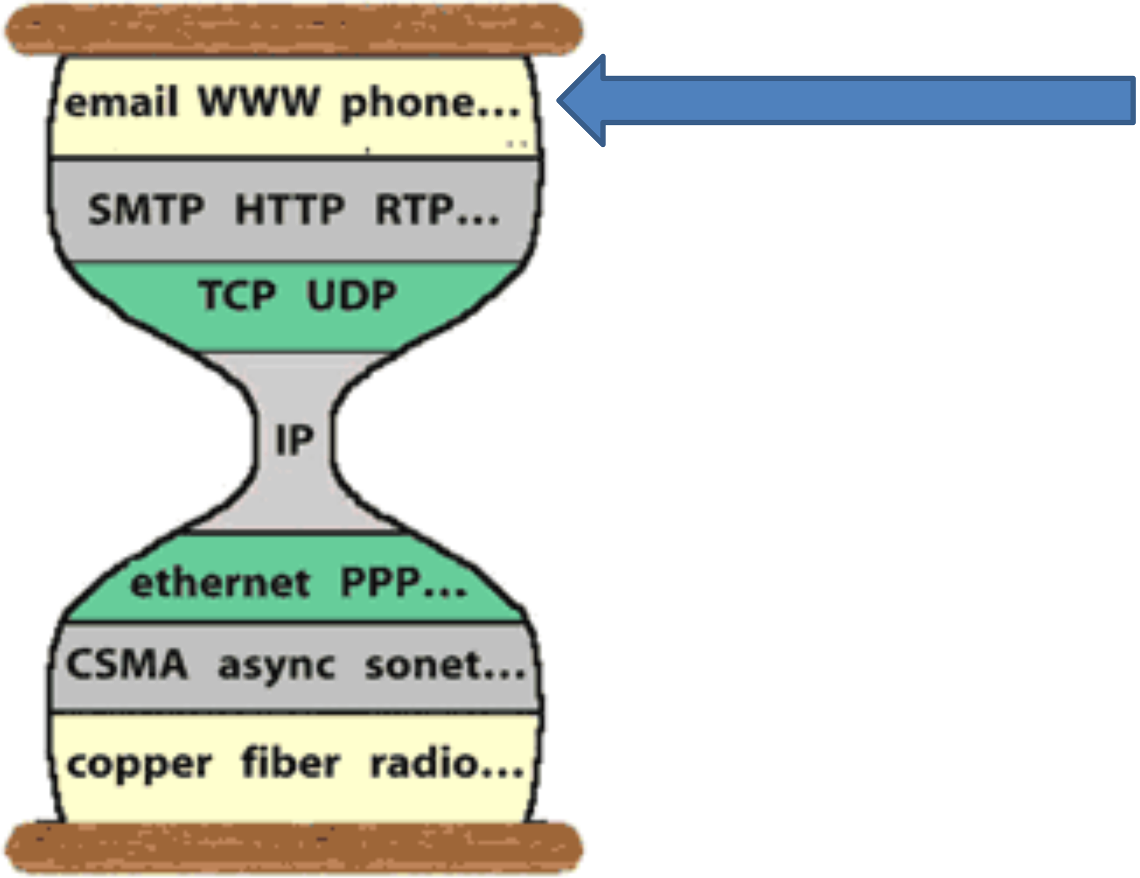
# What we covered in lecture-1

- Internet: made of a huge number of hosts and routers, interconnected by physical and wireless links
- Hosts: run bunch of protocols to exchange data with each other
- Routers: run bunch of protocols in order to move data to their destinations
- Protocols are organized in layers:
  - Application protocols
  - Transport protocols
  - Network protocols
  - Link layer protocols
- Very quick intro to git, Vagrant, Docker (more during discussion sections)

# Application Layer



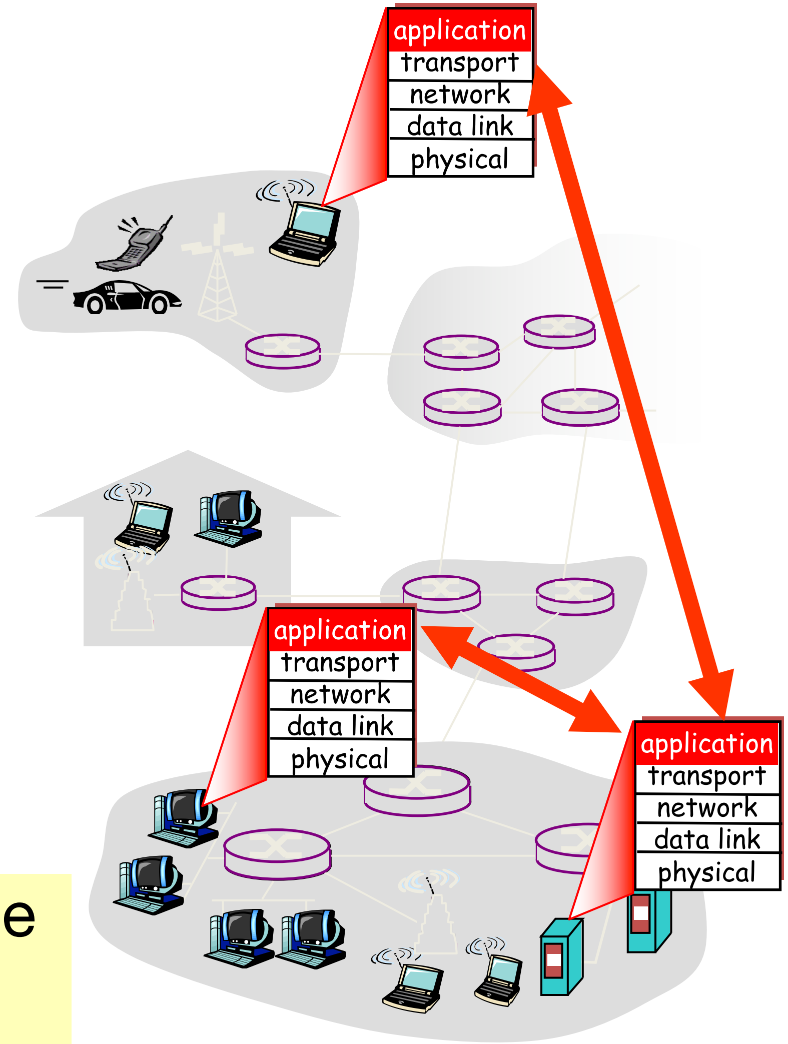
You will learn:

Principles of creating network applications

Details of several application-level protocols

# Some popular network applications

- Web
- E-mail
- Instant messaging
- P2P file sharing
- Multi-user network games
- Video streaming (e.g., YouTube)
- Voice-over-IP (e.g. skype)



