Poll Everywhere

pollev.com/naomila

Compared to previous homeworks, how did you find the difficulty of HW3?

Easier — About the Same — Harder

CSE333 Systems Programming

Networking: Server Sockets & HTTP

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Administrivia

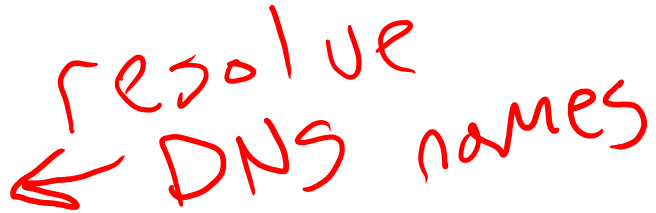
- No class on Monday (Memorial Day)
- Two little networking programs due next week:
  - EX10 (client) due Wednesday
  - EX11 (server) due Friday
    - Refer to lecture and section code to do these two exercises!!!!
- HW4 out later today
  - Using server sockets to serve a website over HTTP

Lecture Outline

- **Server sockets**
- HTTP

Last time

- TCP Client using sockets:
  1) Figure out the IP address and port to connect to
  2) Create a socket
  3) Connect the socket to the remote server
  4) `read()` and `write()` data using the socket
  5) Close the socket



Socket API: Server TCP Connection

- Pretty similar to clients, but with additional steps:

1) Figure out the IP address and port on which to listen
2) Create a socket
3) `bind()` the socket to the address(es) and port
4) Tell the socket to `listen()` for incoming clients
5) `accept()` a client connection, which creates a *new* socket for it
6) `read()` and `write()` to that new client socket
7) `close()` the client socket
8) Rinse and repeat

The idea with listener sockets

https://www.yelp.com/biz_photos/tea-era-mountain-view-2?select=_EH3rI_MLPFIlColyFFxaw

Servers

- The goals of a server socket are different than a client socket
  - Want to bind the socket to a particular *port* of one or more IP addresses of the server
  - Want to allow multiple clients to connect to the same port
    - OS uses client IP address and port numbers to direct I/O to the correct server file descriptor

- Servers can have multiple IP addresses ("*multihoming*")
  - Usually have at least one externally-visible IP address, as well as a local-only address (127.0.0.1)

Step 1: Figure out IP address(es) & Port

- Step 1: `getaddrinfo()` invocation may or may not be needed (but we'll use it)
  - Do you know your IP address(es) already?
    - Static vs. dynamic IP address allocation
    - Even if the machine has a static IP address, don't wire it into the code – better to look it up dynamically or use a configuration file
  - Can request listening on "all" IP addresses by passing `NULL` as `hostname` and setting `AI_PASSIVE` in `hints.ai_flags`
    - Effect is to use address 0.0.0.0 (IPv4) or :: (IPv6)

Step 2: Create a Socket

- Step 2: `socket()` call is same as before
  - Can directly use constants or fields from result of `getaddrinfo()`
  - Recall that this just returns a file descriptor – IP address and port are not associated with socket yet

Step 3: Bind the socket

```
int bind(int sockfd, const struct sockaddr* addr,
         socklen_t addrlen);
```

- Looks nearly identical to **`connect()`**!
- Returns `0` on success, `-1` on error

- Some specifics for `addr`:
  - **Address family:** `AF_INET` or `AF_INET6`
    - What type of IP connections can we accept?
    - POSIX systems can handle IPv4 clients via IPv6 so use `AF_INET6` ☺
    - `AF_UNSPEC` doesn't work as expected: it can bind to v4-only socket
  - **Port:** port in network byte order (**`htons()`** is handy)
  - **Address:** specify *particular* IP address or *any* IP address
    - "Wildcard address" – `INADDR_ANY` (IPv4), `in6addr_any` (IPv6)

