Lecture 4

The Application Layer

Objectives (Part 1)

- Understand **how network applications are written** (at a high level)
- Understand **what transport layer services** applications rely on
- Understand **what application architectures** are used

Question:
What applications do you use in your everyday life?

What do you do over the Internet

in your everyday life?

Some network apps

- Search (DuckDuckGo)
- Web
- E-mail
- Video streaming (YouTube, Hulu, Netflix)
- Voice over IP (e.g., Skype, hangouts)
- Social networking (Facebook)
- Multi-user network game

- real-time video conferencing (Teams, Zoom, Skype, Facetime),
- text messaging
- …
- remote login
- P2P file sharing
- multi-user network games

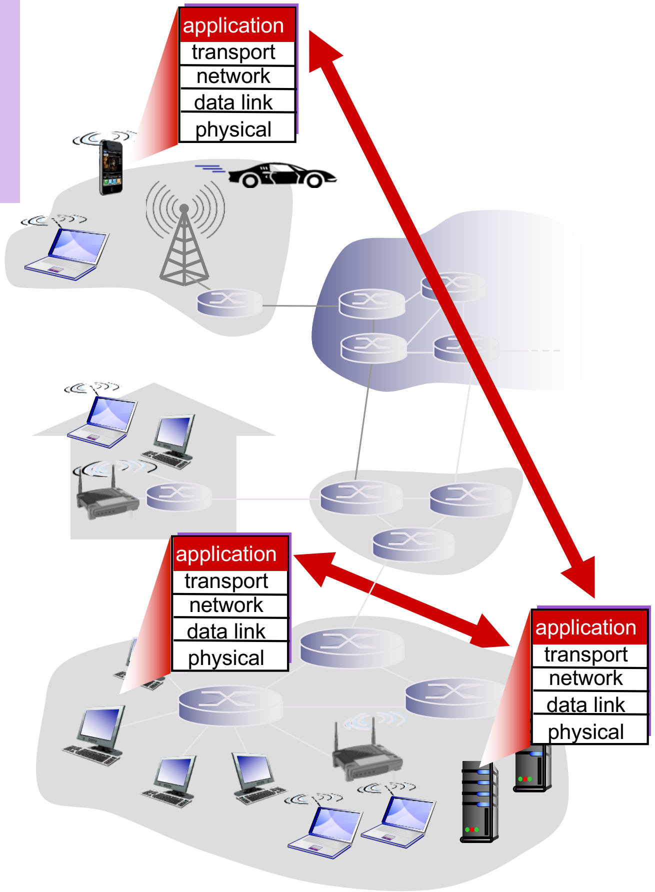
- Emerging ones:
  - AR/VR/MR/XR

Creating a network app

write programs that:

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

no need to write software for network-core devices



Question:
Do all end systems play the the same role?

Example: Use your laptop to do a Bing Search

Hints:

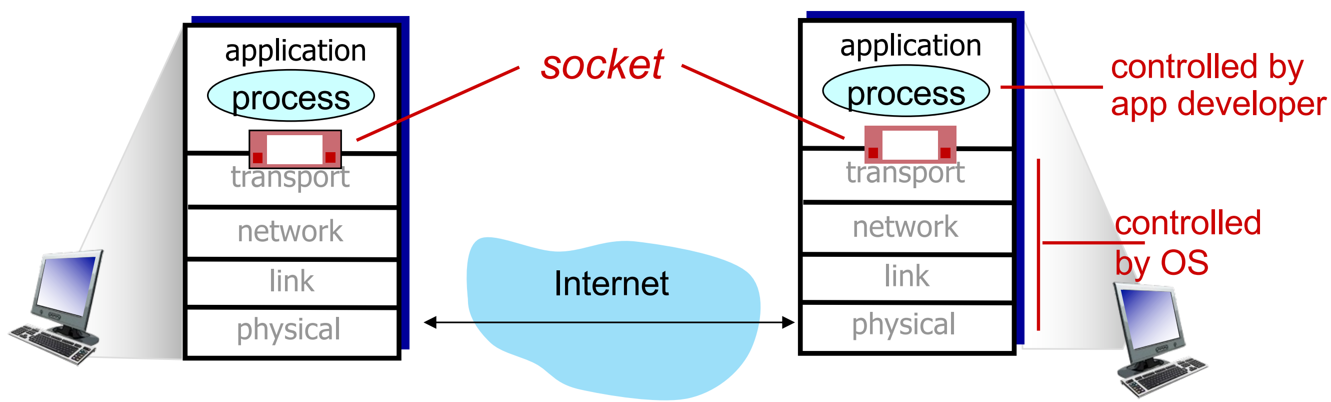
Role of your laptop

Role of Bing Server

Same or different requirements on the Bing Server and your laptop?

What is a network application?

- Two or more **processes** running in **end systems** that send **messages** over a network to accomplish some goal
  - Each process is a running instance of a program
  - Programs can be written in any programming language (C, Java, Python, Go, etc.)
  - Programs use the **socket API (Application Programming Interface)** to send messages



Processes

- Software programs run as **processes** in an operating system
- In Unix like operating systems (Mac OSX or Linux) you can use the `ps` command (process status) to list the processes
- In Windows, you can use the task manager to look at the processes

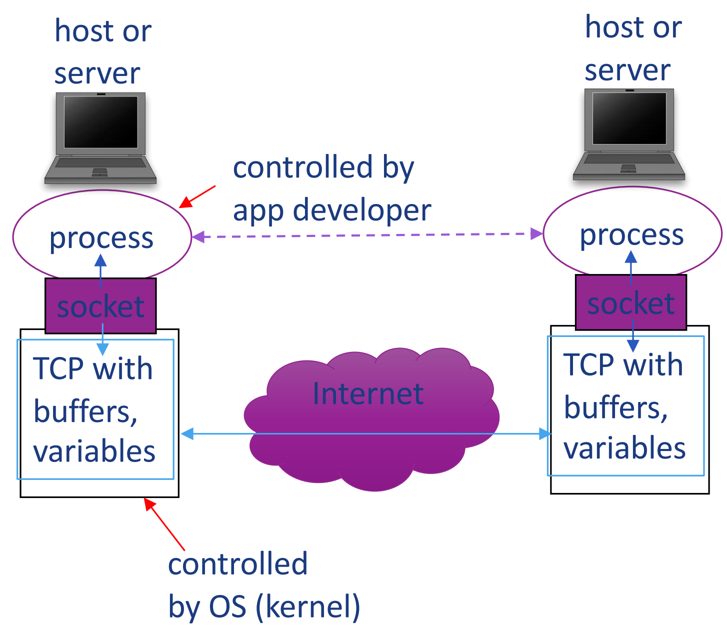
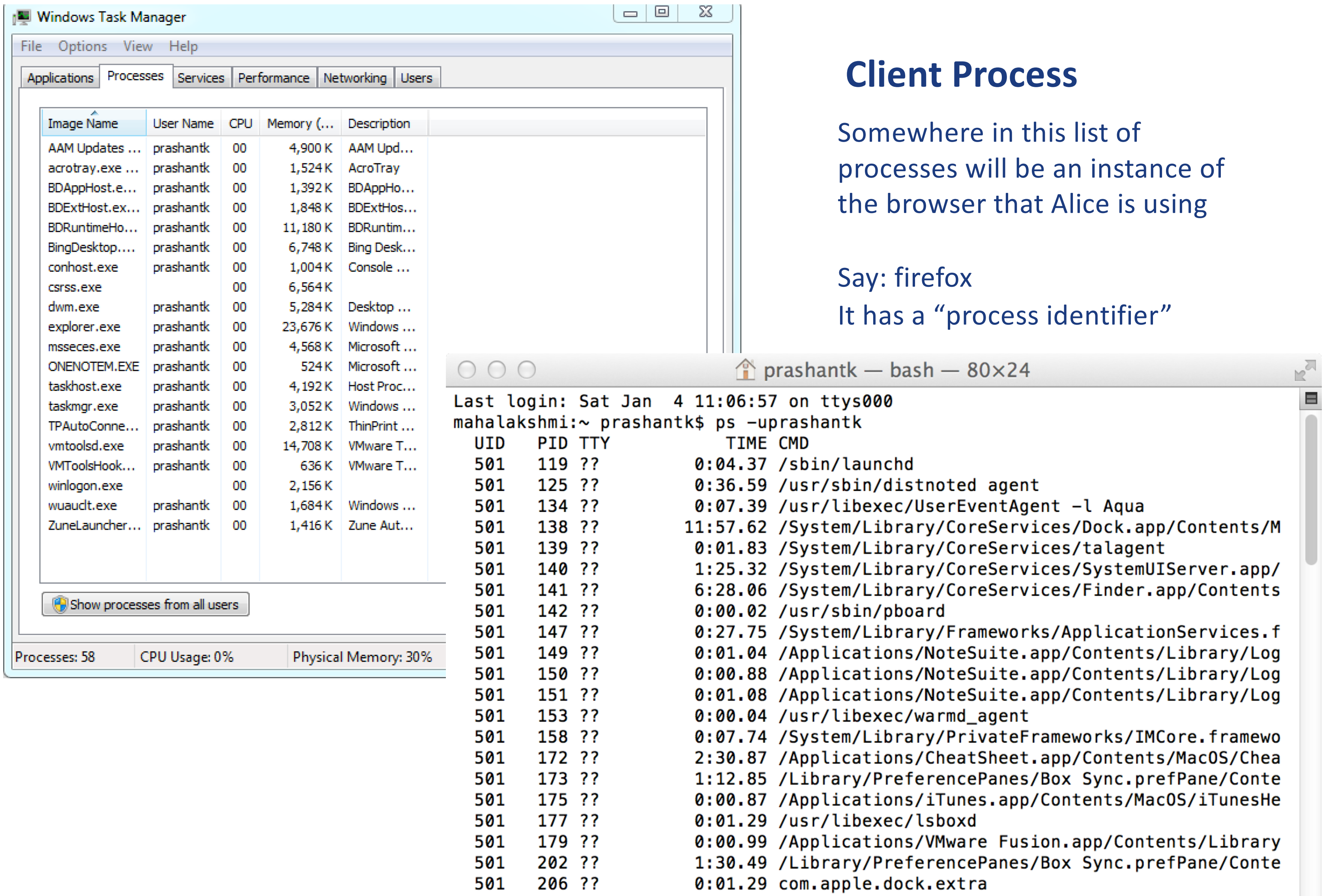


