My approach to project #1

The assignment handout describes the functional requirements and grading rubric for the web server project. You are free to approach the project however you see fit. I believe that you will find using good coding practices, constant design patterns and appropriate data structures makes completing the assignment much easier. However, you are free to take any approach you like to this project (with one caveat – the work must be 100% your own, no generative ai and no borrowing code from classmates).

I'm providing this document, which describes my own approach to the project, for those of you who would like a little more guidance. In this document I outline the functions I used to complete the project along with some hints and warnings about common pitfalls I've seen other students make over the years.

I'm a big believer in iterative design – writing complex programs one step at a time, testing after each step. I recommend completing the project in two parts. First tackle the socket library and reading/writing over the network. One you are sure you can communicate over a TCP connection then move on and implement the HTTP protocol.

Step 1: Write an Echo Server

The first thing I did was to write a program that accept an incoming network connection, reads in a line of text and echoes that line back out to the user. You can test the functionality of this program using the command line program netcat (nc) found on any *nix based system.

For this step you will need just two functions: main() and processConnection()

```
// **************************************************************************
// * main – does main() suff ☺
// **************************************************************************
int main(int argc, char* argv[])
   1. Process the command line arguments.
   2. Create and fill in the socket address structure.

   3. // Bind socket address to a socket
   4. Loop until exitLoop == true.
         a. Pick port number and fill out the struct sockaddr_in structure.
            You need to fill in three fields for that structure:
              i.  .sin_family should contain the address family we are using.
                  We are using IPv4 so specify PF_INET;
             ii.  .s_addr should contain the address you want to listen on.
                  This is mostly applicable on systems with multiple
                  interfaces. Since we don't care which one to use, we use
                  INADDR_ANY
            iii.  .sin_port should contain the port we want to listen on. Any
                  port number larger than 1028 is fine.

         b. Call  bind() system call.
         c. Test to see if it succeeded
         d. If success, set exitLoop to true.
         e. If failure
              i.  If error indicates port is in use, pick another port.
             ii.  Otherwise, there is something wrong with your program, you
                  should exit and debug.
   5. Configure the socket to accept connections with the listen() system
      call.
   6. while true:
         a. Wait for a connection using the accept() system call.
         b. Deal with the connection by calling processConnection().
         c. Close the connection.
```

```
// ***************************************************************************
// * processConnect()
// ***************************************************************************
void processConnection(int socketFD) will:
   1. Loop until the word "CLOSE" is sent over the network
         a. Initialize std container (std::string, std::array etc) that will
            hold the whole line.
         b. Initialize a 10 byte buffer
         c. Loop until the line terminator is found.
               i.  Read up to 10 bytes from the network with the read() system
                   call.
              ii.  Append the bytes received to the end of the container (may
                   be less than 10).
         d. Send the whole contents of the container back to the client.
            Don't forget to convert it back to raw binary data.
         e. Check for the word "CLOSE"
   2. return
```

Testing

To test your program, open two terminals.

1. In the first, run your program in one and make note of the port number your program is using.
2. In the second terminal, run the program netcat (nc), making sure to use the -c flag so it sends <CR><LF> at the end of each line. Any line you type should be sent back to you. If you send a line with the word CLOSE, the connect connection should close. You usually must hit return one more time before netcat notices the connection is gone.

```
$> nc -c localhost <PORTNUMER>
```

A few common problems to watch out for:

- When binding – don't forget to pick a new port number if the number you pick first is in use. I use the complex formula "port = port + 1" each time through the loop ☺
- Don't forget to deal with network vs host byte ordering when printing the port number to the terminal.
- Most network protocols, including HTTP do not use null-terminated strings. You cannot use the c-string or std::string library on data you read from or send to the network. Treat everything as raw data. If you want to use the string library to help with parsing, you must convert the data first.
- The transport layer is not line oriented. Each time you call read you will get some number of bytes (assuming there is data to be read). What you read may not be a full line, or it may be several lines. The "lines" in HTTP are terminated with <CR><LF>, which look to the network like any other data.
    - When I'm coding, I make all my buffers really small (10 bytes) so I'm forced to remember that I have to reassemble things. It's less efficient (you wouldn't want to do that in the real world) but it causes me to be sure I'm reassembling things correctly.

Step 2: Add HTTP GET functionality (CSCI 471)

You should not need to change main() much, if at all. I created five new functions down into six parts, each with its own function.

