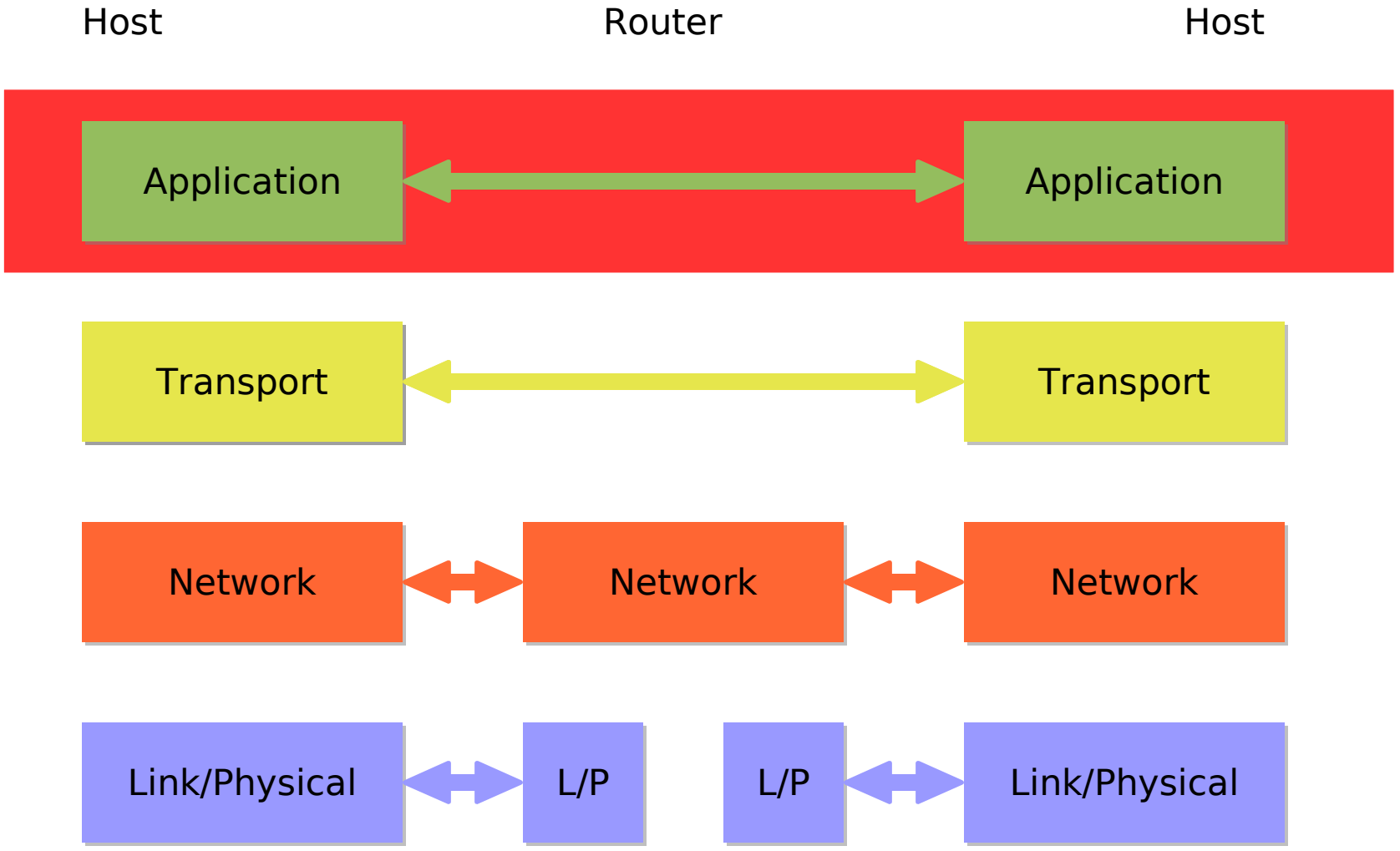
School of Computer and Communication Sciences
EPFL

http://lcawww.epfl.ch

Today's Objectives

- Conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- Learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - Multimedia: streaming, RTSP
  - SMTP / POP3 / IMAP

Application Layer

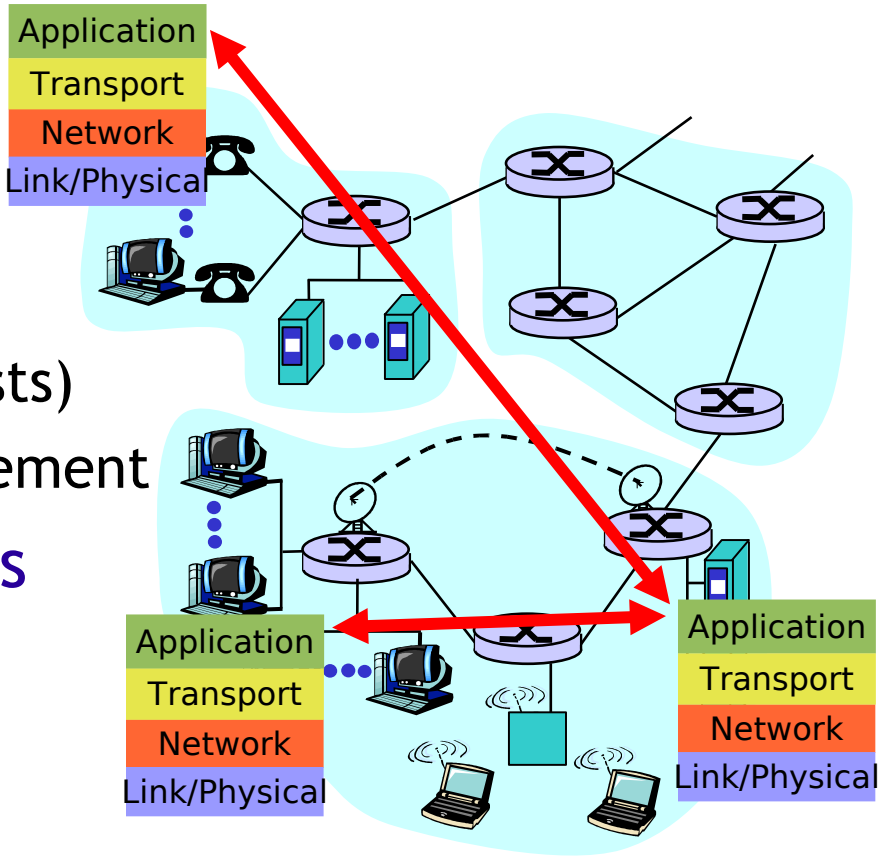


Network Applications: Some Jargon

- Process: program running within a host
  - Within same host, two processes communicate using interprocess communication (defined by OS)
  - Processes running in different hosts communicate with an application-layer protocol
- User agent: interfaces with user “above” and network “below”
  - Implements user interface & application-level protocol
    - Web: browser
    - E-mail: mail reader
    - streaming audio/video: media player

Applications and Application-Layer Protocols

- Application: communicating, distributed processes
  - E.g., e-mail, Web, P2P file sharing, instant messaging
  - Running in end systems (hosts)
  - Exchange messages to implement
- Application-layer protocols
  - One “piece” of an app
  - Define messages exchanged by apps and actions taken
  - Use communication services provided by lower layer protocols (TCP, UDP)



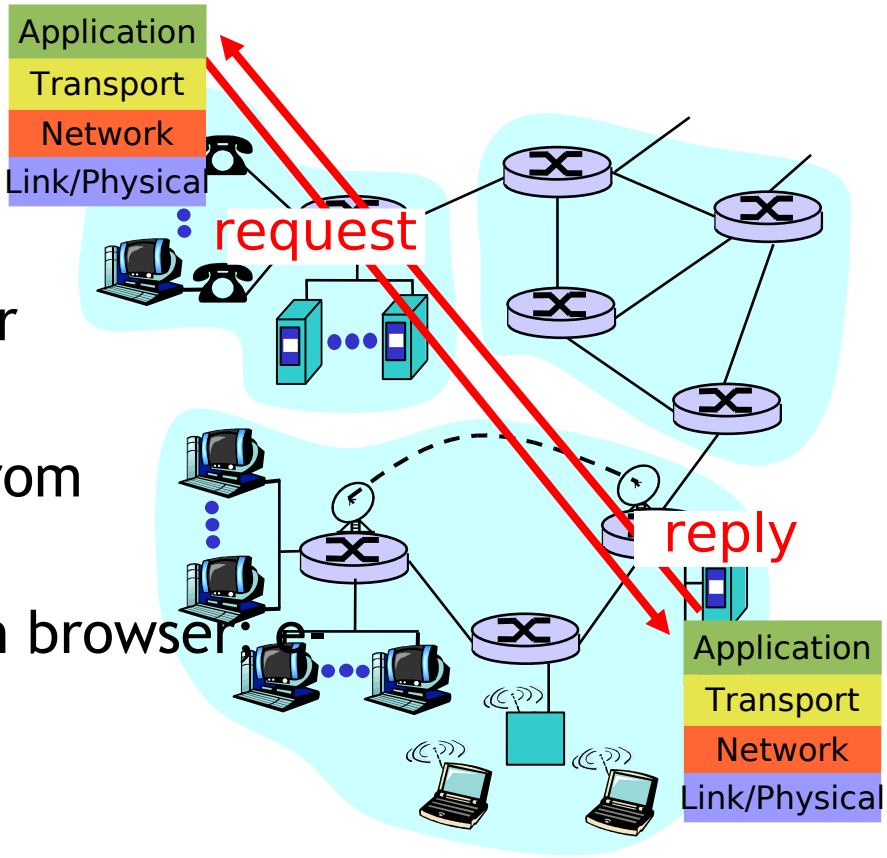
Application-Layer Protocol Defines

- Types of messages exchanged, e.g., request & response messages
- Syntax of message types: what fields in messages & how fields are delineated
- Semantics of the fields, i.e., meaning of information in fields
- Rules for when and how processes send & respond to messages

- Public-domain protocols:
  - Defined in RFCs
  - Allows for interoperability
  - E.g., HTTP, SMTP
- Proprietary protocols:
  - eg, KaZaA, skype

Client-Server Paradigm

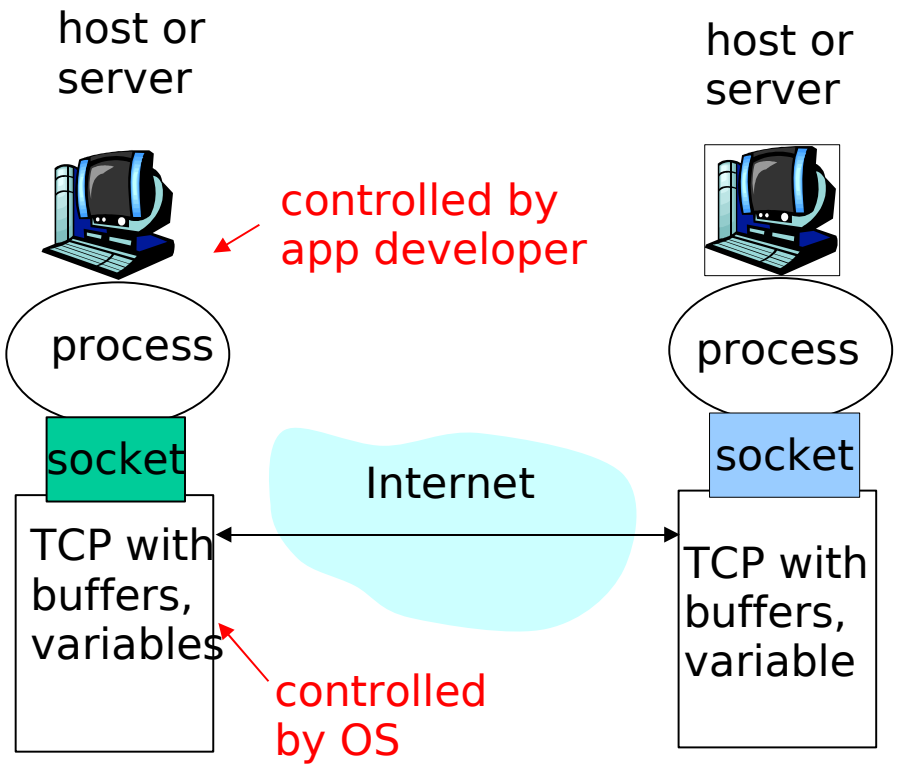
- Typical network app has two pieces: client and server



- Client:
  - initiates contact with server ("speaks first")
  - typically requests service from server,
  - Web: client implemented in browser; e-mail: in mail reader
- Server:
  - provides requested service to client
  - e.g., Web server sends requested Web page, mail server delivers e-mail

Processes Communicating across Network

- Process sends/receives messages to/from its socket
- Socket analogous to door
  - sending process shoves message out door
  - sending process assumes transport infrastructure on other side of door which brings message to socket at receiving process



- API: (1) choice of transport protocol; (2) ability to fix a few parameters (lots more on this later)

Addressing Processes

- For a process to receive messages, it must have an identifier
  - Every host has a unique 32-bit IP address
- Q: does the IP address of the host on which the process runs suffice for identifying the process?
- Answer: No, many processes can be running on same host

- Identifier includes both the IP address and port numbers associated with the process on the host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- More on this later

What Transport Service does an App Need?