Figure modified from Kurose and Ross

Client Process

Somewhere in this list of processes will be an instance of the browser that Alice is using

Say: firefox
It has a “process identifier”



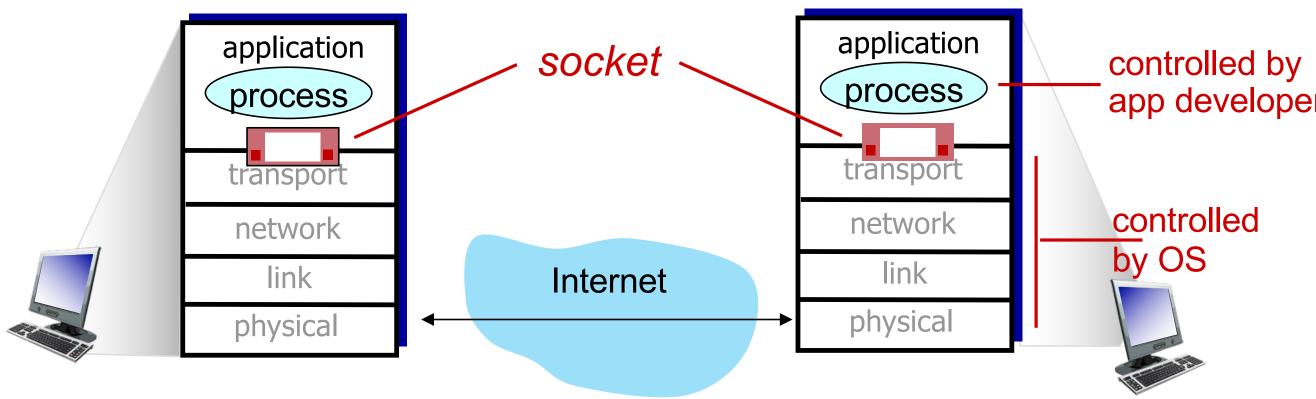
Client and server processes

- Alice's browser is a **client process** running on her computer
- It communicates with a **server process** that is running somewhere on some machine in Amazon.com's network
- Client initiates the contact (through the URL)
- Server responds to the contact
    - It has information about the client in the message that the client sends to the server

What is a socket?

- Interface between the **application layer** and the **transport layer**
- At a high level
  - Program **opens** a socket, specifying which transport layer **service** it wants to use
  - Can use **send** and **receive** functions on that socket
  - **Closes** socket to release resources

From a programming perspective, similar to APIs for **file I/O**



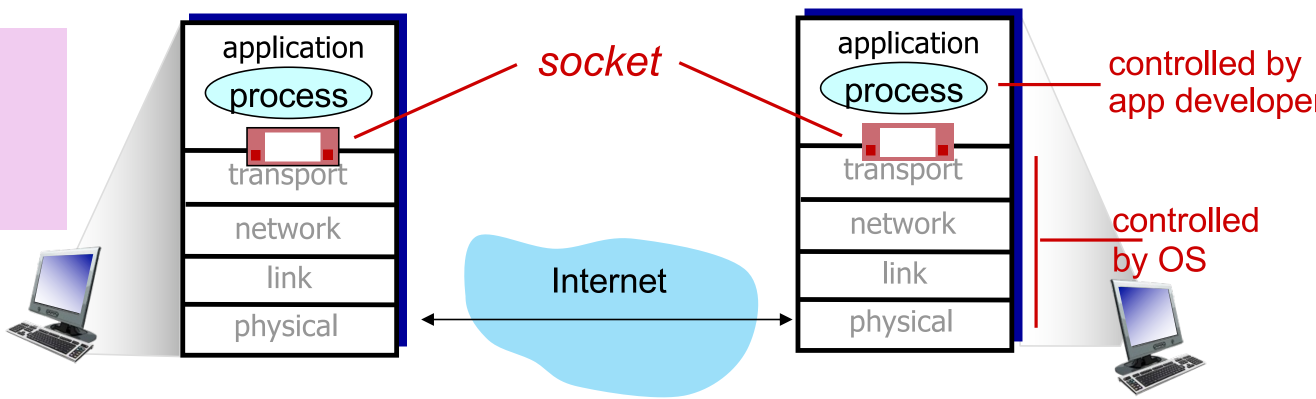
Sockets

- Programming interface used by a process to send and receive messages
  - Like the door to a house
- Sending process sends the message to the socket
  - Assumption: There is something outside the door to transport the message to the door of the receiving process
- At the door of the receiving process (socket), the message is received and pushed to the process
- Application developer has little control over what is outside the door, only controls the process
  - Can select from a set of transport protocols and some parameters

How do we specify where to send data?

- A socket is identified by:
  - **Service**: transport layer protocol (TCP vs UDP)
  - **IP address**: 32-bit address that uniquely identifies the host
  - **Port**: number between 0 and 65535 that identifies this socket (vs all other network applications running on this host)

In our letter analogy:
IP address -> building address
Port -> office number



Slide modified from Kurose & Ross

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?
  - *A:* no, *many* processes can be running on same host

- *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

What transport services can we use?

- UDP (User Datagram Protocol)
  - **Best-effort** (unreliable) data transfer
  - Sender can inject messages at whatever rate they choose, and UDP will pass them to the network layer. But it does **not** provide delivery guarantees (no recovery of lost packets)
- TCP (Transmission Control Protocol)
  - Connection-oriented **reliable, in-order** data transfer
  - Guarantees that sent data is received (unless connection breaks)
  - **Flow control** to avoid overwhelming receiver
  - **Congestion control** to throttle sending when network is overloaded

Discussion Questions

- Why would we ever use UDP?
- What other services would be useful for a network to provide?
  - And why aren't there Internet transport protocols to support them?

What transport service does an app need?

data integrity

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

timing

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

throughput

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

security

- encryption, data integrity, …

Transport service requirements: common apps

application	data loss	throughput	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 video:10kbps-5Mbps	yes, 100’s msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100’s msec
text messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *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, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

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,

Q: Why is there 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	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

Network Application Architectures

- How are network applications and their communication patterns **structured**?