Application processes communicate with each other using application protocols

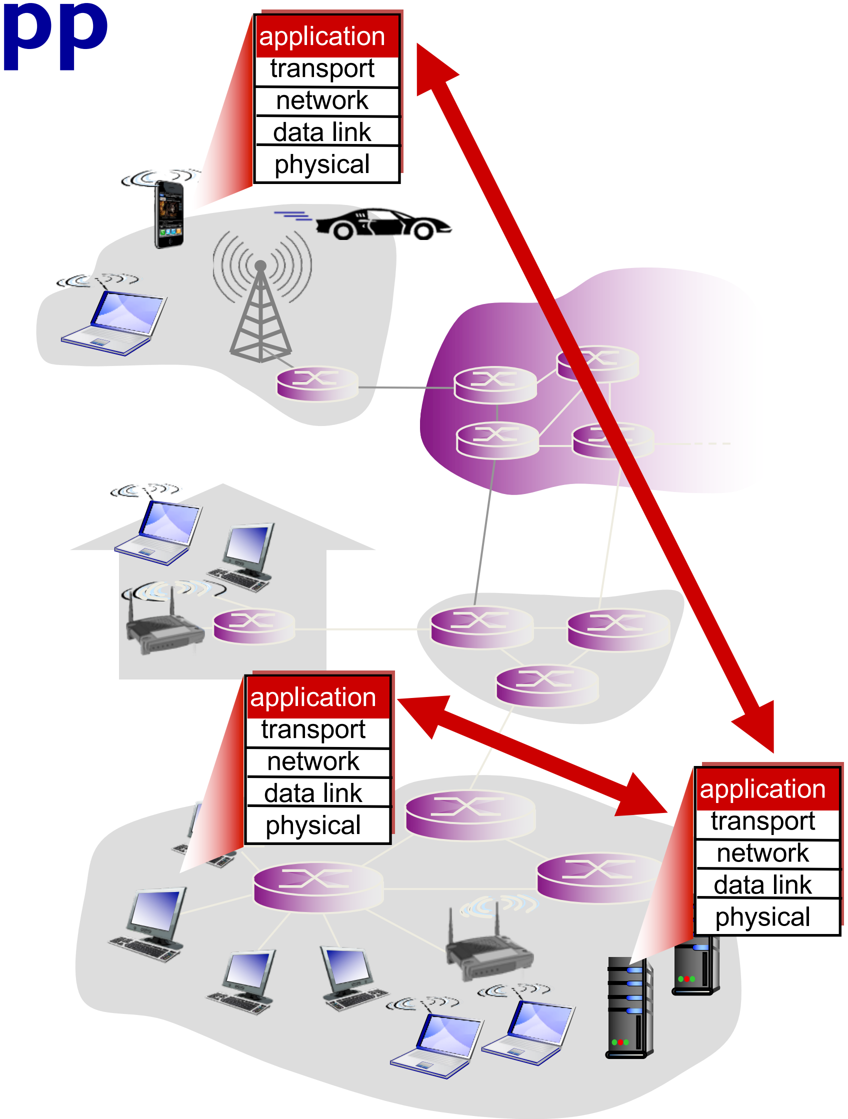
# Creating a Network App

write programs that:

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

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



# Application architectures

Possible structure of applications:

- client-server
- peer-to-peer (P2P)

# Client-Server Architecture

servers:

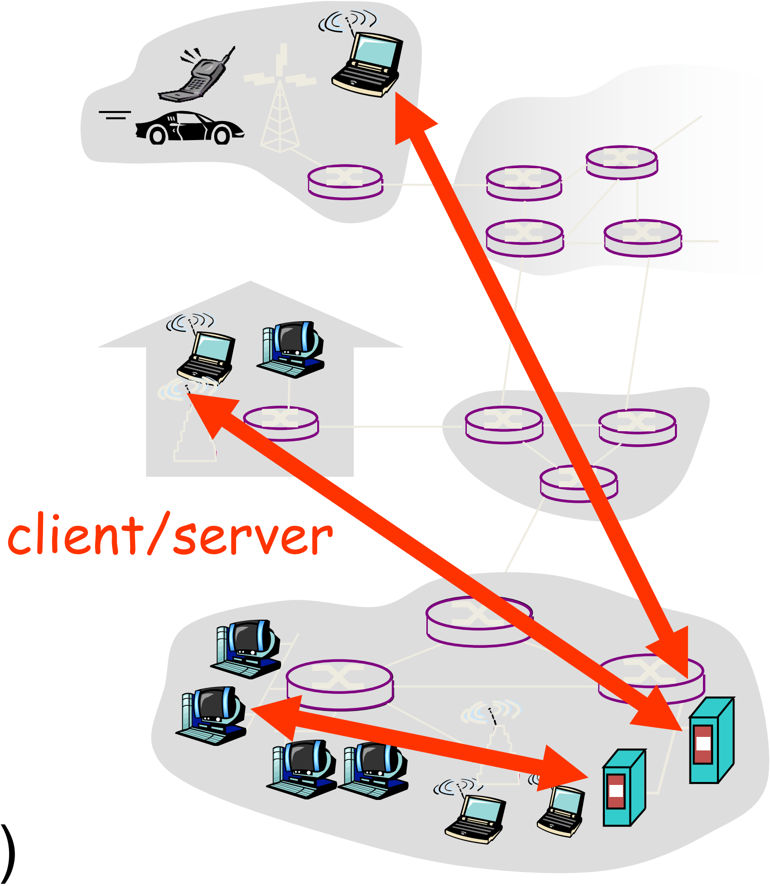
- Reachable by IP address
- **always-on**, waiting for incoming requests from clients

clients:

- Initiate communication with server

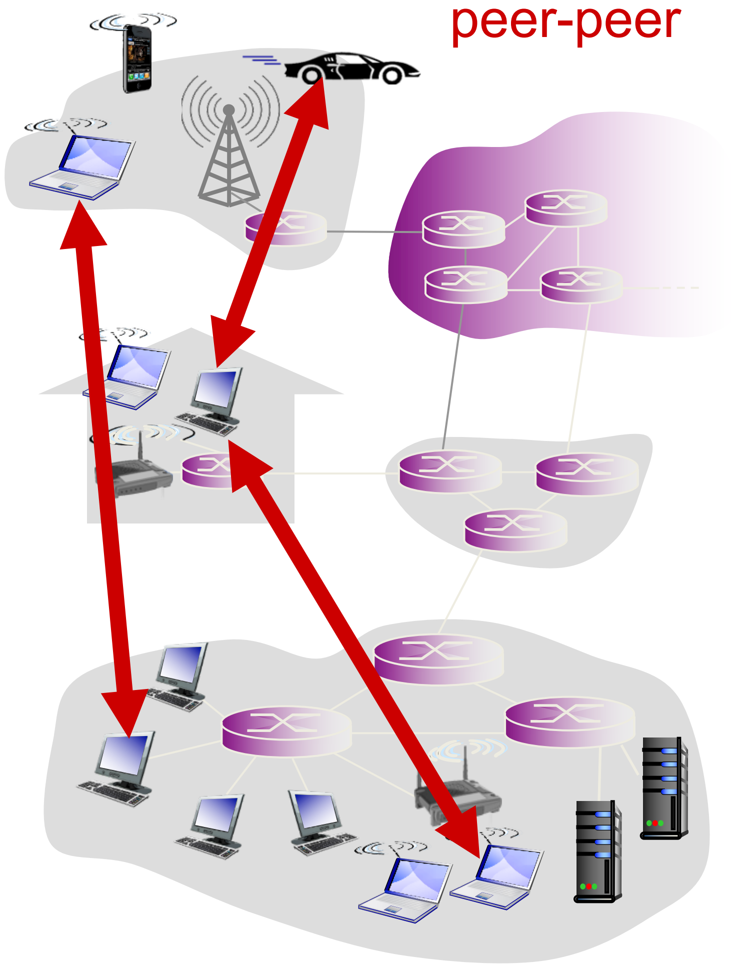
Q: How does a client process *identify* the server process with which it wants to communicate?

