Design Document: httpserver.cpp

Cristian C. Castillo

Goal

The goal is to make a single threaded HTTP/1.1 server. This server needs to respond to GET(read) and PUT(write) requests to “files” with the file’s name length being 10 and consist of only ASCII characters. The server will store its files in the directory of the current server location, where they shall persist (live within the HTTP Server directory) and be present upon invocation. A design document is expected, along with a concise README.md file in the Git directory. Your code must build httpserver.c/cpp using the make command produced by a Makefile. The server must be tolerant of varying client requests, respond adequately to a errorones request, and appropriately generate the needed content upon a successful HTTP request.

Assumptions

My presumed assumption is arguments shall be read from from the client socket when the server side gets the HTTP header. Standard input (0)/outputs(1) shall use special system calls to read/write in lieu of the input/output through the iostream’s norms (e.g. cout in C++); imitating the library in its entirety. From the previous assignment, we shall further advance our knowledge and technical abilities by successfully integrating them with sockets, providing a functional client to http server bilateral relationship that will transmit messages via text files. It is most presumed that this will be a daunting challenge that involves extensive research, one which requires applying the theoretical teachings by integrating systems and designs of complex systems to mitigate complexity through modularization, abstraction, naming, hierarchy, and finally layering. By completing this project one will have acquired ample knowledge regarding socket programming/concepts, how HTTP requests function, and acquired the minimal ability to solve complex systems.

Description of data structures

The *argv[] is declared within the int main() parameters and is used to help access the files and keep track of the number of inputs given by the user in this program. *argv[] is an array containing the names of all the inputs made through the terminal when executing the ./httpserver program.

The code must use fixed-size buffers, and may allocate no more than 4KiB of Memory for each HTTP headers both sent and received (either via malloc() or as a direct variable declaration). The data that follows it (for a PUT) may be much longer. Similarly, response headers will be less than 4 KiB, but the data that follows may be longer.

`struct sockaddr_storage :` This structure is used to define a socket address which is used for the following functions: bind(), connect(), getpeername(), getsockname(), etc… On line 63 of the HTTP Server program, this storage structure defined the address of the socket in reference to the original 1st socket that supports protocol-specifics. Upon a sockaddr_storage cast, this structure maps onto the sa_family, but when as a protocol-specific address...it shall map onto a structure that is of type sa_family_t and identifies the protocol's address family. This enabled us to create processes that were bound to clientSocket.

`struct addrinfo :` This structure is used by getaddrinfo(), an essential component to setting up our connection configurations. The structure contains the following features:

- ai_flags
- ai_family
- ai_socktype
- ai_protocol
- ai_addrlen
- ai_addr

All were used to set up, excluding ai_flags. The ai_family set up the AF_INET, the socket was of type SOCK_STREAM which maps to ai_socktype (vs other methods, for e.g. DGRAM connectionless implementation). The ai_protocol which helped specify our protocol for the returned socket addresses. The ai_protocol was set to IPPROTO_TCP (0), which in case must be set as is to conform with SOCK_STREAM, and zero meaning TCP implementation. Lastly, a pointer to the socket address is placed in the ai_addr field, along with the length of the socket address to ensure accept() functionality is invoked accordingly.

`socklen_t :` An unsigned opaque integral type of length of at least 32 bits, where values of 2^32 -1.

`getaddrinfo :` The get address feature was a wonderful treat that was integrated to the HTTP Server program on line 44. The used parameters are self explanatory: getaddrinfo(host_address, portNumber,&hints,&res); The host_address is e.g localhost/127.0.0.1, the user also has the ability to establish a custom port number, in this case holding the second parameter. The hints and res we will go into depth soon, but prior to this function, gethostbyname() was used to do DNS lookups (associating various information to domain names) by loading the data into a struct sockaddr_in. However, in this program we used this function to mitigate lines of code and to keep up with industry standards. In regards to the hints, this points to a struct addrinfo which helped us assign us to the American Familly_INET (which is an address family that helped designate the type of addresses that the socket may communicate with, in this case our IP addresses). Also tying into our SOCK_STREAM features via hints, which is known as TCP (Transmission Control Protocol which establishes a connection between the client and server prior to the data being sent). Finally the res, is the carrier one might say, presenting itself as a pointer to a linked list, which when you run valgrind in its entirety will become apparent as it requires freeaddrinfo(res) to successfully free the linked list; res* in our program.