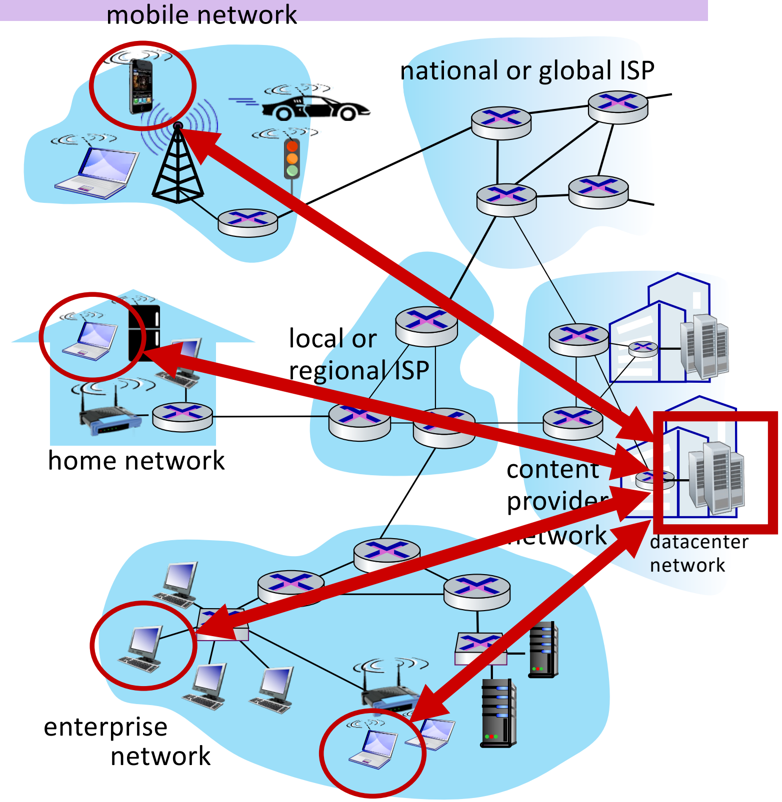
Application architectures

possible structure of applications:

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

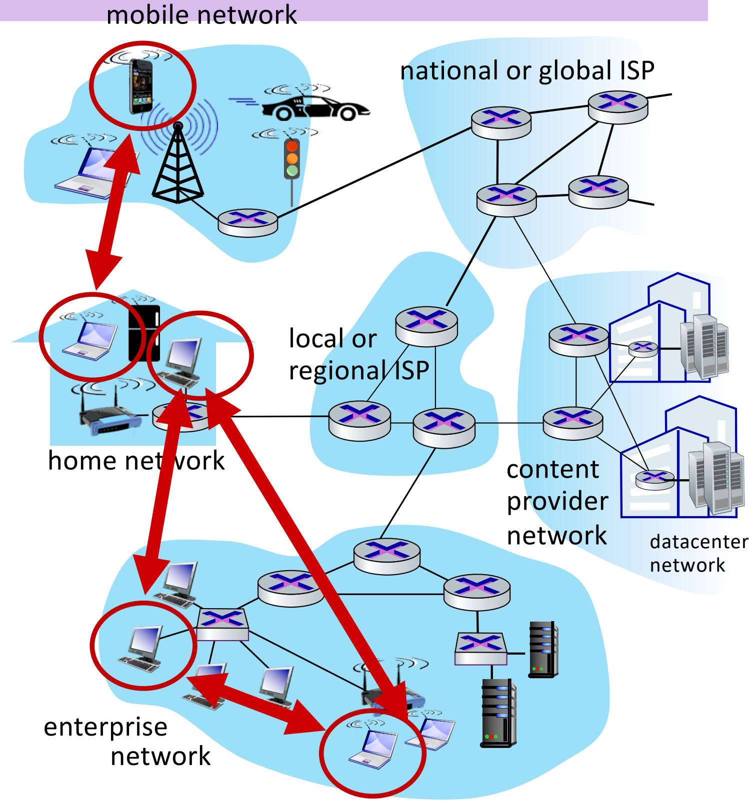
Client-Server Architecture

- Dedicated **servers** process requests from **clients** (users)
- Servers
  - always on and connected to the network (often located in data centers)
  - fixed, publicly known IP addresses
  - accept incoming client requests on a pre-determined port
- Clients
  - may come and go
  - don’t need fixed public IP address
  - don’t communicate directly with each other
- Most common paradigm
  - Used by Web, email, most messaging apps, and more



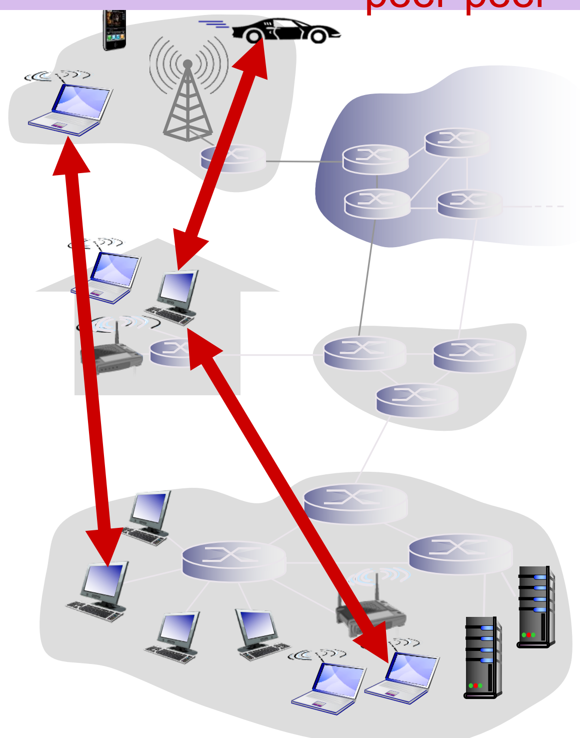
Peer-to-Peer (P2P) Architecture

- No dedicated servers
- **Peers** (users) communicate directly with each other
- Peers are **not** necessarily always on/connected
  - great for creating interesting research problems
  - not so great for a reliable service
- Very popular in the 2000s for file sharing
  - Current examples are BitTorrent, Bitcoin (in theory), Signal calls, some multiplayer games



P2P architecture (2)

- *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



Discussion Questions

- When might we choose a P2P architecture over client-server?

Summary

- Network applications use socket interface to access transport layer services
- Transport layer provides reliable (TCP) or unreliable (UDP) data transfer services
- Application communication patterns can use client-server or peer-to-peer paradigms (or hybrid)
- "Default" application communication: client-server architecture over TCP

TELCOM 2310: Applications of Networks

Lecture 4, Part 2: Web and HTTP

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

World Wide Web

- First “killer application” for the Internet
- NOT the same as the Internet
  - Even though we often use the terms that way (e.g. “I found this recipe on the Internet”)
- Distributed database of documents and other resources linked through Hypertext Transport Protocol (HTTP)

Web Pages

- **Web page** (document) consists of **objects**
  - Objects can be HTML files, CSS style sheets, Javascript files, images, videos, PDFs, text files, …
- Web page typically includes **base HTML file** and several **referenced objects**
- Each object identified by **URL (Uniform Resource Locator)**

Format: `protocol://host-name[:port]/directory-path/resource`
Example: `http://www.someschool.edu/someDept/pic.gif`

Resources: URIs, URNs, URLs

- The target of HTTP requests is called a “resource”
- Resources can be documents, photos, or anything.
- Resources are identified by a Uniform Resource Identifier (URI)
- Uniform Resource Locators (URLs) are the most common kind of URI
- Uniform Resource Name (URN) is another kind of URI for globally unique persistent identifiers within a namespace

URL syntax

- A URL is composed of five parts
  - A protocol specification (http, ftp, gopher)
  - An authority (domain name) (google.com, pitt.edu)
  - A port specification (usually defaults to :80 for http and :443 for https)
  - A relative path specifier (/docs/web/http, /~edmonds, /)
  - A fragment identifier (#schedule) or query (?q=”Cool search terms”)
- Tim Berners-Lee says “Cool URIs don’t change”

tp://**www.example.com**:80/path/to/my

Domain Name

http://www.example.com:80/path/

Protocol

---

com**:80**/path/to/myfile.html?key1=valu

Port

n:80**/path/to/myfile.html**?key1=value1

Path to the file

html?key1=value1&key2=value2#Some

Parameters

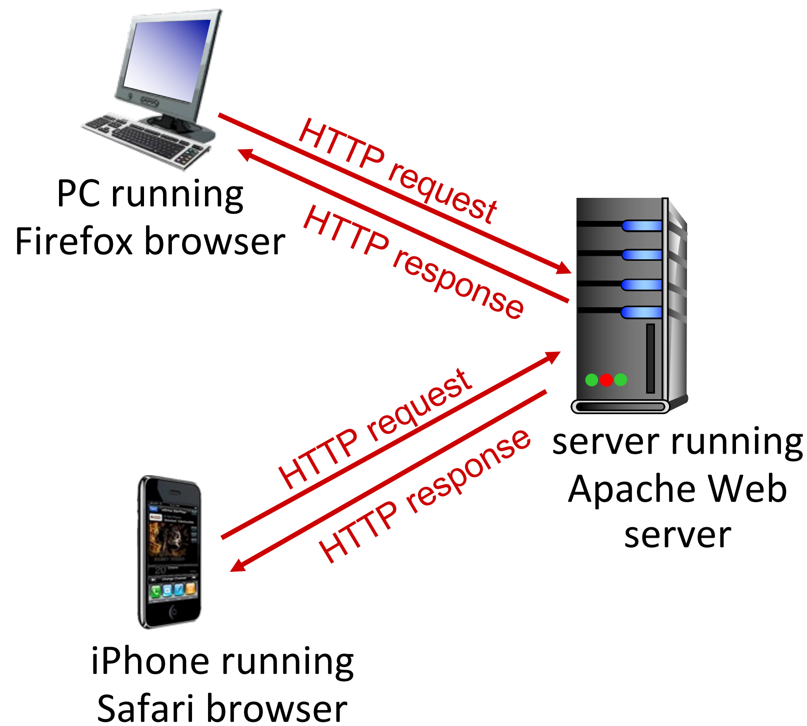
---

lue2#SomewhereInTheDocument

Anchor

Accessing Web Pages

- Client-server architecture, with TCP transport
- **Web browsers (clients)** request web pages from **web servers**
- **HTTP (Hypertext transfer protocol)** defines the communication protocol between web browsers and servers



