Jordan Mai
1645861

1. **Project Description:**
    a. Httpserver is a client-server program that is supposed to receive and send bytes that were delivered from a client server. It connects to a port with the client and once the client sends something, the server's job is to perform the task if possible and send back a response code. The program should send tasks such as sending over files contents and possibly writing from a file. A response code will also be sent to the client to send a message. The program will end once tasks are completed or if manually closed by the user.

2. **Program Logic - PSEUDOCODE**
    a. Handle_connection
        i. While (able to receive bytes from client curl using recv()):
            1. Receive the request line and store in a buffer
            2. Parse each word separated by space. To obtain the request, body of file, version, content length and content. Then store each in a separate buffer.
            3. IF(request is "GET"):
                a. Perform error checks to see if request is correct and able to open file for reading
                b. Open body file for reading
                c. Read content and store in a separate buffer
                d. Create a response header and send required code to the client.
            4. IF(request is "PUT"):
                a. Perform Error checks
                b. Open body file for reading if possible, else create new file
                c. Read in the content to a buffer
                d. Created a response header with content of file at the end
                e. Send to client to be written into their buffer
            5. IF(request is "HEAD"):
                a. Repeat "GET" but only send header and not body for response
        ii. Close(connection)

3. **Data Structures**
    a. Array- An array is used to store some byte amount to be used. We will use an array buffer to store bytes taken in from the client and also use arrays to write information into. The array needs to contain all the information and is also used to build a string of response headers.

b. **Files -** A file is used to put bytes of content into to be kept outside of a program or in this case to be sent over to users. For this program, a file is used to read and write bytes from the client server.

4. **Functions**
    a. **Int main(int argc, int argv[]):**
        i. This function opens a socket and attempts to connect to the same port as a client server. This function checks for any errors when reading from argv[] and makes sure that all arguments are correct. It will persist for a while loop to keep waiting and receiving connection until closing. The error to check for is the correct amount of argument inputs and if connection is possible.
    b. **Strtouint16(char numer[]):**
        i. Converts a string to a 16 bit unsigned integer
        ii. Return 0 if string is out of range
    c. **create _listen_socket(unit16_t port):**
        i. Creates a socket for listening to the port
        ii. Closes and creates an error message on failure
    d. **Handle_connection(int connfd):**
        i. This function receives every byte that is sent from the client server. It will continue to keep receiving. This function will handle all necessary requests (“GET”, “PUT”, “HEAD”) only . This function will parse through the entire buffer and take out necessary elements to be put into different buffers. The function continues with a while loop until everything is received from the client side. This program also checks for errors such as “400 Bad Request” if the request that was sent doesn’t match the appropriate components, or “501 Not Implemented if the request is properly formatted but request is valid but not necessary for this assignment. If everything works, this function will send separate helper functions for each different request. Generate a “500 Internal Server Error” if an error on the program.

    e. **send_get(int connfd,char*body,char*version):**
        i. This function will be responsible for handling “GET” requests, in which we will read from a file and write and send bytes to the client. Firstly, the function will parse the first line and then attempt to open the body of the first line to read(). If a file exists, the function will read every single byte from the file and generate a “200 OK” response to be sent. If the file doesn’t exist, we will generate a “404 Not Found” request. If we don’t have permission to access a file, we will generate a “403 Forbidden. Once we build the response and send() to the client, we will return back to main().

f. **send_put(int connfd, char*body, char*version, char* content_num):**
    i. This function will be responsible for handling “PUT” requests, in which we will open a file for writing and write all the bytes to this file that was sent from the client. Firstly we parse the request line and attempt to open the body for writing. If a file doesn’t exist, we will create one and write all the bytes that were sent to the newly created file. We will then generate a “201 Created”. If a file already exists, we will override everything from the existing file and send “200 OK”. If there is no permission on the file we will generate a “403 Error”. Once we created our response header and send() to client, we will go back to main()
g. **send_head(int connfd, char*version, char*body):**
    i. Same as send_get, except when we build our response header, we do not include a body.

5. **Question**

    a. **What happens in your implementation if, during a PUT with a `Content-Length`, the connection was closed, ending the communication early? This extra concern was not present in your implementation of `shoulders`. Why not? Hint: this is an example of complexity being added by an extension of requirements (in this case, data transfer over a network).**

        i. **If during a PUT and the connection closes after Content-Length, there should be an error on the client’s server showing that a connection has been closed and the data could not be processed. The server won’t be able to process anything because the correct request was not given, and should not process anything because it doesn’t want to risk the client sending the same request again and everything will be doubled. The server should not do anything and wait for connection to reconnect. This extra concern was not presented in shoulders because we didn’t have to wait for everything to come at once. For shoulders, we would just read/write in everything that is received and not have to deal with any delays or bytes not sending. Since data is transferred over a network, it needs to be perfect when sending over bytes are else the server and client won’t know if everything was sent over properly, which will throw an error to be safe.**

    b. **How does your design distinguish between the data required to coordinate/control the file transfer (GET or PUT files), and the contents of the file itself?**

        i. **My design will first distinguish if the request is “GET” or “PUT”, and will send it to the correct helper function. Then based on what request it is, it will parse and take in the specific requirements to be placed in buffers. If it is a “GET” request, my program will check and**

open the required file for reading only. It will specifically only perform a read request and then send the information read to the client. For a “PUT” request, the program will only open or create a file for write only, and send all bytes received into a buffer to be processed. Based on what the client has sent over, there will be specific error checks for each “GET” and “PUT”. File transferring is the same for both, but for “GET” we will send over read data while for “PUT” , we will only receive the data.