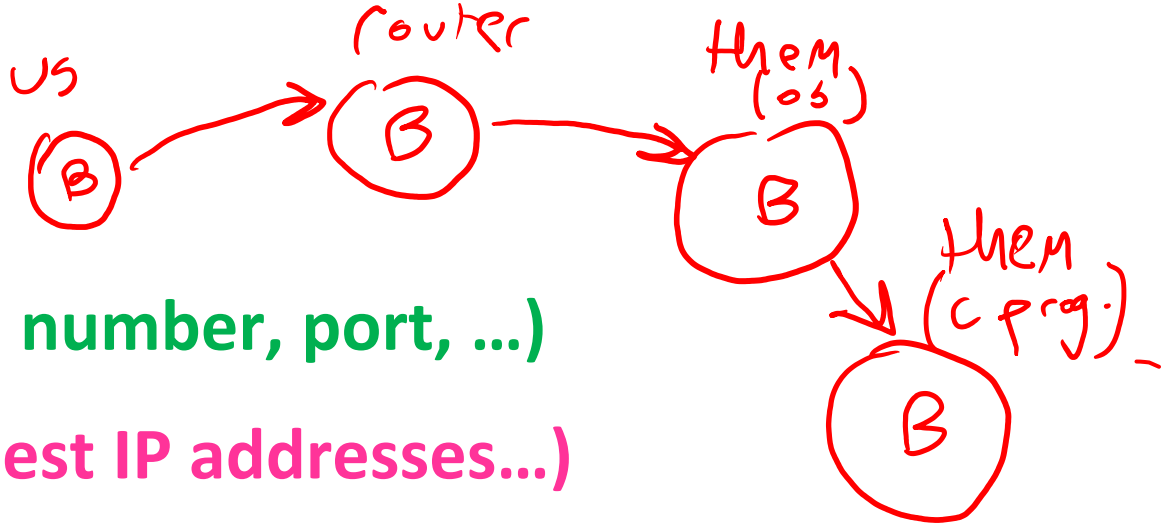
Step 4: Listen for Incoming Clients

- `int listen(int sockfd, int backlog);`
  - Tells the OS that the socket is a listening socket that clients can connect to
  - Returns `0` on success, `-1` on error
  - `backlog`: maximum length of connection queue
    - Gets truncated, if necessary, to defined constant `SOMAXCONN`
    - The OS will refuse new connections once queue is full until server **`accept`**`()`s them (removing them from the queue)
  - Clients can start connecting to the socket as soon as **`listen`**`()` returns
    - Server can't use a connection until you **`accept`**`()` it

Poll Everywhere

pollev.com/naomila

We're using a TCP socket to make a simple 1:1 chat program. What do we need to include in the buffer we pass to `write()`?



A. The user's chat message

B. All of the above + TCP info (sequence number, port, …)

C. All of the above + IP info (source & dest IP addresses…)

D. All of the above + Ethernet info (source & dest MAC addresses)

E. I disavow technology and go to live a quiet life in the mountains

Example Wallflower 

- See `server_bind_listen.cc`
  - Takes in a port number from the command line
  - Opens a server socket, prints info, then listens for connections for 20 seconds
    - Can connect to it using netcat (`nc`)

Step 5: Accept a Client Connection

output telling us IP & ports of the other end

```
int accept(int sockfd, struct sockaddr* addr,
           socklen_t* addrlen);
```

- Returns a new (different from `sockfd`), active, ready-to-use socket file descriptor connected to a client (or `-1` on error)
  - `sockfd` must have been created, bound, *and* listening
  - Pulls a queued connection or waits for an incoming one
- `addr` and `addrlen` are output parameters
  - `*addrlen` should initially be set to `sizeof(*addr)`, gets overwritten with the size of the client address
  - Address information of client is written into `*addr`
    - Use **`inet_ntop()`** to get the client's printable IP address
    - Use **`getnameinfo()`** to do a *reverse DNS lookup* on the client

Example Parrot 

- See `server_accept_rw_close.cc`
  - Gets a port number from the command line
  - Opens a server socket, prints info, then listens for connections
    - Can connect to it using netcat (`nc`)
  - Accepts connections as they come
  - Echoes any data the client sends to it on `stdout` and also sends it back to the client

