Assignment 1 Design

I. Introduction

A. Objective

The goal of this assignment is to implement a single-threaded HTTP server that responds to GET and PUT commands to read and write files, respectively.

B. Statement of Scope

The HTTP server will listen for GET or PUT requests, followed by a 10 character ASCII file name, and then optionally content-length on a user-specified port. A GET request to the server followed by a filename will return the contents of a file that is stored persistently in the same directory or return an error if the file is not found. A PUT request to the server, followed by a file name and data will send the data to be put into a new file denoted by the file name. Each response header to a GET or PUT request will be a valid HTTP response that specifies the HTTP version, the status code, and content length.

II. Restrictions

The program will not utilize standard libraries for HTTP or any of the C library FILE* functions except for when printing to the screen. The system calls: read and write are used for file I/O. The only other system calls: socket, bind, listen, accept, connect, send, recv are used for socket programming. The port numbers that the user can specify are restricted to port numbers above 1024 unless the server is run with sudo.

III. Program Design

1. Determine which port number to listen from using the function **getPort()**, which takes in the second command line argument and returns the user-specified port or the default port 80, if no port is specified. User-specified ports must be above 1024 and the program will exit if an invalid port is provided.

2. A **sockaddr_in struct** named **server_addr** is declared. **inet_aton()** is used to convert the first command line argument, which specifies a hostname or an IP address, to a network address for the socket. The port number is then assigned to the socket as well.

3. The server socket is created with a file descriptor, **server_socket**, that refers to it.

4. Socket options are set using **setsockopt()** with the option **SO_REUSEADDR** to allow the reuse of addresses and ports. The use of this function is suggested from the first link cited.

5. **bind()** is used to assign **server_addr** to the server socket.

6. Listen for incoming connections on the server socket.

7. A **sockaddr struct** named **client_addr** is declared.

8. An **httpObject struct** named **request** is declared. This struct is to be populated with the request type, file name, status code, HTTP version, and content length.

9. An infinite loop is implemented so that the server will continuously accept client requests until it is terminated. Once a connection request is accepted, the following will occur:
   a. The **executeFunctions()** function is called, where the HTTP header from the client socket will be received and stored in a buffer.
      i. The header will then be parsed with **parse_request()**, where the HTTP version and file name will be checked for validity. If both are valid, the request type, file name, HTTP version, and content length (if present) are extracted and put into the **httpObject request**. Then, **parse_request()** returns to **executeFunctions()**.
      ii. If the request is a GET request, call the function **get_request()**. The file specified by the file name in the header is opened. If it fails to open, the appropriate status code is determined, put into the **httpObject request**, and the HTTP response is constructed and sent to the client. If the file is opened successfully, the content length is determined, the HTTP response is constructed and sent to the client, then the contents of the file will be continuously read and sent to the client until the end of the file is reached. The file is then closed before **get_request()** returns to **executeFunctions()**.
      iii. If the request is a PUT request, call the function **put_request()**. The file specified by the file name in the header is opened. If it fails to open, status code 500 is returned. This is the appropriate status code because unlike GET, if the named file does not exist, one with the specified file name will be created. The HTTP response will then be created and sent to the client. If the file is opened, or created, successfully, we check if the content length of the file is noted in the header with a 200/201 request. If content length is specified, the server stops reading receiving data after the specified number of bytes and looks for another header. If it isn't, the server receives data until the client closes the connection or the end of the file is reached. The HTTP response is constructed and sent to the client, the file is closed, and **put_request()** returns to **executeFunctions()**.
      iv. If the request is not a GET or PUT, then it's a bad request. The HTTP response is constructed with a 400 status code.
   b. The file descriptor of the connected socket is closed.

10. **EXIT_SUCCESS**

IV. Pseudocode

A. struct httpObject:

```
char[] type
char[] filename
char[] httpversion
int status code
content length
END
```

B. struct flags:

```
bool exists
bool isFirstParse
bool nameIsValid
END
```

C. executeFunctions(comm_fd, httpObjects request, buf, flag):

Sscanf from buf (type, filename, httpversion, flag) delimited by space into request
Return/Break if file name or http version is not valid
While loop for parsing header
    Recv header
    Parse for content-length
    Break if no more header
Call Get or Put request depending on type

D. parse_request(comm_fd, httpObject request, char buf, flag):

If is first time calling parse
    Check http version
    Check if name is valid
(Else parsing for content-length)
If content length string exists
    Store into request.content length
Else (no content length string in header)
    break and leave content length empty
END