`struct stat :` This function was used in our program to return and access information about a file. Stat requires a const char path, which in this case is our file name, and a pointer to a factitious buffer that is filed by the stats which is pointed by stat. In this program buf.st_size was used to store our content length (known as child->content_length), which was a detrimental key feature that helped accurately send the correct expected amount to our HTTP Server program upon a Put Request. The bus.st_size in this program returned the total size in bytes of our referenced file, and stored the bytes in integers.

`fstat :` This function is identical to stat(), except that the file to be stat-ed is specified by the file descriptor fd, in this case corresponding to line 138 of our HTTP Server's source code, where stat pertained to fetch_file_fd (a file descriptor that fetched a file). The significance was on par to how we implemented stat for our Put Request functionality, but in this regards fstat was vital to delivering our content length upon a HTTP/1.1 200 OK Request!

`struct http_node :` A generic simple node struct that contains only the content length, and surprisingly one of the most significant pieces to the success of this program. This node stores content-length in the main program and can access via the struct with an arrow to access the member from that stores the data throughout the program's lifespan.

Assignment Question

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 dog. Why not? Hint: this is an example of complexity being added by an extension of requirements (in this case, data transfer over a network).

The contents may not finish its registry to the HTTP Server's local file. Ergo, in lieu of such an event the status response shall present itself absent, due to undefined behavior (in this case the registry of content was not allowed to finish) to the file. From the unexpected added complexity of such behavior, since a relationship amongst file and server vice versa exist, the absence of a closure will prompt a redundancy of an infinite state of hang. This multiplicative complexity was not present in the first assignment, due to the absence of socket concepts, however both share similarities, in terms how data is sifted, but noting that sockets are managed with more care (One one think that the Endianness and how bits are managed has mass significance, but in regards to the HTTP protocol, the protocol itself does not strictly abide). The transmission control protocol (TCP) in itself handles how the format of the HTTP Protocol will adhere to in regards to protocol, and how that communication is established between client and server. Nevertheless, the last multiplicative complexity was the dealing of multiple clients, which in reference has a relationship to multiple curls. With such demands, it was difficult to reason the complexity upon growing request, and adding one additive feature that seemed exquisite, caused the handling of sockets to invoke a behavior state of brittleness; delicate upon interaction which we regret heavily. Overall, the concepts of interaction of requirements and how we increase efficiency of each component clearly indicates that we observed symptoms of complexity at its finest . By upping the generality of the capabilities of the field descriptors, which resulted in the socket's abilities to now transmit data resulted in an increase of complexity that was difficult to navigate, but by layering such interconnections, and restraining the flow we managed to transfer data with ease, and allow a solid design to take place. I do want to note that dog.c did not observe such multiplication complexities, because we lacked demand and the generality was specialized to a minimal of four function commands (e.g open, read, etc...). However, we noted that Kernel mode (privileged mode) can be dangerous to our non volatile memory, and that the virtual machine should be utilized religiously. To conclude, we successfully designed a complex system by adhering to Nawab's lecture and methodologies to conquer and mitigate complexity by using the five techniques: Naming, Hierarchy, Layering, Abstraction, and Modularization!

