Karen Shay West
9 Shannon Marie Way, North Easton, MA 02356

508-844-9776 http://www.linkedin.com/in/karenshaywest KarenWest15@gmail.com

SUMMARY OF RECENT PROJECT OF MY OWN IN C on Linux–

- **parts of a basic web server in C, currently being completed and further expanded**
- **project descriptions taken from a class I did not take but instead implemented the projects, with a section of it borrowed from another class's project descriptions, to help get started with the web server**
- see my Linked In Project Section for details and github links to files:

- **Web Server in C:**
  - **I implemented parts of a basic web server in C, taken from another online class's description, to get me started with the project description that follows.** The description below is taken from a class I did not take, but decided to implement their project to better learn how web server's work, while also interacting it with the world I know better, the physical computing environment, which overlaps with my embedded systems past. I also have plans to learn other ways of doing web servers, in the newer languages for that purpose that people use (such as PHP). That will come from a finance project at a later date, to learn HTML, CSS, PHP, and SQL, with PHP being the new thing there. There is another exercise planned to add to those Javascript, Ajax and Json. For now though, this will be my web server in C, interfaced to a physical computing environment.
  - **The parts I wrote for this web server project I used to get started were in the file listed in the next section called server.c:**
    - **"lookup"**, to return the type of web page requested from the client browers, such as html, css, gif, ico, jpg, js (javascript), php, png
    - **"parse"**, to extract the absolute path and query from the client web browser's web page request line, and return standard web error messages if the request had an error
    - **"load"** to read all bytes from a web page if html, dynamically allocate memory on the heap, and save a pointer to the web page content and length, or if a php web page request, invoke the php interpreter to run the requested php script, and return in content what the php script returned and it's length
    - **"indexes"**, which returns a path/to/a/directory with either the html or php file extension on the end of it, if it actually exists.
  - **The parts that were given in the online class project from which I learned better how to implement a web server in C, with plans to expand it in the following ways to a physical computing interface environment.**
    - The socket communications between this server and the client browser were done by the online course people.
    - The directory structure for this basic web server in C was as follows, and I simply filled in the above functions described in the file called server.c.
    - The directory contained these files for getting started below.
    - .
    - ├── Makefile
    - ├── public
    - │ ├── cat.html -- has an IMG tag whose src attribute is cat.jpg

- │ ├── cat.jpg
- │ ├── favicon.ico
- │ ├── hello.html – has a form that's configured to submit via GET a text field called name to hello.php
- │ ├── hello.php – mostly HTML but inside its body is a bit of PHP code:
    - <?= htmlspecialchars($_GET["name"]) ?>
    - The <?= notation just means "echo the following value here". htmlspecialchars , meanwhile, is just an atrociously named function whose purpose in life is to ensure that special (even dangerous!) characters like < are properly "escaped" as HTML "entities." http://php.net/manual/en/function.htmlspecialchars.php
    - Anyhow, $_GET is a "superglobal" variable inside of which are any HTTP parameters that were passed via GET to hello.php . More specifically, it's an "associative array" (i.e., hash table) with keys and values. Per that HTML form in hello.html , one such key should be name !
- │ └── test
- │ │ └── index.html
- │ │ └── index.php
- └── server.c – implements web server that knows how to serve static content (i.e., files ending in .html , .jpg , et al.) and dynamic content (i.e., files ending in .php).
- **Usage: server [p port] /path/to/root**
  - This program's name is server .
  - To specify a (TCP) port number on which server should listen for HTTP requests, include p as a commandline argument, followed by (presumably) a number. The brackets imply that specifying a port is optional. (If you don't specify, the program will default to port 8080.)
  - The last commandline argument to server should be the path to your server's "root" (the directory from which files will be served).
- **To Test:**
- **Server is started with: ./server public**
    - Listening on port 8080 (should be output)
    - Using "public" as server's root – the directory from which files will be served.
- under the public dir is the test/index.html file – you do only specify directories or files underneath the public directory after the localhost:8080 to request web pages.
- **Tests:** in client browser while this server is running on same machine:
- http://localhost:8080/test/index.html (starts a video of singer)
- http://localhost:8080/cat.jpg (shows photo of a happy cat)
  - you should also see
  - GET /cat.jpg HTTP/1.1
  - in your terminal window, which is the "request line" that your browser sent to the server
  - Below that you should see all of the headers that your browser sent to server followed by
  - HTTP/1.1 200 OK
  - which is the server's response to the browser
- http://localhost:8080/test/index.php (should do a dir listing) – still getting to work