- Bandwidth
  - some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
  - other apps ("elastic apps") make use of whatever bandwidth they get
- Data loss
  - some apps (e.g., audio) can tolerate some loss
  - other apps (e.g., file transfer, telnet) require 100% reliable data transfer
- Timing
  - some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

Transport Service Requirements

Application	Data loss	Bandwidth	Time Sensitive
file transfer	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	yes, 100's msec
stored audio/video	loss-tolerant	video:10kbps-5Mbps	yes, few secs
interactive games	loss-tolerant	same as above	yes, 100's msec
instant messaging	no loss	few kbps up elastic	yes and no

Internet Transport Protocols Services

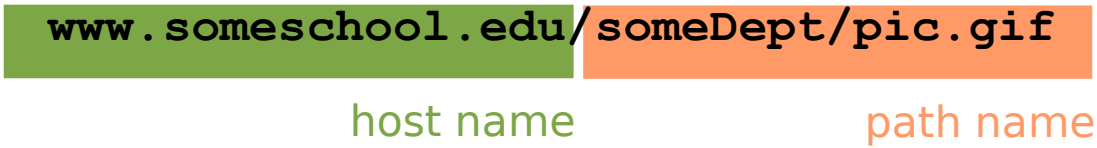
- TCP service:
  - connection-oriented: setup required between client and server processes
  - reliable transport between sending and receiving process
  - flow control: sender won't overwhelm receiver
  - congestion control: throttle sender when network overloaded
  - does not provide: timing, bandwidth guarantees
- UDP service:
  - unreliable data transfer between sending and receiving process
  - does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee
- Q: why bother? Why is there a UDP?

Internet Apps: Application, Transport Protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Dialpad)	typically UDP

Web and HTTP

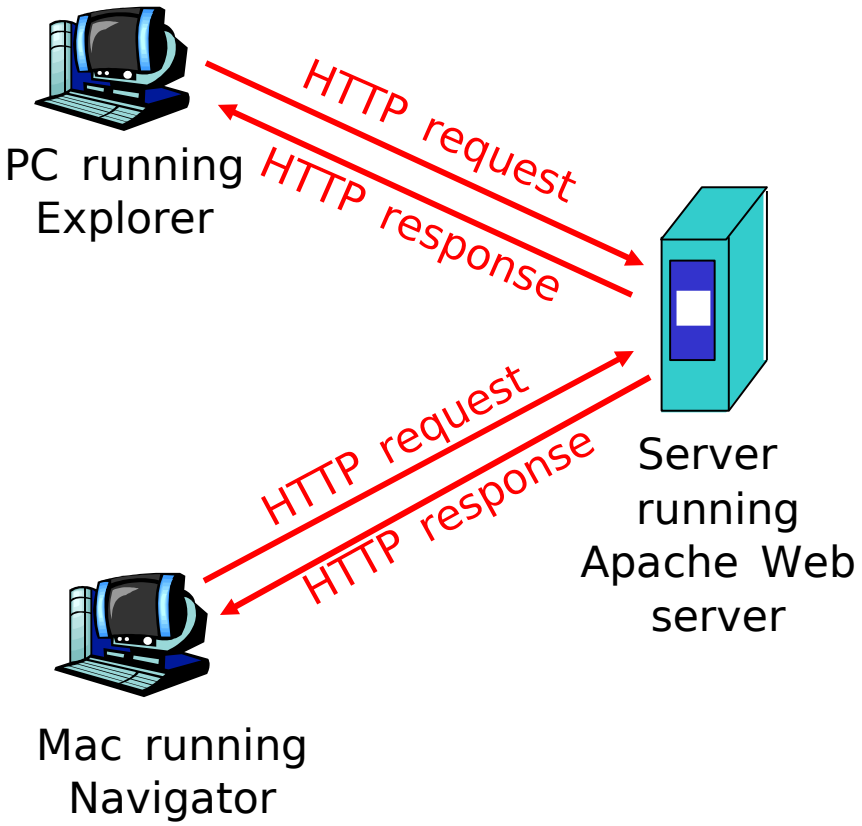
- First some jargon
  - Web page consists of objects
  - Object can be HTML file, JPEG image, Java applet, audio file,…
  - Web page consists of base HTML-file which includes several referenced objects
  - Each object is addressable by a URL
- Example URL:



HTTP Overview

HTTP: hypertext transfer protocol

- Application layer protocol of the Web
- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068



HTTP Overview (continued)

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

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

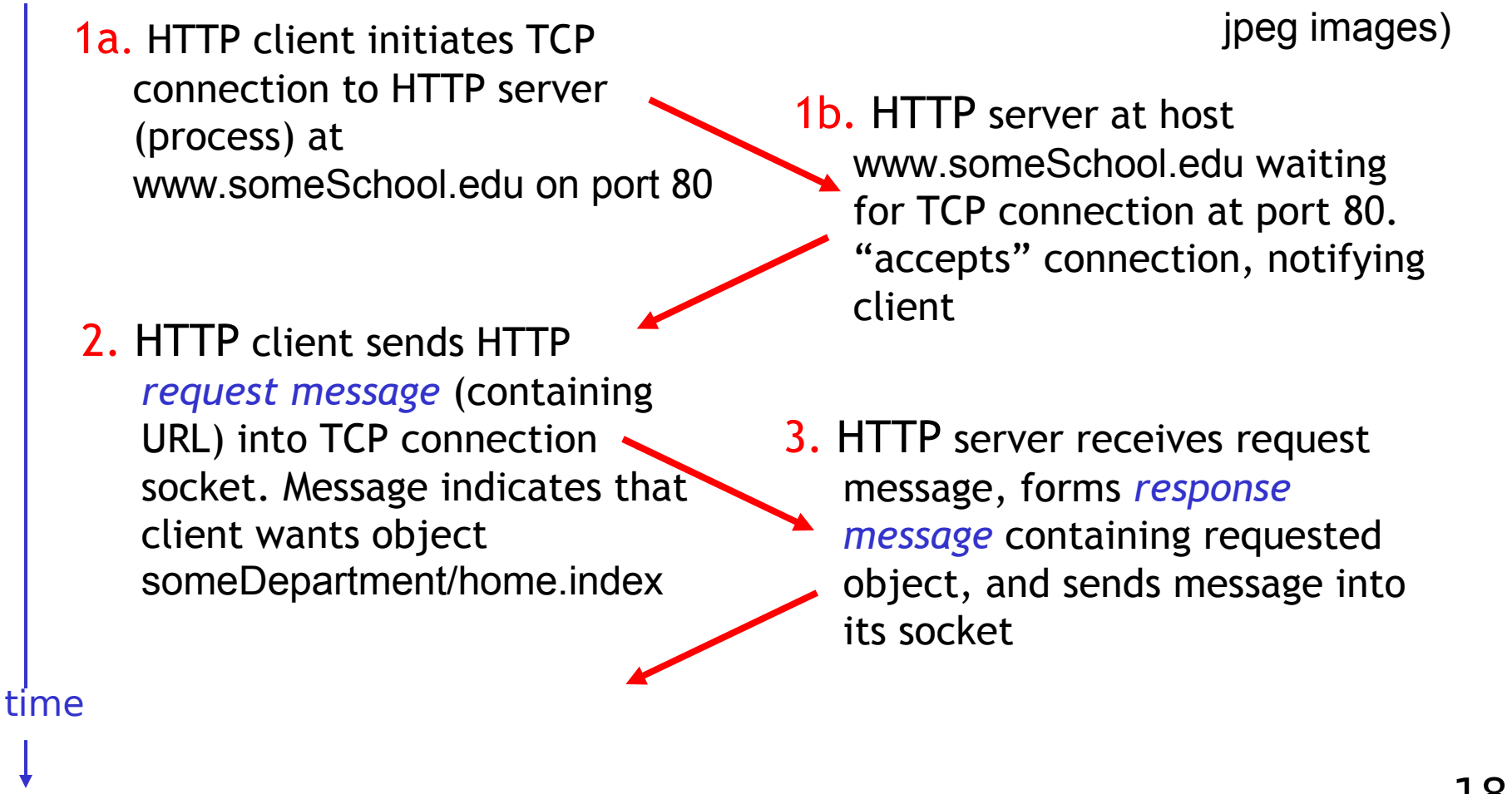
HTTP Connections

- Nonpersistent HTTP
  - At most one object is sent over a TCP connection.
  - HTTP/1.0 uses nonpersistent HTTP
- Persistent HTTP
  - Multiple objects can be sent over single TCP connection between client and server.
  - HTTP/1.1 uses persistent connections in default mode

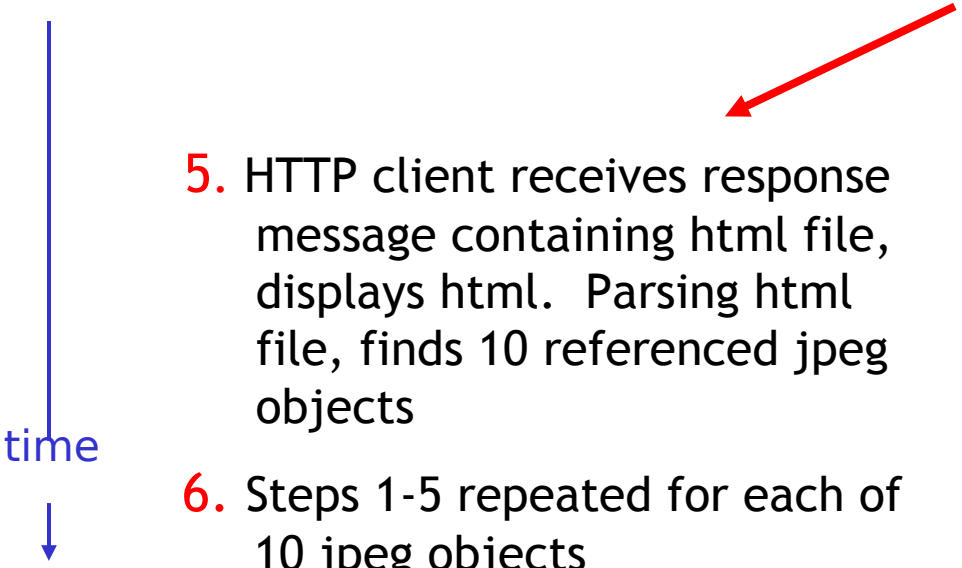
Nonpersistent HTTP

Suppose user enters URL **`www.someSchool.edu/someDepartment/home.index`**

(contains text, references to 10 jpeg images)



Nonpersistent HTTP (cont.)



