Web Essentials

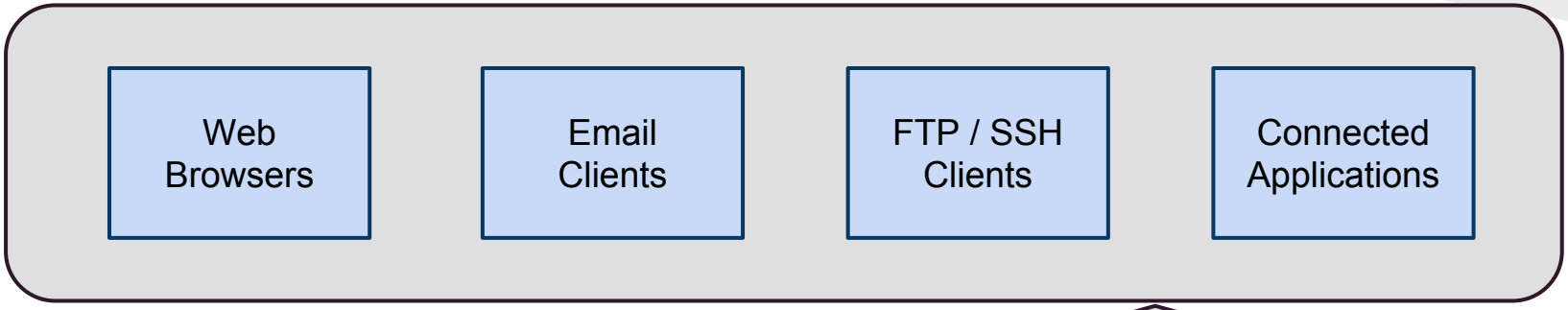
CMPS 369 – Lecture 2

Today's Topics

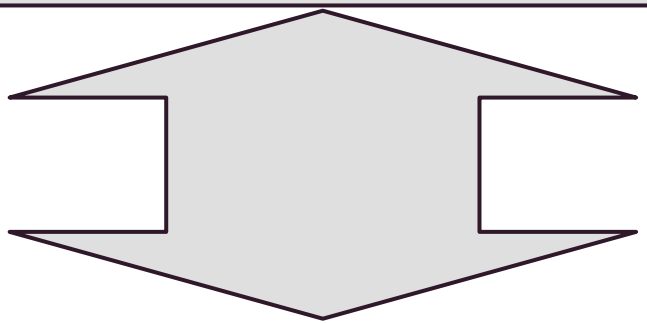
- Networking basics - TCP / IP
  - The basics of IP packets and routing
  - Using Node.js to perform TCP communication
- How the web evolved
  - Understanding a URL
- The HTTP Protocol
  - Using Node.js to perform HTTP communication

3 Fundamental Players

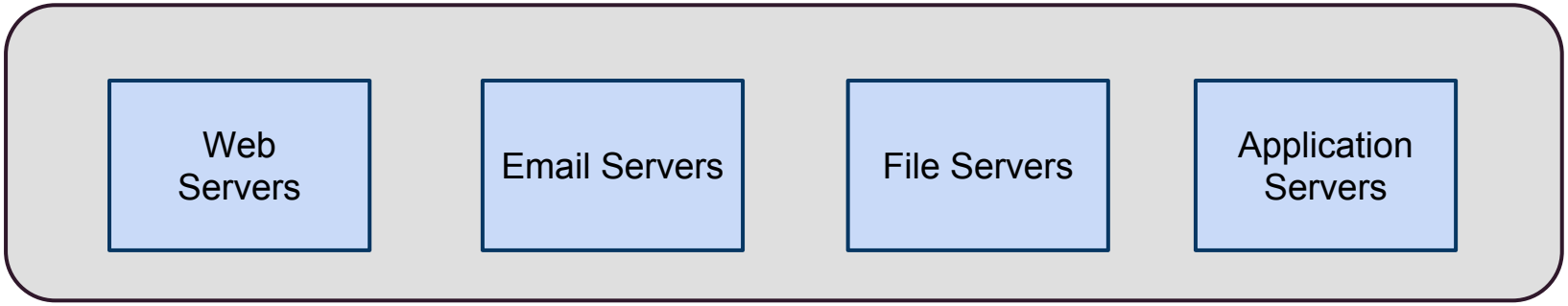
(1) Clients



(2) Communication Protocols



(3) Servers



Communication Protocols

The only thing that goes over the internet is 1's and 0' - a "protocol" defines how those binary numbers are interpreted

- A protocol can be standardized (HTTP, SMTP, FTP) or custom to an applications
  - However the days of "custom" protocols are numbered, if not over...