- **Another way to test it:**
- open up Chrome’s developer tools, per the instructions at https://developer.chrome.com/devtools
- Then, once open, click the tools' Network folder, and then, while holding down Shift, reload the page.
- Not only should you see Happy Cat again. You should also see the below in your terminal window.
- GET /cat.jpg HTTP/1.1
- HTTP/1.1 200 OK
- You might also see the below.
- GET /favicon.ico HTTP/1.1
- HTTP/1.1 200 OK
- What's happening is, by convention, a lot of websites have in their root directory a favicon.ico file, which is a tiny icon that’s meant to be displayed a browser’s address bar or folder. If you do see those lines in your terminal window, that just means Chrome is guessing that your server, too, might have favicon.ico file, which it does!
- **A Walkthrough Demo type test:**
- http://localhost:8080/cat.html (shows photo of a happy cat – with a margin around him, unlike when it was just the jpg, due to Chrome's default CSS properties)
- If you look at the **Chrome developer tools' Network folder** (possibly after reloading, if they weren’t still open), you should see that Chrome first requested cat.html followed by cat.jpg , since the latter, recall, was specified as the value of that img element’s src attribute that we saw earlier in cat.html .
- To confirm, take a look at the **developer tools' Elements folder**, wherein you’ll see a prettyprinted version of the HTML in cat.html. You can even change it but only Chrome’s inmemory copy thereof.
- To tinker with the **developer tools' Styles folder**, even though this page doesn’t have any CSS of its own, you can see and change (temporarily) Chrome’s default CSS properties via that folder.
- If you **look at the page’s source code (as via the developer tools' Elements folder)**, you’ll see your name embedded within the HTML! By contrast, files like cat.jpg and cat.html (and even hello.html) are "static" content, since they’re not dynamically generated.
- **To test code via a command line rather than with a browser, this is one technique.** So let’s show you one other technique.
- Open up a second terminal window and position it alongside your first. In the first terminal window, execute
- ~/server public
- from within your own ~/workspace/webserver directory, if the server isn’t already running.
- Then, in the second terminal window, execute the below. (Note the http:// this time instead of https:// .)
- curl i http://localhost:8080/
- If you haven’t used curl before, it’s a commandline program with which you can send HTTP requests (and more) to a server in order to see its responses. The i flag tells curl to include responses' HTTP headers in the output. Odds are, whilst debugging your server,
- you’ll find it more convenient (and revealing!) to see all of that via curl than by poking

around Chrome's developer tools.

- Incidentally, take care not to request cat.jpg (or any binary file) via curl , else you'll see quite a mess!

