CS 43: Computer Networks
HTTP and the Web

Kevin Webb

Swarthmore College

February 1, 2022

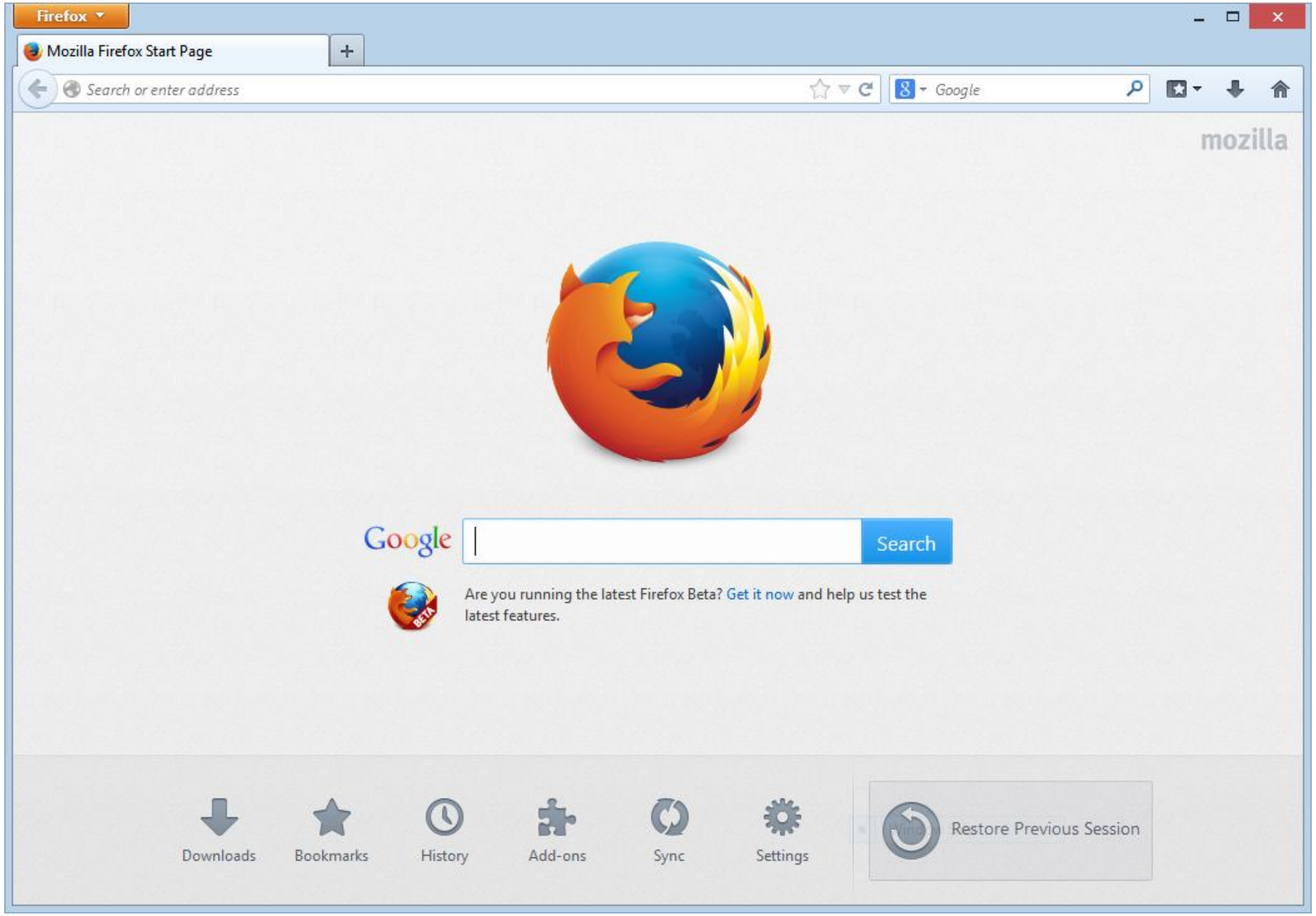
Announcements / Reminders

- Register your clicker!

- CS Mentorship program needs your help

- Clicker frequency test

What IS A Web Browser?



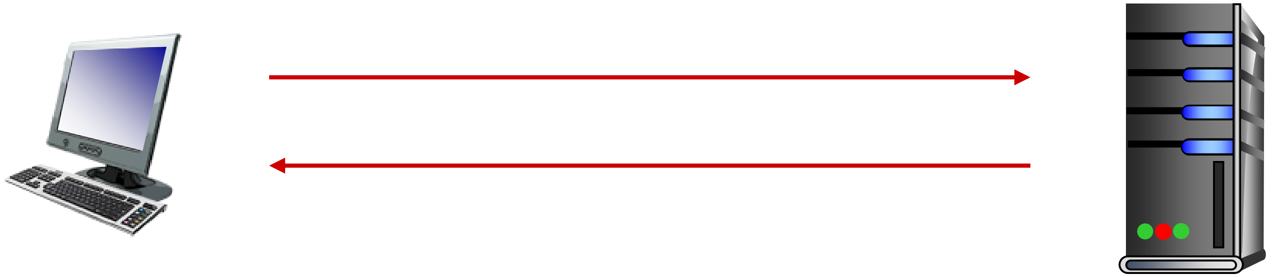
Are you running the latest Firefox Beta? Get it now and help us test the latest features.

HTTP Overview

1. User types in a URL.

   http://some.host.name.tld/directory/name/file.ext

HTTP Overview



2. Browser establishes connection with server.

Looks up "some.host.name.tld"

Calls connect()

HTTP Overview

3. Browser requests the corresponding data.

```
GET /directory/name/file.ext HTTP/1.0
Host: some.host.name.tld
[other optional fields, for example:]
User-agent: Mozilla/5.0 (Windows NT 6.1; WOW64)
Accept-language: en
[Blank line]
```

HTTP Overview

4. Server responds with the requested data.

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 1299
Date: Sun, 01 Sep 2013 21:26:38 GMT
[Blank line]
(Data data data data…)
```

HTTP Overview

5. Browser renders the response, fetches any additional objects, and closes the connection.

HTTP Overview

5. Browser renders the respo
additional objects, and close

