# ECE 435 – Network Engineering
# Lecture 5

Vince Weaver

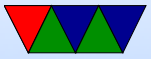
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

2 February 2026

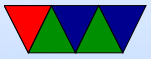
# Announcements

- HW#1 was due
- HW#2 was posted. Write a mini-webserver.

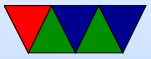


# Aside on Modern Web Design

- A lot more involved than the simple HTML from last time
- Like all things in this class, we often teach stuff from the 90s that was understandable because modern stuff is overly-complicated
- If need to generate web content these days often use some sort of tool that hides everything
- For example, the "official" way to create a personal website at UMaine is using a wordpress blog

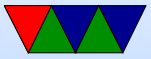


- You might also use high-level things like wikis and git/markdown

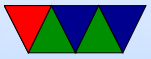


# More modern website notes

- Have to sit through talks where UMaine website team says their plans. It makes a 90s web designer sad
  - Designed for mobile + mobile browsers first
  - Designed for touch, swipe, full-screen menus
  - Lots of graphics and animations with minimal actual content
  - Less frequently accessed info removed or hidden behind firewall (possibly for ADA Title 2 reasons)
- Even further they plan to make website optimized for AI

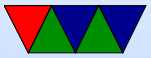


scraping as they think students don't use web browsers anymore but instead ask AI



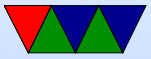
# Web-servers (Historical)

- Famously netcraft had a list (meme netcraft reports BSD is dying)
- NCSA was first popular one (free)
  License said you had to ship unmodified code, so often shipped with patches alongside
- Apache ("a patchy" version of NCSA) took over
- Microsoft IIS
- Other companies like Sun/Netscape/SGI (commercial)



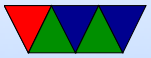
# Web-servers (Recent)

- nginx ("engine-x")
  - Designed to be faster than Apache (Apache has lots of RAM overhead)
  - Solve c10k problem (having 10k concurrent socket connections at once)
  - Now there's the c10M problem
- lighthttpd ("lightly")



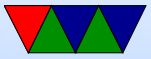
# simple web server

- Listen on port 80
- Accept a TCP connection
- Get name of file requested
- Read file from disk
- Return to client
- Release TCP connection



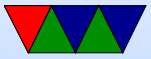
# Aside: How could you make this faster?

- Cache things so not limited by disk
  (also cache in browser so not limited by network)
- Make server multithreaded



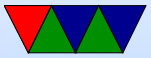
# http

- HyperText Transfer Protocol
  RFC 2068 (1997), RFC 2616 (1999), RFC 7230 (2016)
- Make ASCII request, get a MIME-like response
- Connect with TCP socket
- Plain text request, followed by text headers
- Expects carriage returns in addition to linefeeds
- Influences from e-mail servers

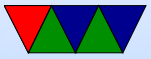


# http Commands

- GET *filename* HTTP/1.1
  get file
- HEAD
  get header (can check timestamp. why? see if cache up to date)
- PUT
  send a file
- POST
  append to a file (send form data)

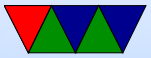


- DELETE
  remove file (not used much)
- TRACE
  debugging
- CONNECT, OPTIONS



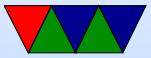
# http three digit status codes

- 1xx – informational – not used much
- 2xx – Success – 200 = page is OK
- 3xx – Redirect – 303 = page moved
- 4xx – Client Error – 403 = forbidden, 404 = not found
- 5xx – Server Error – 500 = internal, 503 = try again



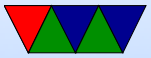
# Example http request from browser

```
GET / HTTP/1.1
Host: 471-pi3:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/109.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

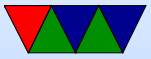


# Selected http request headers (included after GET)

- Host: server you are requesting
  This lets multiple hostnames share one single IP address
- User-Agent (browser info). Can you lie? Can you leak info?
- Accept-*: type of documents can accept, compression, character set
- Authorization: if you need special permissions/login
- Referer [sic] URL that referred to here

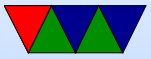


- Cookie: deals with cookies
  Statelessness – how do you remember setting, logins, shopping cart, etc. “cookies”. Expire. Can be misused.
- If-Modified-Since – caching



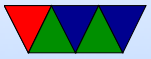
# Example http response