Communication Protocols

Of course, there are various "layers" of protocols

At the lowest level, you need a **method** of sending binary numbers **to a specific machine**

IP: Internet Protocol

- Provides addressing and message chunking
- All of the internet (and all higher level protocols) rely in IP

You'll study IP in more depth in the "Network" class -- CMPS 327

Internet Protocol

All devices on the internet are primarily considered IP *hosts*.

An IP *host* must have the ability to decode IP packets and understand the *routing* required by IP

Hosts include servers, client computers, mobile devices… but more importantly… **routers and switches**.

IP basics

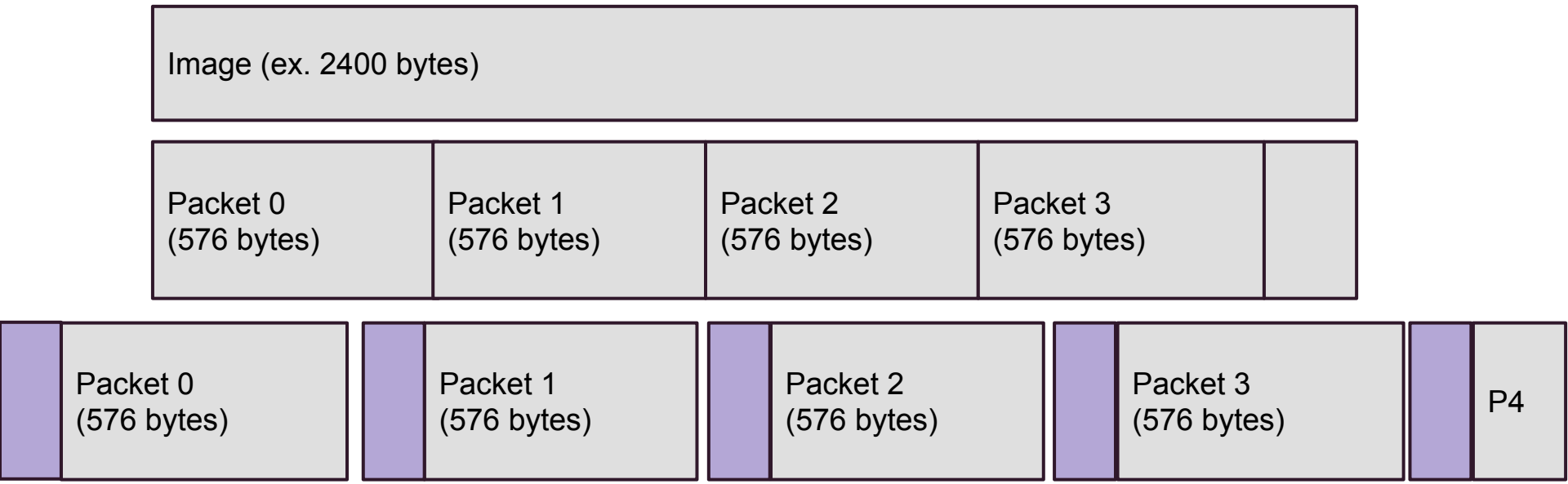
When a program wants to send data to another program - the data could be of any arbitrary size

- Lets consider shipping an image to a destination machine.
    - You need to know how to address the destination machine
    - But you also need to “standardize” the payload into a series of **packets**.