Something to Note

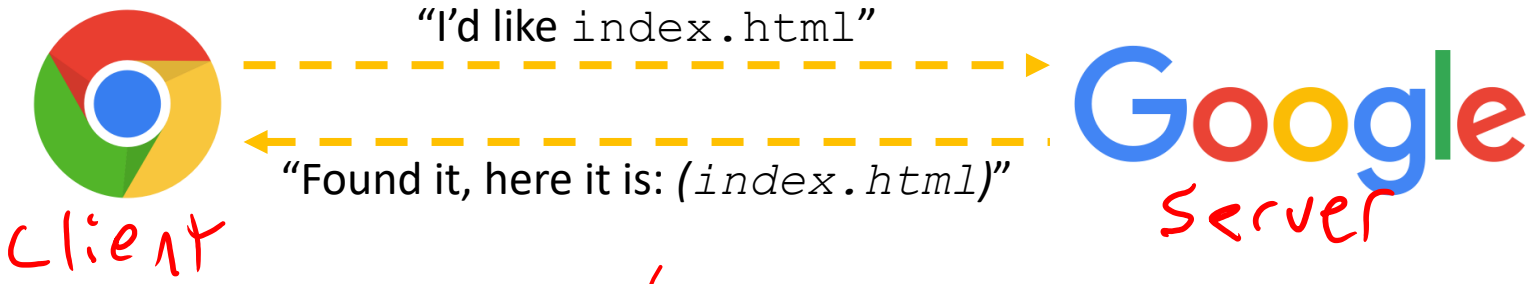
- Unlike `netcat`, our server code is not concurrent
  - Single thread of execution
  - The thread blocks while waiting for the next connection
  - The thread blocks waiting for the next message from the connection

- A crowd of clients is, by nature, concurrent
  - While our server is handling the next client, all other clients are stuck waiting for it ☹

Lecture Outline

- Server sockets
- **HTTP**

HTTP Basics



- A client establishes one or more TCP connections to a server
  - The client sends a request for a web object over a connection and the server replies with the object's contents

- How do we **formalize** the way our web browser asks Google's web server for data?
  - An application layer protocol 😎

What exactly are protocols again?

- A protocol is a set of rules governing the format and exchange of messages in a computing system
  - What messages can a client exchange with a server?
    - What is the syntax of those messages?
    - What do they mean?
    - How are errors conveyed?

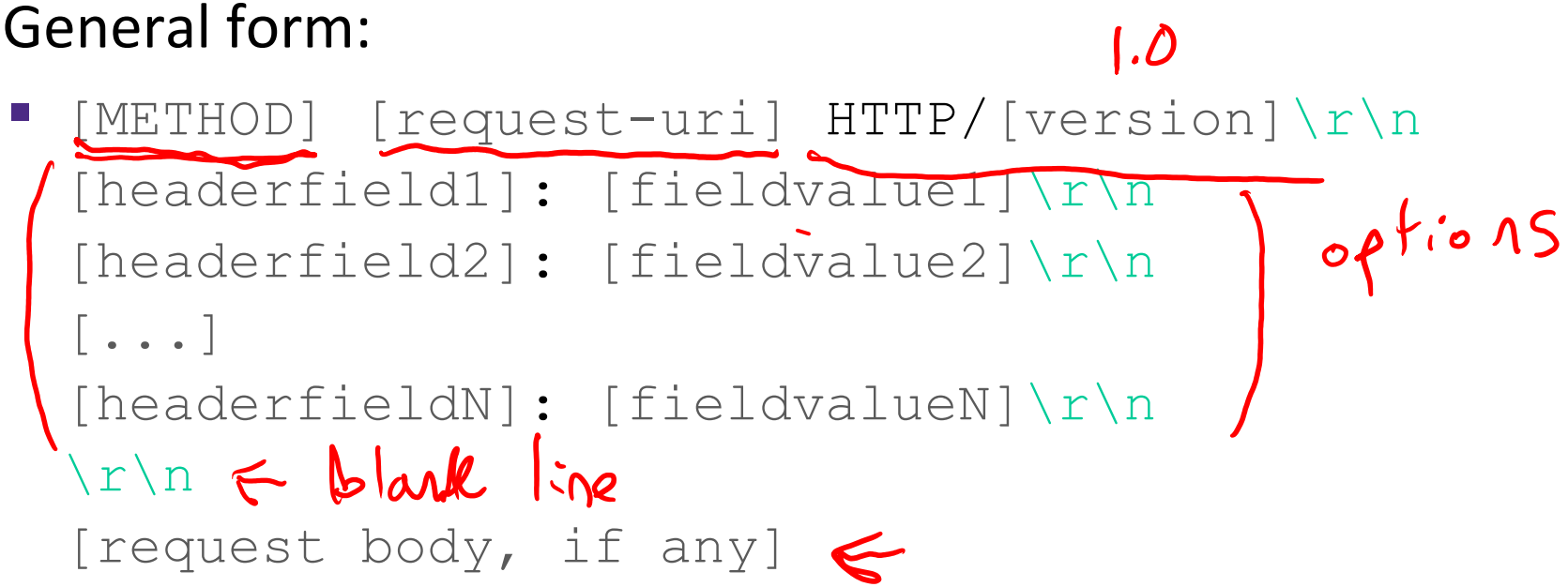
- A protocol is (roughly) the network equivalent of an API