```
<html>
  <head>
    <title>Page title!</title>
  </head>

  <body>
    <p>a paragraph of text</p>

    <img src="http://site/cat.jpg">
    <img src="http://site/dog.jpg">
    ...
  </body>
</html>
```

HTTP Overview

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. Browser renders the response, fetches other objects, and closes the connection.

It’s a document retrieval system, where documents point to (link to) each other, forming a “web”.

HTTP Overview (Lab 1)

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. ~~Browser renders the response, fetches other objects,~~ and closes the connection.

It’s a document retrieval system, where documents point to (link to) each other, forming a “web”.

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet demo.cs.swarthmore.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at example server. Anything typed is sent to server on port 80 at demo.cs.swarthmore.edu

2. Type in a GET HTTP request:

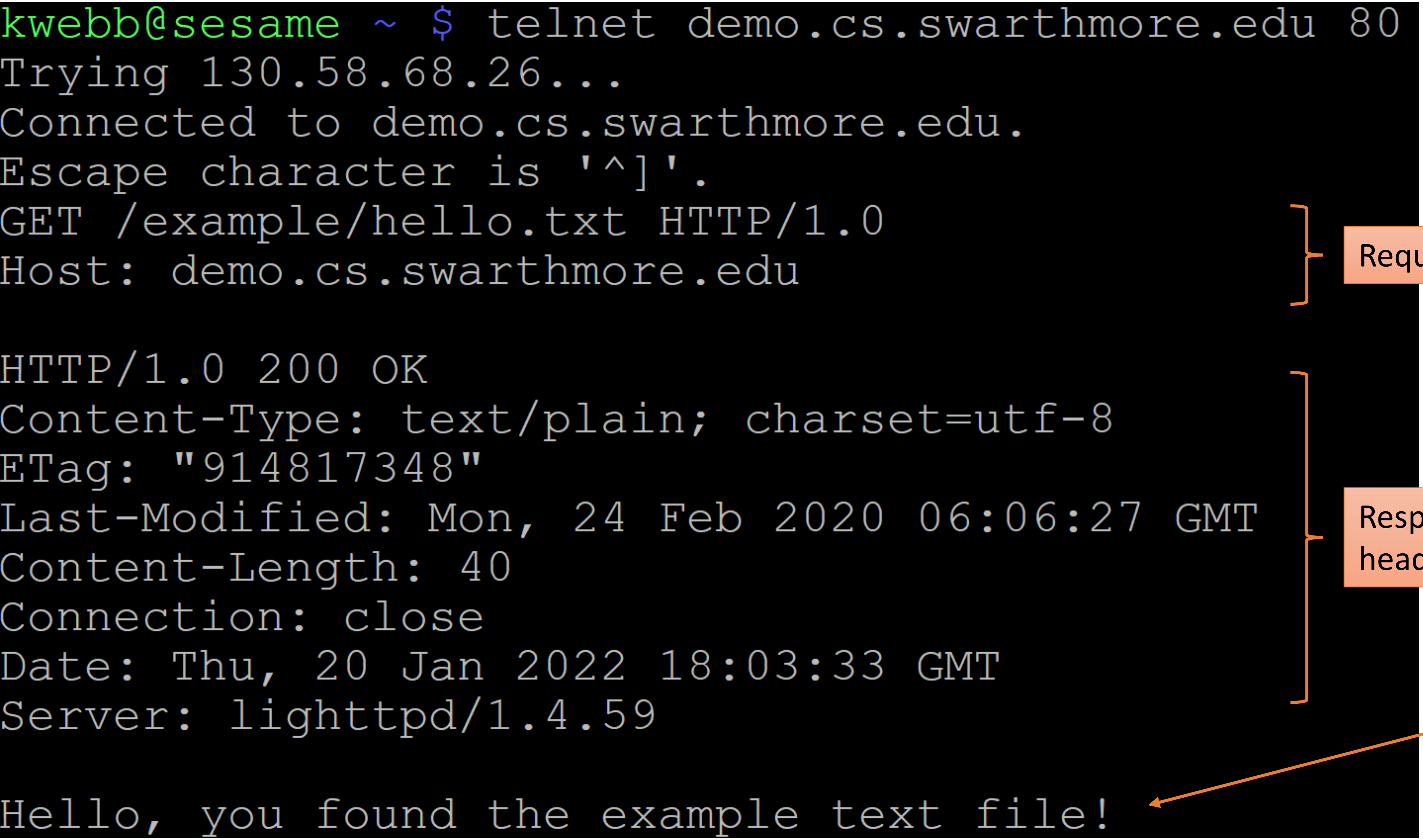
```
GET / HTTP/1.0
Host: demo.cs.swarthmore.edu
(blank line)
```

By typing this in (hit enter twice), you send this minimal (but complete) GET request to the HTTP server.

3. Look at response message sent by HTTP server!

Example (live demo)

Example



```
kwebb@sesame ~ $ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^]'.
GET /example/hello.txt HTTP/1.0
Host: demo.cs.swarthmore.edu

HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
ETag: "914817348"
Last-Modified: Mon, 24 Feb 2020 06:06:27 GMT
Content-Length: 40
Connection: close
Date: Thu, 20 Jan 2022 18:03:33 GMT
Server: lighttpd/1.4.59

Hello, you found the example text file!
```

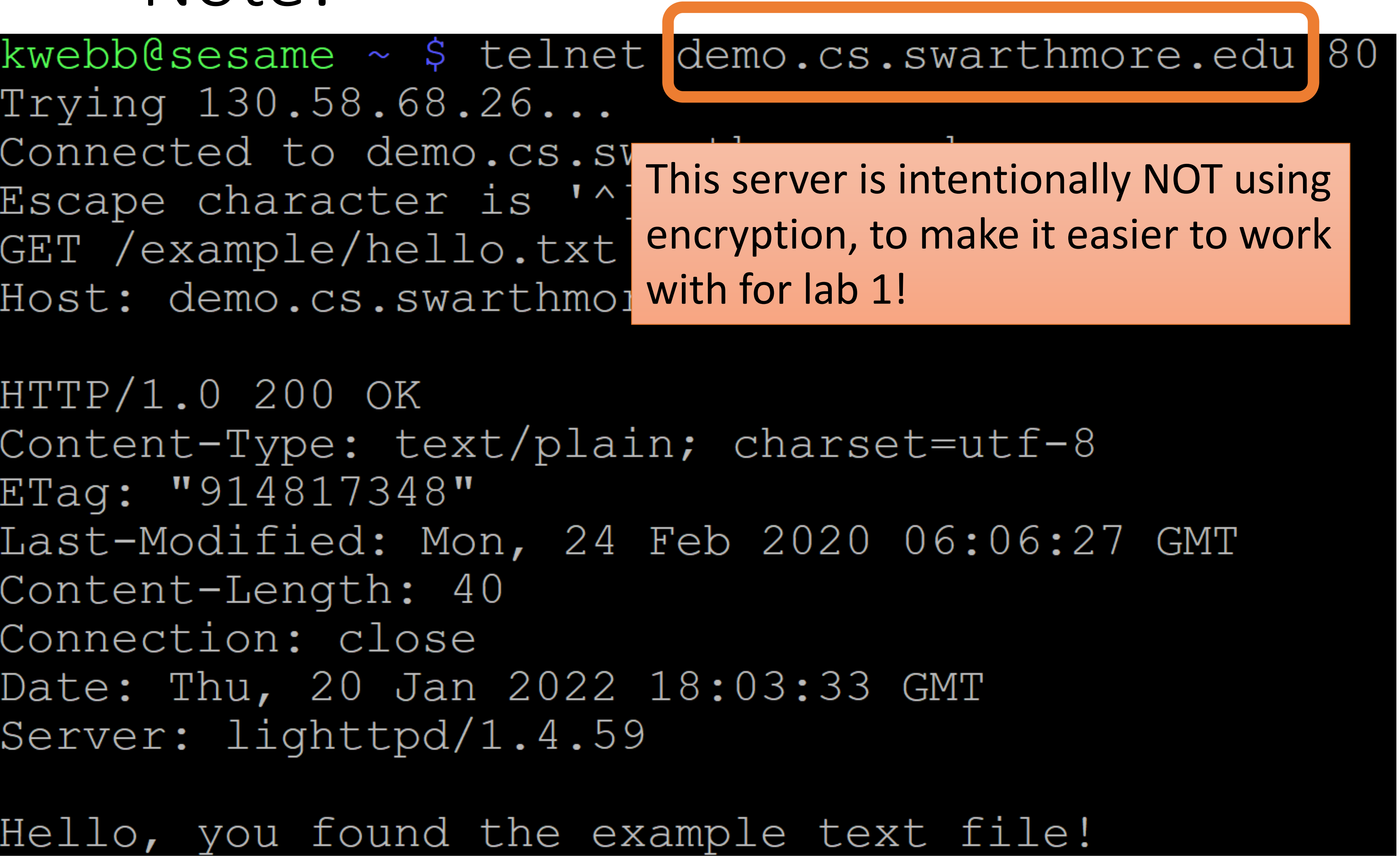
Request

Response headers

Response body

(This is what you should be saving to file in lab 1.)

Note!