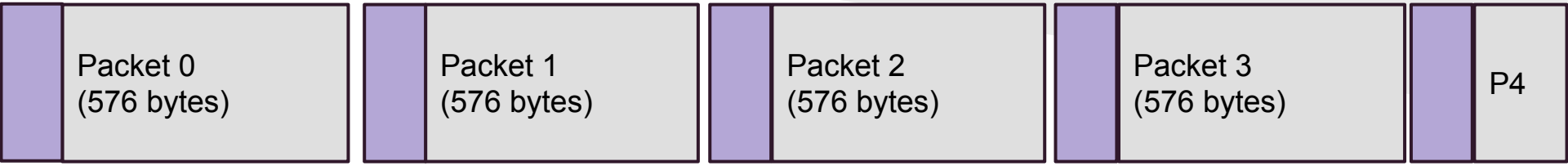
IP Packets

A large chunk of data must be broken into smaller chunks, adhering to the IP standards

- Largest packet a host is *required* to handle is 576 bytes
- Tacked onto each packet is a **header**, containing information such as IP version (IPv4, IP v6), total packet size, and other flags required to perform routing



IP Routing



Each packet contains a destination in its header

IP Address

An IPv4 address (dotted-decimal notation)

172 . 16 . 254 . 1

10101100 .00010000 .11111110 .00000001

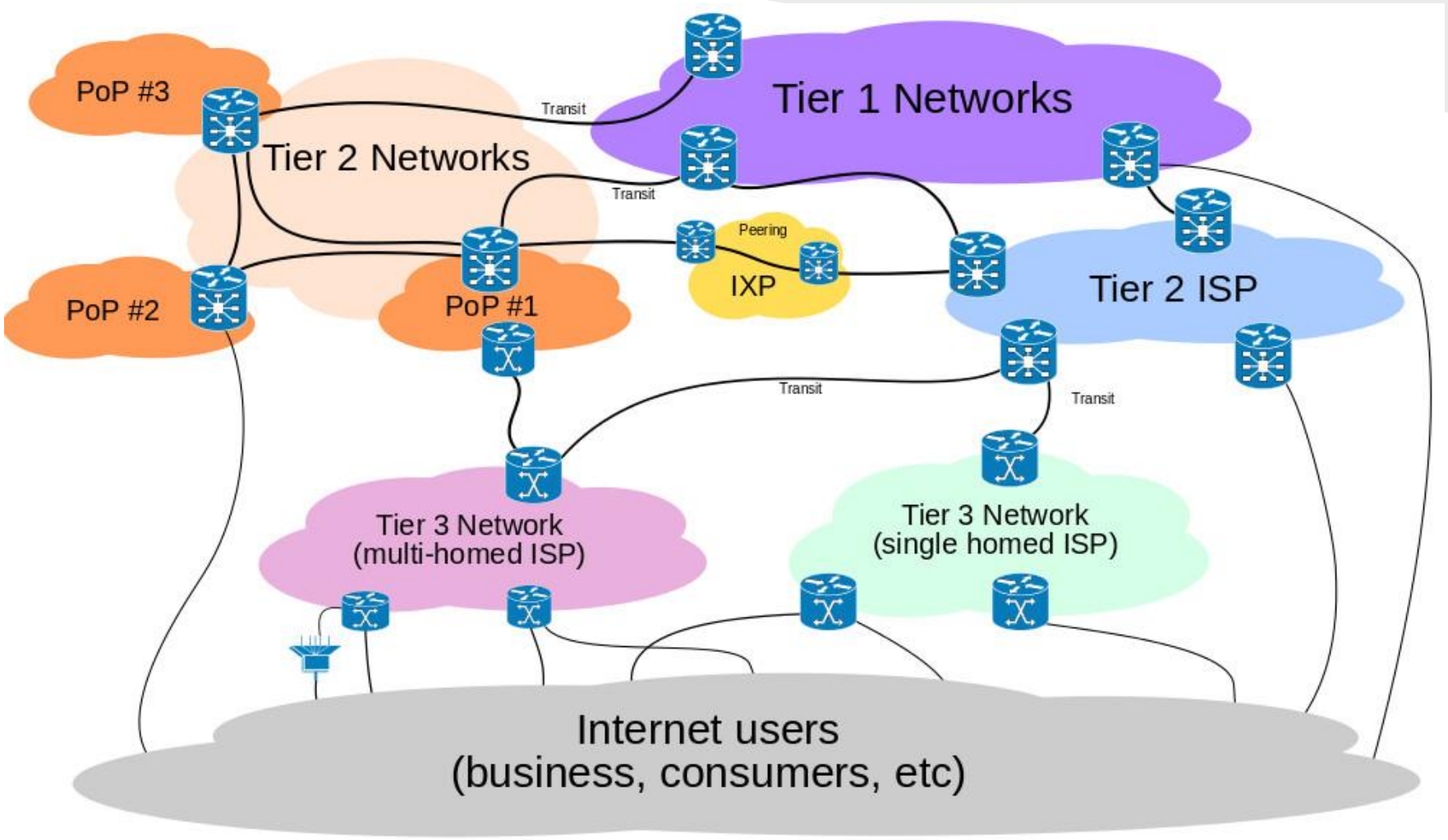
The packet is sent to the first known host – likely a switch providing primary internet access to the machine