A: Using port numbers via the socket API (Application Program Interface)



# P2P 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



# How Processes Communicate?

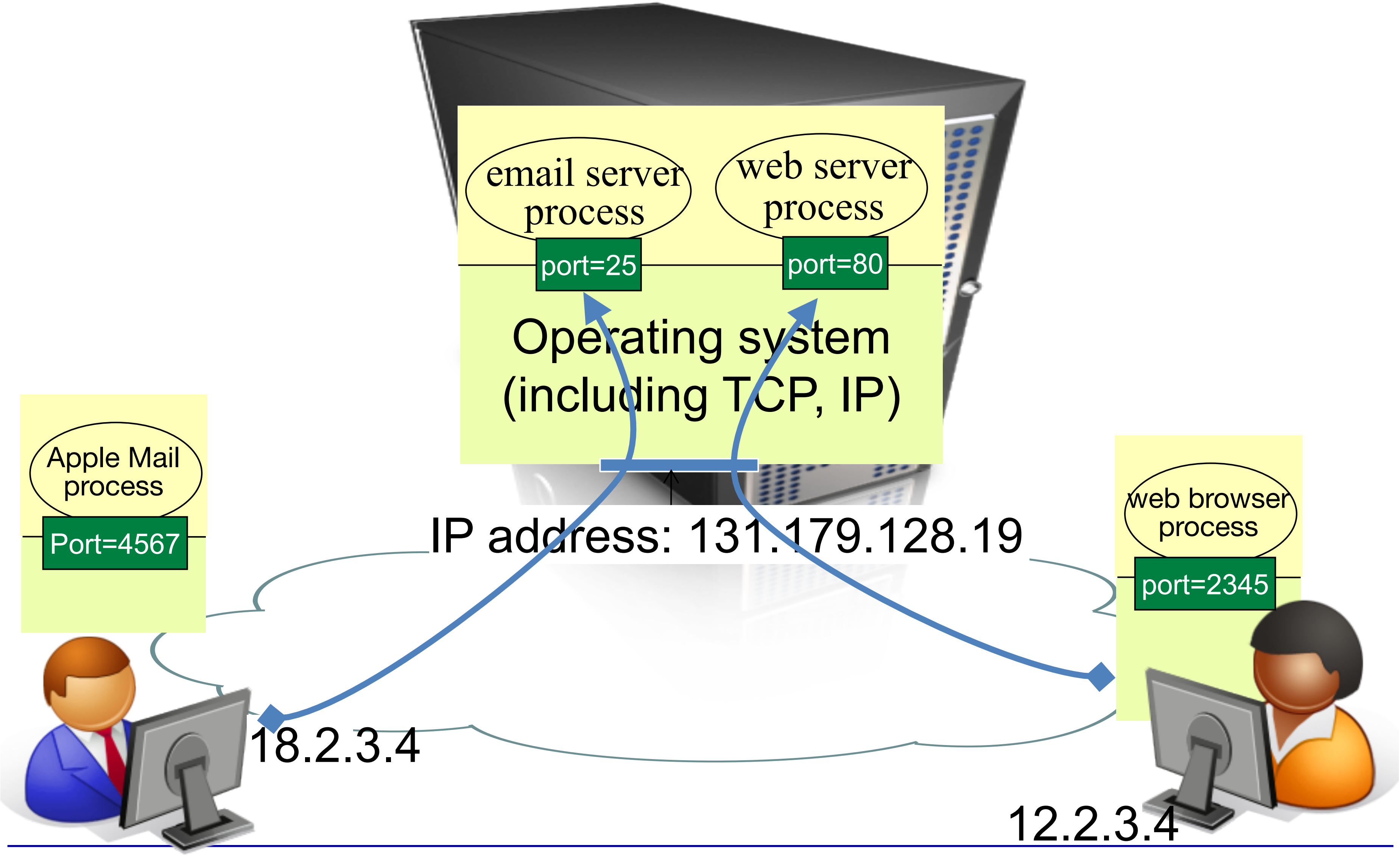
- When client process wants to communicate with server process
  - Decide which transport protocol to use
    - Can tolerate loss?
    - Is time sensitive?
    - Should be secure?
    - Should be private?
  - Figure out server end-point address
  - Use API to connect/send/receive info/packets from the server

- What could be server endpoint address?
- What could be the APIs?

# App Requirements

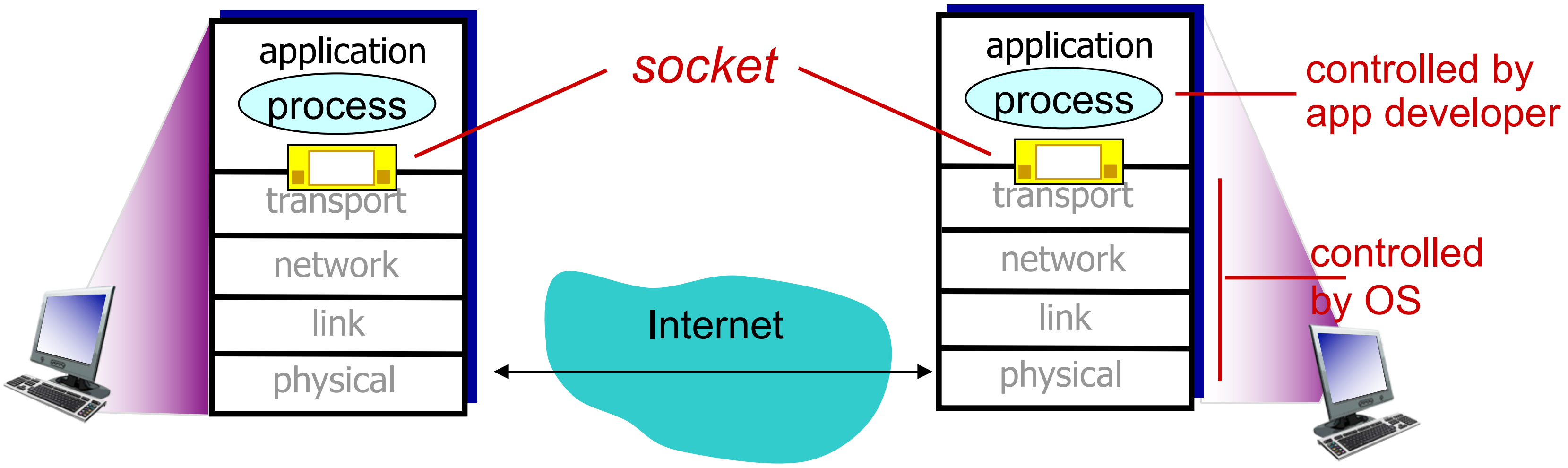
| App | Data Loss? | Time-Sensitive? | Secure? | Private? |
|---|---|---|---|---|
| Web | No | Yes? **No?** | Yes / No | ~~Yes~~ |
| Tor | ? | No | Yes | Yes |
| Mail | No | No | ~~Yes~~ | ~~Yes~~ |
| VoIP | Yes | Yes | Yes | ? |
| YouTube | Yes \| ? | No | Yes | No |
| Games | Both | Both | Yes | Let’s hope |
| Messaging | No | No/Yes | Yes | Let’s hope |