Design

The program shall have various phases, but the initial step is checking the amount of arguments passed to the program httpserver. If only ./httpserver localhost or ./httpserver 127.0.0.1 is present, then our default port number is 80, however, if the user enters a port number for the third argument that has a port value above 1024, then that will be the port number for the server. If the number of arguments is less than 2 or greater than 3, then the server won't start. When entered the correct information and number of arguments, the program will then set up the server with information about IPv4 version, connection type TCP, the host address, and the port number. Then we will use the socket, setsockopt, bind, and listen system call to set everything in order to be able to start a server client socket connection via the accept system call. Once the connection is established successfully, we will system call read and use the buffer (4 KiB memory equivalent to 4096 Bytes) from the accepted functions read to parse the HTTP header's *request*, *file name*, and *protocol*. To further solidify program design we integrated Regex (regular expressions) that help parse HTTP headers, hence bringing forth the robustness principle to ensure ease of use to the user. We validate the data from the HTTP request by the client to the server, and we then see if the *request* is a "GET" or a "PUT" request. If it's a "GET" request, we will fetch the file from the server (server's directory) and check if we received it correctly by checking if the file descriptor is NOT -1. If opened correctly, then respond to the client's terminal "HTTP/1.1 200 OK" and provide the user with the size of the file contents. In addition for content length we dynamically allocate memory according to file size, then read from the file and write to the client terminal one line at a time, and terminate this process when the bytes read from the file is 0. We finish "GET" off by closing the file that's in the server and free the dynamically allocated buffer. If it's a "PUT" request, we use the function *strtok* to sift through the header file received on the server side by a client request, and parse the content length using the buffer (4 KiB memory) which contains the header information. Once content length is received, we will open a new file on the server and check if we opened it successfully by checking if the file descriptor is NOT -1. If the file opened successfully and the content length of the file is 0, then we inform the client through the terminal stating “HTTP/1.1 201 Created” and close the file, not forgetting to append the carriage return and newline sequence to ensure proper response to the client . If however the content length is greater than 0, then read from the HTTP body (file’s contents) received on the server’s side and write it into the server made file until we have read up to the content length provided by the HTTP header received on the server’s side. Upon a PUT Request that motions itself as an empty file, not limited to NULL (‘\0’ or \0), the program recognizes this behavior and observes the content length of 0, prompting a speedy response to the client of 201 Created; indicating a success that the file was stashed in the server. Such files of any length will persist in the server, on and off (the server), and be readily made available upon a client request. The files will live within the HTTP Server directory, and for testing purposes can be observed, retrieved, etc. Nevertheless, the program will follow the flow and read until the EOF (End of File), some systems acknowledge this as 0 and others -1. In this case, the generic meaning of EOF where it shows the most significance is when the client is presented with the task of manually shutting down the connection upon an unknown content length. Nevertheless, the server itself is quite tolerable, friendly to errors, and retains its connection throughout the entirety of its use. The program responds to the “GET” or a “PUT” as a request, however, it will close the client socket and exit the program if no such call is made. When the client is done with the server without any errors, the server socket will close, free the address used by getaddrinfo (linked listed), and end the program gracefully. All modularization of miscellaneous helper functions where migrated to the Httpserver_definitions, to mitigate the cognitive burden of the mass amount of interconnections, which resulted in a flawless, but comprehensible program that adheres to formalities in regards to code structure/refactoring. Lastly, by following and adhering to lecture practices, the team was able to modularization segments of components (e.g Regex , PUT, GET), refactor, optimize, and work harmoniously while maintaining a an abstract level of hierarchy/layering to integrate the components successfully. Hence, a fully functional well put complex system design but with simplicity.

Test Case

The following test cases were implemented to ensure program functionality, design, and performance:

- Program is tested vigorously by Valgrind for memory leaks

- Unit testing of modularized components/segments were completed successfully upon each box ranging from the Regex parsing, helper functions, not limited to exhaustive amounts of data over multiple iterative GET/PUT requests of vast amounts. The goal was to break the program down by any means, and reinforce a monolithic structure that sustains a strong lifespan.

- Each component was modularizatized into abstract boxes and placed into the httpserver_definitions.cpp file. The design of each box was to constrain complexity from within the driver program, so that fellow programmers may comprehend this source code with ease. Hence, mitigating lines of code from the driver program to ensure that the hierarchy of abstraction is visible while achieving successful results.

- Testing can be done by typing in terminal: valgrind ./httpserver (optional --leak-check=full). Resulting in total heap usage: 52 allocs, 52 frees, 86,742 bytes allocated. Server is not in run mode.

- Testing can be done by typing in terminal: valgrind ./httpserver localhost 8080 --leak-check=full. Resulting in total heap usage: 3 allocs, 3 frees, 74,280 bytes allocated. Server is not in run mode.