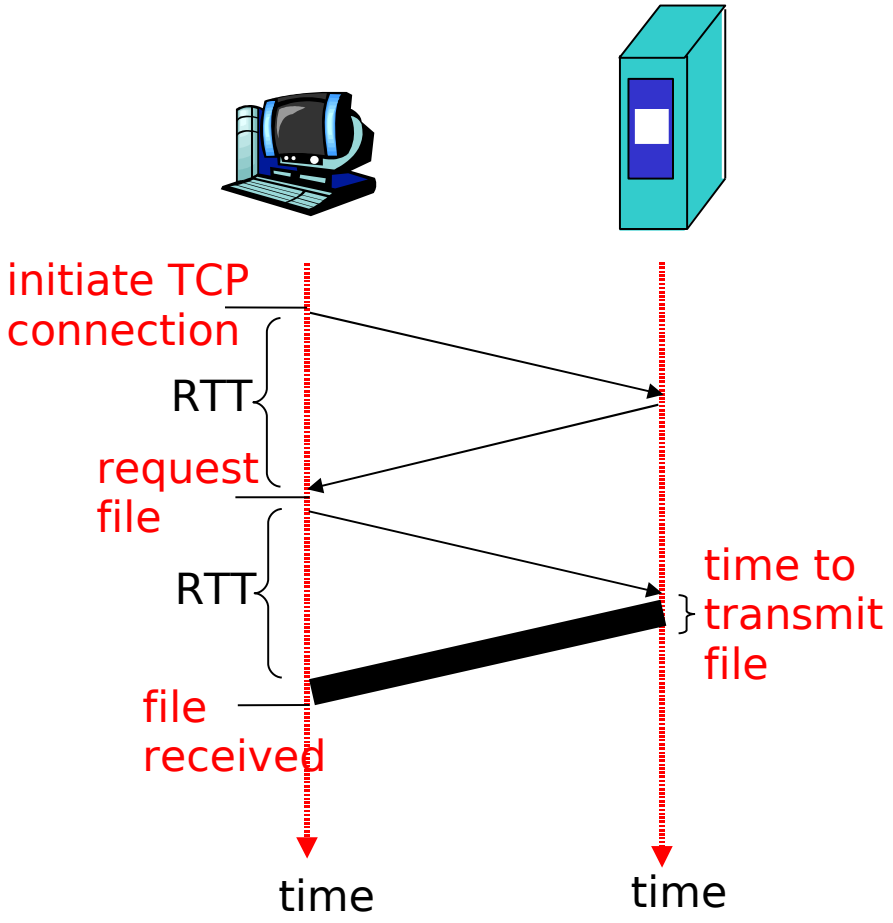
4. HTTP server closes TCP connection.

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

Response Time Modeling

- Definition of RTT:
  - time to send a small packet to travel from client to server and back.
- Response time:
  - one RTT to initiate TCP connection
  - one RTT for HTTP request and first few bytes of HTTP response to return
  - file transmission time
  - total = 2RTT+transmit time



Persistent HTTP

Nonpersistent HTTP issues:

- requires 2 RTTs per object
- OS must work and allocate host resources for each TCP connection
- but browsers often open parallel TCP connections to fetch referenced objects

Persistent HTTP

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server are sent over connection

Persistent without pipelining:

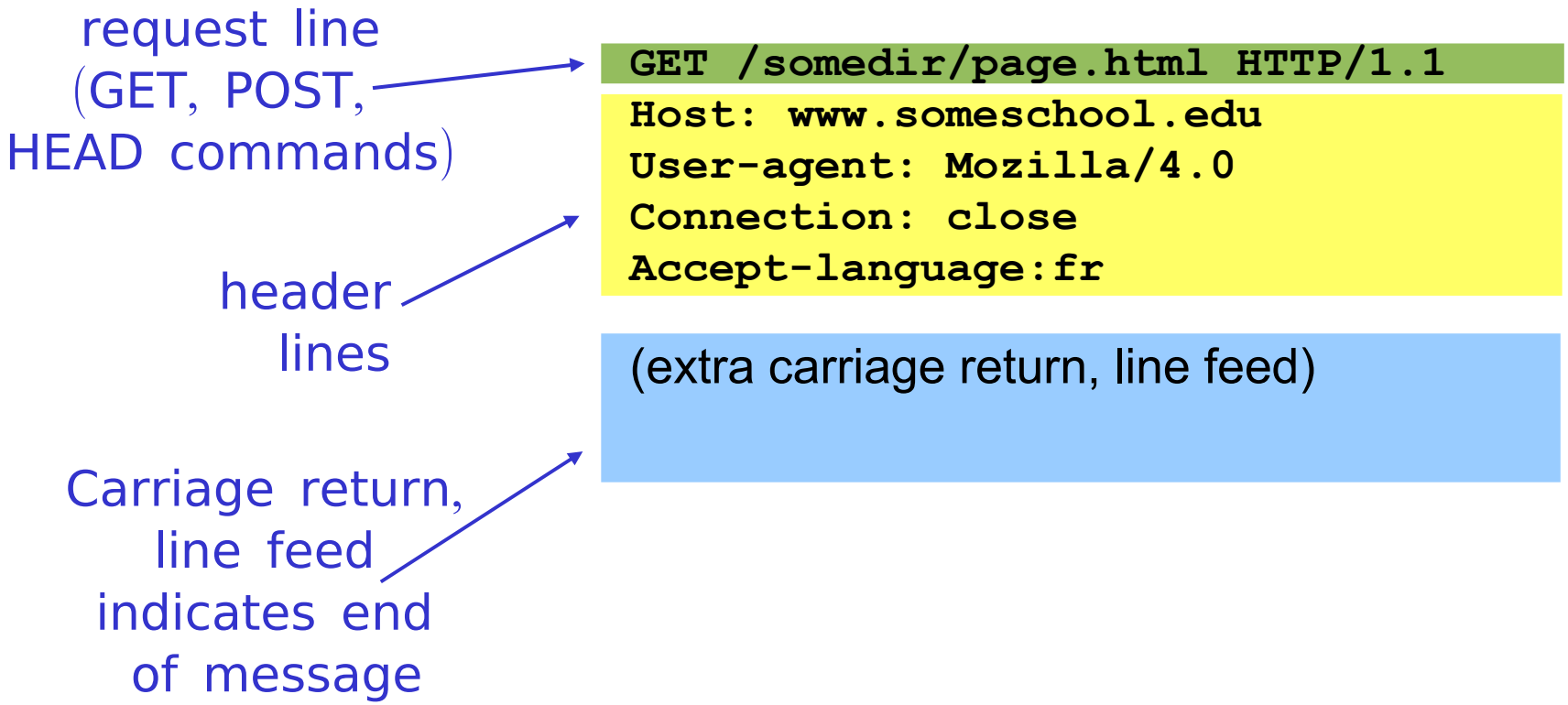
- client issues new request only when previous response has been received
- one RTT for each referenced object

Persistent with pipelining:

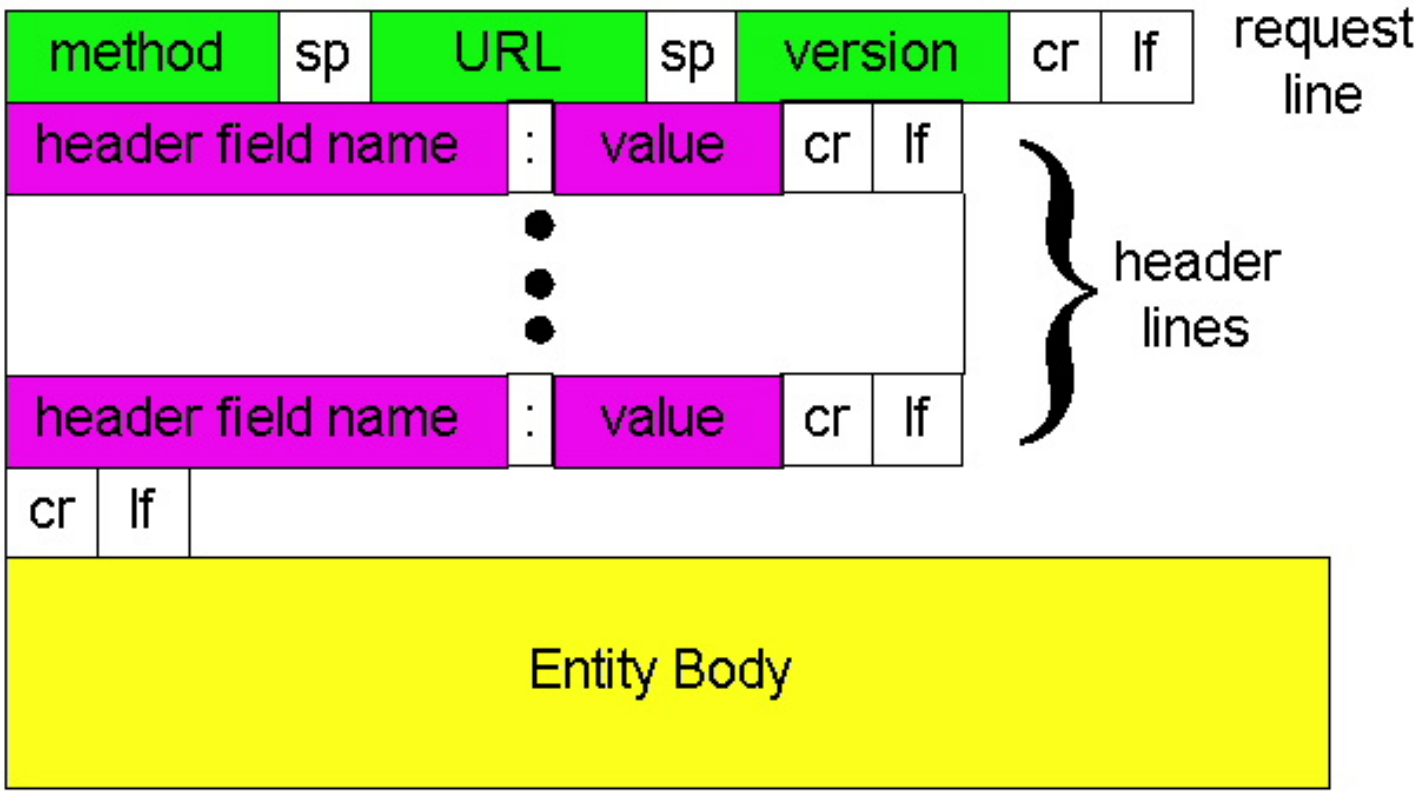
- default in HTTP/1.1
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP Request Message

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



HTTP Request Message: General Format



Uploading Form Input

Post method:

- Web page often includes form input
- Input is uploaded to server in entity body

URL method:

- Uses GET method
- Input is uploaded in URL field of request line:

```
www.somesite.com/animalsearch?monkeys&banana
```

Method Types

HTTP/1.0

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP Response Message

status line
(protocol
status code
status phrase)

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

data, e.g.,
requested
HTML file

HTTP Response Status Codes

In first line in server->client response message.

A few 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 (Location:)

400 Bad Request

- request message not understood by server

403 Forbidden

- requested document exists but access denied

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (Client Side) for Yourself

1. Telnet to your favorite Web server:

```
telnet icawww1.epfl.ch 80
```

Opens TCP connection to port 80 (default HTTP server port) at icawww1.epfl.ch.
Anything typed is sent to port 80 at icawww1.epfl.ch

2. Type in a GET HTTP request:

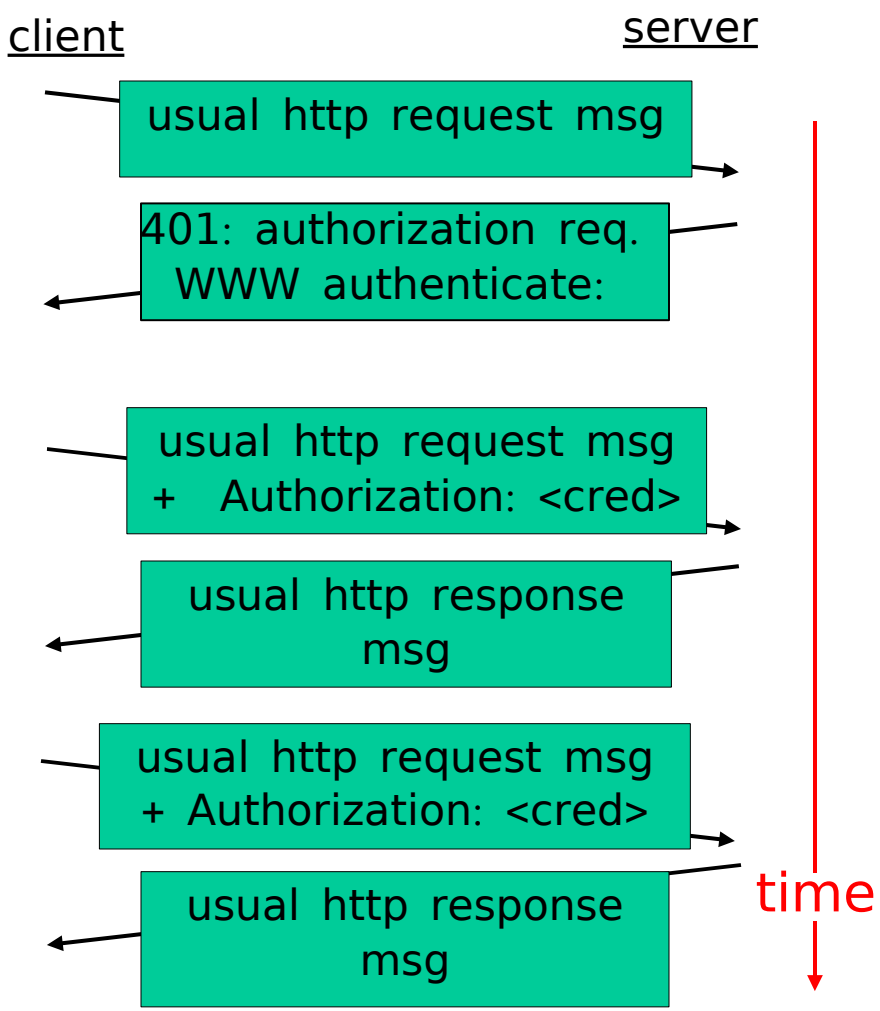
```
GET http://icawww1.epfl.ch/sc250_2005/ HTTP/1.0
```

By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. Look at response message sent by HTTP server!