# TCP/IP Addressing

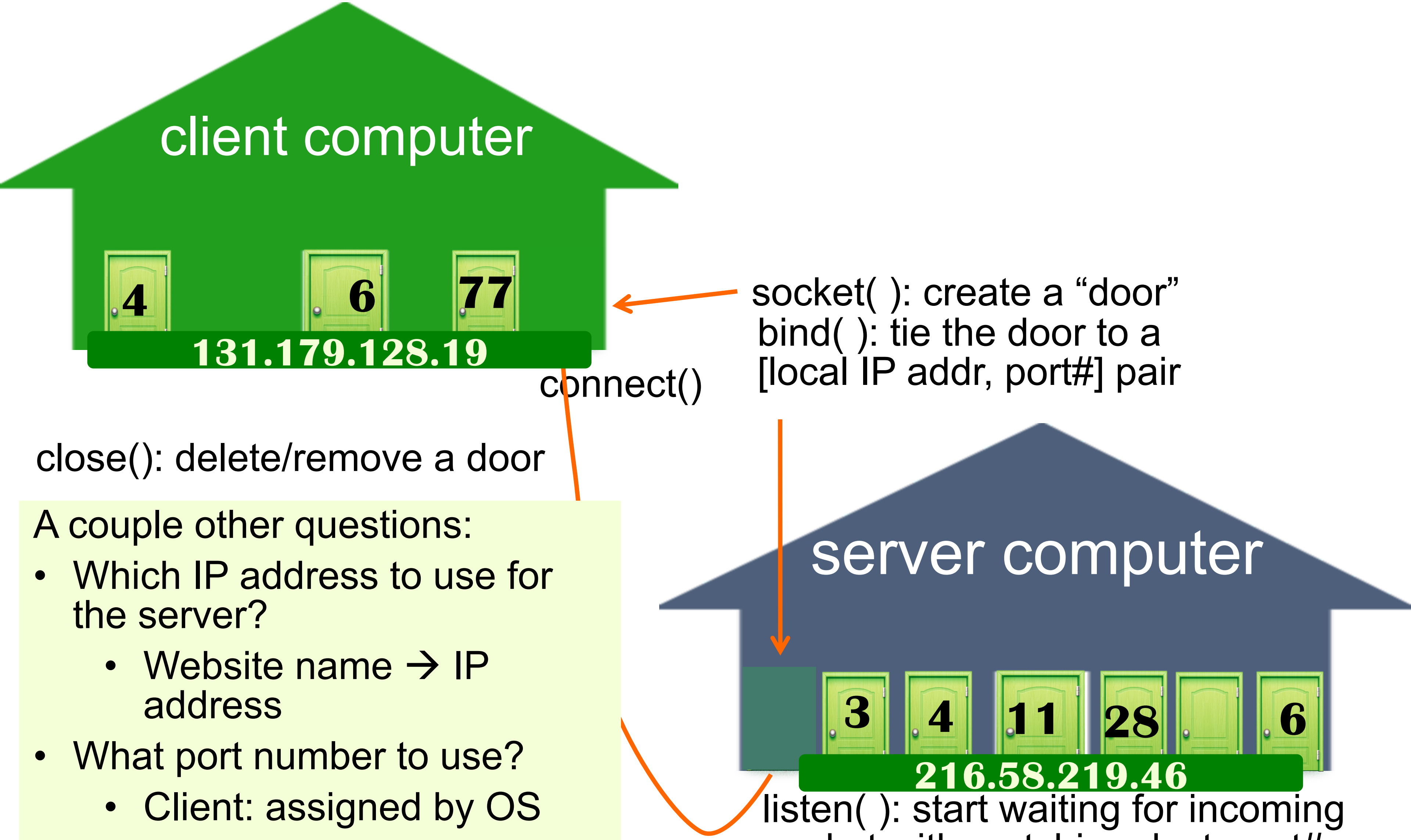


# Berkeley Socket API

- Process sends/receives messages to/from its socket
- Socket a low-level APIs for application to communicate with remote applications
  - Higher level APIs: AJAX, RPC, etc.



# Socket: Analogous to a Door



socket( ): create a “door”
bind( ): tie the door to a [local IP addr, port#] pair

connect()

close(): delete/remove a door

listen( ): start waiting for incoming packet with matching dest. port#

A couple other questions:

- Which IP address to use for the server?
  - Website name → IP address
- What port number to use?
  - Client: assigned by OS
  - Server: defined standards

# What is "socket"

- An API between an app and kernel

socket(): Create a socket

bind( ): bind a socket to a local IP address and port #

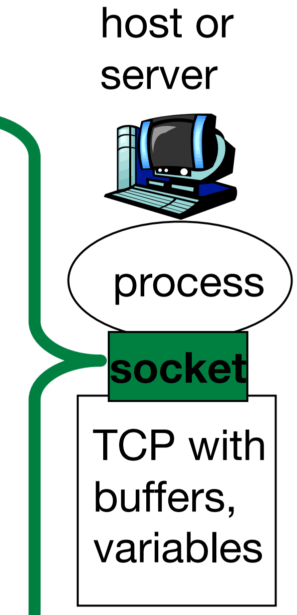
connect( ): initiating connection to another socket

listen( ): passively waiting for connections

accept( ): accept a new connection

write( ): write data to a socket

read( ): read data from a socket



## Establishing a socket on the *client* side:

- Create a socket with the socket() system call
- Connect the socket to the server using the connect() system call
- Send and receive data.
  - There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