- **server.c – a tour of what was written by the online class people, and what I wrote:**
- **In server.c, only the lookup(), parse(), load(), and indexes() functions were written by me, as described in the previous section.**
- Atop the file are a bunch of "feature test macro requirements" that allow us to use certain functions that are declared (conditionally) in the header files further below. Defined next are a few constants that specify limits on HTTP requests sizes. We've (arbitrarily) based their values on defaults used by Apache, a popular web server. See http://httpd.apache.org/docs/2.2/mod/core.html,
- Defined next is BYTES , a constant the specifies how many bytes we'll eventually be reading into buffers at a time.
- Next are a bunch of header files, followed by a definition of BYTE , which we've indeed defined as an 8bit char , followed by a bunch of prototypes.
- Finally, just above main are a just a few global variables.
- **main()**
- Atop main is an initialization of what appears to be a global variable called errno . In fact, errno is defined in errno.h and is used by quite a few functions to indicate (via an int), in cases of error, precisely which error has occurred. See man errno for more details.
- Shortly thereafter is a call to getopt , which is a function declared in unistd.h that makes it easier to parse commandline arguments. See man 3 getopt if curious. Notice how we use getopt (and some Boolean expressions) to ensure that server is used properly.
- Next notice the call to start (for which you may have noticed a prototype earlier). More on that later.
- Below that is a declaration of a struct sigaction via which we'll listen for SIGINT (i.e., controlc), calling handler (a function defined by us elsewhere in server.c) if heard.
- And then, after declaring some variables, main enters an infinite while loop.
- Atop that loop, we first free any memory that might have been allocated by a previous iteration of the loop.
- We then check whether we've been "signalled" via controlc to stop the server.
- Thereafter, within an if statment, is a call to connected , which returns true if a client (e.g., a browser or even curl) has connected to the server.
- After that is a call to parse , which parses a browser's HTTP request, storing its "absolute path" and "query" inside of two arrays that are passed into it by reference.
- Next is a bunch of code that decodes that path (decoding any URLencoded characters like %20) and "resolves" the path to a local path, figuring out exactly what file was requested on the server itself.
- Below that, we ascertain whether that path leads to a directory or to a file and handle the request accordingly, ultimately calling list , interpret , or transfer .
- For directories (that don't have an index.php or index.html file inside them), we call list in order to display the directory's contents.)
- For files ending in .php (whose "MIME type" is text/xphp), we call interpret .
- For other (supported) files, we call transfer .
- And that's it for main ! Notice, though, that throughout main are a few uses of continue ,

the effect of which is to jump back to the start of that infinite loop. Just before continue in some cases, too, is a call to error (another function we wrote) with an HTTP status code. Together, those lines allow the server to handle and respond to errors just before returning its attention to new requests.

- **connected()**
- connected is below main .
- memset() function fills the first sizeof(client socket address) bytes of the memory area pointed to by (client socket address) with the constant byte zero.
- accept(): extracts the first connection request on the queue of pending connections for the listening socket (server), sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket (for this client it just connected to).
- **error()**
- error calls reason() to determine the reason for the failure of obtaining the request for the client and places it in a phrase string.
- It **forms a template string** as follows:
- "<html><head><title>%i %s</title></head><body><h1>%i %s</h1></body></html>"
- It then **renders the template** as follows, **into a body string and it's length**:
- char body[(strlen(template) - 2 - ((int) log10(code) + 1) - 2 + strlen(phrase)) * 2 + 1];
- int length = sprintf(body, template, code, phrase, code, phrase);
- It then **adds the headers and responds with the error code, header, body, and length to the client:**
- char* headers = "Content-Type: text/html\r\n”
- **freedir()**
- This function exists simply to facilitate freeing memory that’s allocated by a function called scandir that we call in list .
- **handler()**
- This function (called whenever a user hits controlc) essentially tells main to call stop by setting signaled , a global variable, to true .
- **htmlspecialchars()**
- This function, named identically to that PHP function we saw earlier, escapes characters (e.g., < as <) that might otherwise "break" an HTML page. We call it from list , lest some file or directory we’re listing have a "dangerous" character in its name.
- **indexes() - I wrote this function.**
- returns a path/to/a/directory with either the html or php file extension on the end of it, if it actually exists.
- the function, given a /path/to/a/directory , returns
  - /path/to/a/directory/index.php if index.php actually exists therein, or /path/to/a/directory/index.html if index.html actually exists therein, or NULL . In the first of those cases, this function should dynamically allocate memory on the heap for the returned string.
- **interpret()**
- This function enables the server to interpret PHP files. It’s a bit cryptic at first glance, but in a nutshell, all we’re doing, upon receiving a request for, say, hello.php , is executing a line like QUERY_STRING="name=Alice" REDIRECT_STATUS=200 SCRIPT_FILENAME=/path/to/public the effect of which is to pass the contents of hello.php to PHP’s interpreter (i.e., phpcgi), with any HTTP parameters supplied via an "environment variable" called QUERY_STRING . Via load (a function we wrote), we