E. get_request(com_fd, httpObject request, buf, flag):

Memset buf since all objects are parsed into request already
Open file from filename for reading
IF error
    Set status code to error number
    Construct response
    Send constructed response
ELSE (get request succeeds)
    Set status code to 200
    Get content length of file being read and store into request
    Construct response
    Send constructed response
    Read and send the file
Close file
END

F. put_request(com_fd, httpObject request, buf, flag):

Memset buf since all objects are parsed into request already
Check whether file name already exists and set exists flag accordingly
Open file from filename for writing
Check to see if open is successful

```
        IF content length exists:
            While content length is not 0:
                Write from buf to file
                Content length-bytes received
            IF content length is 0 (meaning all of data was received)
                IF file already existed
                    Set status code to 200
                ELSE
                    Set status code to 201
            ELSE(data was not all received)
                Throw error 500
        ELSE (content length does not exist)
            While client does not close connection(recv != 0)
                Read from buf to file until EOF
            Set success
        Construct response
        Close file
    END
```

G. construct_response(comm_fd, httpObject request):

```
        Char length_string[20]
        Convert request→content_length from a long int to to a string and store in
            length_string
        Char response[50]
        Copy request→httpversion to response
        Concatenate " " to response
        Concatenate getCode(request→status_code) to response
        Concatenate "\r\n" to response
        Concatenate "Content-Length: " to response
        Concatenate length_string to response
        Concatenate "\r\n\r\n" to response
        send(comm_fd, response, strlen(response), 0)
    END
```

V. Testing

A. Classes of Tests

White-box testing will be utilized while testing the functionality of the program. Although the use of C library FILE* functions are restricted for the functionality of the program, they will be used during testing as a convenient way to debug and keep track of values such as loop iterations and filenames. These print statements will be removed before submission to

ensure clean and readable code. The program will be incrementally developed, so these tests are performed at every stage. These tests are described as follows:

a. First the tests were focused on establishing a connection between the server and client using socket system calls

b. After the connection was established, we started testing how to successfully parse the HTTP requests into a HTTPobject struct.

c. After being able to parse the HTTP requests, we then focused on how to successfully pass back a file from a GET request along with the header.

d. Dealing with PUT requests was next along with how to receive the information and store it locally

- **a)** Requests with the "default" content-length were tested to see basic correctness.
- **b)** Requests with an overridden content-length were tested to ensure only the specified amount of bytes would be written.
- **c)** Requests with no content-length header were tested to see whether data would be written until connection was closed.

e. Exception handling with correct HTTP error codes were next. Many tests were done to make sure the correct codes were being sent when given a "faulty" request.

Black-box testing will also be utilized through manual testing. The code will be tested using a variety of different combinations of inputs. The test inputs include the following:

a. Testing if 127.0.0.1 and localhost were interchangeable
   i. Set server to http://localhost and client to connect to 127.0.0.1
   ii. Set server to 127.0.0.1 and client to http://localhost

b. Making sure the **while loops** for system calls work correctly by reducing the buffer size forcing the loop to run multiple times
   i. While loop for parsing the header
   ii. While loop for sending a file for a get request
   iii. While loop for receiving information for a put request

c. Testing **Get Requests**
   i. A normal valid get request
   ii. A get request with invalid files
      1. Nonexistent file
      2. File name with a non alphanumeric char
      3. File name less than 10 ASCII
      4. File name more than 10 ASCII
      5. A file that has no read permission

d. Testing **Put Requests**
   i. Normal valid put request
   ii. A put request with invalid files

1. Nonexistent file
2. File name with a non alphanumeric char
3. File name less than 10 ASCII
4. File name more than 10 ASCII

iii. A file that has no read permission

iv. A file that has no write permission

B. Expected Output

a. A get statement with a valid request will send an appropriate response header followed by the contents of the specified file stored in the same directory as httpserver.cpp.

b. A put statement with a valid request will send an appropriate response header and write contents of the file specified as an argument in the same folder as the client to **either** a new file created by open(2) or overwrite an existing file on the server directory.

VI. 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).

In our implementation, if the connection is closed during a PUT with a `Content-Length`, the target file would be written up to the point where connection was ended and no response is sent. This means that before-or-after atomicity is not achieved.

This concern does not apply to Dog because it is a single program designed to compile and run on the same instance. To use the HTTPserver successfully, there has to exist two instances: the server and the client. The server is a continuously listening program run on one instance (terminal) separate from the client (remote terminal), which has to connect to the server first.

VII. Citations

1. setsockopt(): https://www.geeksforgeeks.org/socket-programming-cc/