- Testing can be done ./httpserver localhost 8080 with no flags, while the server is running with the following command: valgrind ./httpserver localhost 8080. Since the program is concurrently in run mode, 64 bytes will be in use in 1 block, a total heap usage of 59 allocs, 58 frees, and 94,882 bytes allocated. The amount reachable is 64 bytes in 1 block, but this due to getaddrinfo(), a linked list, that has yet to be freed because the HTTP Server is active while conducting this test. Should the HTTP Server not be in run mode, you will have all no leaks as expected.

- Binary files were tested and compared with the diff -s foo1 foo2. Both are identical when generated and compared to the original file. You may also compare the original file interchangeably with foo1 and foo2.

- The following command was used to further test our program's capabilities by scraping websites: curl https://www.google.com > hello.txt. Successfully retrieving information upon curl.

- Files persist with this program should the client disconnect from the server. To test this feature, start the server, and go into a remote directory or desktop. In the terminal type in the following command: curl -T foo localhost:8080/1234567890 -v. This command will put the file, along with its contents, in the http server directory from your remote location. You may then retrieve this file from your location by typing in the following command in the terminal: curl localhost:8080/1234567890 > new_file01. To compare if your contents indeed were retrieved successfully, you may apply the following command: diff foo new_file01 -sq. For further reassurance, you may locate file 1234567890 within the HTTP Server directory, and compare all three contents interchangeably to ensure contents were recorded accurately.

- The HTTP Server program is to remain connected during all requests, excluding a Put Request with no content-length (resulting in a closed connection that is conducted manually with no client response from the server). The following test checks for prolonged connection persistence, in the terminal type the following: curl http://localhost:8080/123456789[a-b], a and b as any integer that pertains to files. This command will generate a Get Request, returning the content sequentially in the order that it was called, with the content, status msg e.g 200, 404, etc..., on each iteration, all while maintaining a connected status.

- The HTTP Server program was tested with the bible's length of 4.5 million bytes on a (Put Request) curl -T bible localhost:8080/abc123456[0-9] -v for a consecutive ten iterations,all while successfully keeping the connection intact to the server. This command may also be implemented as curl -T bible localhost:8080/123456789[0-9], the iteration numbers within the brackets may be of the clients choosing in regards to range.

- The HTTP Server program was tested with the bible's length of 4.5 million bytes on a (Get Request) curl 'localhost:8080/abc123456[0-9]' -v | grep "Connection" for a consecutive ten iterations, all while successfully keeping the connection intact to the server. This command may also be implemented as curl localhost:8080/123456789[0-9] -v from within the brackets may be of the clients choosing in regards to range.

- The HTTP Server program when prompted with multiple Get Request/Put Request will remain open during the entire process, patiently awaiting for the next curl upon completing its objective, and will respond accordingly with the appropriate response should a file be present, internal error, forbidden, created, etc... all while keeping the connection alive.

- The HTTP Server program will acknowledge a file of content-length 0 and null upon a Put Request, and will retrieve them successfully upon a Get Request.

Functions

`char *strtok(char *str, const char *delim)`
A macro used to break apart a string dependent upon a pattern(delimiter) that is in the string *str itself. This function returns a pointer to the first value up to where the delimiter is found in the string, if nothing in front of the first delimiter however, the first values after the delimiter is returned. A null pointer is returned if there are no values left to retrieve.

`void *memset(void *str, int c, size_t n)`
Used to fill a block of memory with a particular value. Where the starting pointer to the address of memory is to be filled with the provided value, given the size. Memset was declared in our dog definitions to re-initiate the state of our buffer; setting all bits to zero by using '\0' for the second argument. The third argument is simply the size for this buffer in question; 4Kib as stated in requirements.

`int strcmp(const char *str1, const char *str2)`
Takes two string arguments, compares them, if equal, returns integer value 0 indicating equivalence. In our driver program dog.cpp, this feature was implemented as the first argument corresponds to the file arguments, and the second argument a string dash argument.