then read the interpreter’s output into memory (via load). And then we respond to the browser with (dynamically generated) output like:
    - HTTP/1.1 200 OK
    - XPoweredBy: PHP/5.5.91ubuntu4.12
    - Contenttype: text/html
    - <!DOCTYPE html>
    - <html>
    - <head>
    - <title>hello</title>
    - </head>
    - <body>
    - hello, Alice
    - </body>
    - </html>
- Even though the PHP code in hello.php is prettyprinted, it’s output isn’t quite as pretty.
- **popen()** -- That function opens a "pipe" to a process (phpcgi in our case), which provides us with a FILE pointer via which we can read that process’s standard output (as though it were an actual file). Notice how this function calls load , though, in order to read the PHP interpreter’s output into memory.
- **list()**--function that generates a directory listing. Notice how much code it takes to generate HTML using C, thanks to requisite memory management. **(They pointed out here that with PHP this part is easier).**
- **load()—this I wrote**--to read all bytes from a web page if html, dynamically allocate memory on the heap, and save a pointer to the web page content and length, or if a php web page request, invoke the php interpreter to run the requested php script, and return in content what the php script returned and it's length
    - 1. reads all available bytes from file
    - 2. stores those bytes contiguously in dynamically allocated memory on the heap
    - 3. stores the address of the first of those bytes in *content
    - 4. stores the number of bytes in *length
    - Note that content is a "pointer to a pointer" (i.e., BYTE**), which means that you can effectively "return" a BYTE* to whichever function calls load by dereferencing content and storing the address of a BYTE at *content . Meanwhile, length is a pointer (i.e., size_t*), which you can also dereference in order to "return" a size_t to whichever function calls load by dereferencing length and storing a number at *length .
- **lookup()—this I wrote**--
    - returns
    - text/css for any file whose path ends in .css (or any capitalization thereof),
    - text/html for any file whose path ends in .html (or any capitalization thereof),
    - image/gif for any file whose path ends in .gif (or any capitalization thereof),
    - image/xicon for any file whose path ends in .ico (or any capitalization thereof),
    - image/jpeg (not image/jpg) for any file whose path ends in .jpg (or any capitalization thereof),
    - text/javascript for any file whose path ends in .js (or any capitalization thereof),
    - text/xphp for any file whose path ends in .php (or any capitalization thereof), or

    - image/png for any file whose path ends in .png (or any capitalization thereof), or
    - NULL otherwise.
- **parse()—this I wrote**--to extract the absolute path and query from the client web browser's web page request line, and return standard web error messages if the request had an error.
    - the function parses (i.e., iterates over) line , extracting its absolutepath and query and storing them at abs_path and query , respectively.
    - **abs_path:**
    - Per 3.1.1 of http://tools.ietf.org/html/rfc7230 (http://tools.ietf.org/html/rfc7230), a requestline is defined as
        - method SP requesttarget SP HTTPversion CRLF
    - wherein SP represents a single space ( ) and CRLF represents \r\n . None of method , requesttarget , and HTTPversion meanwhile, may contain SP .
    - Per 5.3 of the same RFC, requesttarget , meanwhile, can take several forms, the only one of which your server needs to support is
        - absolutepath ["?" query]
- whereby absolutepath (which will not contain ?) must start with / and might optionally be followed by a ? followed by a query ,which may not contain " .
- Ensure that requestline (which is passed into parse as line) is consistent with these rules. If it is not, respond to the browser with 400 Bad Request and return false .
- Even if requestline is consistent with these rules, if method is not GET , respond to the browser with 405 Method Not Allowed and return false
- if requesttarget does not begin with / , respond to the browser with 501 Not Implemented and return false
- if requesttarget contains a " , respond to the browser with 400 Bad Request and return false
- if HTTPversion is not HTTP/1.1 , respond to the browser with 505 HTTP Version Not Supported and return false
- If all is well, store absolutepath at the address in abs_path (which was also passed into parse as an argument). You may assume that the memory to which abs_path points will be at least of length LimitRequestLine + 1.
- **query:**
    - Store at the address in query the query substring from requesttarget . If that substring is absent (even if a ? is present), then query should be "" , thereby consuming one byte, whereby query[0] is '\0' . You may assume that the memory to which query points will be at least of length LimitRequestLine + 1.
    - For instance, if requesttarget is /hello.php or /hello.php? , then query should have a value of "" . And if requesttarget is /hello.php?q=Alice , then query should have a value of q=Alice .
- **reason()**--This function simply maps HTTP "status codes" (e.g., 200) to "reason phrases" (e.g., OK).
- **redirect()**--this function redirects a client to another location (i.e., URL) by sending a status code of 301 plus a Location header.
- **request()**--When the server receives a request from a client, the server doesn’t know in advance how many characters the request will comprise. And so **this function iteratively reads bytes from the client, one buffer’s worth at a time, calling realloc as needed to store the entire message (i.e., request). Notice this function’s use of**