Internet Topology

Routing performed by each host is essentially a simple process:

- When receiving an IP packet, check destination
  - If the host is the destination, OK - done.
  - If the host can connect directly to the destination, forward it there.
  - Otherwise, forward it to another host who hopefully has a better idea…

Internet Topology



Internet Topology

All the routers and switches on the internet communicate with each other using their own protocol - **Border Gateway Protocol**

- They maintain routing tables to help them decide where to sent packets
- Routers and switches perform load balancing, making sure packets are sent efficiently
- Packets sent from one machine to another may take drastically different routes

Reliability

IP and the data-link layer provide ECC to ensure that a received packet actually contains the data sent.

If the data is corrupted, the packet is ***dropped***.

- IP does not guarantee delivery of a packet!
- It does not resend dropped packets!
- It does not ensure the packets arrive in order!

TCP - 1974

TCP: Transmission Control Protocol

TCP builds on IP to provide reliability

- Packets are assigned sequence numbers
- Received packets are acknowledged (with return packets)
- Lost packets are resent
- Data "arrives" in order

This creates overhead, but generally considered "acceptable" overhead when reliability is needed.

TCP/IP implementation

On a given machine, software must be present to implement TCP/IP

- This is typically part of the operating system kernel.
- The operating system provides an ***abstraction*** called a **socket**.
  - A socket is a "connection" between two machines.
  - It is a 2-way channel
  - It is associated with **port numbers** on each side
    - This allows machines to open many ports for communication simultaneously

Sockets - Introduction

Server Socket: **Listens for a connection** on a "well-known" port

Client Socket: **Connects** to a server (address/port)

Once the server gets the connection request, a **new socket** is created

- The original server-socket can return to listening
- The socket remains connected until explicitly closed.

Sockets - Introduction

The operating system provides an API for socket programming.

You can program with sockets in C, C++, Java, C#, and essentially any other language that can run on your machine.

Lets see an example of doing so in C++, and then in Node.js

Sockets - C++

I'm using the Win32 API, however the topic is similar on Linux and Mac OS X

- We'll create a server program, listening on port 3000 for incoming connections.
- A client program will connect to the server
  - Both programs will run on my machine
  - We will use IP address 127.0.0.1
  - 127.0.0.1 is reserved for "local machine" - or "localhost"
- The client will send "hello", the server will send back "HELLO" - an upper-case echo server.

Sockets - Node.js

Now lets write the same **server**, with Node.js

- We'll have the Node.js server listen to port 3001
- It will return the **lower case** echo response.

Note - I'm going to continue to use the same C++ code to connect to the server.

Reminder: All the source code is posted on **moodle**. You should try to install node.js on your own machine and run the demo yourself. Note - I have provided a node.js based client as well.

HTTP

Echo servers are tons of fun, but what about the actual web?

- A web server listens with a TCP socket on port 80
  - Port numbers < 1024 are typically "reserved"
  - 80 is reserved for HTTP, other protocols correspond with other port numbers (by convention)
- A web browser connects to a machine at port 80
  - The browser sends a request
  - The server responds with information (a web page in all likelihood).
  - The format of the request/response is HTTP

HTTP: The URL

URL stands for **U**niform **R**esource **L**ocator

Format: **scheme**://**domain**:**port**/**path**

The **scheme** represents the protocol being used

http, ftp, attachment, mailto, etc.