1. **readRequest()** : Read the request and return a status code and a file name if we can find one.
2. **sendLine** (): Takes a socketID and arbitrary std::string, converts it to a char array, adds the line terminator and sends it to the client.
3. **send404()** : Uses sendLine() to send back the 404 error code and message.
4. **send400** : Uses the sendLine() function to send back the 400 error code and message.
5. **send200**: Uses the sendLine() function to send the 200 AND sends the contents of the file (if the file is not found, call send404()).

The behavior of each of the functions is described below.

```
// ************************************************************************
// * processConnection.
// ************************************************************************
void processConnection(int sockfd) {
   1. Use readRequest(sockfd, &filename)to get the HTTP request
         o  if returnCode is 400 use send400() and exit processConnection()
         o  if returnCode == 404 use send404() and exit processConnection()
         o  if returnCode is 200 use send200()
   2. Exit the function and return 0, so the main loop will close the
      connection and wait for the next request.
```

```
// *************************************************************************
// * Read the header. Return the appropriate status code and filename
// *************************************************************************
int readRequest(sockeFD, std::string &filename)
   1. Set the default return code to 400
   2. Read everything up to and including the end of the header.
   3. Look at the first line of the header to see if it contains a valid GET
         a. If there is a valid GET, find the filename.
         b. If there is a filename, make sure it is a valid filename
            according to the specs of the assignment.
              i.  If the filename is valid set the return code to 200.
             ii.  If the filename is invalid set the return code to 404.
```

```
// *************************************************************************
// * Send one line (including the line terminator <LF><CR>)
// *************************************************************************
sendLine(socketFD, std::string &stringToSend)
   1. Convert the std::string to an array that is 2 bytes longer than the
      string.
   2. Replace the last two bytes of the array with the <CR> and <LF>
   3. Use write to send that array.
```

```
// ***************************************************************************
// * Send a 404
// ***************************************************************************
send404(socketFD)
   1. Using the sendLine() function you wrote, send the following:
   2. Send a properly formatted HTTP response with the error code 404
   3. Send the string, "content-type: text/html" to indicate we are sending a
      message
   4. Send a blank line to terminate the header
   5. Send a friendly message that indicates what the problem is (file not
      found or something like that)
   6. Send a blank line to indicate the end of the message body.
```

```
// ***************************************************************************
// * Send a 400
// ***************************************************************************
send400(socketFD)
   1. Using the sendLine() function you wrote, send the following:
   2. Send a properly formatted HTTP response with the error code 400
   3. Send a blank line
```

```
// ***************************************************************************
// * Send a 200
// ***************************************************************************
sendFile(socketFD, filename)
   1. Use the stat() function call to find the size of the file.
   2. If stat fails you don't have read permission or the file does not
      exist.
         a. Send a 404 by calling send404()
         b. exit the send200() function.
   3. Using the sendLine() function you wrote send the header:
   4. Send a properly formatted HTTP response with the code 200
   5. Send the content type depending on the type of file (text/html or
      image/jpeg)
   6. Send the content-length
         a. Note - if the content length and/or file type are not sent
            correctly, your browser will not display the file correctly.
   7. Send the file itself.
         a. Open the file.
         b. Allocate 10 bytes of memory with malloc() or new[]
         c. While #of-bytes-sent != size-of-file
               i.  Clear out the memory with bzero or something similar.
              ii.  read() up to 10 bytes from the file into your memory buffer
             iii.  write() the number of bytes you read
   8. when you are done you can just return.  Since you set the content-
      length you don't send the line terminator at the end of the file.
```

Testing

The repository includes four files you can test with (although any HTLM or JPG should work. You can use a web browser for testing – I think it is kind of fun to see my web server working in the real world. However, depending on your browsers settings you might run into trouble with encryption. The default for most web browsers these days is to require encryption (HTTPS). Some will let you use the plaintext HTTP protocol, but it can take some work.

If you have trouble with TLS and your web browser, you can use the command line browser 'curl' to test. Open two terminal windows:

1. In the first window
   a. Run your web server making note of the port number it is using. Remember that the files your web server will read must be in the same directory as the executable.
2. In the second terminal window:
   a. Set the working directory to something other than the directory that contains your program and the sample files. If you don't do this you will overwrite the files you are reading.
   b. Run the program curl with the URL you want to fetch. For example:

   ```
   curl --output <filename> --verbose http://localhost:<portnumber>/file1.html
   ```

3. Use the program md5sum to compare the file you received with the original file.

   ```
   md5sum output <path-to-your-program>/file1.hml
   ```

You should also use Wireshark to look at the data your program is sending/receiving. I find this helpful when I can't figure out what is going on because it shows you what is really happening, not what you think is happening, on the network itself.