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

aside

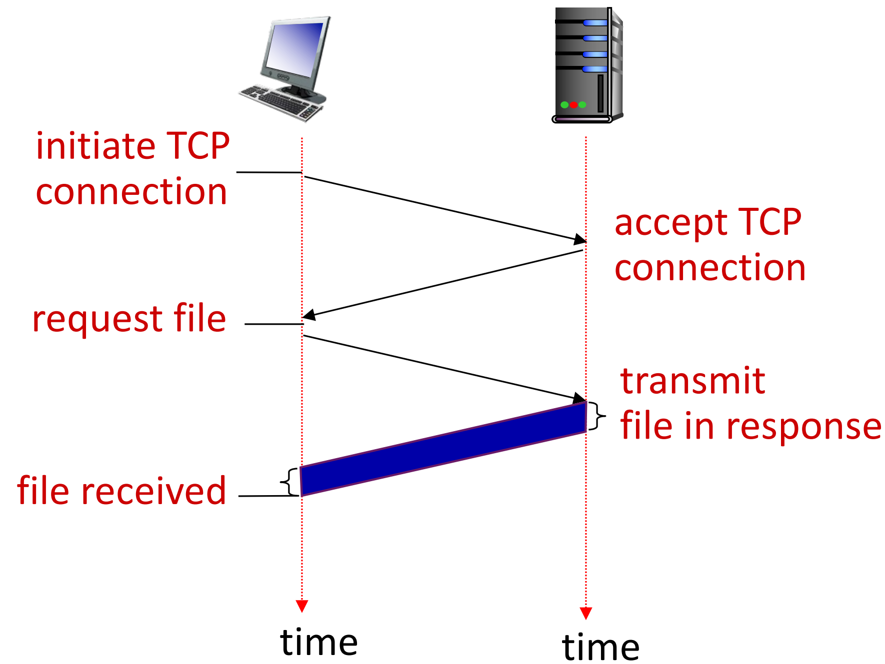
protocols that maintain "state" are complex!

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

HTTP Communication Steps

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**

1. Client **initiates TCP connection** to server at www.someSchool.edu on port 80
2. Server listening for TCP connections on port 80 **accepts** the connection
3. Client sends **HTTP request** message for object someDepartment/home.index to server on TCP connection
4. Server receives request and sends **HTTP response** message containing requested object to client on TCP connection



Client-to-Server Message Format

- HTTP Request Message
  - Request line: method, resource, and protocol version

request line

header lines

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
(blank line)
```

carriage return line feed indicates end of message

Client-to-Server Message Format

- HTTP Request Message
  - Request line: method, resource, and protocol version
  - Request headers: provide info or modify request
  - Body: optional data (e.g., to “POST” data to server)

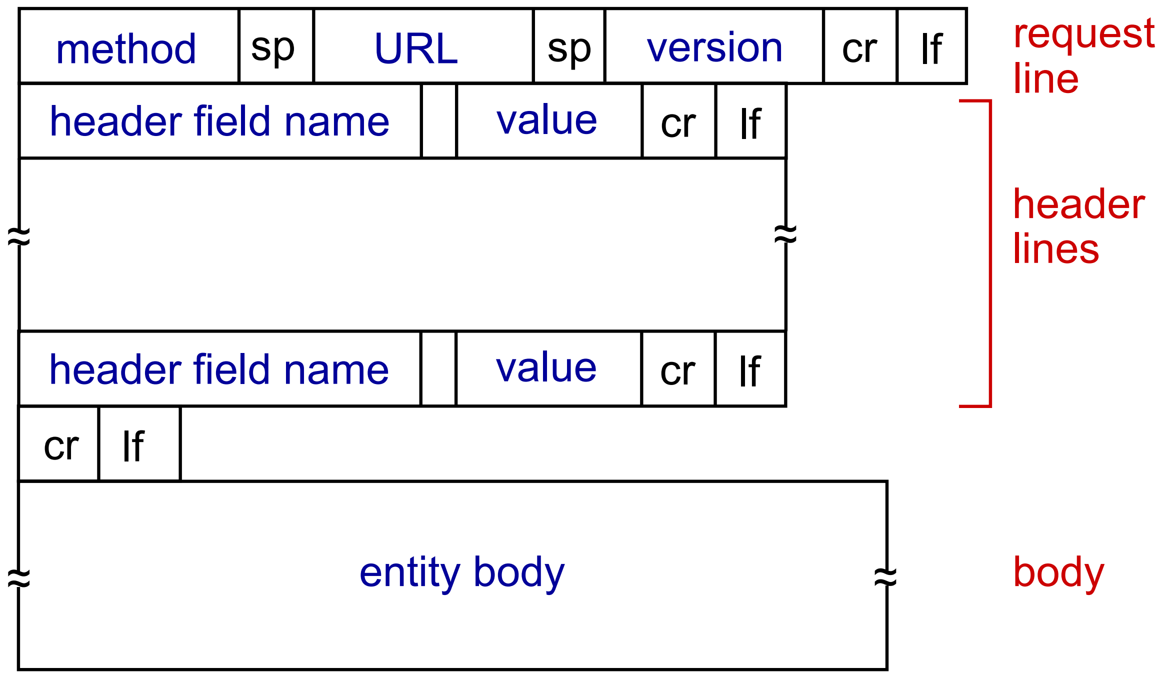
request line

header lines

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
(blank line)
```

carriage return line feed indicates end of message

HTTP Request General Format



Method types (HTTP 1.1)

- GET, HEAD
  - Retrieve information
  - HEAD is like GET, but leaves out object – used for debugging
  - Can specify user data as part of URL: **`www.somesite.com/animalsearch?monkeys&banana`**
- POST
  - Send information (e.g., web forms)
- PUT
  - Upload file in entity body to path specified in URL field
- DELETE
  - Delete file specified in the URL field

Server-to-Client Message Format

- HTTP Response Message
  - Status line: protocol version, status code, status phrase
  - Response headers: provide information
  - Body: optional data

status line
(protocol, status code, status phrase)

header lines

data
e.g., requested HTML file

```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Jan 2017 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 2006 ...
Content-Length: 6821
Content-Type: text/html
(blank line)
data data data data data ...
```

HTTP Status Code Examples

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)

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

HTTP is stateless

- Each request-response treated independently
  - Servers not required to retain state
- Good: Improves scalability on the server-side
  - Failure handling is easier
  - Can handle higher rate of requests
  - Order of requests doesn't matter
- Bad: Some applications need persistent state
  - Need to uniquely identify user or store temporary info
  - e.g., Shopping cart, user profiles, usage tracking, …

Question

- How does a stateless protocol keep state?