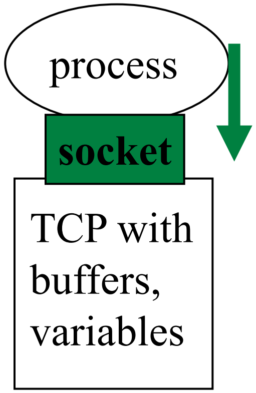
## Establishing a socket on the *server* side:

- Create a socket with the socket() system call
- Bind the socket to [address,port#] using the bind() system call.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call.
  - This call typically blocks until a client connects with the server.
- Send and receive data

# Applications

So far we've talked

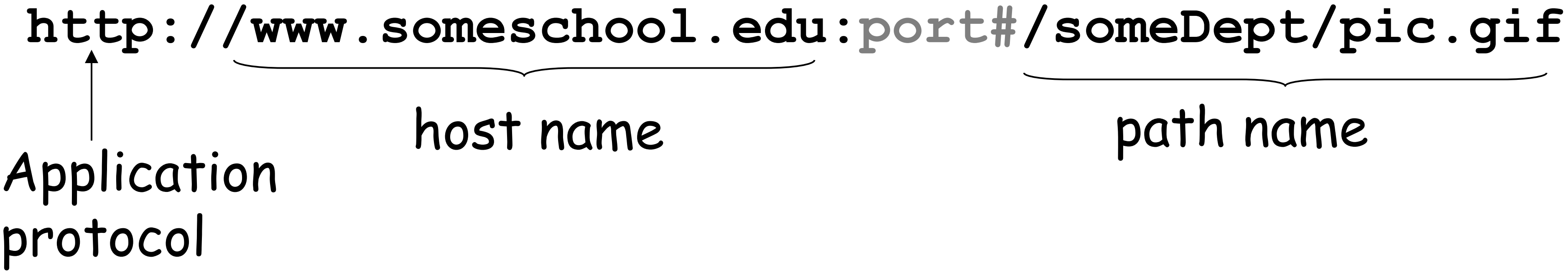
- Application process (executing application program)
- Application protocol (used by application processes to exchange data)
- Exactly how data is exchanged
  - Socket
  - Transport protocol



- Lets look at exactly what data is exchanged

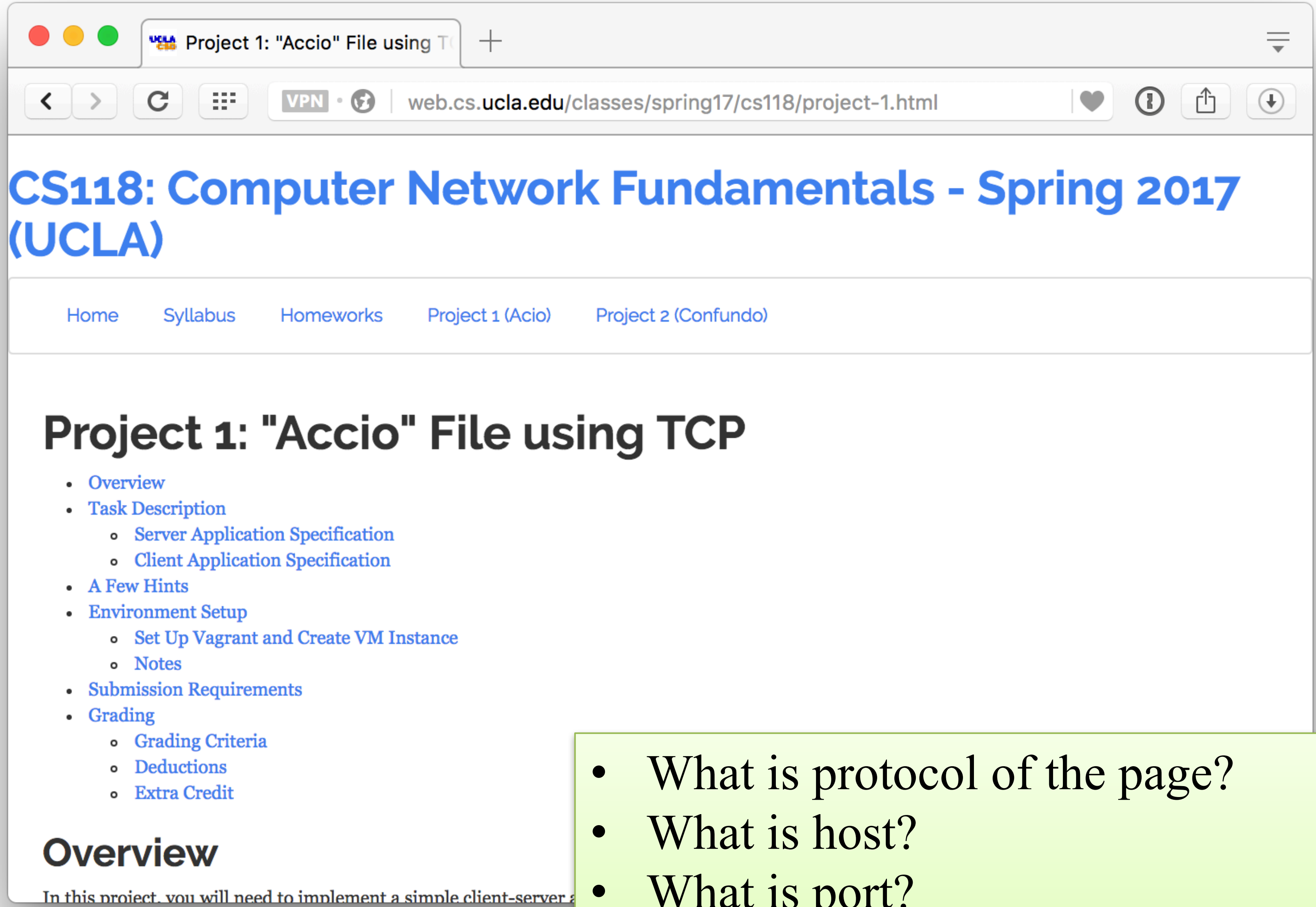
# Web and HTTP

- Web page: normally consists of
  - base HTML-file, which includes
  - several referenced objects
- An object can be another HTML file, JPEG image, audio file,…
- Each object is addressable by a URL (Universal Resource Locator )



`http://www.someschool.edu:port#/someDept/pic.gif`

- http://2130706433:80/index.html



Project 1: "Accio" File using T

web.cs.ucla.edu/classes/spring17/cs118/project-1.html

# CS118: Computer Network Fundamentals - Spring 2017 (UCLA)

Home Syllabus Homeworks Project 1 (Acio) Project 2 (Confundo)