The **domain** is the server address

127.0.0.1, www.google.com, www.ramapo.edu

Port is self-explanatory, **path** will be discussed later

URL -> Socket

When you type http://www.ramapo.edu into a web browser you tell it to…

- Establish a TCP/IP connection with a machine named "www.ramapo.edu" at port 80 (default)
- And to initiate an HTTP request for the path "/" (default)

The first step is to convert www.ramapo.edu into an IP address

Domain Name Services

Each computer has an IP address (or several) for a machine that is responsible for *resolving* domain names to IP addresses

- If you "ping" a domain, you can find the IP address
- Machines that do this are called **name servers**.

```
PING www.ramapo.edu (192.107.108.90) 56(84) bytes of data.
PING www.facebook.com (31.13.69.160) 56(84) bytes of data.
```

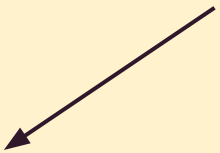
- Top-level domains (**.com, .net, .edu**) are maintained in global registries by large companies (Verisign)
- At this highest level, maintained by **ICANN** - Internet Corporation for Assigned Names and Numbers

Domain Name Services (DNS)

Each web domain (www.ramapo) is registered at a top level domain (.**edu**).

The TLD maintains global registries, accessible to the public.

```
>:  whois ramapo.edu
--------------------------
Domain Name: RAMAPO.EDU
Registrant:
   Ramapo College
   505 Ramapo Valley Road....
..
Name Servers:
   NS1.RAMAPO.EDU          192.107.108.15
   NS2.RAMAPO.EDU          192.107.108.14
```



Will know all the ramapo named servers (i.e. pages.ramapo.edu)