HTTP Sessions

- HTTP Sessions provide a mechanism for stitching together a series of independent requests
- A web session is a sequence of network HTTP request and response transactions associated to the same user
- Sessions can store variables – such as access rights and localization settings – for every interaction a user has with the web application for the duration of the session
- Can apply to anonymous or authenticated (logged-in) users
- When authenticated, sessions allow the ability to track requests to:
  - Apply access controls to create a secure web application
  - Regulate access to user data
  - Increase the usability of the application through user preferences

Examples of HTTP Sessions

- eCommerce - shopping carts, storing address & payment
- Complex Web Applications - Gmail, Github
- Social media - Facebook, Twitter
- Media and News - NYTimes limit of 10 articles per month

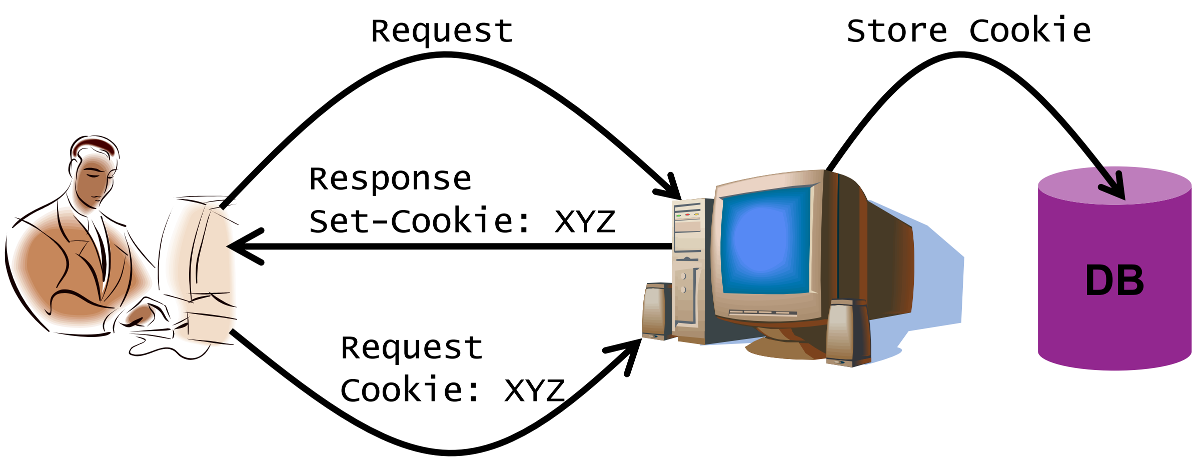
Cookies

- A cookie is a small piece of data sent by a server to a browser and stored on the user's computer while the user is browsing.
- Cookies can be attached to every HTTP request using the cookie HTTP Header
- This allows us to track a little bit of state across what would otherwise be completely independent HTTP requests and responses
- Cookies are primarily used for: (1) Session management, (2) personalization and (3) tracking.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

High-level State Maintenance: Cookies

- Client-side state maintenance
  - Client stores small state on behalf of server
  - Client sends state in future requests to the server



Cookie-Based Session Management

- This cookie is sent back to the server when the user tries to access certain pages
- The cookie allows the server to identify the user and retrieve the user's session from the session database, so that the user’s session is maintained.
- A cookie-based session ends when the user logs off or closes the browser.
- You can use Passport.js, which integrates nicely with Node.js and Express
- Session IDs should be random, very hard to guess identifiers (why?)

Image from HTTP Cookies in ASP.NET Web API

How Cookies Work

The Set-Cookie HTTP response header sends cookies from the server to the browser/device.. A simple cookie is set like this:

```
Set-Cookie: yucky-cookie=strawberry
```

```
Set-Cookie: yummy-cookie=oatmeal
```

Now, with every new request to the server, the browser will send back all previously stored cookies to the server using the Cookie header.

```
GET /sample_page.html HTTP/2.0 Host: www.example.org
```

```
Cookie: yummy_cookie=oatmeal; yucky-cookie=strawberry
```

Types of Cookies

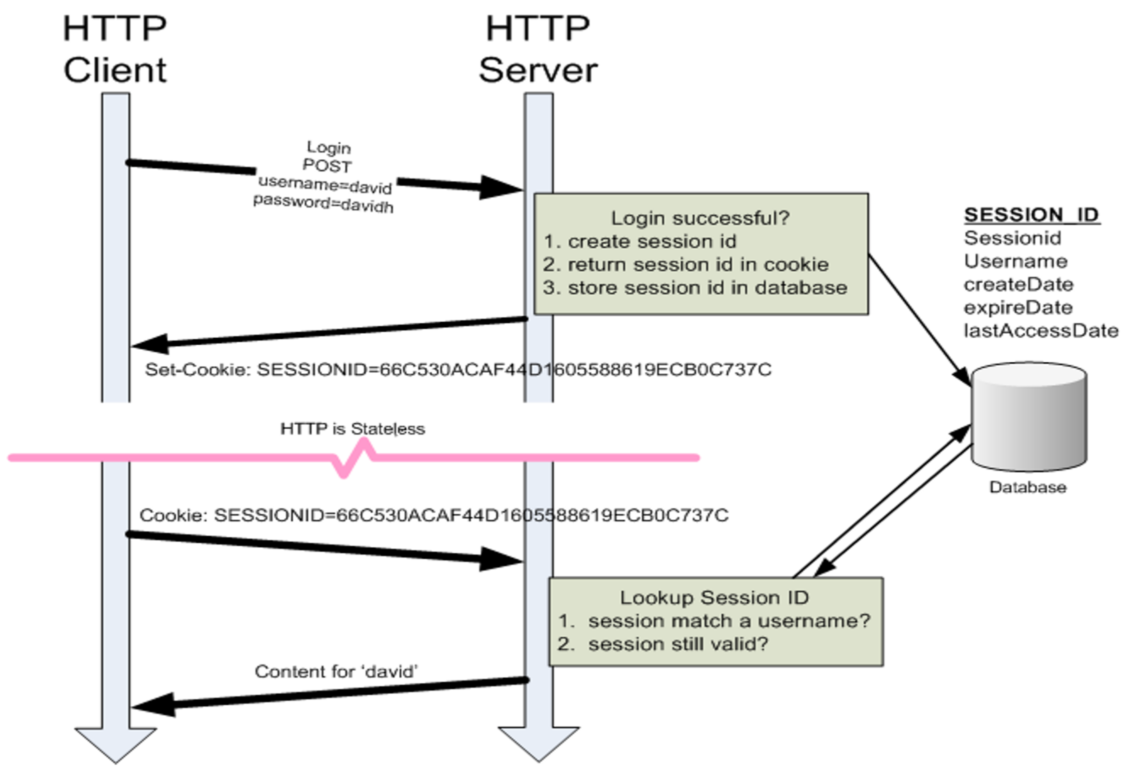
- Session Cookie - This type of cookies dies when the browser is closed because they are stored in the browser’s memory. Can be used for shopping carts or anonymous user preferences that don't need to be saved across a multiple browser sessions
- Persistent or Permanent Cookie - Stored in a file or database in the browser. Expire at a specific date (Expires) or after a specific length of time (Max-Age).
- Third Party Cookie - A cookie set by a different domain from the server. These cookies are used for tracking patterns and advertising
- Secure Cookie - Cookies that are only transmitted over an encrypted connection. Browser won't send the cookie over an insecure connection
- Zombie or Evercookies or Supercookies - Cookies that get recreated after they are deleted and persist on the client all the time. Popular with advertising and analytics trackers. VERY HARD TO DELETE

Cookie Attributes

- **Name** - Specifies the name of a cookie for retrieving the cookie
- **Value** - Specifies the value of cookie. Max size of all cookies is 4093 bytes per domain
- **Secure** - Specifies if the cookie should only be transmitted over encrypted HTTPS connections. Default is false
- **Domain** - Specifies the domain name associated with the cookie. Helps the browser determine when to send a cookie with HTTP requests (don't want to send all cookies to all servers)
- **Path** - Specifies a server path ("/", "/users/", "/login") for sending the cookie.
- **HTTPOnly** - Means the cookie will only be available on the HTTP protocol, not accessible to JavaScript
- **Expires** - Specify when the cookie expires and should no longer be sent with HTTP Requests. If set to 0 the cookie will expire when the browser closes

```
Set-Cookie: id=a3fWa; Expires=Wed, 13 Nov 2019 07:28:00
GMT; Secure; HttpOnly
```