pointers, dynamic memory allocation, pointer arithmetic, and more. Ultimately, it keeps reading bytes from the client until it encounters \r\n\r\n (aka CRLF CRLF), which, according to HTTP's spec, marks the end of a request's headers. Note that **read() is quite like fread except that it reads from a "file descriptor" (i.e., an int) instead of from a FILE pointer (i.e., FILE*).**

- **respond()**--It's this function that actually sends to a client an HTTP response, given a status code, heads, a body, and that body's length. For instance, it's this function that sends a response like the below.
  - HTTP/1.1 200 OK
  - XPoweredBy: PHP/5.5.91ubuntu4.12
  - Contenttype: text/html
  - <!DOCTYPE html>
  - <html>
  - <head>
  - <title>hello</title>
  - </head>
  - <body>
  - hello, Alice
  - </body>
  - </html>
- Know that dprintf is quite like printf (or, really, fprintf) except that the former, like read , writes to a "file descriptor" instead of to a FILE* .
- **start()**--start is the function that configures the server to listen for connections on a particular TCP port!
- **stop()**--stop does the opposite, freeing all memory and ultimately compelling the server to exit, without even returning control to main .
- **transfer()**--This function's purpose in life is to transfer a file from the server to a client. Whereas interpret handles dynamic content (generated by PHP scripts), transfer handles static content (e.g., JPEGs). Notice how this function calls load in order to read some file from disk.
- **urldecode()**--This function, also named after a PHP function, URLdecodes a string, converting special characters like %20 back to their original values.

- **Current Status of Web Server in C: Works for HTML static page requests, but not for a PHP request to return a directory, and the simple Perl script mentioned below has not yet been tried.**
- **Purpose – the plans for expanding the web server implemented in C described above:**
  - There are two parts to this assignment: a mandatory part and an optional bonus part. For the mandatory part, you will be required to write a C/C++ program that implements a web server. Your web server will conform to HTTP/1.x for the purposes of client requests, and you will need to process client HTTP GET requests for web pages hosted on the server machine. You will need to use sockets to implement the communication between a client on one machine and the server on either the same or a remote machine.
  - For the bonus part, described in more detail below, you will be required to add functionality to your web server to support interaction with a physical device, a case where computers interact with the physical world through a collection of sensors and actuators. This forms a physical computing environment.
  - **Web server Description:**

- The Basic HTTP Protocol: The basic structure of interaction between a web client and web server is as follows:
  - Client sends request (from a suitable browser)
    - GET filename HTTP/version
    - optional arguments
    - a blank line
  - Server sends reply
    - HTTP/version status-code status-message
    - additional information
    - a blank line
    - content
- You will need to ensure the information sent back from the server is formatted as described above.
- The additional information send back in a server reply is of the form:
  - Content-type:text/plain
    - text/html
    - image/gif
    - image/jpeg
    - xxx/yyywhere xxx/yyy can be:

- **The Server**
  - You can assume the server forks a child process for each incoming request.
  - Additionally, you can assume each request involves establishing a new connection with the server, rather than maintaining any notion of a session for multiple requests.
  - You should write your web server in a file called webserv.c. For execution, you should bind the server to a specific port.
  - To run your web server you should type:
    - $./webserv port-number
      - where port-number should be a value in the range 5000-65536 such as 8080.
    - The server will be required to handle a series of requests, such as listing the contents of a directory on the server machine, retrieving a file for viewing on the client, and running cgi scripts. In particular, you will be required to handle static and dynamic content requests. In the former case, the content is simply retrieved from a pre-existing file on the server. For dynamic content, you will be expected to run a program on the server, to process data and generate an HTML compliant file for sending back to the client.
    - Your web server should handle HTML status codes 200 (successful request), 404 (Not Found), and 501(Not Implemented). For status code 501, the server does not recognize the request method. Further information about status codes can be found here.
- **The Client**
  - The client can be any web browser of your choice. Your requests from the client should be in the form:
    - http://ip.address.of.server:port-number/request
    - ip.address.of.server -- this is the IP address of the server machine
    - port-number -- this is the numeric port on which the server listens. Together with the IP address, this identifies an end-point of communication (or socket) to