User-Server Interaction: Authorization

- Authorization : control access to server content
  - Authorization credentials: typically name, password
  - Stateless: client must present authorization in each request
    - authorization: header line in each request
    - if no authorization: header, server refuses access, sends "WWW authenticate:" header line in response



Cookies: Keeping "State"

Many major Web sites use cookies

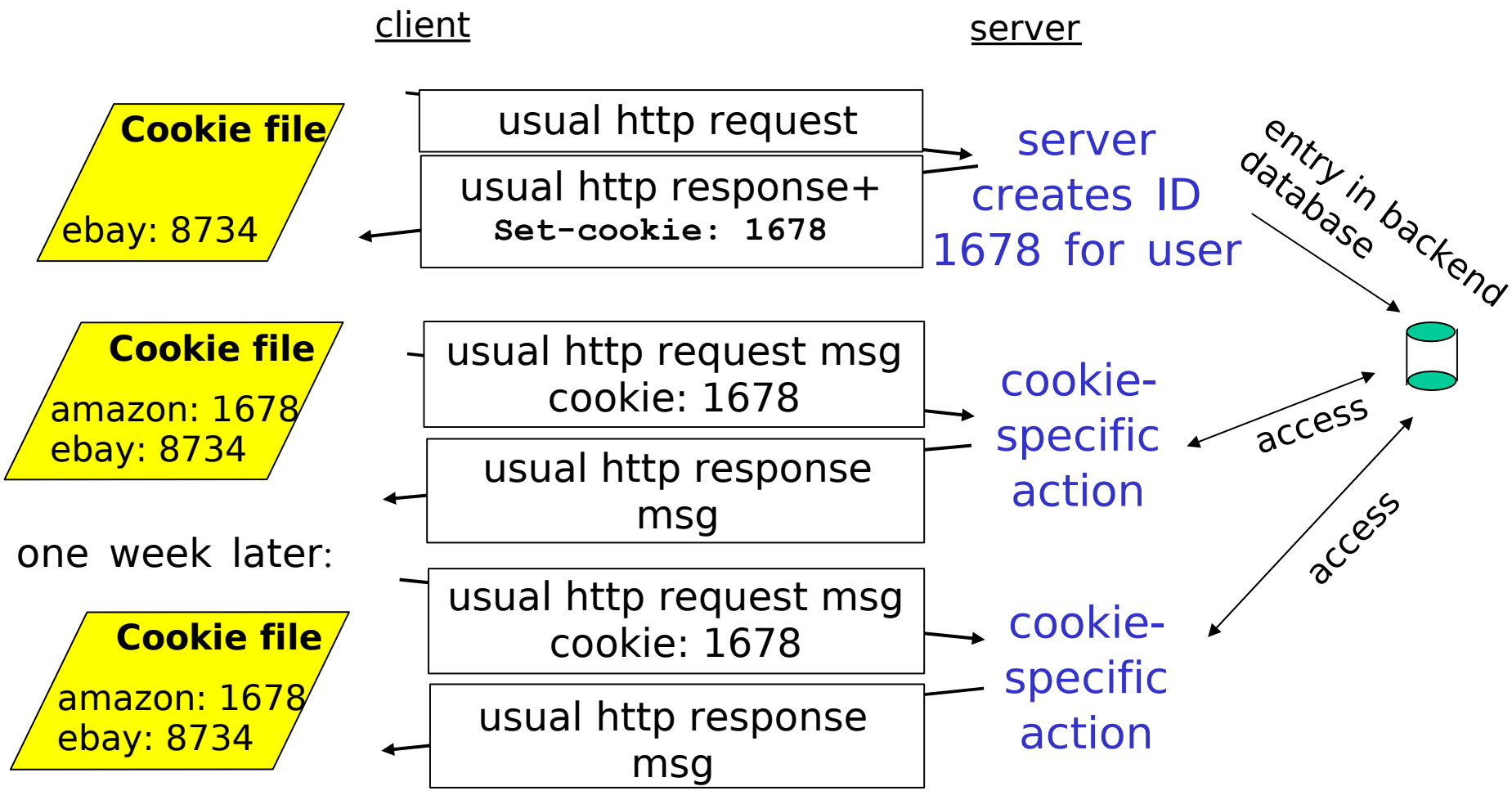
Four components:

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

Example:

- Susan accesses Internet always from same PC
- She visits a specific e-commerce site for first time
- When initial HTTP requests arrives at site, site creates a unique ID and creates an entry in backend database for ID

Cookies: Keeping "State" (cont.)



Cookies (continued)

- What cookies can bring:
  - 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
  - search engines use redirection & cookies to learn yet more
  - advertising companies obtain info across sites

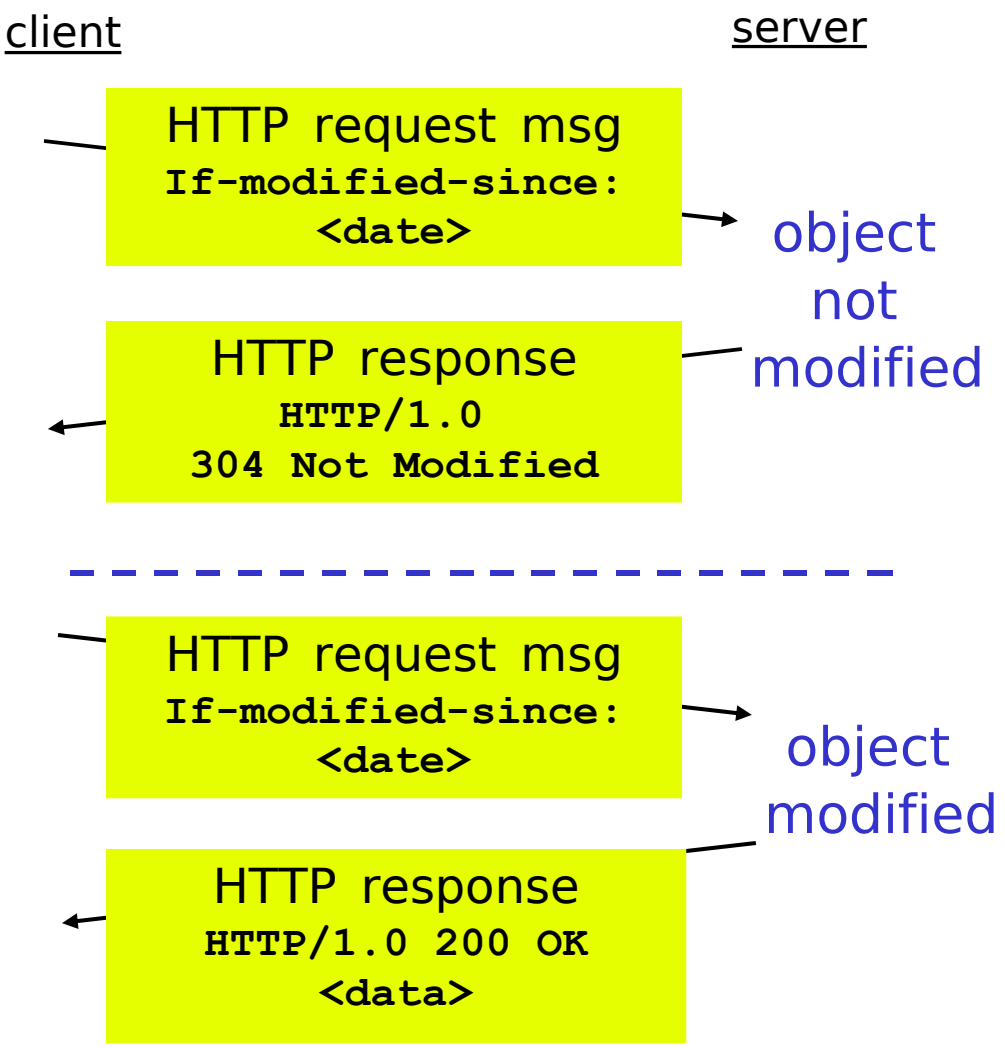
Conditional GET: Client-Side Caching

- Goal: don't send object if client has up-to-date cached version
- client: specify date of cached copy in HTTP request

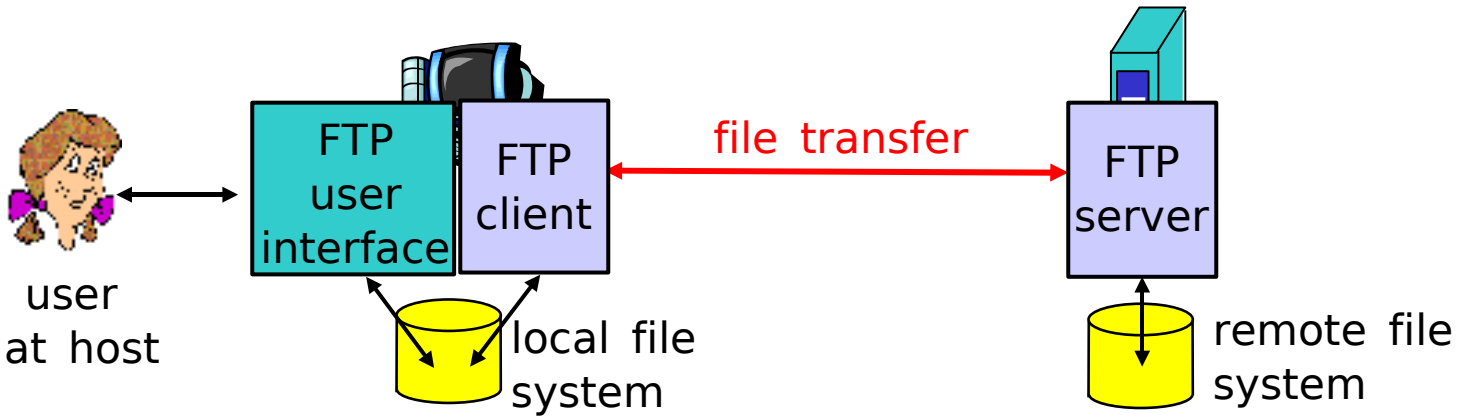
```
If-modified-since:
  <date>
```

- server: response contains no object if cached copy is up-to-date:

```
HTTP/1.0 304 Not
  Modified
```



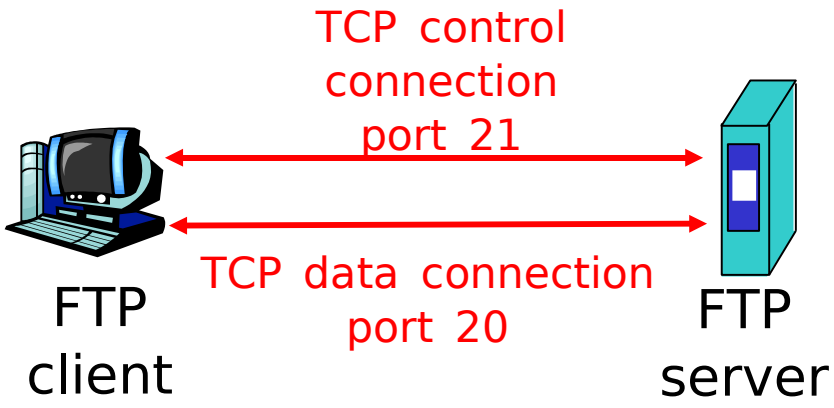
FTP: The File Transfer Protocol



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

FTP: Separate Control, Data Connections

- FTP client contacts FTP server at port 21, specifying TCP as transport protocol
- Client obtains authorization over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives a command for a file transfer, the server opens a TCP data connection to client
- After transferring one file, server closes connection.