How Cookies Work



HTTP Client Server interaction with cookies. Image from Hacking Articles

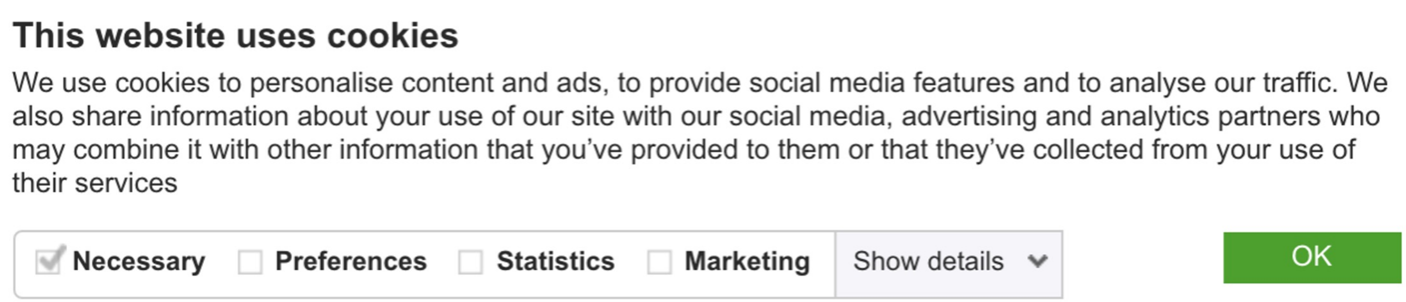
HTTP Cookies: Use and Abuse

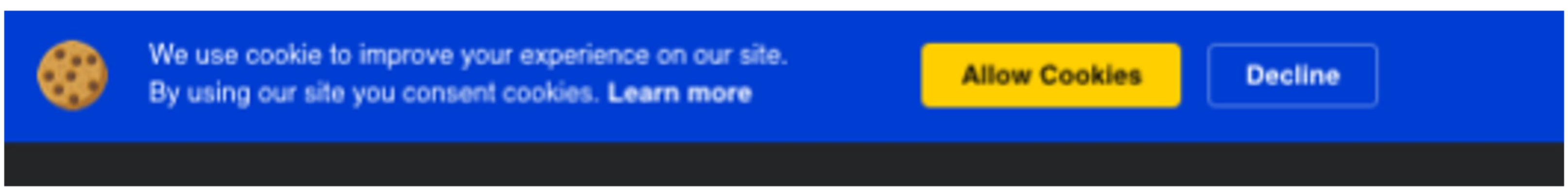
What cookies can be used for:

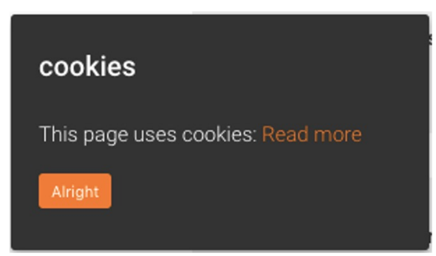
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

cookies and privacy:

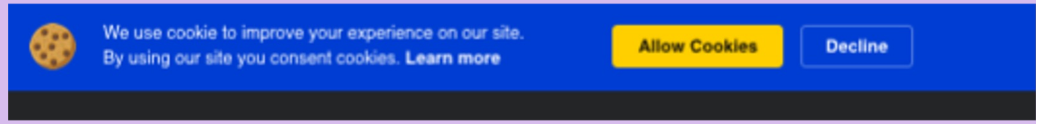
- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites


This website uses cookies
We use cookies to personalise content and ads, to provide social media features and to analyse our traffic. We also share information about your use of our site with our social media, advertising and analytics partners who may combine it with other information that you've provided to them or that they've collected from your use of their services
Necessary
Preferences
Statistics
Marketing
Show details
OK


We use cookie to improve your experience on our site.
By using our site you consent cookies. Learn more
Allow Cookies
Decline


cookies
This page uses cookies: Read more
Alright

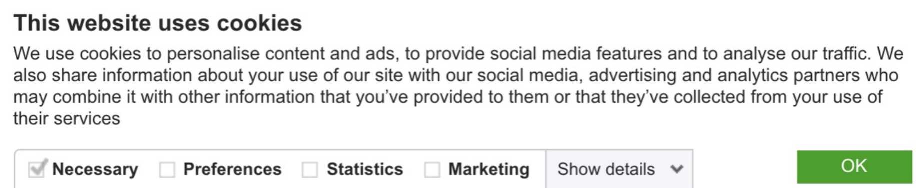
Cookie Laws

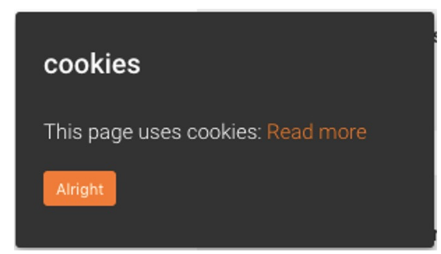


The EU Directive 2009/136/EC of the European Parliament means that before somebody can store or retrieve any information from a computer, mobile phone or other device, the user must give informed consent to do so.

Enforced in the US via the General Data Protection Regulation (May 25, 2018) which establishes that a business must have a “legal basis” for collecting personal data from individuals located in the EU.

The California Consumer Privacy Act (CCPA) will become effective on January 1, 2020.



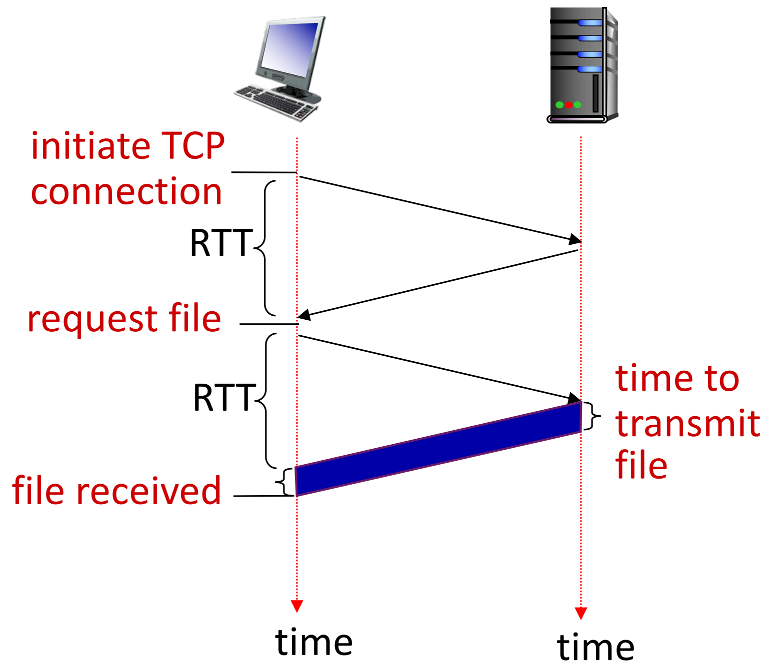


HTTP Performance

RTT (round trip time): time for a packet to travel from client to server and back

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



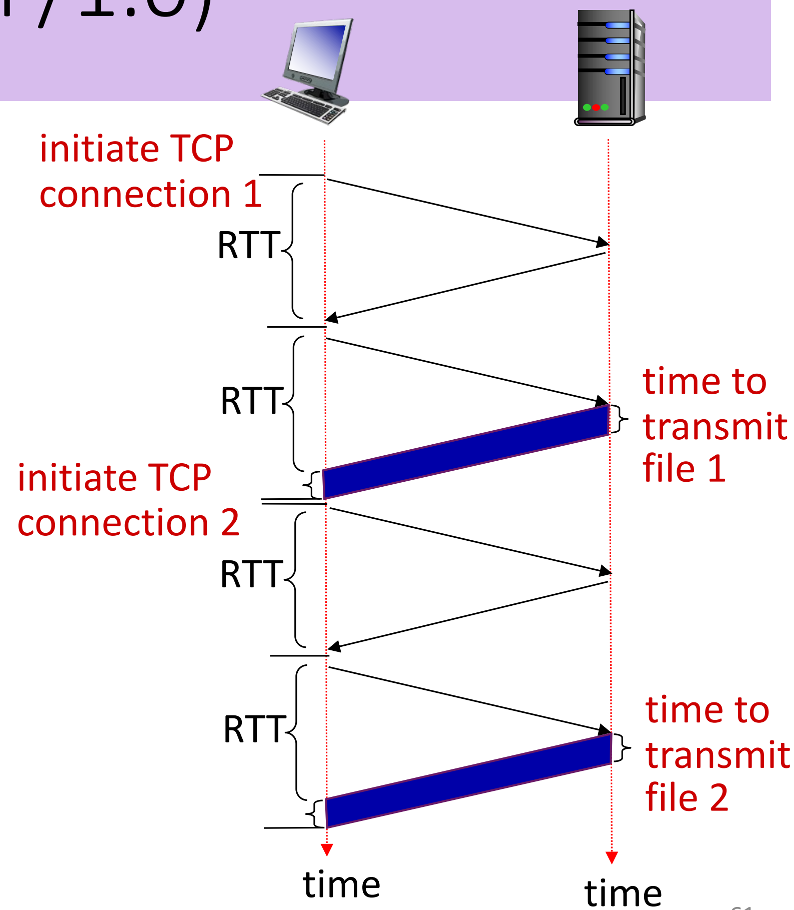
HTTP response time = 2RTT+ file transmission time

HTTP Performance

- Most webpages include more than one object…how do we retrieve all of them?