```
kwebb@sesame ~ $ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.sw
Escape character is '^
GET /example/hello.txt
Host: demo.cs.swarthmo

HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
ETag: "914817348"
Last-Modified: Mon, 24 Feb 2020 06:06:27 GMT
Content-Length: 40
Connection: close
Date: Thu, 20 Jan 2022 18:03:33 GMT
Server: lighttpd/1.4.59

Hello, you found the example text file!
```

This server is intentionally NOT using encryption, to make it easier to work with for lab 1!

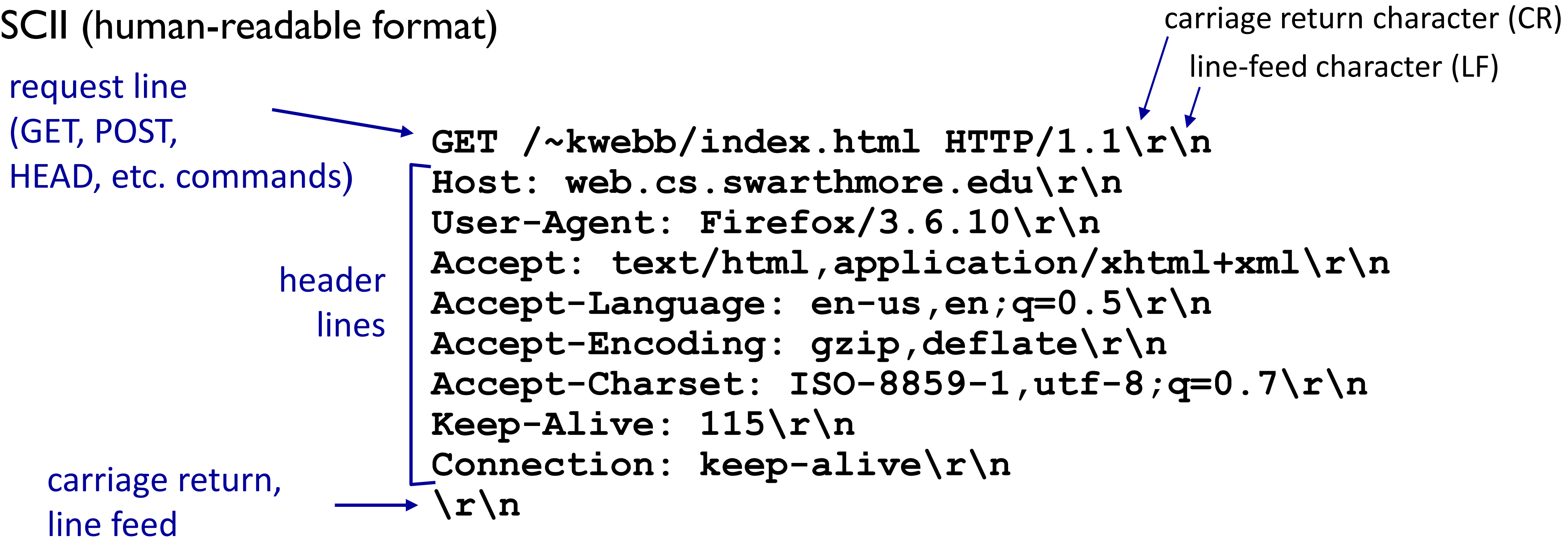
HTTPS (live demo)

- Telnet transfers unencrypted data ("clear text")
  - Great for learning
  - Not so great for real world security / privacy

- For a similar (interactive) command line experience with encryption:
  - `openssl s_client -crlf -connect server.name:443`

HTTP request message

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



request line (GET, POST, HEAD, etc. commands)

header lines

carriage return, line feed

carriage return character (CR)

line-feed character (LF)

```
GET /~kwebb/index.html HTTP/1.1\r\n
Host: web.cs.swarthmore.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