which the client connects.
    - request -- this is either a subdirectory on the server that you wish to list, the name of a html file, or a cgi file. In the latter case, this will be a reference to a script on the server that you must execute to perform some command. The content of a cgi script, such as test.cgi, must be set executable on the server and must refer to a shell or Perl script such as the following:
        - test.cgi (set executable using chmod 755 test.cgi):
        - #!/bin/sh
        - # test.cgi -- a simple test
        - printf "Content-type: text/plain\n\nThis is a test!\n"
        - To execute a Perl script, you can issue a request such as:
        - http://ip.address.of.server:port-number/request.cgi
        - where request.cgi is an executable Perl script on the server having contents such as:
        - #!/usr/bin/perl
        - # perl-test.cgi -- a simple Perl script test
        - print "Content-type: text/plain\n\nThis is a Perl test!\n";
- **Basic Test Cases**
    - You should be able to support all the following test cases:
        - A request for a directory listing
        - A request for a valid (and non-existing) html file. NOTE: A non-existent request corresponds to an
        - HTTP error status code of 404.
        - A request for a static image (in either gif or jpeg format, having a file ending of .gif, .jpg or .jpeg)
        - A request for a cgi script that requires execution of a basic shell command, executed using sh
        - A request for a perl script in a cgi file to process raw data and format it into an html file.
        - A request for a dynamically-created image using gnuplot on the server. Information about gnuplot can be found at: http://www.gnuplot.info/
        - For the latter case, above, you can assume the request specifies a cgi file describing a perl script. The perl script will process data as described in the next subsection.
        - **Dynamic Content using Gnuplot:** In this case, you are required to execute a program on the server called my-histogram, as follows:
            - $my-histogram file pattern1 pattern2 ... patternN
            - file specifies the name of a file you wish to search for all occurrences of a given regular-expression pattern or
            - string sequence. For example:
            - $my-histogram file "and" "but" "so" "he.*lo"
            - will tally all occurrences of the words "and", "but" and "so" in file, along with all strings that match the pattern "he.*lo" such as "hello" etc. You can assume all regular-expression patterns that are acceptable to grep -e are valid. You can assume the number of pattern arguments is limited to 5.
            - Once my-histogram has tallied all occurrences of the matching strings for

each pattern, you should plot the results as a histogram using gnuplot. You can pipe the output of my-histogram to gnuplot using a Perl command as follows:

- open (GNUPLOT, "|gnuplot"); # Notice the vertical bar for a pipe
- After which piping commands to gnuplot is analogous to writing to a file.
- my-histogram can be written in any language you choose, including C/C++, Perl, etc. You are also free to use shell commands such as "grep -e" if you wish, or the built-in Perl regular expression features. You should format the output of your gnuplot histogram, to show "frequency" up the y-axis and the labelled patterns on the x-axis, so there is one frequency bar per pattern.
- Once you have done this, you should command gnuplot to output the histogram to a file that records the information in gif or jpeg format. After this, you need your cgi script to send your gnuplot gif or jpeg image back to the client for viewing.

- **Pretty Printed Output**
  - Just as this webpage has been formatted using html, you should write an executable on the server that is invoked as part of your CGI script to pretty-print your histogram.
  - Specifically, you are to embed the histogram image file in a HTML page that has a 16pt RED font title and white background*. The title should read: "My Webserver".
  - The title should be centered on the page. Below it, should be a blank line (spacing of which is your choosing) followed by the histogram, which is also centered.
  - *If you so wish, you can experiment with the generated HTML content, producing image backgrounds and additional details. The base case should be formatted as described, however.