`int sscanf(const char *str, const char *format, ...)`
This function reads/parses data from a string rather than standard input or input from the keyboard. The first argument is a string from where we want to read/parse the data from. The second argument is the format/style in which we want to parse from the first argument. So if we want to parse for example 3 strings from the first argument(%s %s %s), the last arguments 4,5,6 represented by "…" are getting those parsed strings in that order respectively. The output is the number of items read from the string, and returns -1 if the parsing is unsuccessful.

`char *strcat(char *dest, const char *src)`

This function appends the input string src to the end of the input string dest. The output of this function is the string dest concatenated with src.

`void freeaddrinfo(struct addrinfo *ai)`

This function frees the linked list ***res** which is dynamically allocated memory. The function should also free **addrinfo** structures returned by **getaddrinfo()**, along with any additional storage associated with those structures.

`bool f_void_check_file(char *parseFile)`

This function checks that all resource names passed from the client to the server must be 10 ASCII characters long, must consist only of the upper and lowercase letters of the (English) alphabet (52 characters), and the digits 0–9 (10 characters). If a request includes an invalid name, the function will respond accordingly.

`bool f_void_check_file_len(char *fileLength)`

Checks if the file name being PUT to the server by a client is of 10-ASCII character length. If a request includes an invalid name length, the function will respond accordingly.

`void f_void_server_arg_error()`

This function will be triggered when the user(you) sets up the server the wrong way and will respond to the user accordingly. After you compile the server you will have a binary called httpserver, which you will execute with ./httpserver <address> <port> (but <port> is optional and will default to 80 when not specified).

`void f_void_setsocket_error(int setsocket)`

This function is called upon when the socket connection on the server side is unsuccessful, it will return an integer which indicates an error with -1. The specific code in the global variable errno will respond to the user the problem accordingly indicating the problem.

`void f_void_bind_error(int bind)`

This function is called upon when the bind connection on the server side is unsuccessful, it will return an integer which indicates an error with -1. The specific code in the global variable errno will respond to the user the problem accordingly indicating the problem and close the bind socket.

`void f_void_listen_error(int socket_listen)`

This function is called upon when the listen connection on the server side is unsuccessful, it will return an integer which indicates an error with -1. The specific code in the global variable errno will respond to the user the problem accordingly indicating the problem and close the listen socket.

`int fstat(int fd, struct stat *buf)`

This function helps us obtain the content length of a file that's passed through a file descriptor. However the buffer is obtained by creating a structure (struct stat buf) which will contain the content length itself.

`void f_client_req_found(int file,int clientsocket)`

This function is for when the user sends a GET(read) request to the server. The server checks if the file exists, if so, the server terminates the child process and responds to the client with “HTTP/1.1 200 OK\r\n\r\n” along with the contents of the file(Note: “\r\n\r\n” is new line). On the new line the client will see “Content-Length: X \r\n\r\n” where X is a number written in the form of the data type “char”(ex: “278”).

`void f_client_req_created(int int_client_sockd)`

This function is called upon when the client sends a PUT request to the server. The server will respond by printing to the client terminal via the dprintf function "HTTP/1.1 201 Created\r\nContent-Length: 0\r\n\r\n".

`void f_void_400(int clientsock)`

This function is called upon when the client makes a 400 bad request to the server. The server will respond to the client by printing to the client terminal via the dprintf function “HTTP/1.1 400 BAD REQUEST\r\nContent-Length: 0\r\n\r\n".

`void f_void_client_error_forbid(int int_client_sockd)`

The input is the socket connection between server and client. This function is called upon when the client triggers a 403 Forbidden on the server side. The server will respond to the client by printing to the client terminal via the dprintf function "HTTP/1.1 403 FORBIDDEN\r\nContent-Length: 0\r\n\r\n".