Why do we have these \r\n (CRLF) things all over the place?

```
GET /~kwebb/index.html HTTP/1.1\r\n
Host: web.cs.swarthmore.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

A. They’re generated when the user hits ‘enter’.

B. They signal the end of a field or section.

C. They’re important for some other reason.

D. They’re an unnecessary protocol artifact.

How else might we delineate messages?

(What are the good/bad properties of each of these ideas?)

A. There’s not much else we can do.

B. Force all messages to be the same size.

C. Send the message size prior to the message.

D. Some other way (discuss).

HTTP is all text…

- Makes the protocol simple
  - Easy to delineate message (\r\n)
  - (Relatively) human-readable
  - No worries about encoding or formatting data
  - Variable length data

- Not the most efficient
  - Many protocols use binary fields
    - Sending “12345678” as a string is 8 bytes
    - As an integer, 12345678 needs only 4 bytes
  - The headers may come in any order
  - Requires string parsing / processing

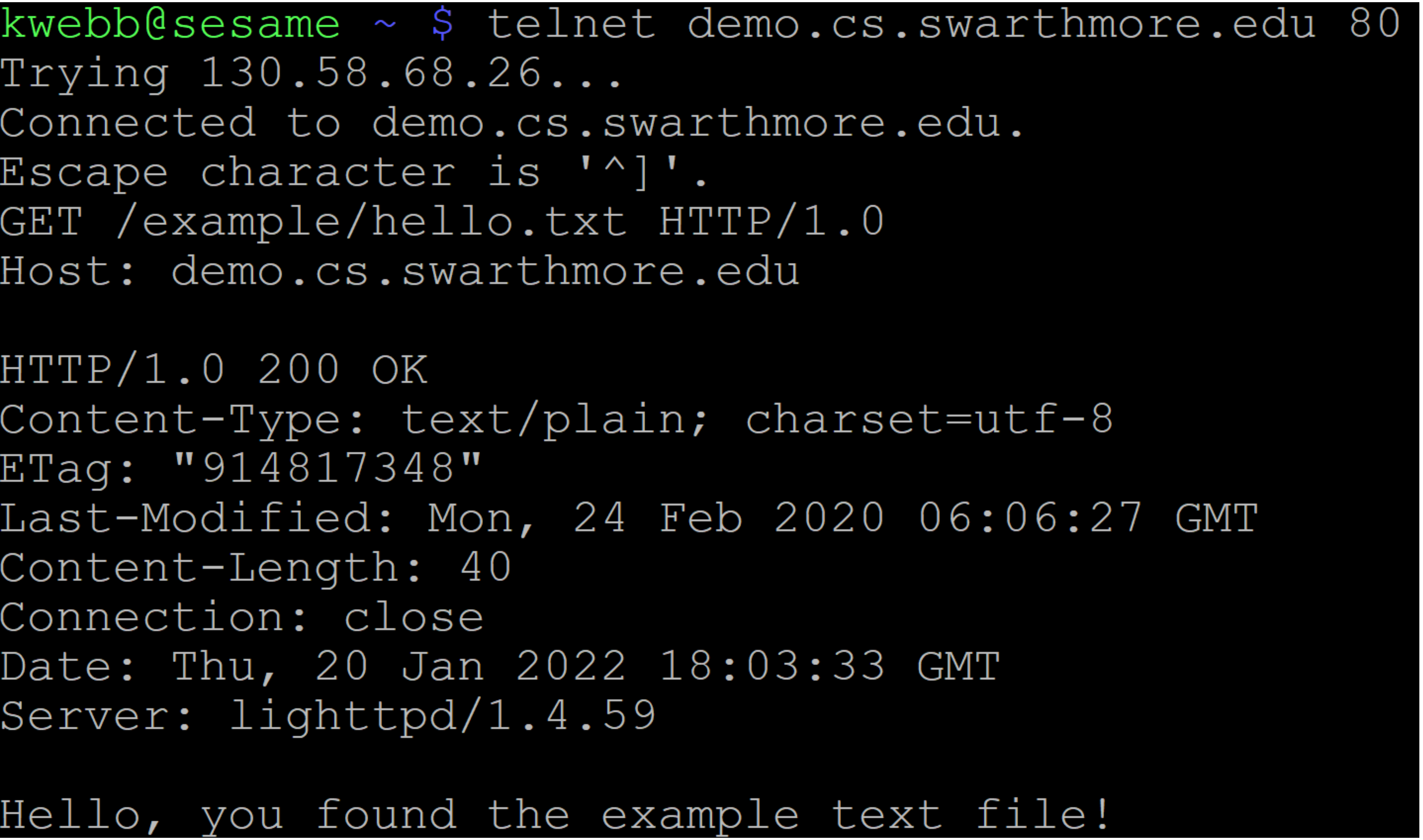
HTTP is all text…

- The HTTP **PROTOCOL** is all text
  - That is, the messages that are required (request and response)
  - All headers are text

- The BODY of a message might **NOT** be text

- This distinction is critically important for lab 1!
  - Fine to use string functions on HTTP messages
  - You better not use string functions on body data

Visualizing HTTP: telnet

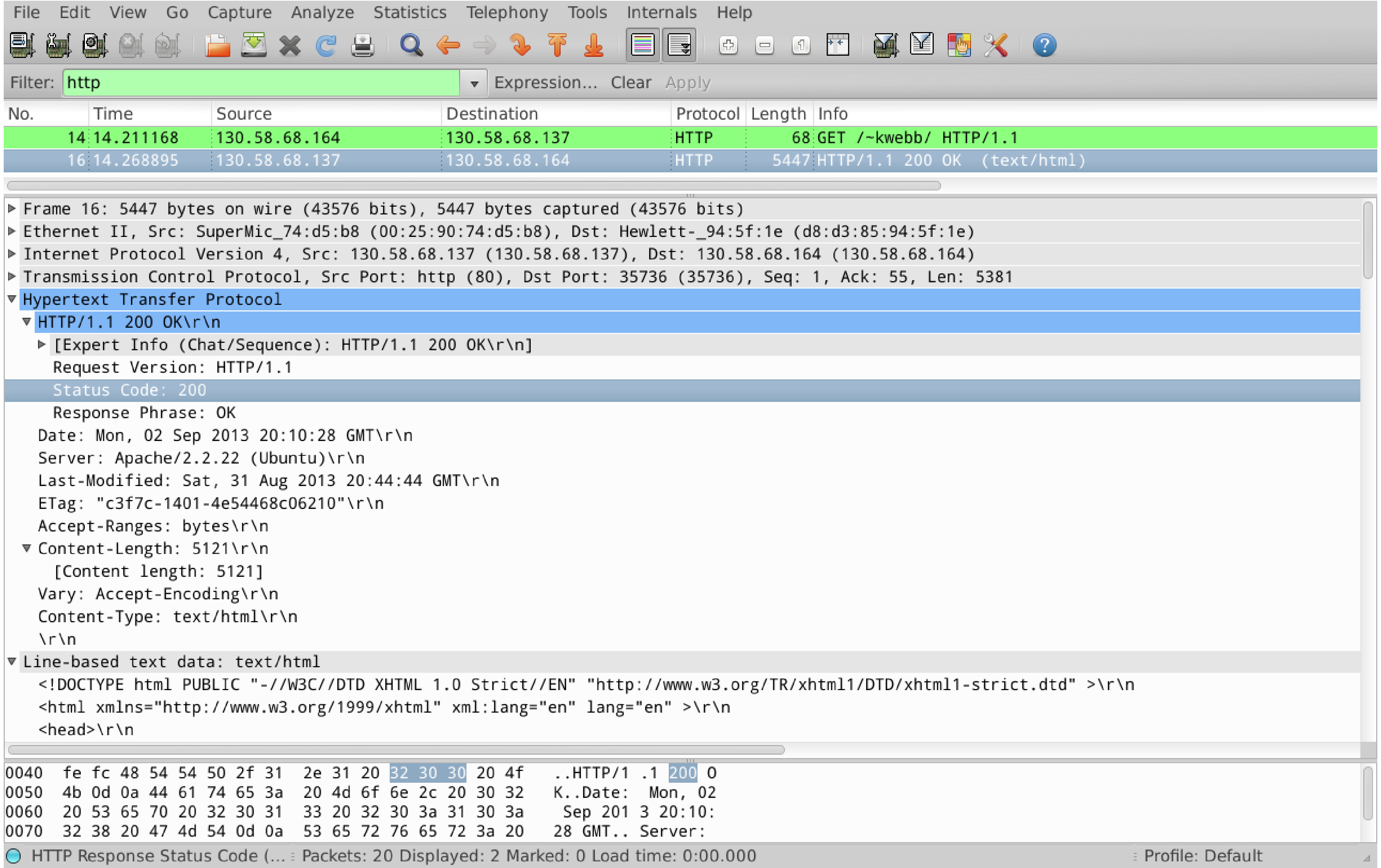


```
kwebb@sesame ~ $ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^]'.
GET /example/hello.txt HTTP/1.0
Host: demo.cs.swarthmore.edu

HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
ETag: "914817348"
Last-Modified: Mon, 24 Feb 2020 06:06:27 GMT
Content-Length: 40
Connection: close
Date: Thu, 20 Jan 2022 18:03:33 GMT
Server: lighttpd/1.4.59

Hello, you found the example text file!
```

Visualizing HTTP: wireshark



Filter: http

```
No.   Time       Source         Destination    Protocol Length Info
   14 14.211168  130.58.68.164  130.58.68.137  HTTP         68 GET /~kwebb/ HTTP/1.1
   16 14.268895  130.58.68.137  130.58.68.164  HTTP       5447 HTTP/1.1 200 OK  (text/html)
```

```
▶ Frame 16: 5447 bytes on wire (43576 bits), 5447 bytes captured (43576 bits)
▶ Ethernet II, Src: SuperMic_74:d5:b8 (00:25:90:74:d5:b8), Dst: Hewlett-_94:5f:1e (d8:d3:85:94:5f:1e)
▶ Internet Protocol Version 4, Src: 130.58.68.137 (130.58.68.137), Dst: 130.58.68.164 (130.58.68.164)
▶ Transmission Control Protocol, Src Port: http (80), Dst Port: 35736 (35736), Seq: 1, Ack: 55, Len: 5381
▼ Hypertext Transfer Protocol
  ▼ HTTP/1.1 200 OK\r\n
    ▶ [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      Request Version: HTTP/1.1
      Status Code: 200
      Response Phrase: OK
    Date: Mon, 02 Sep 2013 20:10:28 GMT\r\n
    Server: Apache/2.2.22 (Ubuntu)\r\n
    Last-Modified: Sat, 31 Aug 2013 20:44:44 GMT\r\n
    ETag: "c3f7c-1401-4e54468c06210"\r\n
    Accept-Ranges: bytes\r\n
  ▼ Content-Length: 5121\r\n
      [Content length: 5121]
    Vary: Accept-Encoding\r\n
    Content-Type: text/html\r\n
    \r\n
▼ Line-based text data: text/html
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >\r\n
    <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" >\r\n
    <head>\r\n
```