- Server opens a second TCP data connection to transfer another file.
- Control connection: "out of band"
- FTP server maintains "state": current directory, earlier authentication

FTP Commands, Responses

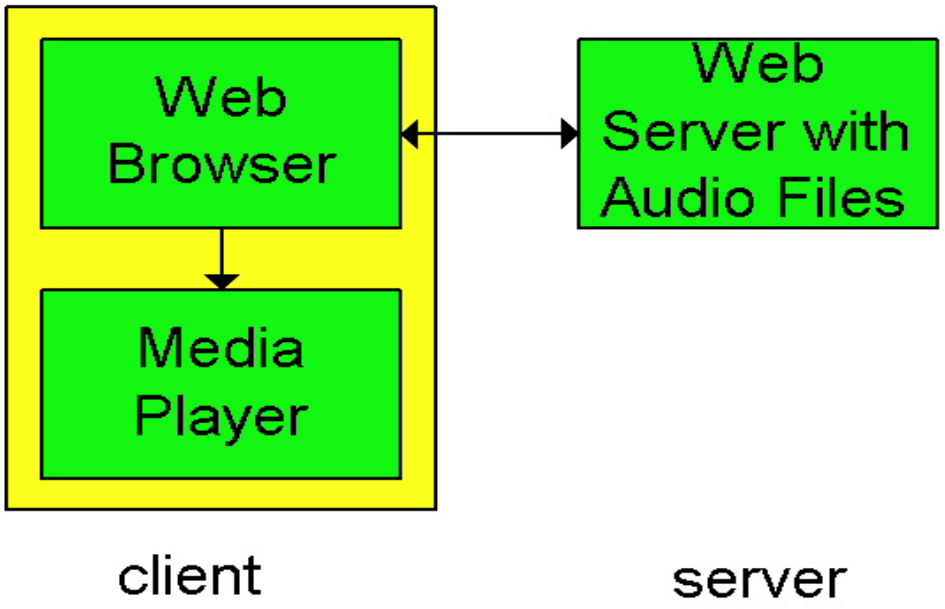
Sample commands:

- sent as ASCII text over control channel
- **`USER`** ***`username`***
- **`PASS`** ***`password`***
- **`LS/DIR`** return list of file in current directory
- **`GET filename`** retrieves (gets) file
- **`PUT filename`** stores (puts) file onto remote host
- **`MGET filenames`** retrieves (gets) multiple files
- **`MPUT filenames`** stores multiple files

Sample return codes

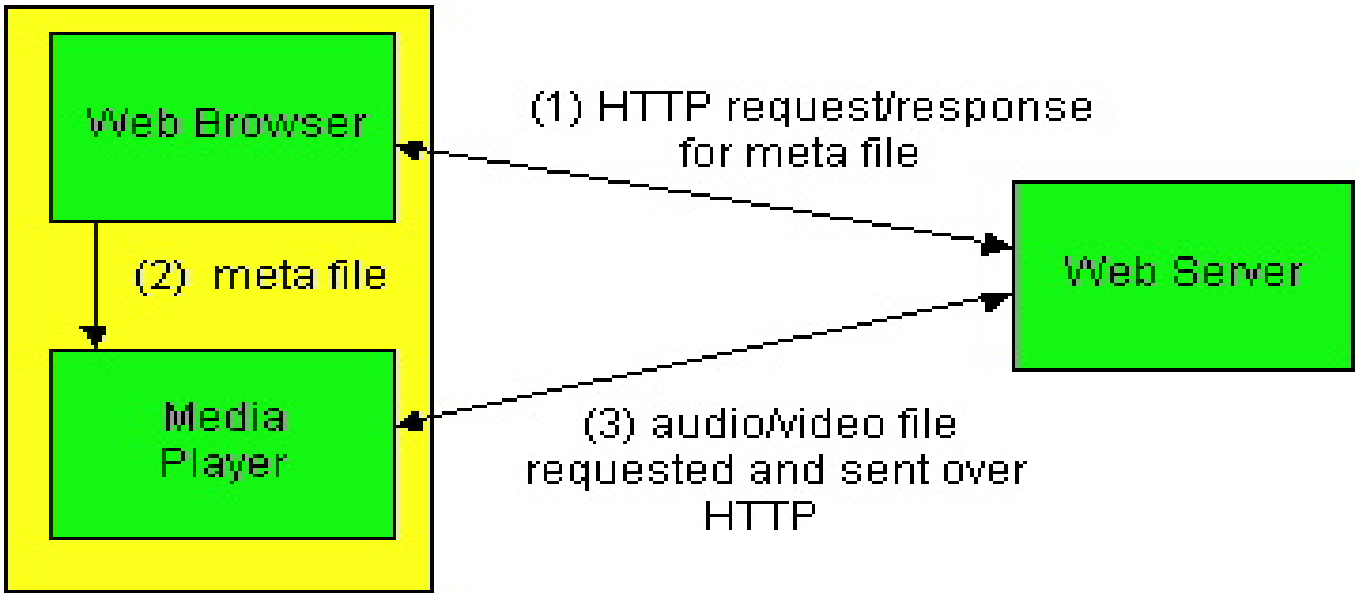
- status code and phrase (as in HTTP)
- **`331 Username OK, password required`**
- **`125 data connection already open; transfer starting`**
- **`425 Can't open data connection`**
- **`452 Error writing file`**

Internet Multimedia: Simplest Approach



- Audio or video stored in file
- Files transferred as HTTP object
  - received in entirety at client, then passed to player
- Audio, video not streamed:
  - no pipelining, long delays until playout!

Internet Multimedia: Streaming Approach

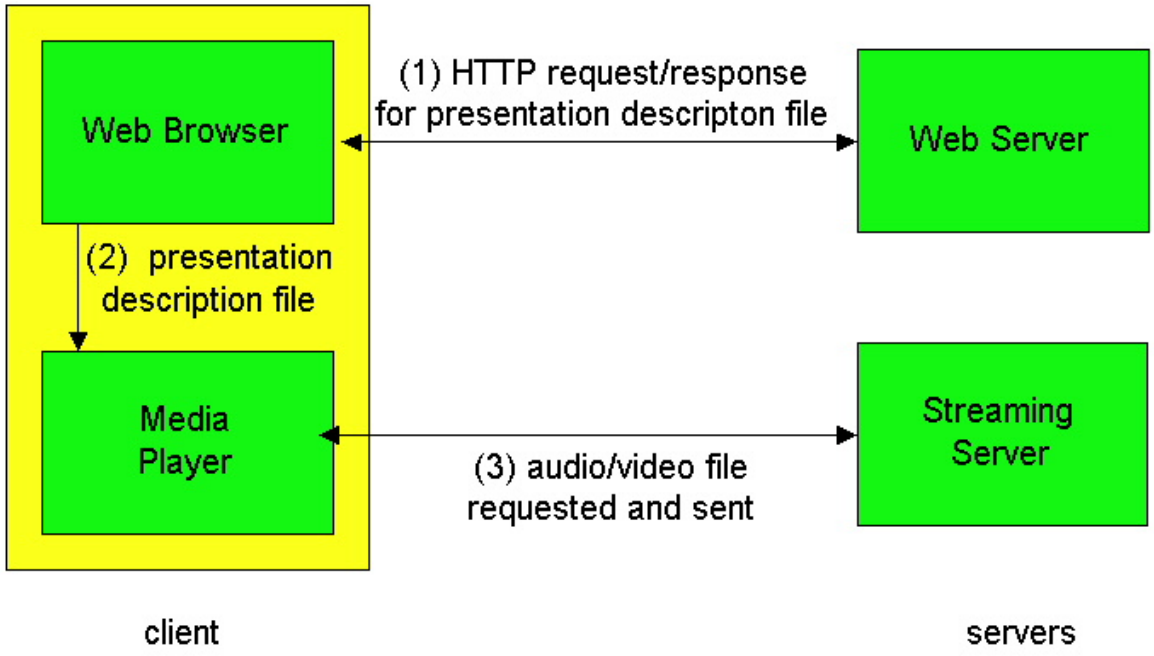


- Browser GETs metafile
- Browser launches player, passing metafile
- Player contacts server
- Server streams audio/video to player

Metafile Example

```
<title>Twister</title>
<session>
        <group language=en lipsync>
                <switch>
                   <track type=audio
                         e="PCMU/8000/1"
                         src = "rtsp://audio.example.com/twister/audio.en/lofi">
                   <track type=audio
                         e="DVI4/16000/2" pt="90 DVI4/8000/1"
                         src="rtsp://audio.example.com/twister/audio.en/hifi">
                 </switch>
              <track type="video/jpeg"
                         src="rtsp://video.example.com/twister/video">
         </group>
</session>
```

Streaming From a Streaming Server



- This architecture allows for non-HTTP protocol between server and media player
- Can also use UDP instead of TCP.

User Control of Streaming Media: RTSP

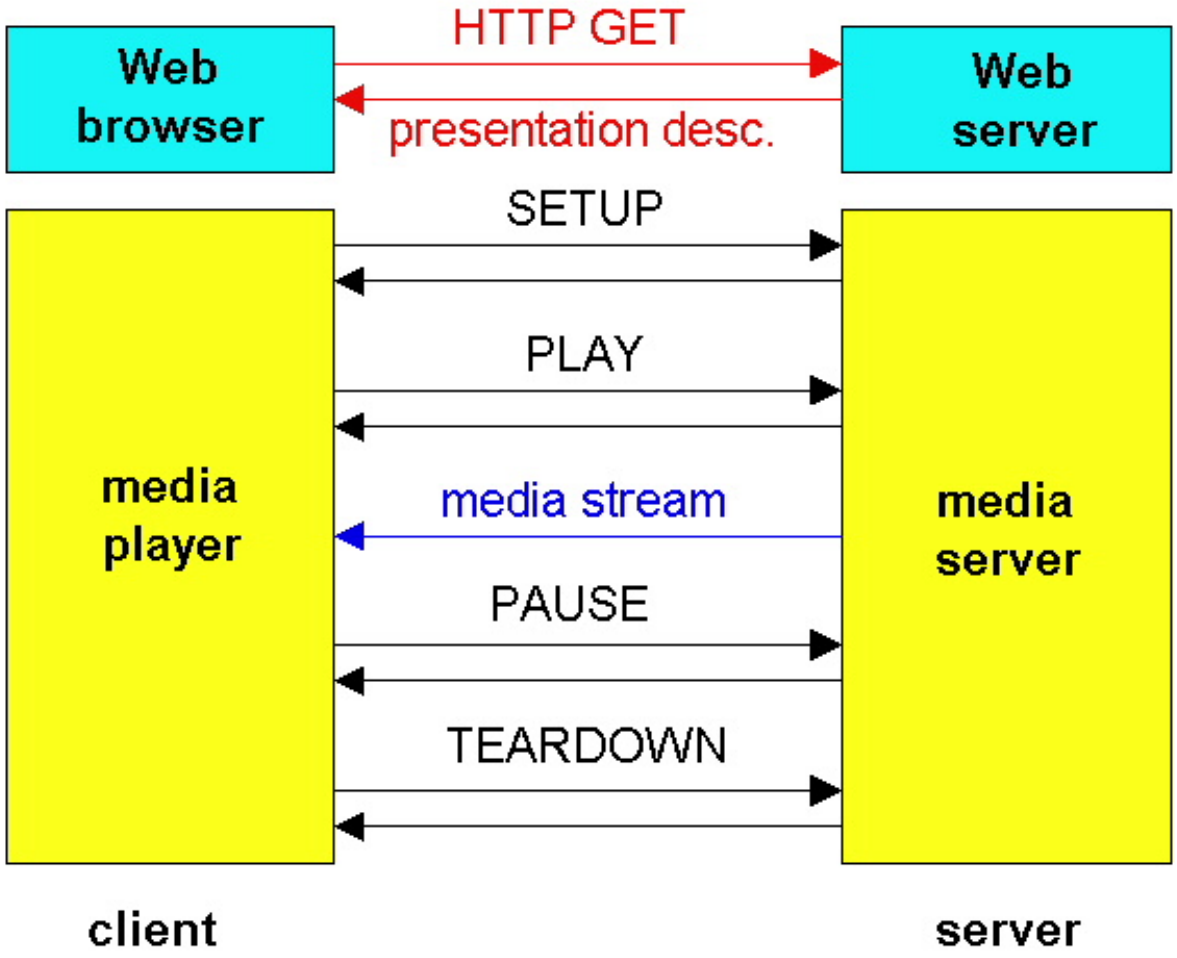
- HTTP
  - Does not target multimedia content
  - No commands for fast forward, etc.
- RTSP [RFC 2326]
  - Client-server application layer protocol
  - For user to control display: rewind, fast forward, pause, resume, repositioning, etc…
  - "Internet remote control"