`void f_void_client_error_not_found(int int_client_sockd)`

The input is the socket connection between server and client. This function is triggered when the client is requesting(GET) a file from the server that doesn't exist, hence, making the server print on the client “HTTP/1.1 404 FILE NOT FOUND\r\nContent-Length: 0\r\n\r\n”.

void f_void_intr_error(int int_client_sockd)
The input is the socket connection between server and client. This function is triggered when the client tries to communicate with the server and the server is unable to fulfill the request due to an error on the server, hence, making the server print to the client "HTTP/1.1 500 Internal Service Error\r\nContent-Length: 0\r\n\r\n"

void f_void_kill_request(int clientsock,struct addrinfo *res)
The input is the socket connection between server and client and the linked list pointer containing all the information regarding the connection. This function gets triggered when the request that is sent to the server is not a PUT or a GET request, which results in the function making the server client socket connection to close, free all the dynamically allocated memory(socket address structures returned in the form of a list pointed to by res), and exit the program.

void f_void_path_error(int int_client_sockd)
The input is the socket connection between server and client. This function is triggered when the system call read(2) fails to read information(client request) from the client and prints a 500 internal error message "HTTP/1.1 500 Internal Service Error\r\nContent-Length: 0\r\n\r\n".

void f_void_getaddrinfo_error(int status,struct addrinfo *res)
The get address feature was a wonderful treat that was integrated to the HTTP Server program on line 44. The used parameters are self explanatory: getaddrinfo(host_address, portNumber,&hints,&res); The host_address is e.g localhost/127.0.0.1, the user also has the ability to establish a custom port number, in this case holding the second parameter. The hints and res we will go into depth soon, but prior to this function, gethostbyname() was used to do DNS lookups (associating various information to domain names) by loading the data into a struct sockaddr_in. However, in this program we used this function to mitigate lines of code and to keep up with industry standards. In regards to the hints, this points to a struct addrinfo which helped us assign us to the American Familly_INET (which is an address family that helped designate the type of addresses that the socket may communicate with, in this case our IP addresses). Also tying into our SOCK_STREAM features via hints, which is known as TCP (Transmission Control Protocol which establishes a connection between the client and server prior to the data being sent). Finally the res, is the carrier one might say, presenting itself as a pointer to a linked list, which when you run valgrind in

its entirety will become apparent as it requires freeaddrinfo(res) to successfully free the linked list; res* in our program.

void f_void_error_on_accept()
A generic message informing the user that connection is reestablishing via the server.

int f_fetch_file(char *char_file)
The input is the file name that the client requested to get back using GET. This function opens a file for reading only using the open(2) system call and returns the file descriptor associated with the file that was just opened.