```
0040  fe fc 48 54 54 50 2f 31  2e 31 20 32 30 30 20 4f   ..HTTP/1 .1 200 O
0050  4b 0d 0a 44 61 74 65 3a  20 4d 6f 6e 2c 20 30 32   K..Date:  Mon, 02
0060  20 53 65 70 20 32 30 31  33 20 32 30 3a 31 30 3a    Sep 201 3 20:10:
0070  32 38 20 47 4d 54 0d 0a  53 65 72 76 65 72 3a 20   28 GMT.. Server:
```

HTTP Response Status Code (... Packets: 20 Displayed: 2 Marked: 0 Load time: 0:00.000 Profile: Default

Request Method Types (“verbs”)

HTTP/1.0 (1996):

- GET
  - Requests page.
- POST
  - Uploads user response to a form.
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1 (1997 & 1999):

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

Request Method Types (“verbs”)

HTTP/1.0 (1996):

- GET
  - Requests page.
- POST
  - Uploads user response to a form.
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1 (1997 & 1999):

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field
- TRACE, OPTIONS, CONNECT, PATCH
- (+) Persistent connections

Request Method Types (“verbs”)

HTTP/1.0 (1996):

- **GET**
  - Requests page.
- **POST**
  - Uploads user response to a form.
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1 (1997 & 1999):

- **GET, POST**, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field
- TRACE, OPTIONS, CONNECT, PATCH
- (+) Persistent connections

Requests with user input / form data

GET (in-URL) method:

- uses GET method
- input is uploaded in URL field of request line:

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

POST method:

- web page often includes form input
- input is uploaded to server in request entity body

GET vs. POST

- GET should only be used for *idempotent* requests
  - Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

GET vs. POST

- GET should only be used for *idempotent* requests
  - Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

How many of the following operations are idempotent?

I. Incrementing a variable
II. Assigning a value to a variable
III. Allocating memory
IV. Compiling a program

A. None of them
B. One of them
C. Two of them
D. Three of them
E. All of them

GET vs. POST

- GET should only be used for *idempotent* requests.
  - Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

- POST should be used when…
  - A request changes the state of the server (or underlying DB)
  - Sending a request twice would be harmful
    - (Some) browsers / sites warn about sending multiple POST requests
  - Users are inputting non-ASCII characters
  - Input may be very large

When might you use GET vs. POST?

	GET	POST
A.	Forum post	Search terms, Pizza order
B.	Search terms, Pizza order	Forum post
C.	Search terms	Forum post, Pizza order
D.	Forum post, Search terms, Pizza Order	
E.		Forum post, Search terms, Pizza Order

HTTP response message

status line
(protocol
status code
status phrase)

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

response
header
lines

response body,
e.g., requested
HTML file

HTTP response status codes

- Status code appears in first line of server-to-client response message.