## Project 1: "Accio" File using TCP

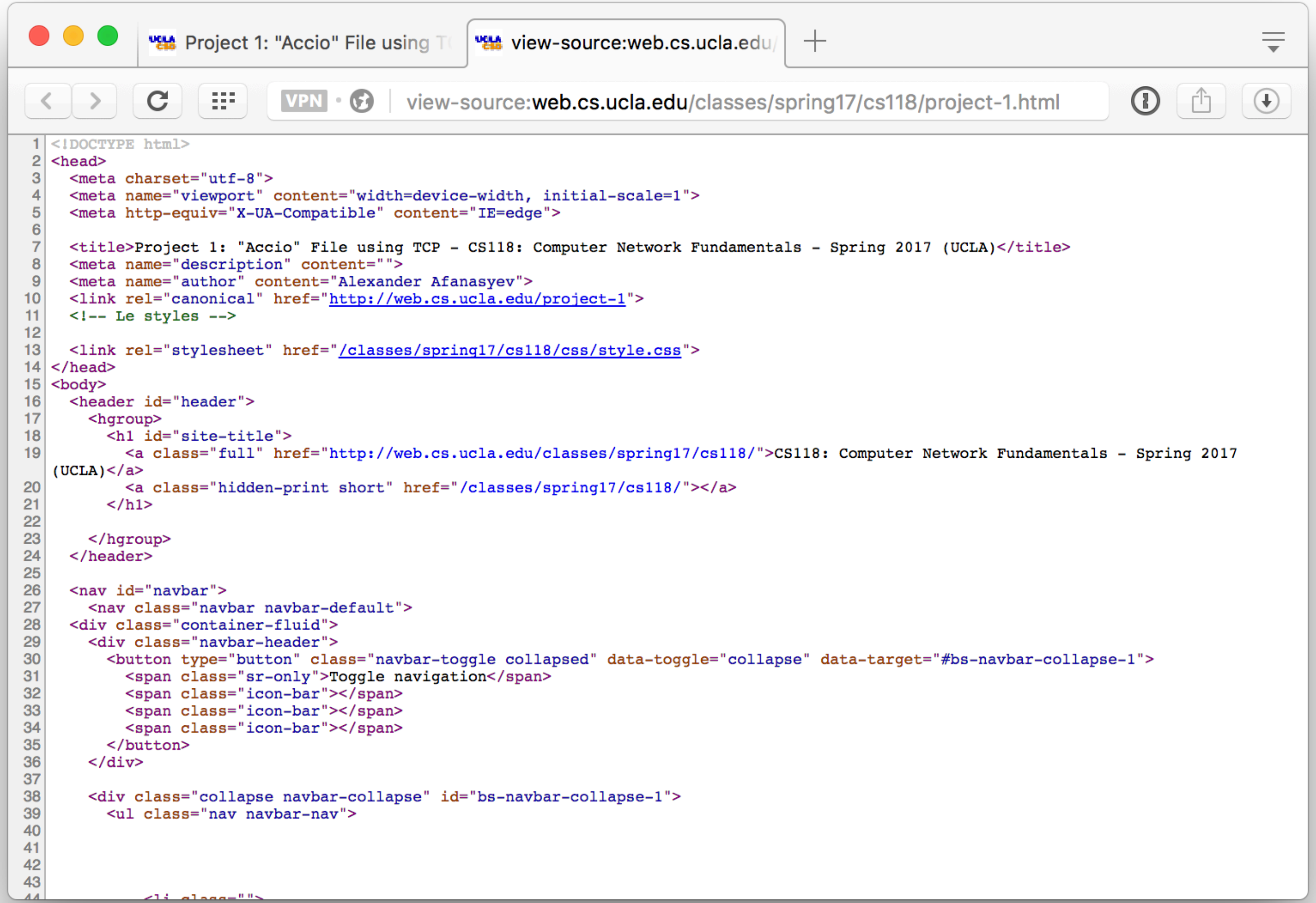
- Overview
- Task Description
  - Server Application Specification
  - Client Application Specification
- A Few Hints
- Environment Setup
  - Set Up Vagrant and Create VM Instance
  - Notes
- Submission Requirements
- Grading
  - Grading Criteria
  - Deductions
  - Extra Credit

## Overview

In this project, you will need to implement a simple client-server a

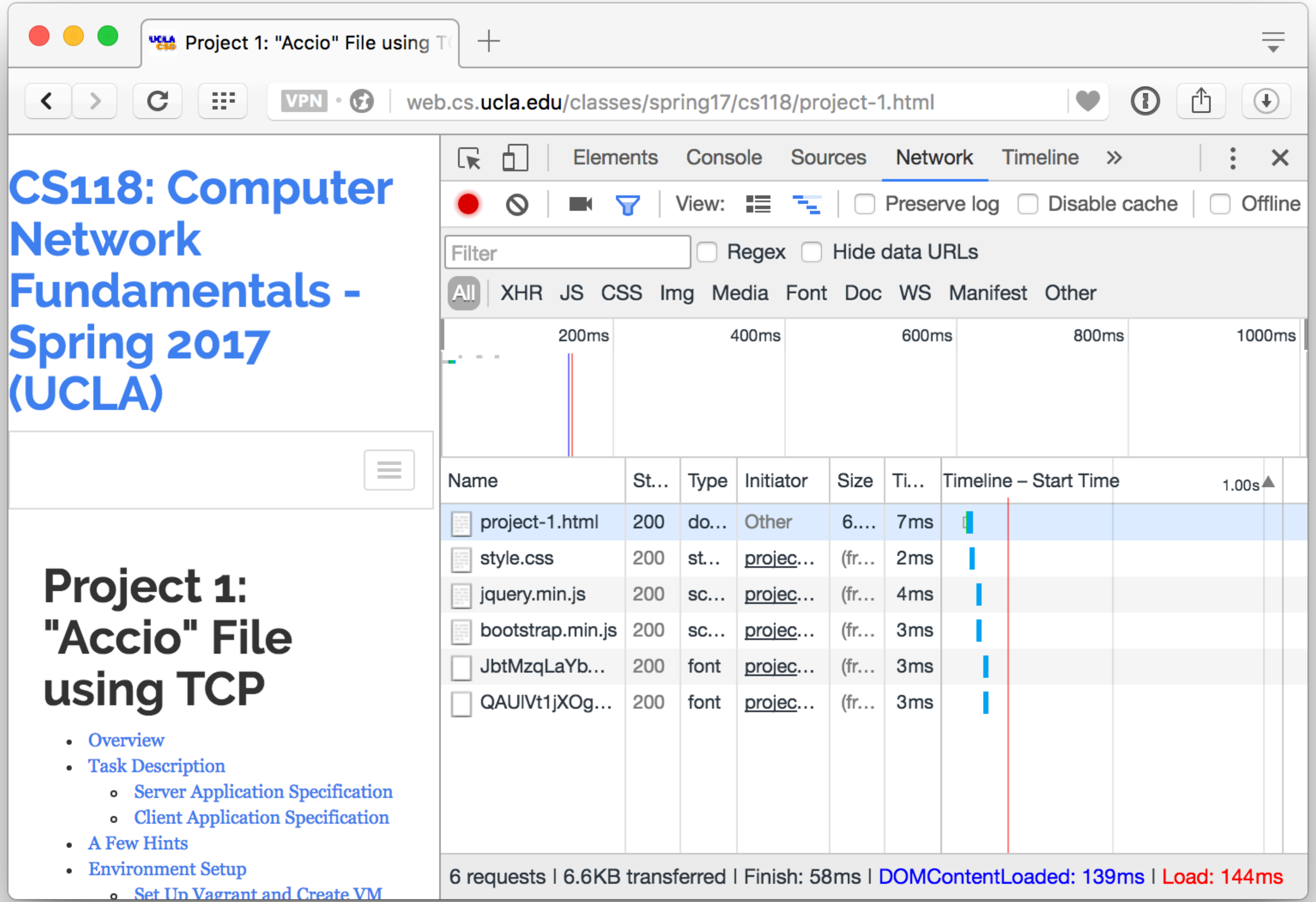
- What is protocol of the page?
- What is host?
- What is port?
- What is path?
- How many objects referenced?