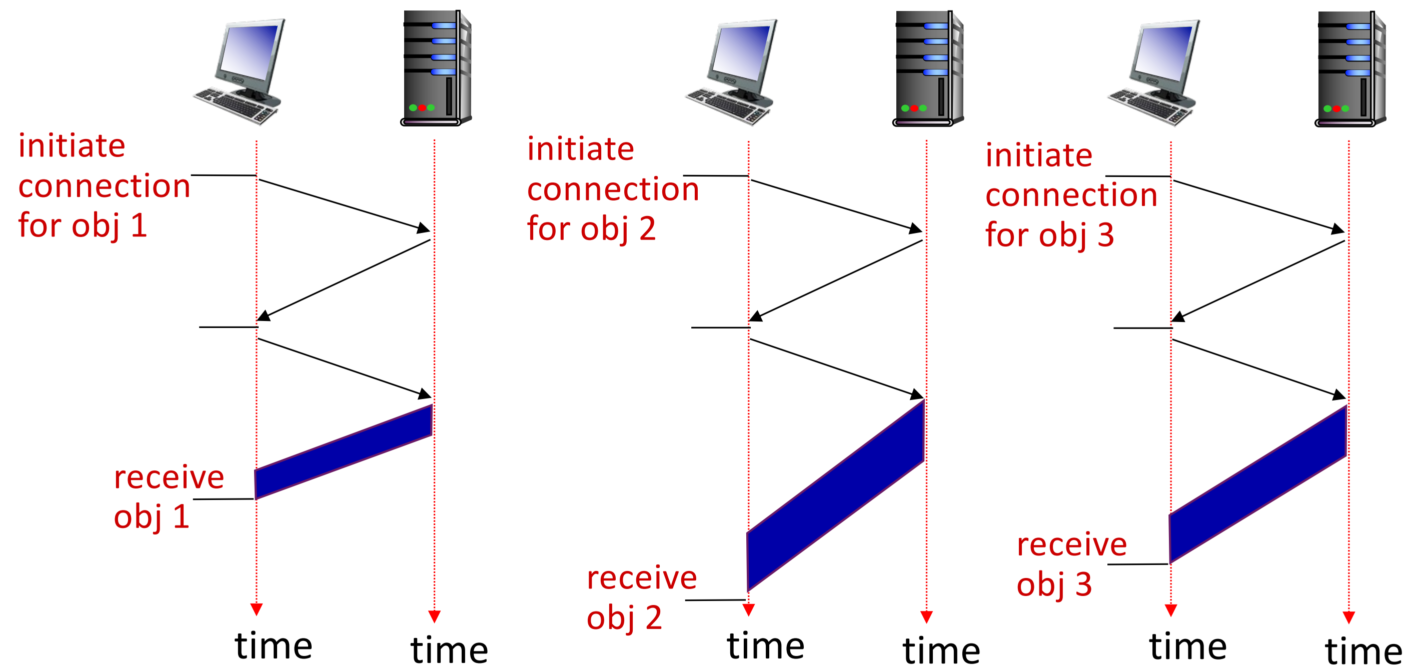
Non-Persistent HTTP (HTTP/1.0)

- Separate TCP connection for each object
- Naively, request objects one at a time (serially)… 2RTT + Transmission time **for each object**



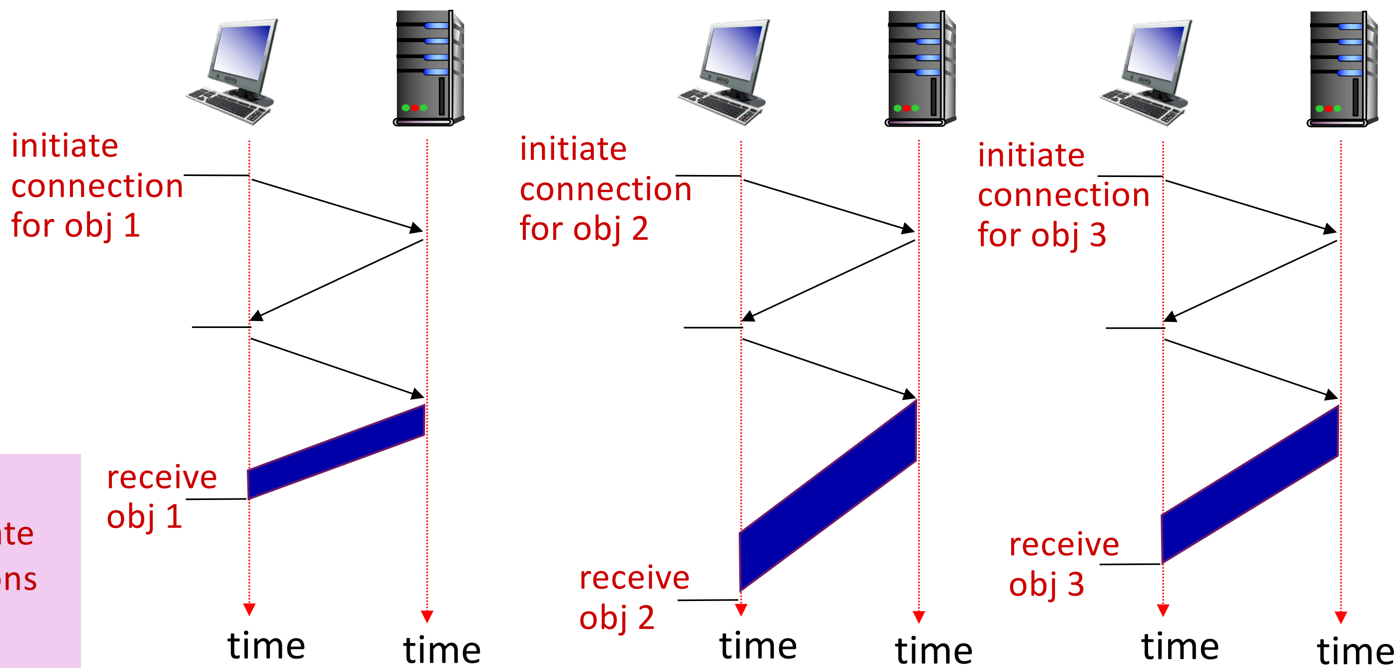
Non-Persistent HTTP (HTTP/1.0)

- **How can we reduce response time?**
- Separate TCP connection per object, but run them in **parallel**



Non-Persistent HTTP (HTTP/1.0)

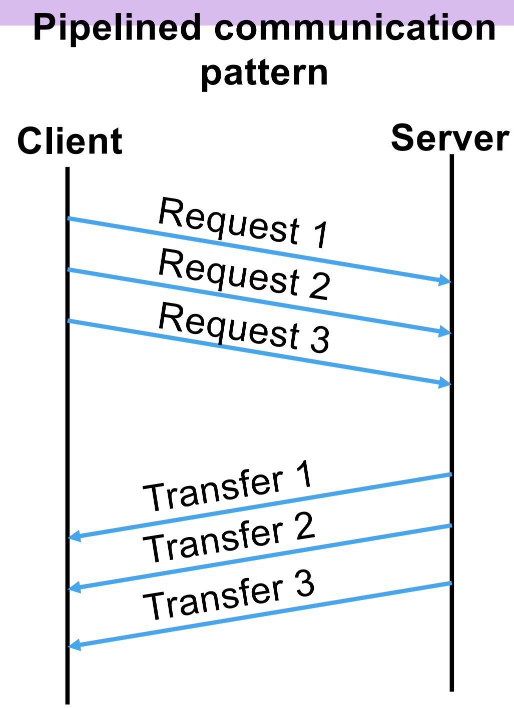
- How can we reduce response time?
- Separate TCP connection per object, but run them in parallel



But, 1) still has OS overhead of multiple TCP sessions; 2) can violate TCP congestion control expectations

Persistent HTTP (HTTP/1.1)

- **Maintain TCP connection across multiple requests**
  - Avoid overhead of setting up and tearing down many connections
  - Better match TCP expectations: allow TCP to learn RTT and bandwidth characteristics, support fair bandwidth sharing
- **Pipelining** to further reduce response time



HTTP Performance

- How long does it take to retrieve **n** small objects?
  - Since objects are small, assume transmission delay is negligible
  - Propagation delay (RTT) dominates
- **Non-persistent, serial**: ~ $2RTT\ x\ n$
- **Non-persistent, m parallel connections**: ~$2RTT\ x \left\lceil \frac{n}{m} \right\rceil$
- **Persistent (non-pipelined)**: ~ $(1 + n)\ x\ RTT$
- **Persistent, pipelined**: ~ $2\ x\ RTT$ for first set of requests, RTT after connection established