- Some common response codes:

`200 OK`

- Request succeeded, requested object later in this message (body)

`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`

- You don’t have permission to read the object

`404 Not Found`

- Requested document not found on this server

`505 HTTP Version Not Supported`

HTTP response status codes

- Status code appears in first line of server-to-client response message.
- Many others too. Search "list of HTTP status codes".
- Some of my favorites:

`420 Enhance Your Calm (twitter)`

- Slow down, you're being rate limited

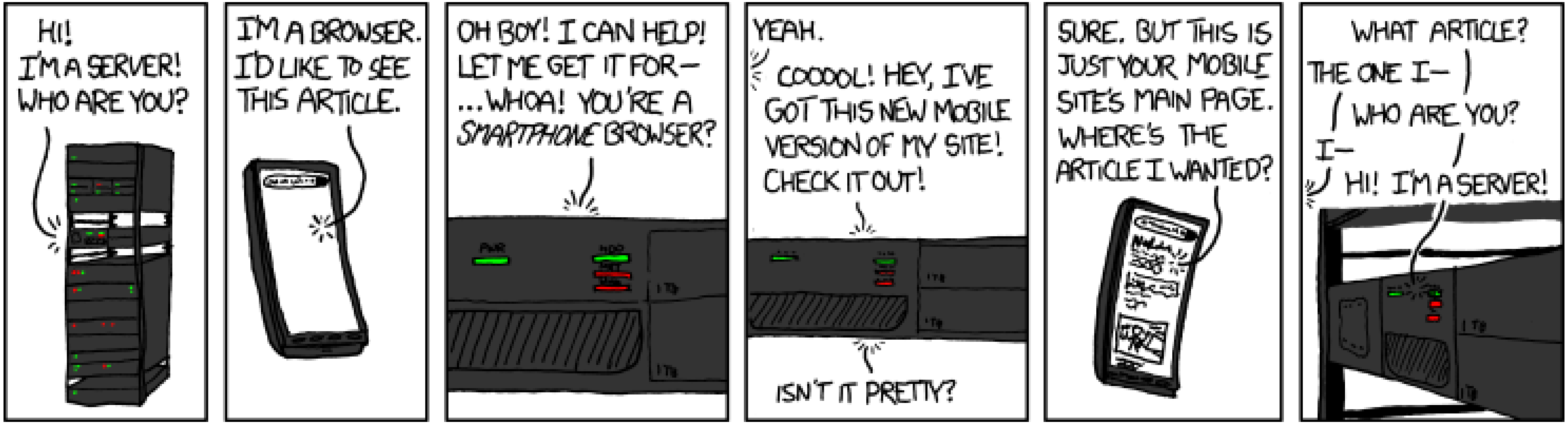
`451 Unavailable for Legal Reasons`

- Censorship?

`418 I'm a Teapot`

- Response from a teapot requested to brew a beverage (announced Apr 1)

State(less) Protocols


HI!
I'M A SERVER!
WHO ARE YOU?
I'M A BROWSER.
I'D LIKE TO SEE THIS ARTICLE.
OH BOY! I CAN HELP! LET ME GET IT FOR—
...WHOA! YOU'RE A *SMARTPHONE* BROWSER?
PWR
HDD
1 TB
YEAH.
COOOOL! HEY, I'VE GOT THIS NEW MOBILE VERSION OF MY SITE! CHECK IT OUT!
ISN'T IT PRETTY?
SURE. BUT THIS IS JUST YOUR MOBILE SITE'S MAIN PAGE. WHERE'S THE ARTICLE I WANTED?
WHAT ARTICLE?
THE ONE I—
WHO ARE YOU?
I—
HI! I'M A SERVER!

(XKCD #869, "Server Attention Span")

State(less) Protocols

- Original web: simple document retrieval
- Server is not required to keep state between connections
  (often it might want to though!)
- Client is not required to identify itself
  (server might refuse to talk otherwise though!)

Keeping state: cookies

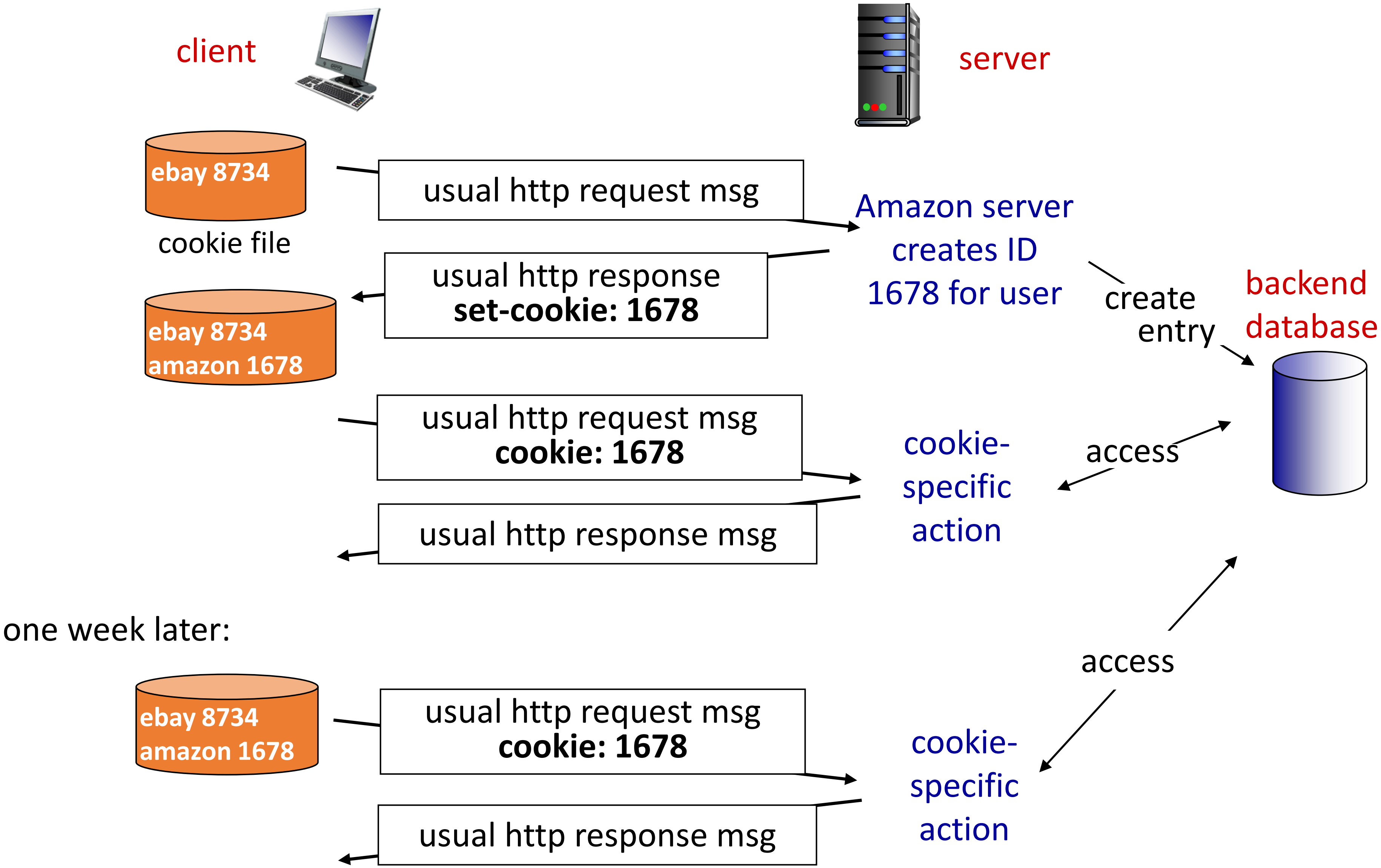
Many web sites use cookies

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 accesses the Web from her PC
- She visits specific e-commerce site for the first time
- When initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID



client

server

ebay 8734

cookie file

usual http request msg

Amazon server creates ID 1678 for user

create entry

backend database

usual http response
set-cookie: 1678

ebay 8734
amazon 1678

usual http request msg
cookie: 1678

cookie-specific action

access

usual http response msg

one week later:

ebay 8734
amazon 1678

usual http request msg
cookie: 1678

access

cookie-specific action

usual http response msg

Cookies

What cookies can be used for:

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

How to keep "state":

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

Cookies: Pros / Cons

- Cookies permit sites to learn a lot about you

- You may supply name and e-mail to sites (and more!)

- 3rd party cookies (from ad networks, etc) can follow you across multiple sites.
  - Ever visit a website, and the next day ALL your ads are from them?

- You COULD turn them off
  - But good luck doing anything on the internet!

HTTP Performance

HTTP Connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects requires multiple connections

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

object: image, script, stylesheet, etc.

Pseudocode Example

non-persistent HTTP