```
HTTP/1.1 200 OK\r\n
Date: Fri, 26 Jan 2024 04:56:25 GMT\r\n
Server: ECE435\r\n
Last-Modified: Sun, 26 Mar 2017 04:31:47 GMT\r\n
Content-Length: 64\r\n
Content-Type: text/html\r\n
\r\n
<html><head><title>Test</title></head>
<body>test</body></html>
```



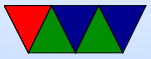
# Selected http response headers

- Content-Encoding,Language,Length,Type
- Last-Modified: helps with caching
- Location: used when redirecting
- Accept-Ranges: partial downloads (downloading a large file, interrupted, can restart where left off)
- Content-Length: length of file being sent
- Content-Type: type of data
- Date: current date
- Server: Name of webserver (is it secure to do this?)

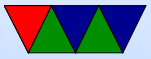


# HW#2 Preview

- Can use existing server code, will connect to it with any web-browser
- Listen on port 8080 (why not 80?)
- Once browser connects, read entire request into buffer (more proper way to dynamically allocate memory?)
- Ignore most of the headers, mostly want to parse the GET request
- Generate headers for response
- Send header and file back to browser over socket

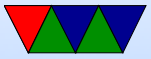


- Handle a few corner cases, like 404 errors



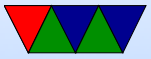
# Homework #2 – Connecting

- If connecting on same machine, can use localhost
  if over network, must use IP address.
- Can find this various ways (`ip addr` on Linux)
- Be aware depending on how your network is set up (firewalls, if behind NAT, etc) you might not be able to connect to your test machine remotely



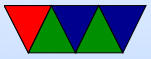
# HW#2 Hints – Reading Request into Buffer

- First be sure you are getting the incoming header. Print it or use strace to verify.
- Some web-browsers might send really big requests, be sure getting it all
  - Use big enough buffer? 4096 bytes? How big?
  - How would a "proper" server do this?
    `malloc()`, `realloc()` if not big enough?
    Overkill for this homework. You can try this, but only if you know what you are doing. Goal of this assignment



is a simple server not perfect server.

- Just use a bigger buffer if necessary and error if you get bigger, don't waste time chasing pointers/segfaults



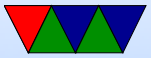
# HW#2 – Parsing the GET Request

- Search for a string and point to location after it?
  - Find a string and point to beginning of it.

```
char *pointer;
pointer=strstr(haystack,needle);
```

  - Look for `"GET "`

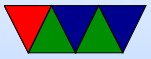
    Actually points to beginning of GET. How to skip ahead?
  - pointer+=4 is one way. (pointer math, ugh)
  - How to get to first space?
  - `strtok(pointer," ");`



Will split the string into chunks, put 0 at end.

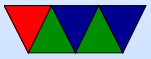
- Also can do this manually;

```
pointer2=pointer;
while(*pointer) {
   if (pointer==' ') {
      *pointer=0;
      break;
   }
   pointer++;
}
printf("%s\n",pointer2);
```



# Homework #2 – Interpreting the Filename

- Be sure to strip off initial /, and if it's just / return index.html
- Do you need to handle spaces in the filename? Thankfully no, URLs can't have spaces

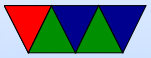


# HW#2 – Generating Response Headers

- Print to stdout to verify what sending, also can use lynx / wget.
- Know how to construct a string on the fly?
  - One way is to have empty string, than use `strcpy()` first bit in. `strcat()` additional strings.
  - Easier might be `sprintf()` If you want formatting you can do things like

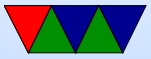
```
sprintf(temp_string,"File size=%d\r\n",filesize);
strcat(out_string,temp_string);
```

  - `snprintf()` might be a bit safer as you can specify



the max length of the string (to avoid overflowing)

- Try not to be too fancy with one gigantic `sprintf()` call as C can evaluate function parameters in arbitrary orders

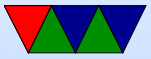


# HW#2 – Calculating Content-length

- How to find size of a file?
- Can read it in, and count. Note: don't use strlen() for this as a binary file might have zeros in it
- Might be better to use `stat()` (`man stat.2`) need .2 (or `man -a`) as there's a command line tool called stat that comes ip first.

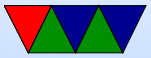
```
#include <sys/stat.h>
struct stat statbuf;