HTTP Performance

- Let *n=10 objects, RTT = 20ms, m=6 connections*

- **Non-persistent, serial**
  - $2RTT\ x\ n$ = 2(20)(10) = 400ms
- **Non-persistent, m parallel connections**
  - $2RTT\ x\ \left\lceil \frac{n}{m} \right\rceil$ = 2(20)(10/6) = 2(20)(2) = 80ms
- **Persistent (non-pipelined)**
  - $(1+n)\ x\ RTT$ = (1+10)(20) = 220ms
- **Persistent, pipelined**
  - $2\ x\ RTT$ = 2(20) = 40ms
  - RTT = 20ms

Note: these calculations assume we know the *n* objects to request up front AND transmission delay is negligible

Exercise: adapt to reflect the fact that we need to receive the base HTML file first to learn about other objects (hint: only affects parallel & pipelined)

Exercise: adapt to reflect transmission delays

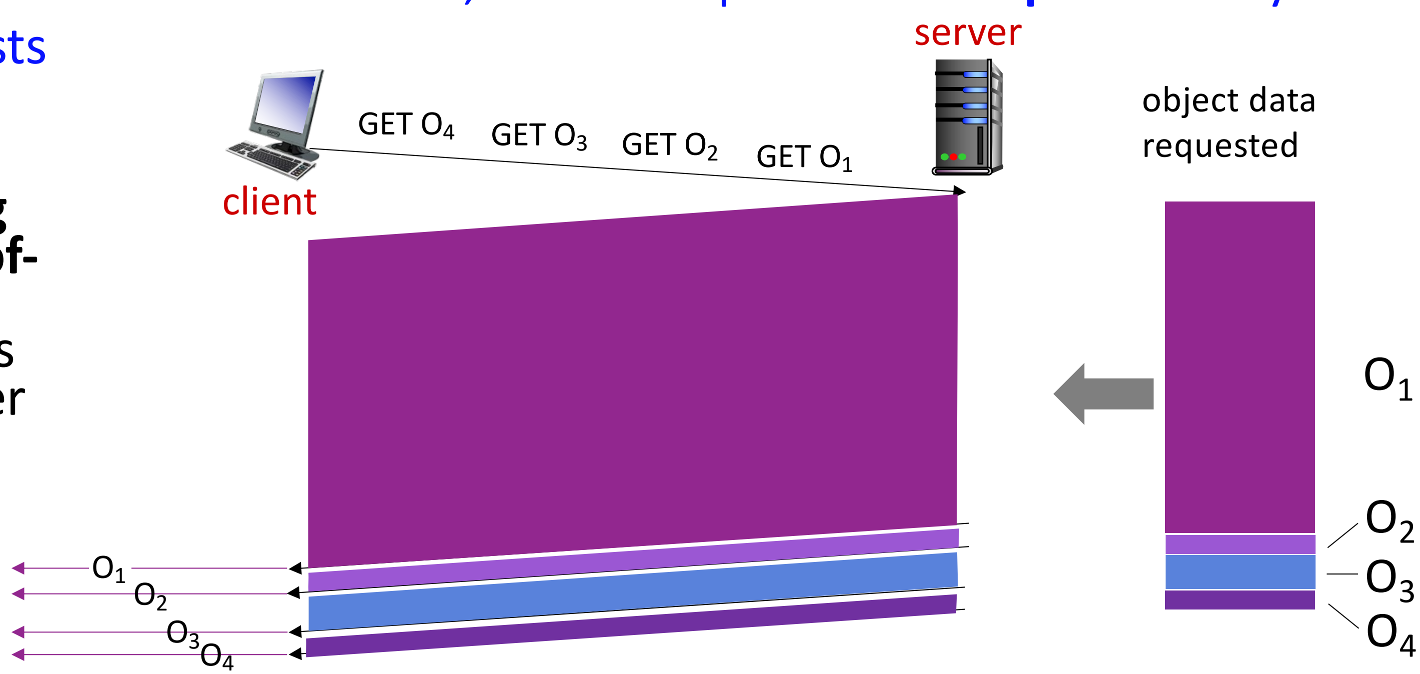
HTTP Performance

- **What if the objects are large?**
- Need to account for **transmission delay**
- Pipelining doesn't help with transmission delay – <u>we still need to transfer all the data</u>
  - Let *F = file size in bits, B = bandwidth in bits/sec, n = number of objects*
  - Very best case; still need to request object and retrieve it: $\sim RTT + \frac{nF}{B}$
- **To improve further, need to find a way to reduce RTT and/or increase B (or reduce data size?)…**

But first…a look at HTTP/2

Even if we can't reduce **total** load time, we can improve **user experience** by prioritizing requests

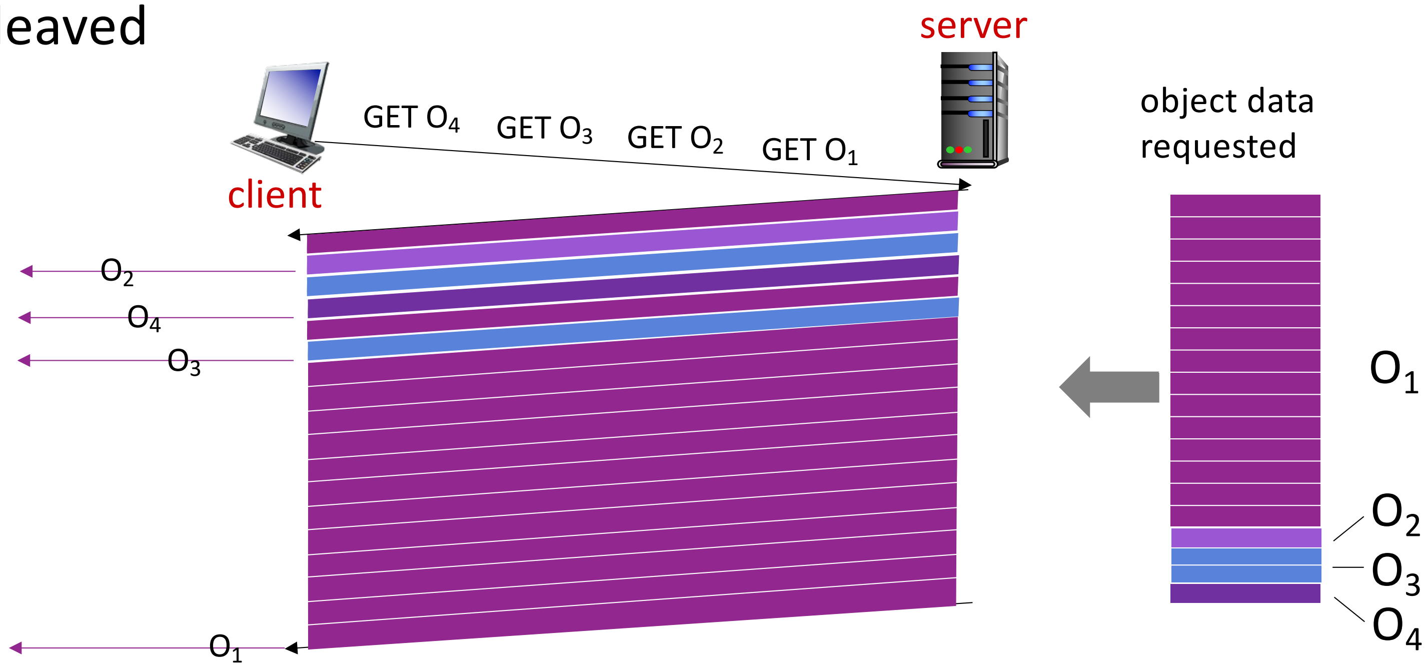
HTTP/1.1 pipelining suffers from **Head-of-Line (HOL) Blocking** because it processes requests in the order they are received



objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2 Multiplexing

- Objects are divided into *frames*, and frames for different objects can be interleaved



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP/2: Key Performance Features

- Objects are divided into *frames*, and frames for different objects can be interleaved
- Requests can be assigned **priorities** and **dependencies** to optimize transmission order
- **Server push** allows server to send responses for objects **before they are explicitly requested**
  - Can determine which objects are needed from **base HTML** file
- Note: new version of HTTP, HTTP/3 was recently standardized (not using TCP, but QUIC)

Summary

- HTTP is the protocol powering the Web application
- Simple, text-based stateless protocol
  - but, cookies provide a mechanism for maintaining state
- Persistent connections, parallel connections, and pipelining can reduce response time
- HTTP/2 aims to improve user experience via request multiplexing