Pseudocode

procedure httpserver.cpp

```
Struct http_node with member content_length
host_address = argv[1]
Struct http_node with member child, member process

If argc < 2 or argc > 3 then
        perror("Server format failure.  Try ./httpserver <ip><port> or sudo ./httpserver
<ip>\n");
   exit(EXIT_FAILURE);

If argc == 2 then
        portNumber = “80”
Else
        portNumber = argv[2]
Struct sockaddr_storact with member client_addr
Struct addrinfo with member hints, member res
Clear all contents of hints
hints.ai_family = AF_INET
hints.ai_socktype = SOCK_STREAM
hints.ai_protocol = IPPROTO_TCP

If ((return_value = getaddrinfo(host_address, portNumber, hints, res)) != 0) then
        Prints error of failure to client terminal
        freeaddrinfo(res)  //clears res’s linked list
        exit(EXIT_FAILURE)
serverSocket = socket(res->ai_family, res->ai_socktype, res->ai_protocol)

If (serverSocket < 0) then
        Prints specific error message of failure to client terminal
        close serverSocket
        exit(EXIT_FAILURE)
If ((return_value = setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR,
&enable, sizeof(int))) < 0) then
        Prints specific error message of failure to client terminal
        close serverSocket
        exit(EXIT_FAILURE)

If ((return_value = bind(serverSocket, res->ai_addr, res->ai_addrlen)) < 0 ) then
        Prints specific error message of failure to client terminal
        close serverSocket
        exit(EXIT_FAILURE)

If ((return_value = listen(serverSocket, 500)) == -1) then
        Prints specific error message of failure to client terminal
        close serverSocket
        exit(EXIT_FAILURE)

while(true)
        If (recv(clientSocket, &data, 1, MSG_PEEK) <= 0) then
                clientSocket = accept(serverSocket, (struct sockaddr *)&client_addr,
                &addr_size);
        Else
                Print to the client terminal ("Warning on accept(). Found bundle for host.
Re-using existing connection!")


        If (read (clientSocket, buffer, 4096) < 0) then
                Print to the client terminal   ("Read path failure from client file\n.)
                Print to the client terminal ("HTTP/1.1 500 Internal Service
                Error\r\nContent-Length: 0\r\n\r\n")

        sscanf(buffer, "%s %s %s", request, file, protocol)
        strcpy (buffer_for_regex,request)

        strcat (buffer_for_regex,file)

        strcat (buffer_for_regex,protocol)

        Regex match (buffer_for_regex with "(GET|PUT)[/]?[a-zA-Z0-9]{10}HTTP/1.1")

        If (Regex match != true) then
                Print to the client terminal("HTTP/1.1 400 BAD
    REQUEST\r\nContent-Length: 0\r\n\r\n")
                continue

        File_name = strtok(file, "/")

        check_file_length = returns true if File_name's length is 10, else returns false

        If (check_file_length == false) then
                Print to the client terminal("HTTP/1.1 400 BAD
    REQUEST\r\nContent-Length: 0\r\n\r\n")
                clear all contents of buffer
                continue

        If (request == "GET") then
                Clear all contents of buffer
                fetch_file_fd = opens(file_name) as read only and returns file descriptor
                If (fetch_file_fd file opened unsuccessfully) then
                        Print to the clients terminal ("HTTP/1.1 404 FILE NOT
    FOUND\r\nContent-Length: 0\r\n\r\n")
                        close fetch_file_fd
                        continue

                If (fetch_file_fd file opened successfully) then
                        Print to the client terminal("HTTP/1.1 200 OK\r\nContent-Length: X
    \r\n\r\n")
                        Struct stat with member buf
                        fstat(fetch_file_fd, &buf)
                        get_length = buf.st_size
```

```
            buff = dynamically allocate bytes returning char* with length (get_length)

            while((child -> content_length = read(fetch_file_fd, buff, sizeof(buff))) != 0)
            send_bytes =  write(clientSocket,buff,child->content_length)
            if(send_bytes == get_length)
                break

            close fetch_file_fd

            if(read all the contents of the file) then
                free buff’s memory
                continue
            Else
                Prints to the client terminal("HTTP/1.1 500 Internal Service Error\r\nContent-Length: 0\r\n\r\n")
                free buff’s memory
                continue



    If (request == “PUT”) then
        content_length = parse content length from the server’s received header by using the function sscanf described under “Functions” in this design document

        clear all contents of buffer
        put_fd_request = return file descriptor from server opening a file
        If (put_fd_request file opened unsuccessfully) then
            Prints to the client terminal ("HTTP/1.1 404 FILE NOT FOUND\r\nContent-Length: 0\r\n\r\n")
        If (content_length == 0) then
            Print to the client terminal("HTTP/1.1 201 Created\r\nContent-Length: 0\r\n\r\n")
            close put_fd_request
        If (content_length > 0) then
            buff = dynamically allocate bytes returning char* with length (get_length)
            while((check_bytes_recv = read(clientSocket, content_length)) > 0)
```

```
                write_bytes =  write_bytes + write(put_fd_request, buff, check_bytes_recv)
                if(write_bytes == content_length) then
                    break

            If (write_bytes == content_length) then
                Print to the client terminal("HTTP/1.1 201 Created\r\nContent-Length: 0\r\n\r\n")
                close put_fd_request
                Continue
            Else
                Print to the client terminal
                close put_fd_request
                continue

            Free buff's memory
    Else
        while((recv_bytes = recv(clientSocket, buffer, sizeof(buffer), 0)) != EOF
            sentByte = write(put_fd_request, buffer, recv_byte)
            clear all contents of buffer
            if(recv_byte == EOF OR sendByte == EOF) then
                Close put_fd_request
  If (request != “PUT” OR request != “GET) then
    close int_client_sockd
    free res’s memory
    exit(EXIT_FAILURE)

close serverSocket
freeaddrinfo res //clears res’s linked list
exit(0)
```