```
for object on web page:
    connect to server
    request object
    receive object
    close connection
```

persistent HTTP

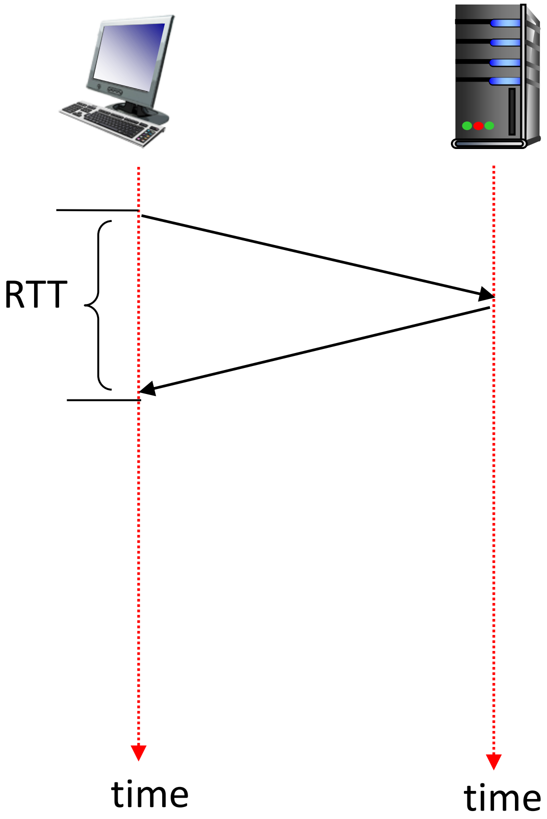
```
connect to server
for object on web page:
    request object
    receive object
close connection
```

Round Trip Time (RTT)

Round Trip Time (RTT): time for a small packet to travel from client to server and response to come back

Connection establishment (via TCP) requires one RTT.

Connection must be established prior to any other communication.



Non-Persistent HTTP Connections can download a website with several objects in…

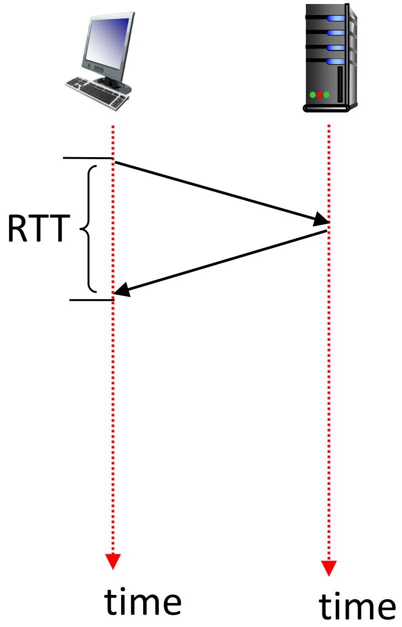
A. One RTT + (File transfer time per object)

B. (One RTT + File transfer time) per object

C. Two RTTs

D. Two RTTs + (File transfer time per object)

E. (Two RTTS + File transfer time) per object



Non-persistent HTTP: response time

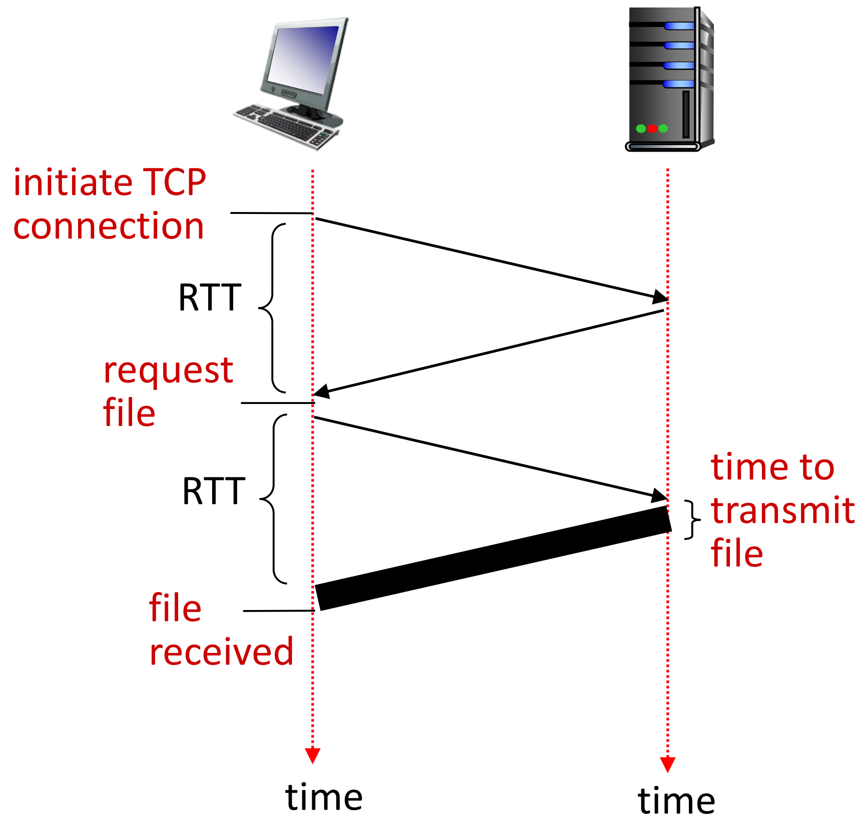
Round Trip Time (RTT): time for a small packet to travel from client to server and back

HTTP response time:

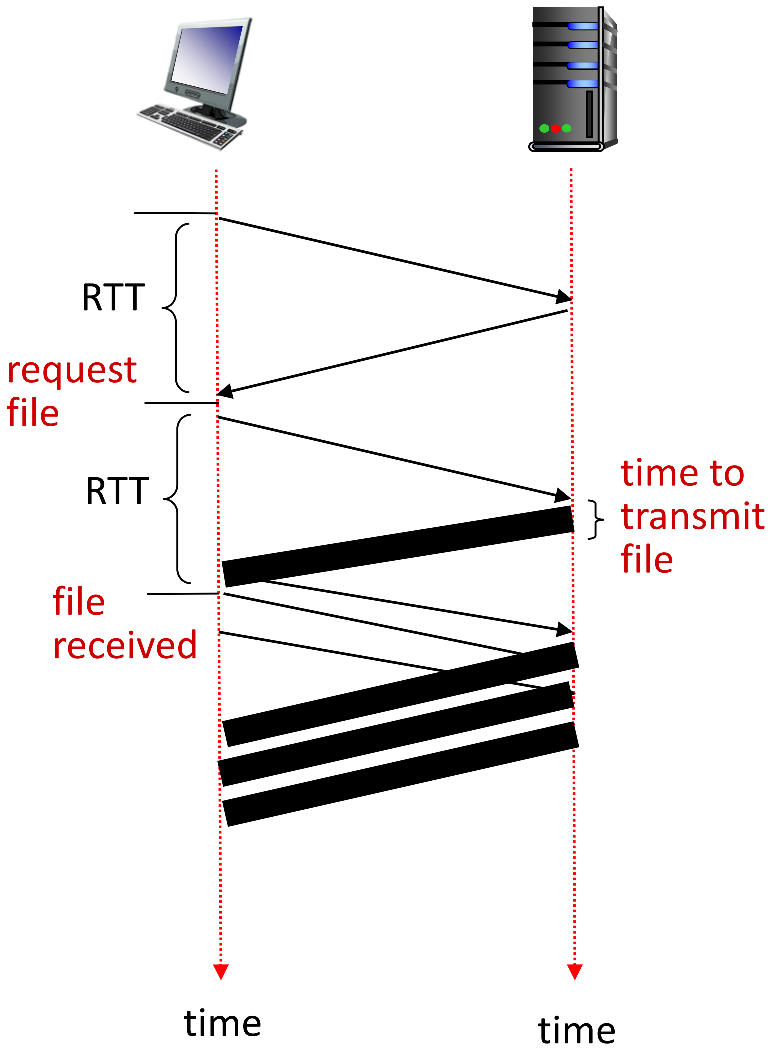
- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2RTT+ file transmission time

  For each object



Persistent Connection