- What it doesn't do:
  - Does not define how audio/video is encapsulated for streaming over network
  - Does not restrict how streamed media is transported; UDP or TCP possible
  - Does not specify how the media player buffers audio/video

RTSP: Out of Band Control

- FTP uses an “out-of-band” control channel:
  - A file is transferred over one TCP connection.
  - Control information (directory changes, file deletion, file renaming, etc.) is sent over a separate TCP connection.
  - The “out-of-band” and “in-band” channels use different port numbers.
- RTSP messages are also sent out-of-band:
  - RTSP control messages use different port numbers than the media stream: out-of-band
    - Port 554
  - The media stream is considered “in-band”

RTSP Operation



RTSP Exchange Example

C: SETUP rtsp://audio.example.com/twister/audio RTSP/1.0
Transport: rtp/udp; compression; port=3056; mode=PLAY

S: RTSP/1.0 200 1 OK
Session 4231

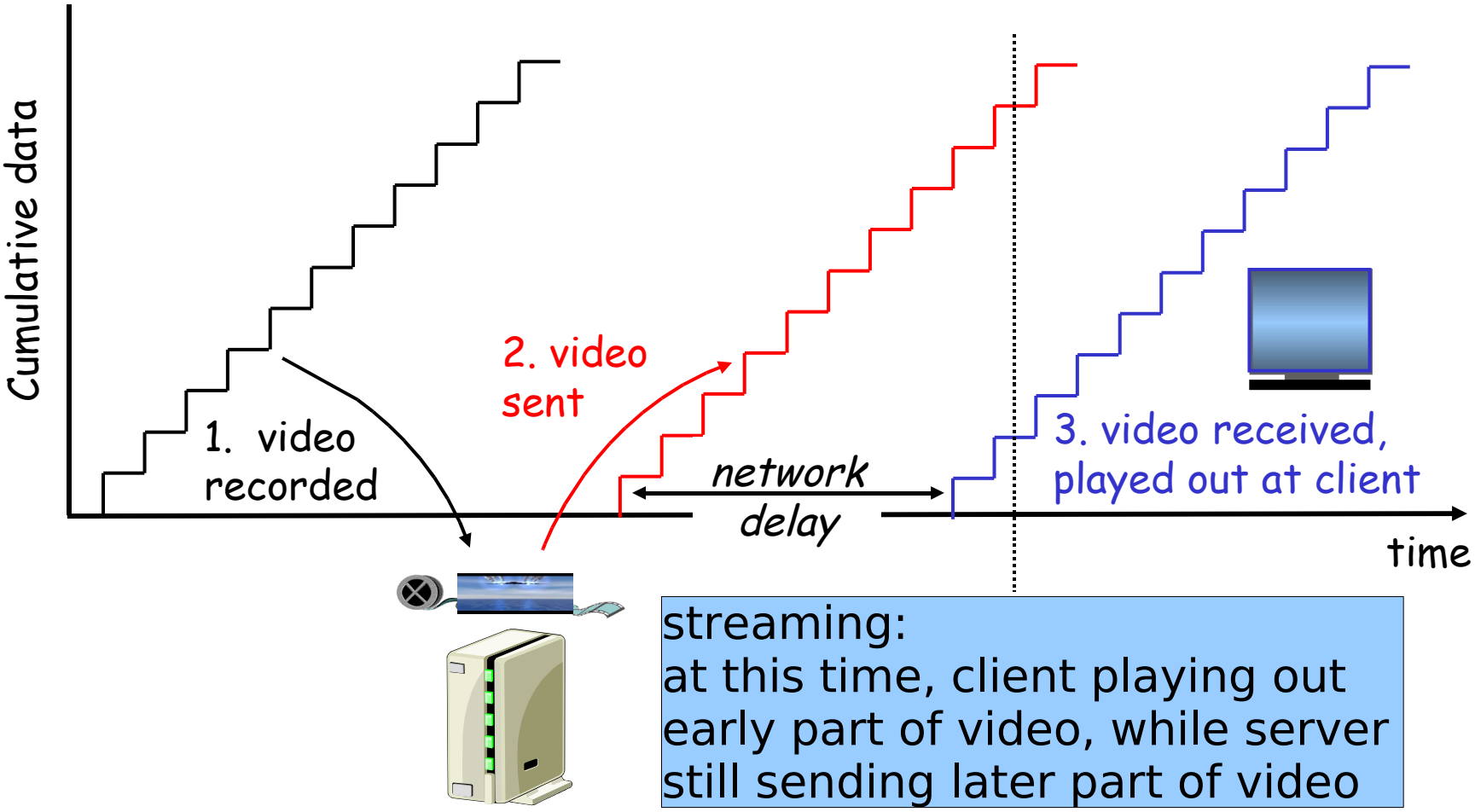
C: PLAY rtsp://audio.example.com/twister/audio.en/lofi RTSP/1.0
Session: 4231
Range: npt=0-

C: PAUSE rtsp://audio.example.com/twister/audio.en/lofi RTSP/1.0
Session: 4231
Range: npt=37

C: TEARDOWN rtsp://audio.example.com/twister/audio.en/lofi RTSP/1.0
Session: 4231

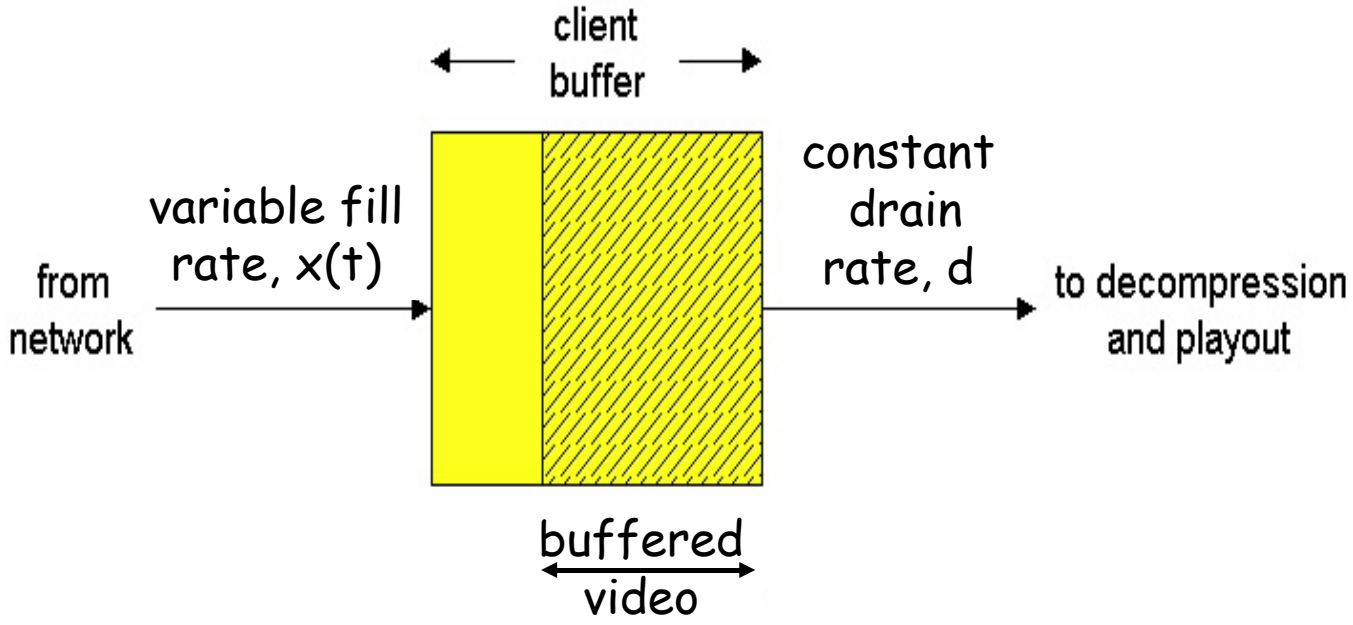
S: 200 3 OK

Streaming Stored Multimedia: Delay



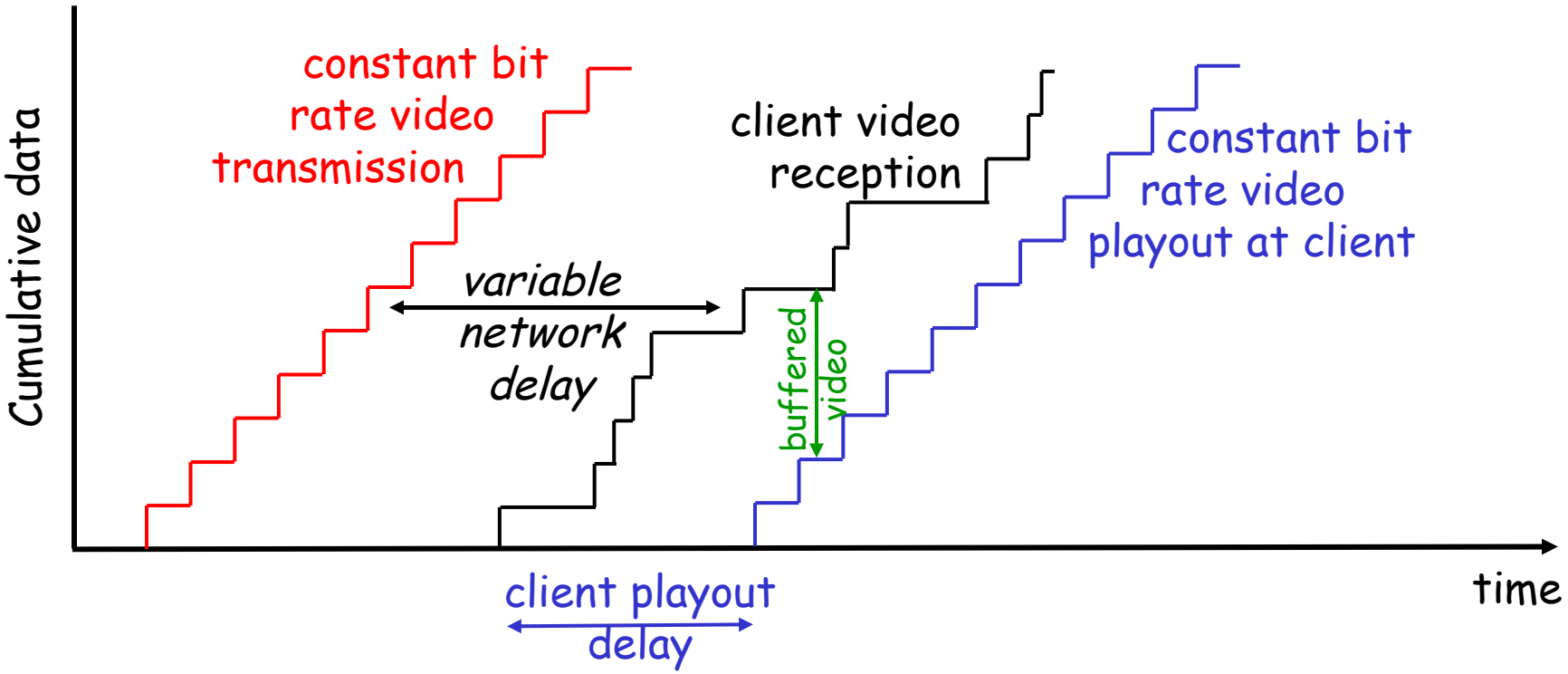
streaming:
at this time, client playing out early part of video, while server still sending later part of video