HTTP

- Hypertext Transport Protocol
  - A request / response protocol
    - A client (web browser) sends a request to a web server
    - The server processes the request and sends a response
  - Typically, a **request** asks a server to retrieve a resource
    - A *resource* is an object or document, named by a Uniform Resource Identifier (**URI**)
  - A **response** contains the bytes of that resource
    - Or, if not, an error code expressing why the bytes aren't there
  - All you could want to know (and more!): https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

HTTP Requests

- ASCII plaintext
- General form:
  - `[METHOD] [request-uri] HTTP/[version]\r\n`
    `[headerfield1]: [fieldvalue1]\r\n`
    `[headerfield2]: [fieldvalue2]\r\n`
    `[...]`
    `[headerfieldN]: [fieldvalueN]\r\n`
    `\r\n`
    `[request body, if any]`



- Demo: use nc to see a real request

HTTP Methods

- Common HTTP methods used when you visit websites:
  - `GET`: "please send me the named resource"
  - `POST`: "I'd like to submit data to you" (*e.g.* file upload)
  - `HEAD`: "Send me the headers for the named resource"
    - Doesn't send resource; often to check if cached copy is still valid

- Other methods, mostly for HTTP APIs:
  - `PUT`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT`, `PATCH`, `...`
    - For instance: `TRACE` – "show any proxies or caches in between me and the server"

HTTP Versions

- All current browsers and servers "speak" HTTP/1.1
  - Version 1.1 of the HTTP protocol
    - https://www.w3.org/Protocols/rfc2616/rfc2616.html
  - Standardized in 1997 and meant to fix shortcomings of HTTP/1.0
    - Better performance, richer caching features, better support for multihomed servers, and much more

- HTTP/2 standardized mid 2010's (published in 2015)
  - Allows for higher performance but doesn't change the basic web request/response model
  - Will coexist with HTTP/1.1 for a long time

Client Headers

- The client can provide zero or more request "headers"
  - These provide information to the server or modify how the server should process the request

- You'll encounter many in practice
  - `Host:` the domain name the user entered into their browser bar
  - `User-Agent:` the name and version of the browser the user is on
  - `Accept:` the content types the client prefers or can accept
  - `Cookie:` an HTTP cookie previously set by the server
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers#request_context

A Real Request

```
GET / HTTP/1.1
Host: attu.cs.washington.edu:3333
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/66.0.3359.181 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
image/apng,*/*;q=0.8
DNT: 1
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: SESS0c8e598bbe17200b27e1d0a18f9a42bb=5c18d7ed6d369d56b69a1c0aa441d7
8f; SESSd47cbe79be51e625cab059451de75072=d137dbe7bbe1e90149797dcd89c639b1;
_sdsat_DMC_or_CCODE=null; _sdsat_utm_source=; _sdsat_utm_medium=; _sdsat_ut
m_term=; _sdsat_utm_content=; adblock=blocked; s_fid=50771A3AC73B3FFF-3F18A
ABD559FFB5D; s_cc=true; prev_page=science.%3A%2Fcontent%2F347%2F6219%2F262%
2Ftab-pdf; ist_usr_page=1; sat_ppv=79; ajs_anonymous_id=%229225b8cf-6637-49
c8-8568-ecb53cfc760c%22; ajs_user_id=null; ajs_group_id=null; __utma=598078
07.316184303.1491952757.1496310296.1496310296.1; __utmc=59807807; __utmc=80
...
```