/* use stat() if have filename, fstat() if have file descriptor */
result=fstat(input_fd,&statbuf);
input_size=statbuf.st_size;
```



# HW#2 – Getting Filetype

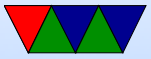
- Easiest way is calculating based on extension
- Take filename, look for . and compare after it
- Can use `strstr()` again, but think of corner cases
  What if multiple dots? What if no dots?



# HW#2 Hints – Sending File Contents

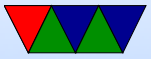
- Reading file into buffer then writing to socket
  - I don't recommend this as you have to dynamically handle different file sizes
  - If you do this, don't use sprintf() with %s to print the contents. Won't work if 0 in file
- Reading/Writing in chunks
  - `open()/read()/write()/close`

```
fd=open(filename,O_RDONLY);
if (fd<0) fprintf(stderr,"Error opening %s\n",filename);
while(1) {
   result=read(fd,buffer,256);
```



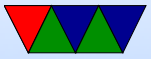
```
        if (result<=0) break;
        write(network_fd,buffer,result);
    }
```

  - `fopen()/fread/fwrite/fclose` (careful! Buffered! And maybe need fdopen() to print to file descriptor).
- Be sure to close afterward.



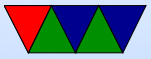
# HW#2 Notes – Knowing Request is Done (part1)

- This probably isn't needed for this assignment, but can be useful if you re-use code for your project
- When reading in data from a socket, you probably want to read in the entirety of a request even though it might be split across multiple reads (so `read()` in a `while(1)` loop)
- You might also want to read all you can and then have your client or server handle the request. However if



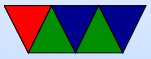
the last `read()` call blocks forever waiting then your program is stuck waiting and can't accomplish anything else

- Is there a way to have interactive programs that are also waiting for socket data?

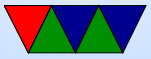


# HW#2 Notes – Knowing Request is Done (part2)

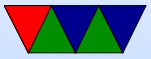
- Can you just assume each read() matches an exact write() from the client?
  - No: TCP is a byte stream, you can't see packet boundaries and they might not correspond to the write() calls on the other side anyway
- Can you infer that there's more data based on the content being sent?
  - Yes, for example if the data read ends in a new-line it



could mean the transaction is done
  - Your protocol can contain info that lets you know how long things are (content-length), or have a signal (like the empty newline in http after headers) that let you know
- Can you have non-blocking read() calls?
  - You can set the fd to be non-blocking
  - The `recv()` call (unlike `read()` has some extra flags that can help. On Linux can pass `MSG_DONTWAIT` which will not-block and just return an error if no data is available

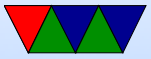


- Note in these cases you have to periodically poll the socket to check for input which might not be optimal
- You can use `poll()` or `select()` to be notified when a fd has data but that's complex
- You can also possibly set up multiple threads with pthreads or similar, with one thread handling the socket I/O



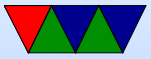
# Homework #2 – Common Issues

- If browser confused, be sure you aren't sending extra zeros. strlen() is your friend
- If browser gets some data but then just spins waiting, be sure your Content-length field is set with the proper size Note it's the size of file you are sending, does not include header size.



# Homework #2 – Debugging

- A powerful tool is using
  `wget -S localhost:8080/test.html`
  which will show you the headers your server is sending and download the file so you can verify the contents.
  Note you might need to install the `wget` tool (easy to do on Linux, maybe more difficult elsewhere)
- The `strace` tool can also be useful as it can show you the bytes being sent by the various syscalls
- If getting segfaults, you might be stuck using `gdb`



# HW#2 Hints – General C annoyances

- When you use a char pointer to point into a string (as when using `strstr()` or `strtok()` remember what you have is a pointer, not a copy of the string you're pointing to. So if the buffer gets freed or re-used your pointer may suddenly point to something different.