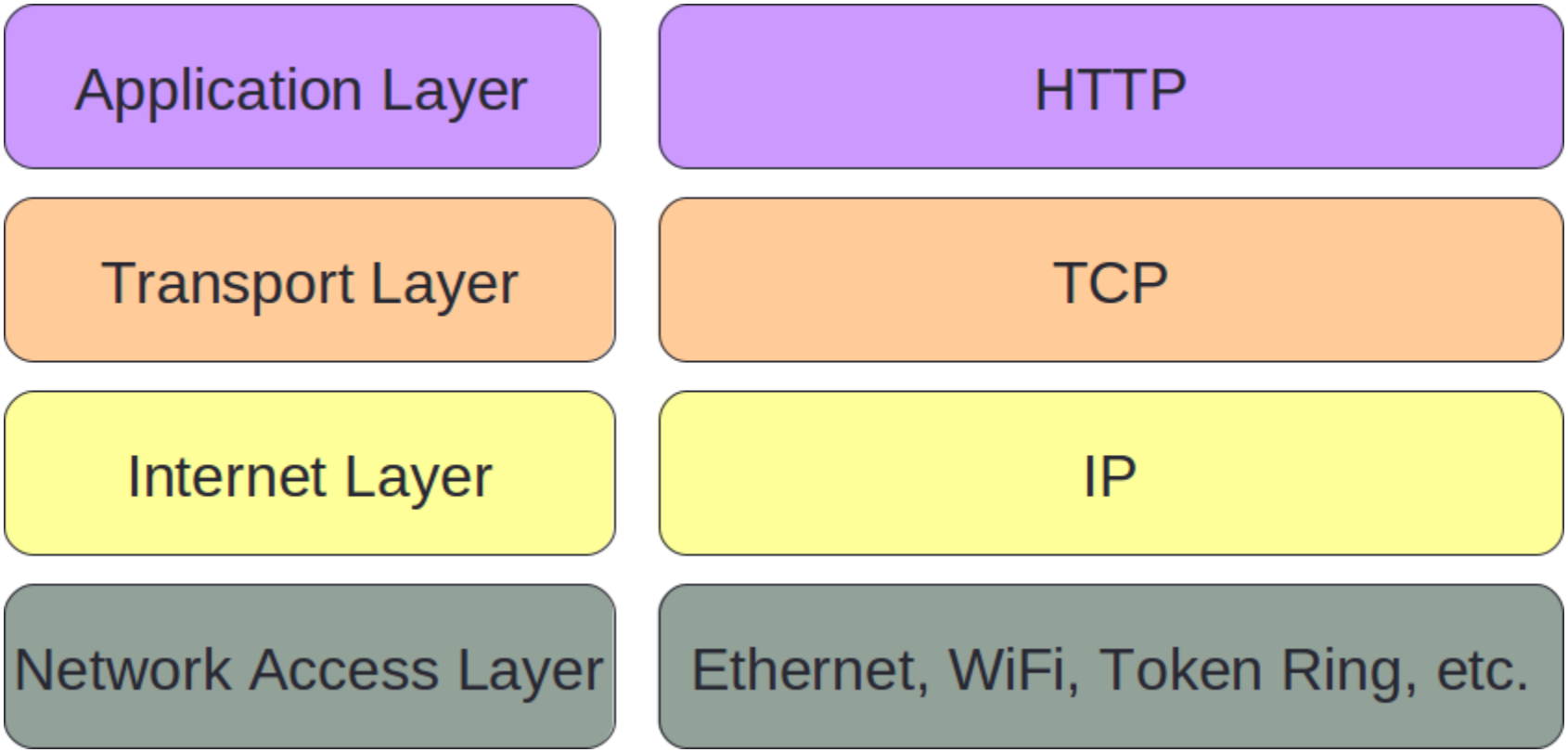
URL -> Socket

Once the name (domain) is resolved to an IP address, the browser connects via **TCP**

At this point - anything *could* be sent. However the browser needs to communicate using a **known protocol**.

HTTP: **H**yper-**T**ext **T**ransfer **P**rotocol

What's with all the protocols?



HTTP Protocol

- HTTP outlines standard “language” of communication between clients and servers
  - It has very little to do with HTML
- Request and Response Model
  - Client (Browsers) issue Requests in a specified format
  - Servers respond with Response messages in a specified format
- When user enters URI into Location Bar:
  - Browser creates an HTTP Request message for the resource indicated by URI
  - Server sends the resource back as part of the Response message
  - Server may also indicate an error

HTTP Messages

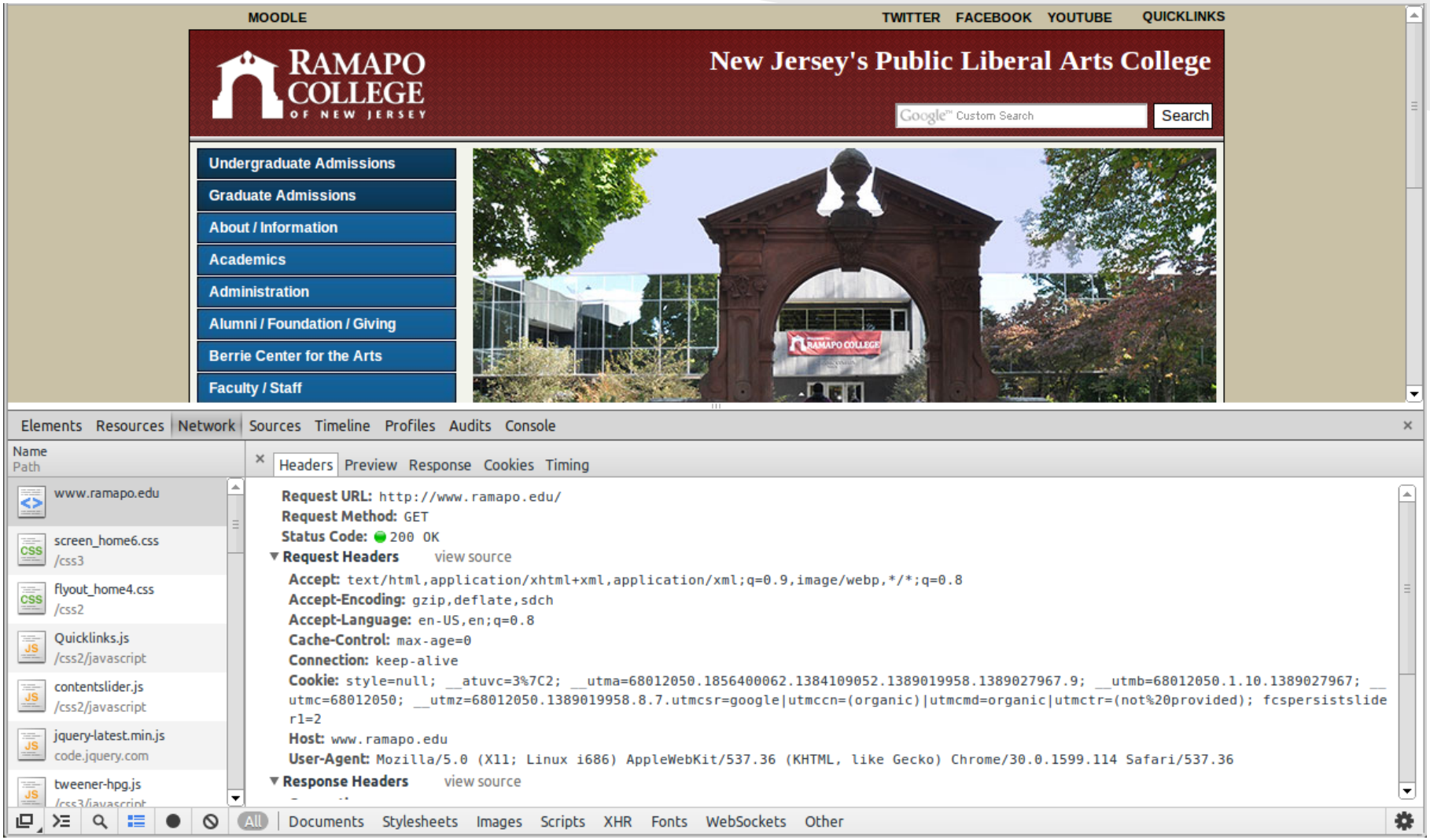
- All HTTP messages are in plain ASCII text
  - Why do you think this is?