HTTP Responses

- General form:
  - `HTTP/[version] [status code] [reason]\r\n`
    `[headerfield1]: [fieldvalue1]\r\n`
    `[headerfield2]: [fieldvalue2]\r\n`
    `[...]`
    `[headerfieldN]: [fieldvalueN]\r\n`
    `\r\n`
    `[response body, if any]`

- Demo: use **`nc -C`** to see a real response
  - The "`-C`" option uses \r\n as line endings

Status Codes and Reason

- *Code*: numeric outcome of the request – easy for computers to interpret
  - A 3-digit integer with the 1st digit indicating a response category
    - `1xx`: Informational message
    - `2xx`: Success
    - `3xx`: Redirect to a different URL
    - `4xx`: Error in the client's request
    - `5xx`: Error experienced by the server
- *Reason*: human-readable explanation
  - *e.g.* "OK" or "Moved Temporarily"

Common Statuses

- `HTTP/1.1 200 OK`
  - The request succeeded and the requested object is sent
- `HTTP/1.1 404 Not Found`
  - The requested object was not found
- `HTTP/1.1 301 Moved Permanently`
  - The object exists, but its name has changed
    - The new URL is given as the "`Location:`" header value
- `HTTP/1.1 500 Server Error`
  - The server had some kind of unexpected error

Server Headers

- The server can provide zero or more response "headers"
  - These provide information to the client or modify how the client should process the response

- You'll encounter many in practice
  - `Server:` a string identifying the server software
  - `Content-Type:` the type of the requested object
  - `Content-Length:` size of requested object
  - `Last-Modified:` a date indicating the last time the request object was modified
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers#response_context

A Real Response

```
HTTP/1.1 200 OK
Date: Mon, 21 May 2018 07:58:46 GMT
Server: Apache/2.2.32 (Unix) mod_ssl/2.2.32 OpenSSL/1.0.1e-fips
mod_pubcookie/3.3.4a mod_uwa/3.2.1 Phusion_Passenger/3.0.11
Last-Modified: Mon, 21 May 2018 07:58:05 GMT
ETag: "2299e1ef-52-56cb2a9615625"
Accept-Ranges: bytes
Content-Length: 82
Vary: Accept-Encoding,User-Agent
Connection: close
Content-Type: text/html
Set-Cookie:
bbbbbbbbbbbbbbb=DBMLFDMJCGAOILMBPIIAAIFLGBAKOJNNMCJIKKBKCDMDEJHMPONHCILPIBL
ADEAKCIABMEEPAOPMMKAOLHOKJMIGMIDKIHNCANAPHMFMBLBABPFENPDANJAPIBOIOOOD;
HttpOnly

<html><body>
<font color="chartreuse" size="18pt">Awesome!!</font>
</body></html>
```

Extra Exercise #1

- Write a program that:
  - Creates a listening socket that accepts connections from clients
  - Reads a line of text from the client
  - Parses the line of text as a DNS name
  - Does a DNS lookup on the name
  - Writes back to the client the list of IP addresses associated with the DNS name
  - Closes the connection to the client

Extra Exercise #2

- Write a program that:
  - Creates a listening socket that accepts connections from clients
  - Reads a line of text from the client
  - Parses the line of text as a DNS name
  - Connects to that DNS name on port 80
  - Writes a valid HTTP request for "/"

```
GET / HTTP/1.1\r\n
Host: <DNS name>\r\n
Connection: close\r\n
\r\n
```

  - Reads the reply and returns it to the client