- **Advanced Features:** If you manage to complete the above parts of the assignment, you should consider these more advanced features. Extra credit will be given if you tackle one or more of the following:
  - **A multi-threaded web server.** Instead of using fork() calls for each client request, you should instead spawn a thread using your own thread creation routines, based on the sigaltstack() method. Specifically, you should *not* use make/get/set/swapcontext() functions or any pre-existing thread packages (e.g., pthreads). Instead, you should write your own thread management code.
  - **A web cache.** In this part of the assignment, you should develop a method to cache files in RAM for subsequent requests. The RAM cache should be a pool of memory of some defined size. You should set this to be a tunable parameter from 4KB-2MB. Upon initialization, the cache is empty, but gets filled for each file request until it is full. At that point you can adopt a simple replacement strategy of your choice (e.g., first-in-first-out, random, or least-recently used). You must specify your replacement method in a README file. To make the cache beneficial, you should support requests that are both in the server's filesystem and also on a remote host. Client requests should provide an optional argument to indicate the remote host location for files that are not stored in the server's local filesystem. To test this feature, you should have the server act like a client for a remote host machine, thereby retrieving the necessary file(s) for placement in the web cache. In turn, these files will be

relayed back to the original client.

- **Server configuration.** For testing purposes, you should provide a way to disable the above advanced features, so that the web server falls back to operating in normal mode (without web caching and threads). This means you should produce only one version of your code. To enable/disable features of your server, you can either use a configuration script, pass in command-line arguments, or (worst-case) use defined constants within your code.

- **Physical Computing (NOTE: I may use what I learned in my embedded systems online class projects for this section, not sure yet, and that is described separately).**
  - To tackle this part of the assignment requires you to have access to an Arduino Uno or similar Arduino-compatible device. These can be purchased for about $5.99 (roughly the price of two coffees) from places such as Microcenter.You can also buy a good quality starter kit from Amazon, which is a little more expensive but includes everything to get going with some basic building projects. If you have access to a Raspberry Pi or other similar single board computer, you can use that too.
  - **NOTE:** 2 years ago, I used a Launch board from TI, Cortex M, with an ARM based microcontroller, and I may use this instead, not sure yet.
    - The idea: as a way to introduce you to physical computing, you are allowed to be creative for this optional part of the assignment. One idea would be to have an Arduino board connected via a serial interface to a server PC running the web server you have created. A client would initiate requests to your web server, which then issued serial commands to your Arduino board. Your Arduino board would run some code (of your choosing) to perform a control operation. For example, a simple control operation would be to turn on/off some LEDs, or to control the speed/angle of a stepper or servo motor. All these operations would be triggered by client requests via a web browser, through which commands are sent via your web server.
    - Another option would be to simply use a web interface to submit requests to upload Arduino "sketches" to your target device. Here, a sketch is a simple program that runs out of the ROM space on the Arduino itself. The Arduino IDE is open source and can be downloaded for most OSes, and there is a reference guide to explain the simple programming language available online.

- **Why tackle the physical computing part of the assignment?**
  - You also get to work with physical devices which is a lot of fun. It makes computing feel "real" rather than abstract. Writing programs that make computers work is one thing, but having those computers control devices is the basis for how robots, 3D printers, UAVs and all sorts of physical devices operate. Where possible, we will try to provide resources that you can borrow, but you are encouraged to invest a small fee in purchasing equipment that will potentially prove extremely valuable in the long run. The costs are, for the most part, less than the price of a text book!
  - This part of the assignment also allows you to be creative. You can come up with any physical computing problem of your own choosing, as long as the control is initiated through the web server.
  - **DEMO (if I ever finish, will post a video):**
    - Basic test cases for static content (including directory listings, correct error reporting, static images and cgi scripts). This should include correct usage of sockets to enable client requests and correct handling of HTML client request formats

- Dynamic content including regular expression handling and histogram/gnuplot generation
- Pretty printed output, including correct embedding of images in HTML-formatted files
- Advanced features -- thread management
- Advanced features -- web cache (including proxy support for remote requests and cache replacement)
- Server configuration
- Program style and documentation (in a README file)