# “Hint” 1


Project 1: "Accio" File using T
view-source:web.cs.ucla.edu/
VPN
view-source:web.cs.ucla.edu/classes/spring17/cs118/project-1.html

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">

  <title>Project 1: "Accio" File using TCP - CS118: Computer Network Fundamentals - Spring 2017 (UCLA)</title>
  <meta name="description" content="">
  <meta name="author" content="Alexander Afanasyev">
  <link rel="canonical" href="http://web.cs.ucla.edu/project-1">
  <!-- Le styles -->

  <link rel="stylesheet" href="/classes/spring17/cs118/css/style.css">
</head>
<body>
  <header id="header">
    <hgroup>
      <h1 id="site-title">
        <a class="full" href="http://web.cs.ucla.edu/classes/spring17/cs118/">CS118: Computer Network Fundamentals - Spring 2017 (UCLA)</a>
        <a class="hidden-print short" href="/classes/spring17/cs118/"></a>
      </h1>

    </hgroup>
  </header>

  <nav id="navbar">
    <nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-navbar-collapse-1">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
    </div>

    <div class="collapse navbar-collapse" id="bs-navbar-collapse-1">
      <ul class="nav navbar-nav">




          <li class="">
```

# “Hint 2”



Project 1: "Accio" File using T

web.cs.ucla.edu/classes/spring17/cs118/project-1.html

# CS118: Computer Network Fundamentals - Spring 2017 (UCLA)

## Project 1: "Accio" File using TCP

- Overview
- Task Description
  - Server Application Specification
  - Client Application Specification
- A Few Hints
- Environment Setup
  - Set Up Vagrant and Create VM

Elements Console Sources Network Timeline »

View: □ Preserve log □ Disable cache □ Offline

Filter □ Regex □ Hide data URLs

All XHR JS CSS Img Media Font Doc WS Manifest Other

200ms 400ms 600ms 800ms 1000ms

| Name | St… | Type | Initiator | Size | Ti… | Timeline – Start Time 1.00s |
|---|---|---|---|---|---|---|
| project-1.html | 200 | do… | Other | 6.… | 7ms | |
| style.css | 200 | st… | projec… | (fr… | 2ms | |
| jquery.min.js | 200 | sc… | projec… | (fr… | 4ms | |
| bootstrap.min.js | 200 | sc… | projec… | (fr… | 3ms | |
| JbtMzqLaYb… | 200 | font | projec… | (fr… | 3ms | |
| QAUlVt1jXOg… | 200 | font | projec… | (fr… | 3ms | |

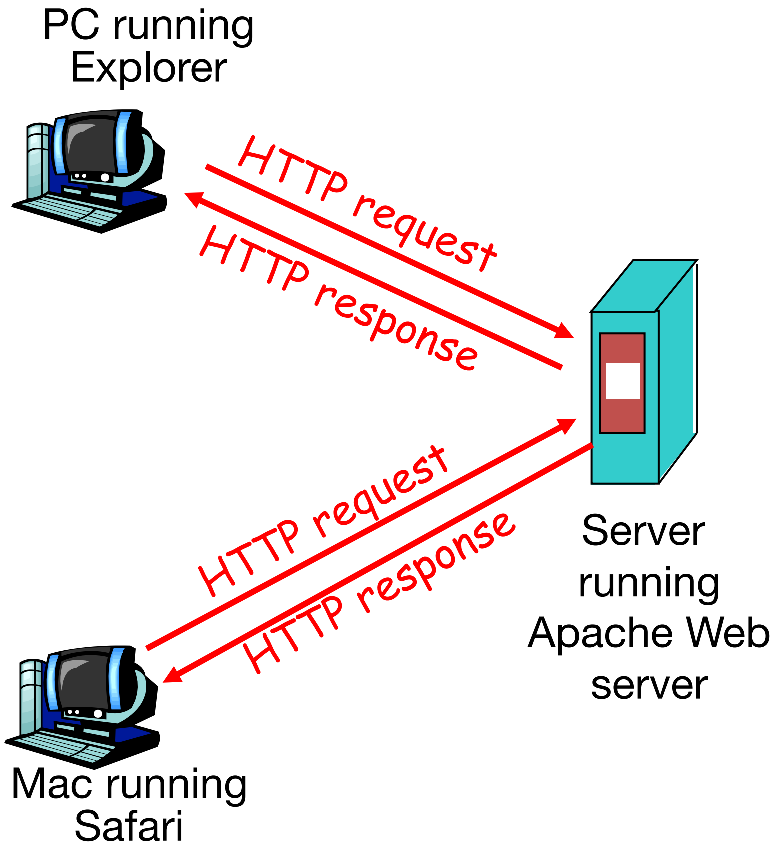
6 requests | 6.6KB transferred | Finish: 58ms | DOMContentLoaded: 139ms | Load: 144ms

# Exploring Content of Web Pages

Quick demo

# HTTP: HyperText Transfer Protocol

- Web’s application layer protocol
- client/server model
  - *client:* browser that requests, receives, “displays” Web objects
  - *server:* Web server sends objects in response to requests
- HTTP/1.0: non-persistent connection
- HTTP/1.1: persistent connection
  - May also pipelining



# More on 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

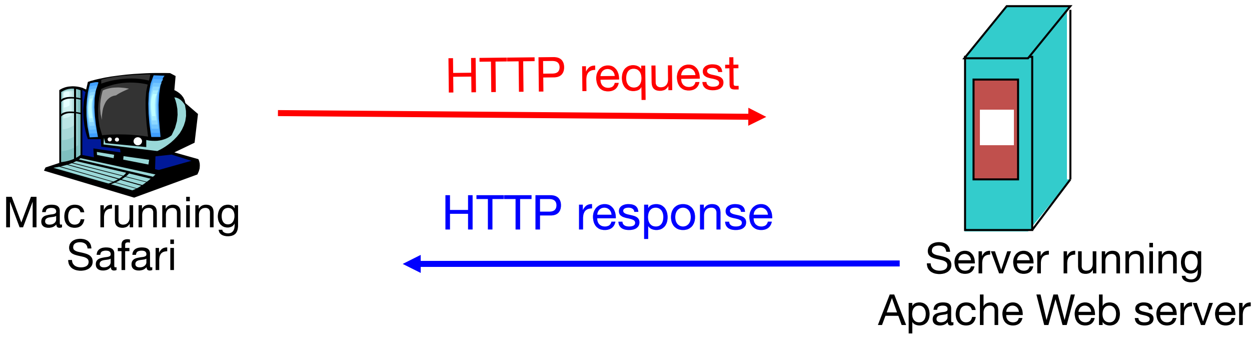