```
<html>
  <head>
    <title>Page title!</title>
  </head>

  <body>
    <p>a paragraph of text</p>

    <img src="http://site/cat.jpg">
    <img src="http://site/dog.jpg">
    ...
  </body>
</html>
```

Comparison

Non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- 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 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

HTTP 2.0 (2015)

- Adds some new features for better efficiency
    - Encodes HTTP messages into a binary format to reduce size
    - Can transmit data from multiple objects concurrently instead of in series
    - (several other smaller changes)

- Most browsers support it

- Major sites support it (those with enough volume to actually benefit)

Other HTTP Verbs

HTTP/1.0 (1996):

- GET
  - Requests page.
- POST
  - Uploads user response to a form.
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1 (1997 & 1999):

- **GET**, POST, HEAD
- **PUT**
  - uploads file in entity body to path specified in URL field
- **DELETE**
  - deletes file specified in the URL field
- TRACE, OPTIONS, CONNECT, **PATCH**
- (+) Persistent connections

CRUD and REST

- Create, Read, Update, Delete (CRUD)
  - Common pattern for storing information in an application

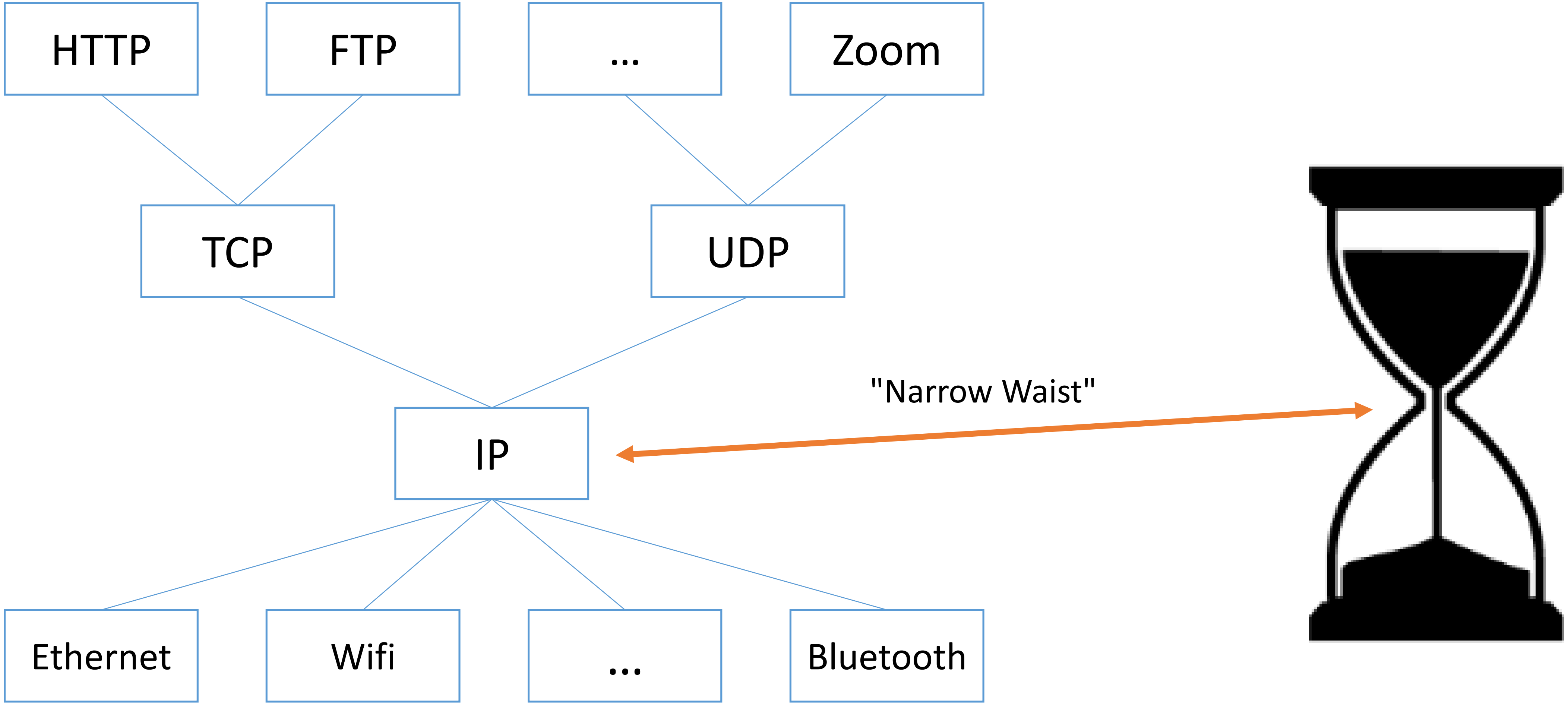
- Example: twitter
  - Create: produce new tweet
  - Read: get tweet(s) from [criteria]
  - Update: edit tweet (settings)
  - Delete: remove tweet

CRUD and REST

- Create, Read, Update, Delete (CRUD)
  - Common pattern for storing information in an application
- Example: twitter
  - Create: produce new tweet
  - Read: get tweet(s) from [criteria]
  - Update: edit tweet (settings)
  - Delete: remove tweet

- Representational state transfer (REST)
  - Use HTTP verbs to implement the common CRUD model
- Create -> PUT (or POST)
- Read -> GET
- Update -> PUT (or PATCH)
- Delete -> DELETE

Internet Protocol Suite ("Hourglass model")



HTTP

FTP

...

Zoom

TCP

UDP

IP

"Narrow Waist"

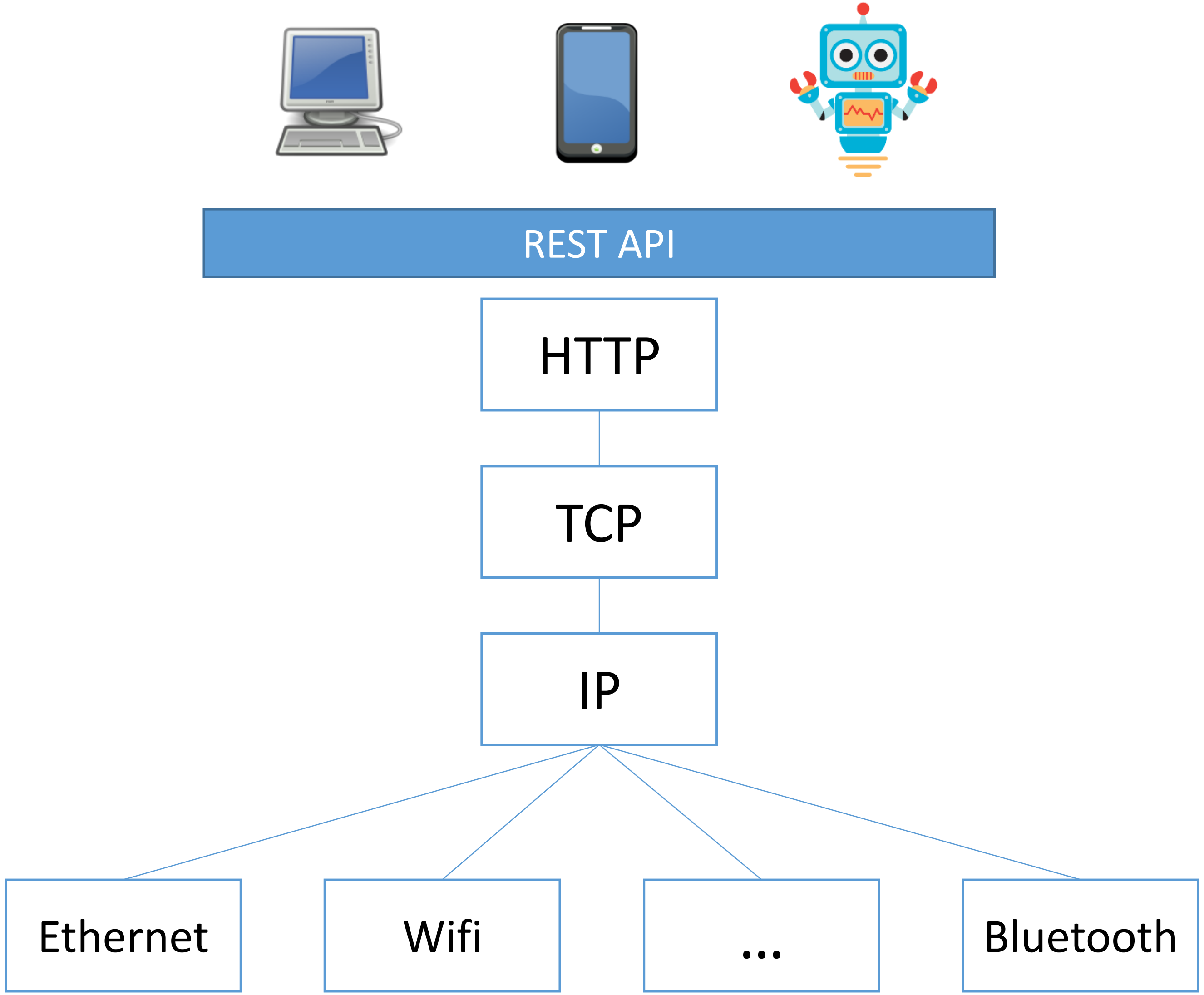
Ethernet

Wifi

...

Bluetooth

If CRUD is your application's model…



REST API

HTTP

TCP

IP

Ethernet

Wifi

…

Bluetooth

Summary

- HTTP is a text-based protocol for document retrieval
  - request and response message types
  - requests have verbs (GET, POST, etc.)
  - responses have status codes / messages
  - message sender can add arbitrary headers
  - CRLFs to delineate messages
- HTTP is stateless, but "cookie" headers allow persistent identification
- Managing connections is important for performance
- REST APIs over HTTP are super common (taking over?)