- Telnet can be used to manually create requests and view responses

```
telnet pages.ramapo.edu 80
...
GET /~sfrees/index.html HTTP/1.1
host:  pages.ramapo.edu
```

- Most browsers have developer tools for this too, which are much better than telnet for debugging purposes

HTTP Messages



HTTP Request Message

- Start Line
  - Request Method
  - Request URI
  - HTTP Version
- Header Field (s)
- Blank Line
- Message Body (Optional)

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Request Methods

- **GET:** Return resource specified by Request URI as body of response
- **POST:** Pass body of this request message as data to be processed by resource specified in Request URI
- **HEAD:** Return same HTTP header fields as if GET was used, but without the body
- **OPTIONS:** Return a list of HTTP methods that may be used to access the resource in Request URI
- **PUT:** Store the body of this message on the server and assign the specified Request URI to it
- **DELETE:** Remove the specified Request URI
- **TRACE:** return a copy of the complete HTTP request message received by server (debugging)

Request Header Fields

```
POST /servlet/EchoHttpRequest HTTP/1.1
host: www.example.org:56789
user-agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
  rv:1.4) Gecko/20030624
accept:  text/xml, application/xml, application/xhtml+xml,
  text/html;q=0.9, text/plain;q=0.8, video/x-mng, image/png,
  image/jpeg, image/gif;1=0.2
accept-language:  en-us, en;q=0.5
accept-encoding:  gzip, deflate
connection:  keep-alive
keep-alive: 300
```

Common Header Fields

- **Host:** Used to support virtual hosts
- **User-Agent:** String identifying program generating the request
- **Accept:** List of MIME extensions the program is willing to accept
- **Accept-Language:** Language preferences
- **Accept-Encoding:** List of compression formats program is capable of accepting

Common Header Fields

- **Connection**: Indicates if the TCP connection should remain open after response is sent (keep-alive or close)
- **Keep-Alive**: Indicates # of seconds to keep connection open
- **Content-Type**: Indicates the MIME type of the request's message body
- **Content-Length**: # of bytes in the message body
- **Referrer**: The URI that sent program to this resource (nothing if user directly enters this URI request)

MIME Extensions

- **M**ultipurpose **I**nternet **M**ail **E**xtension
- Top Level Content Types
  - application, audio, image, message, model, multipart, text, video
- Common MIME Extensions
  - text/html, text/plain
  - image/png, image/jpeg
  - application/octet-stream
  - application/x-www-form-urlencoded
  - Many others

Responding to a Request

```
GET /index.html HTTP/1.1
Host: www.example.com
```

A GET request will specify a **path**.