aside

Protocols that maintain "state" are complex!

past history (state) must be maintained

if server/client crashes, their views of "state" may become inconsistent

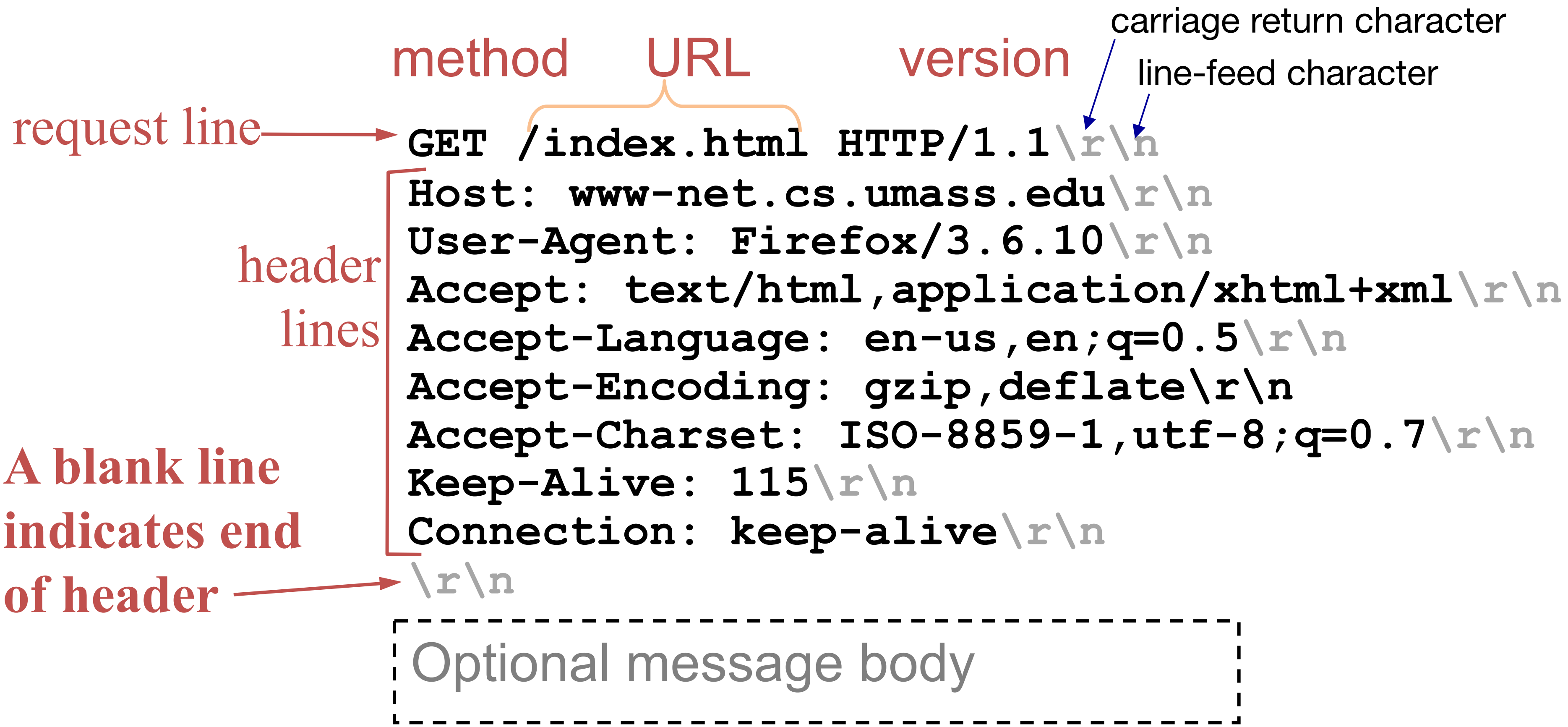
# Now we got the big picture



- Client (browser) speaks first
  - Details about how to set TCP connection: later
- Server answers
  - and then forgets it (stateless)
- Exactly how these two messages look like?

# HTTP request message

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



carriage return character

line-feed character

method URL version

request line

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.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
```

header lines

**A blank line indicates end of header**

Optional message body

# What is in the HTTP Header? Why?

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.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
```

```
HTTP/1.0 301 Moved Permanently\r\n
Location: http://www.google.com/\r\n
Content-Type: text/html; charset=UTF-8\r\n
Date: Wed, 05 Apr 2017 02:25:13 GMT\r\n
Expires: Fri, 05 May 2017 02:25:13 GMT\r\n
Cache-Control: public, max-age=2592000\r\n
Server: gws\r\n
Content-Length: 219\r\n
X-XSS-Protection: 1; mode=block\r\n
X-Frame-Options: SAMEORIGIN\r\n
```

# Method types

## HTTP/1.0

- GET
- POST
- HEAD
  - Requesting the header only (i.e. response does not include the requested object)

## 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 from the server
- and a few others
  - See the protocol specification RFC2616

# HTTP response message

status line (status code, status phrase)

header lines

A blank line

data, e.g., requested HTML file

```
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 ...
```

# Trying out HTTP request for yourself

1. Telnet to a Web server:

```
telnet google.com 80
```

Opens TCP connection to port 80
(default HTTP server port) at cis.poly.edu.
Anything typed in is sent
to port 80 at cis.poly.edu

2. Type in a GET HTTP request:

```
GET / HTTP/1.0
Host: google.com
```

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

3. Look at response message from the HTTP server!

```
✘  19:24 ~ $ telnet google.com 80
Trying 2607:f8b0:4007:806::200e...
Connected to google.com.
Escape character is '^]'.
GET / HTTP/1.0
Host: google.com

HTTP/1.0 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Wed, 05 Apr 2017 02:25:13 GMT
Expires: Fri, 05 May 2017 02:25:13 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
Connection closed by foreign host.
```

# HTTP response status codes

important

- Appears in the 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

  **404 Not Found**

  - requested document not found on this server

  **505 HTTP Version Not Supported**

# Packet Sniffing

- Tcpdump
  - Quick demo
- Wireshark
  - Quick demo