Streaming Multimedia: Client Buffering



- Client-side buffering, playout delay compensate for network-added delay, delay jitter
- Tradeoff end-to-end delay vs. loss

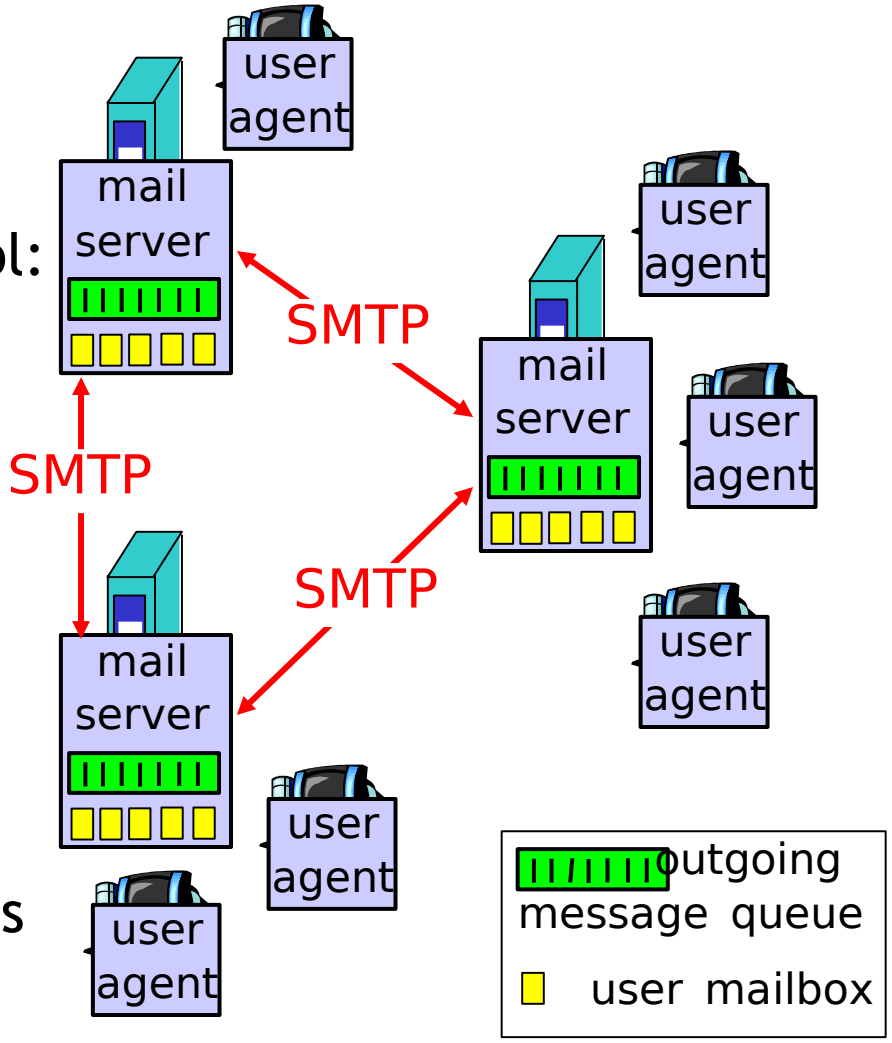
Streaming Multimedia: Client Buffering



- Client-side buffering, playout delay compensate for network-added delay, delay jitter
- Packet arriving late:
  - keep playout delay -> lost information
  - adjust playout delay -> interruption

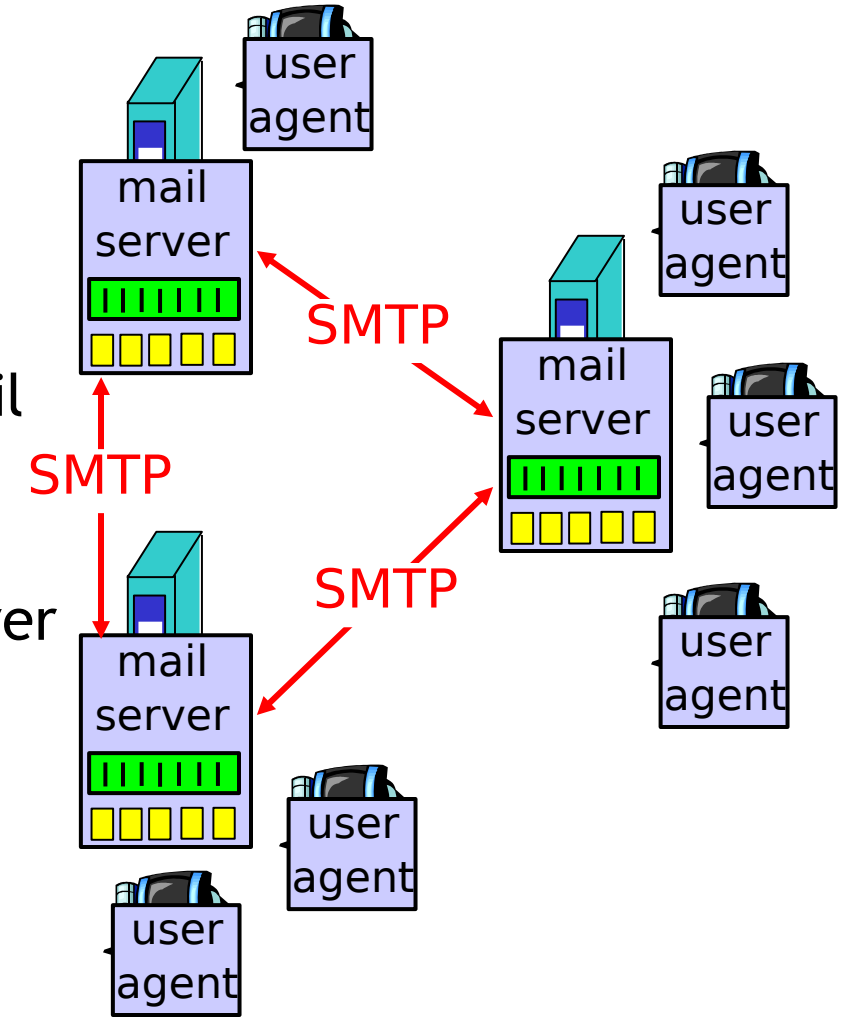
Electronic Mail

- 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., Eudora, Outlook, elm, Netscape Messenger
  - 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

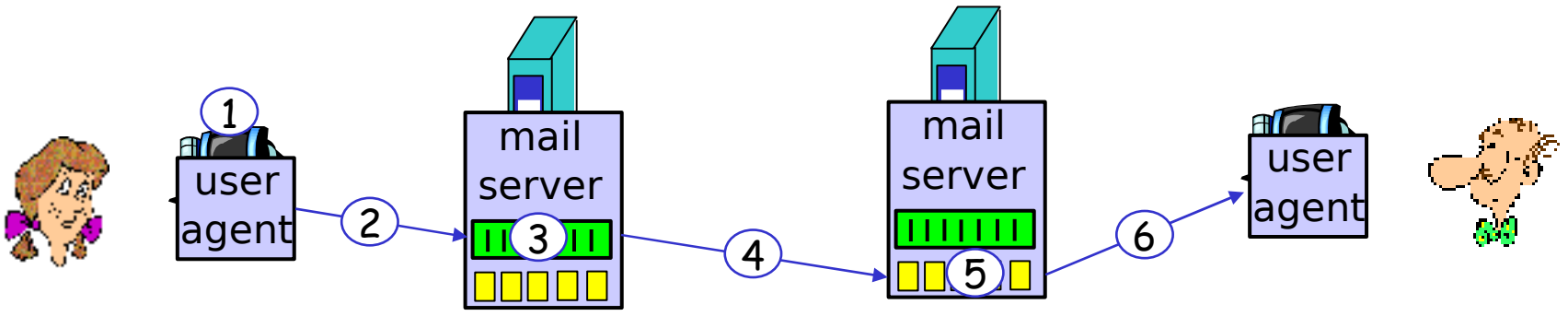


Electronic Mail: SMTP [RFC 2821]

- 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
  - commands: ASCII text
  - response: status code and phrase
- Messages must be in 7-bit ASCII

Scenario: Alice Sends Message to Bob

1) Alice uses user agent (UA) to compose message and "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