Above, its for /index.html

- This likely corresponds to a file on the server
- The server now should respond with the contents of this file
- However - its possible that it couldn't be found, or that the user doesn't have access to it.
- The response will be sent as an HTTP response with all this information

HTTP Response

- Response Message has similar format:
  - Status Line
  - Header Field(s)
  - Blank Line
  - Message Body
    - Often just plain text written in HTML language (text/html)

- Typical Response Status:
  - HTTP/1.1 200 OK
    - HTTP Version
    - Status Code
    - Reason Phrase

Status Codes

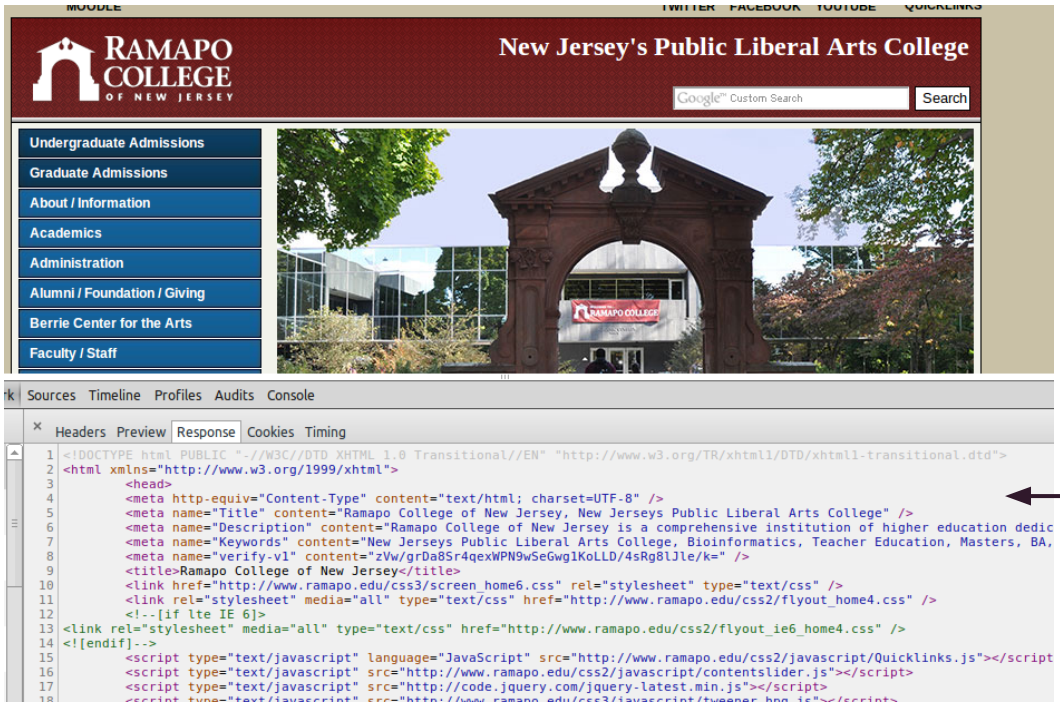
- First digit represents a “class” of status codes
- Common Status Codes
  - 200: OK, Request processed normally
  - 301: Moved Permanently
  - 307: Temporary Redirect
  - 401: Unauthorized Access (password)
  - 403: Forbidden (resource present, by not public)
  - 404: Not Found
  - 500: Internal Server Error

Response Header Fields

- **Date**: Time when response was generated
- **Server**: string identifying the server's software
- **Last-Modified:** last time resource was modified
- **Expires:** (time after which resource is no longer valid)
- **Etag:** hash code of the resource returned
- **Location:** New location of resource (only for redirects)

Response Body

A blank lines comes after the header fields, and if the response was 200, there will be a **body** - consisting of binary or text data.



Chrome is providing some formatting, the HTTP response is just ASCII text

Response Body

Of course - the requested **path** does not need to correspond to a file on the server

- The server could respond with a file from another server (downloads it itself)
- Or the server could simply execute some code associated with the path and return a **dynamic** response

Its the dynamic part that is interesting to us…
… and its what web development is all about

Node.js - A Web Server

Lets look at a really simple Node.js web server

Listens on port 3000 instead of 80

Always responds with a simple HTML page, no matter what path is requested.

We can hit the server with any browser!

Next time

Now we need to learn HTML

Please read Chapters 1 and 2 in the HTML text

We will be covering chapters 1-5 between now and Feb. 6th