HTTP Server Design

Goal: The goal of the assignment is to implement a single-threaded HTTP server that responds to simple GET and PUT requests. The program creates a HTTP server that for each request, the server will read/write “files” consisting of ASCII names that are 10 characters long. The server files will persist in a directory, so that the server is portable and can be run if a directory already exists and contains files.

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).

Answer:
If the connection is closed and ends the communication early, the socket gets closed early and no bytes are read. This was not an issue in dog because there is no extra communication between two modules. Dog only needed to search within the directory and return the output to the terminal. However, in this assignment for the HTTP server, there is a transfer of data between the client socket and the server. There is an extra layer of complexity due to the maintenance of the connection between the socket and server. As a result, there exists potential for an extra layer of failure if the server connection is faulty.

Design:
Overview: The HTTP server program has two main commands, GET and PUT. The GET request will fetch a file from the server and print it to the client. The PUT request will send a file from the client to the server and store it into a filename specified in the request. Depending on the status of the request, a “response” containing a HTTP code and the content length will be returned. The program does not use any FILE* calls and primarily uses the functions open(), close(), write(), read() to read and write files within the socket. In order to obtain the type of the request, the file involved, and the http version of the request, the sscanf() function is used to parse the request as a string in order to get all the needed information. The dprintf() function is used to print response statements and the file contents for a GET request to the socket. This program does not use any special formulas or any non-trivial algorithms.

Structures: The HTTP server is written in C, using char arrays to store and manipulate strings. When sscanf() is called at the beginning of the program, it parses the subsequent three strings from the HTTP header into the char arrays method, filename, and http version. Char arrays are also used to create a buffer for reading and writing to the specified file, and is used to find the content-length within the header. The buffer was set to be an unsigned integer of size 4096, as the buffer length will not be negative for any reason and 4096 is the standard size of directories in unix.

Functions: The program has 5 functions total; main, checkString, checkVersion, check, and getAddr. The main function takes in an int, then a char string for the address and port for the

server. It primarily works to process the request, checking if the method char array matches ‘GET’ or ‘PUT’, otherwise it prints a 500 code to the socket. If the filename does not contain only number 0-9 and characters a-z, A-Z, or is not only 10 characters long, or the http version is not 1.1, then the main function will print a 400 code to the socket. In the main method, the syscalls bind(), listen_fd(), and accept() were used to set up the socket, this code was given by the TA in section.

For a GET request, a stat struct is created to determine whether or not the file exists, if a file has even been given as an argument. If an invalid file was given as the target from the request, then a 404 response is returned and a 403 response if the arguments specifies a directory or a file that the client does not have permissions for. Otherwise, using the stat struct, the size of the file is obtained for the 200 response with the Content-Length to the socket. Then the contents of the file are read using a loop upon “read()” that repeatedly reads the requested file’s content into the buffer until any more remains and then writes it to the client socket with write(). The file and socket is then closed.

For a PUT request, strstr(buffer, “Content-Length”) is used to locate the “Content-Length” of the request and then sscanf it into an ssize_t contentLength. The file derived from the filename from the request is then opened with open(). If the file can’t be opened, a 403 code is printed to the socket, otherwise the server will read the contents of the client socket with a while loop containing “read()”, like we did for get, reading from the client socket rather than an existing file. It will then write to the opened file for the content length specified in the header. The connection to the socket is closed once the bytes written reaches the contentLength. A 201 code is then returned and the file and socket are closed.

The checkString function takes in a char array and checks whether the array contains only digits 0-9 and characters a-z, A-Z. Whenever a valid character is found, the array is incremented and returns whether or not the array pointer is on a null character “\0”.

The getAddr function was given by the TA Daniel as a starting code for how to set up the socket connection to the server. It takes in a char pointer set to name in order to obtain the address of the host that was listed in the arguments when running the HTTP server. It then returns the address.

The check() function, referenced from stackoverflow, takes in an int and if the int is negative, the function returns a warning for a bad input. This function was also provided in Daniel’s example.

The checkVersion() function takes in a char array and uses strstr to determine whether or not the string “1.1” exists within the array, therefore checking whether or not the version is 1.1. It returns 1 if 1.1 exists within the array, and returns 0 if it does not exist.

Testing: For testing GET, we used the command curl localhost:8080/0123456789 -v where the designated address was local host with the port 8080 for returning the contents of file 0123456789. The address, port, and file name were changed depending on what address, port, and file we were testing. We attempted to break the server by changing the length of the file name, attempting to access nonexistent files, and changing various parts of the header. Similarly for PUT, we used the command curl -T test1.txt localhost:8080/0123456789 -v and changed the address, port, and file name depending on the arguments given when the program was initially run. We also used other tags such as --request-target and --header 'Content-Length: 10' to test more specific trickier requests on our server.

Pseudo-code:

```
int main(int argc, char** argv){
        set address to be argv[1]

        If >3 arguments:
                Error: too many arguments
                EXIT_FAILURE
        else if 3 arguments:
                set port = argv[2]
        else if 2 arguments (if no port specified):
                set port = "80"   → the default port for the assignment
        else if 1 argument:
                Error: not enough arguments
                EXIT_FAILURE

        get address info and set to port and host
        *initialize socket* → code given from sections
        while(connection is going){
                create socket comm_fd
                read from socket and place into buf

                //our buffer was wiped
                copy buf into temporary buffer
                // scan in method, filename, and version
                sscanf(buf, "%s /%s %s ", method, filename, httpversion)
                copy temporary buffer back into buf

                if(filename is not 10 characters long or has an invalid character or has invalid
        version){
                        400 error to client
                        Close socket
                if(method is get){
                        create a stat struct
                        fd = stat(filename, &filestats)  → allows us to get file stats
```

```
                If file invalid
                        404 error  to client
                        Close socket
                } else if( not a file at all){
                        403 error  to client
                        Close socket
                } else {
                        size = filestats.size  → sets size of file to the int size
                        200 OK
                        fd = open file as readonly
                        If file invalid
                                403 error  to client
                                Close socket
                        while(there are bytes to read from file to buffer){
                                Write file to client socket
                        }
                Close file fd
                Close client socket

        }
        else if( method is PUT){
                if(filename is not 10 ascii characters long or has invalid characters){
                        400 error  to client
                        Close client socket
                }
                get content length from buffer
                fd = open file if permissions match
                If file invalid
                        403 error  to client
                        Close socket
                }else {
                        200 OK
                        while( there are bytes to read){
                                write to comm_fd from buffer until specified content length
                        }
                }
                Close client socket
        } else{
                400 Error to client
        }
        Close client socket

}
return success;
```

```
}
```