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@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line 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
```

Try SMTP Interaction for Yourself:

- **`telnet servername 25`**
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

The above lets you send email without using email client (reader), and it lets you pretend you are someone else (spammers!)

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 msg
- SMTP: multiple objects sent in multipart msg

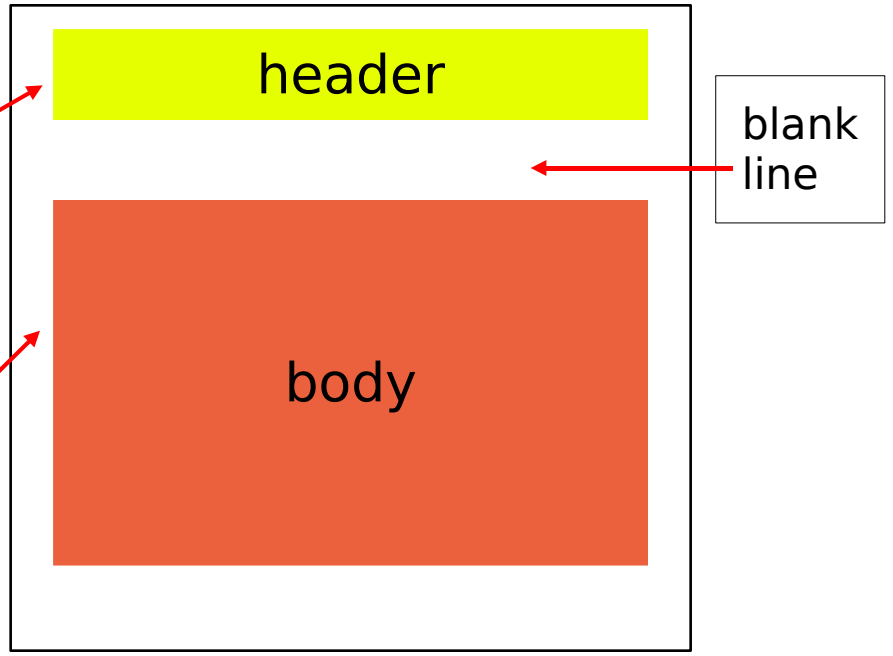
Mail Message Format

SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

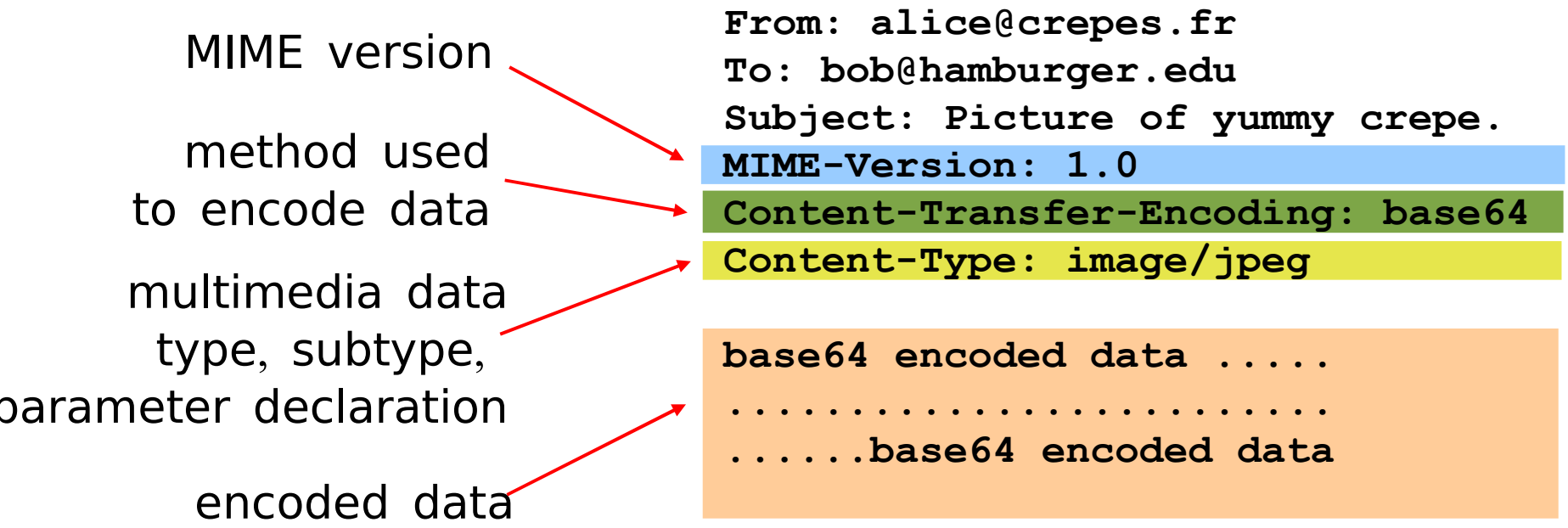
- header lines, e.g.,
  - To:
  - From:
  - Subject:

  different from SMTP commands!
- body
  - the “message”, ASCII characters only



Message Format: Multimedia Extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type



MIME Types

`Content-Type: type/subtype; parameters`

Text

- example subtypes: **`plain`**, **`html`**

Image

- example subtypes: **`jpeg`**, **`gif`**

Audio

- example subtypes: **`basic`** (8-bit mu-law encoded), **`32kadpcm`** (32 kbps coding)

Video

- example subtypes: **`mpeg`**, **`quicktime`**

Application

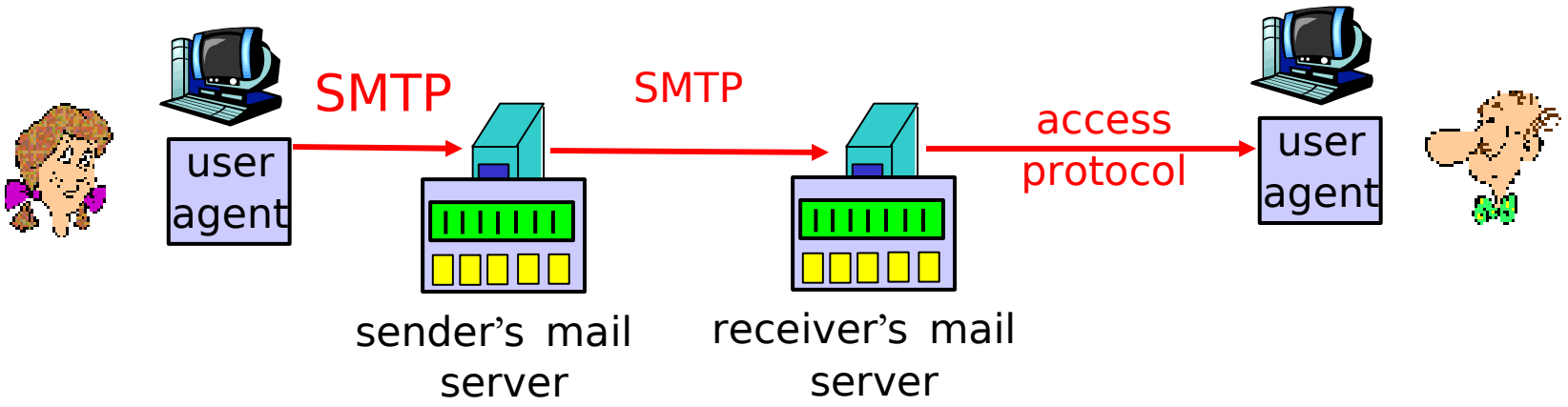
- other data that must be processed by reader before "viewable"
- example subtypes: **`msword`**, **`octet-stream`**

Multipart Type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=StartOfNextPart

--StartOfNextPart
Dear Bob, Please find a picture of a crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
base64 encoded data .....
.........................
......base64 encoded data
--StartOfNextPart
Do you want the recipe?
```

Mail Access Protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - HTTP: 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 "download and delete" mode.
  - Bob cannot re-read e-mail if he changes client
  - "Download-and-keep": copies of messages on different clients
  - POP3 is stateless across sessions
- IMAP
  - Keep all messages in one place: the server
  - Allows user to organize messages in folders
  - IMAP keeps user state across sessions:
    - names of folders and mappings between message IDs and folder name

Finite State Machines (FSM): What for?

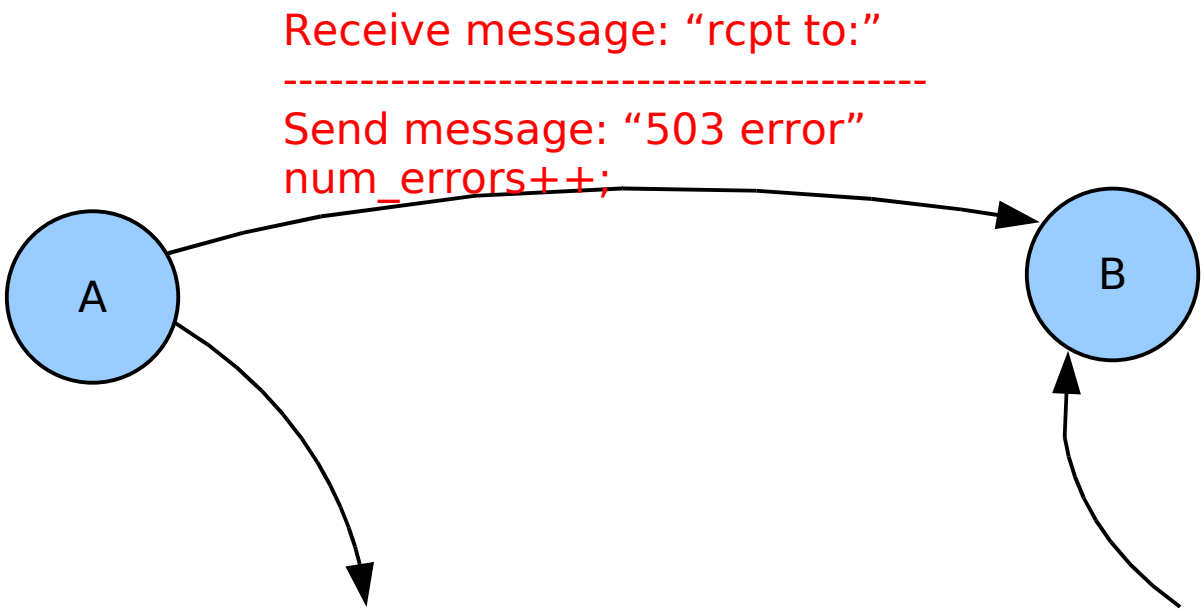
- Recall: a protocol consists of
  - two or more protocol entities
  - messages that these entities exchange in order to carry out some well-defined task
- Need a formalism
  - Specify what the entities do
    - e.g., if mail server receives “rcpt to:” without being preceded by “mail from:”, then generate error “503 Error: need MAIL command”
  - Specify what the protocol achieves
    - e.g., TCP three-way handshake ensures that connection state exists on each side
  - Analyze protocol behavior
    - e.g., prove that it is not possible that a protocol ever gets into a deadlock, i.e., a state from which it is impossible to move out

Finite State Machine: Key Concepts

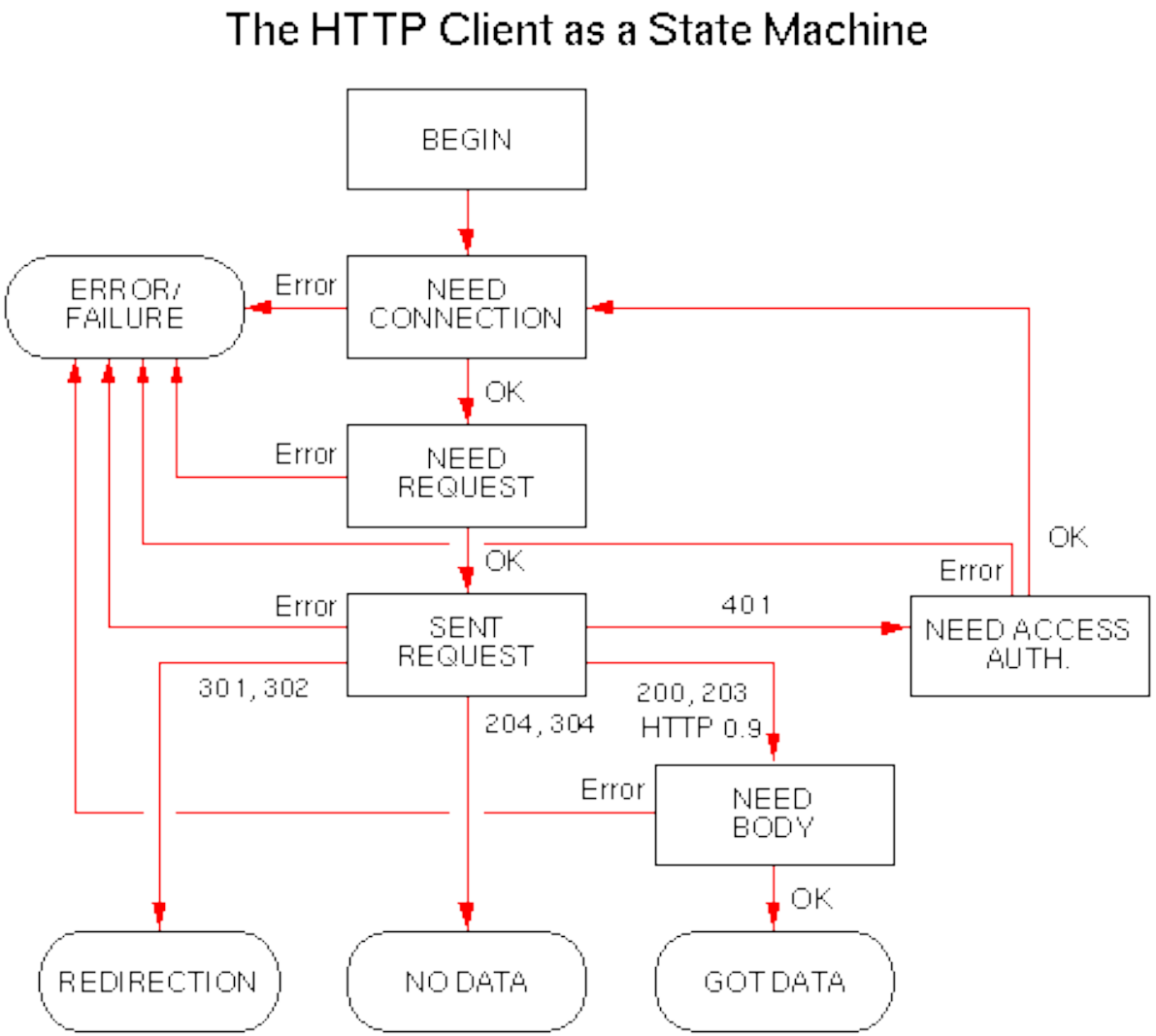
- Finite State Machines (FSMs):
  - FSM receives “signals” and takes “actions”
  - Signals:
    - receipt of a message from other entity
    - receipt of a request from protocol layer above
    - timer times out
  - Actions:
    - send a message to other entity
    - send a notification to protocol layer above
    - start a timer
    - set a variable, do a computation,...
  - State: result of the past, fully determines future behavior
  - Transition: moving from one state to the next in response to external signals

Finite State Machine

- Can be represented as a directed graph
  - Vertex (node): state
  - Edge (arrow): transition, labelled with condition for transition and action taken in response to this transition



HTTP Client FSM



(source: W3C)