Design Document
Assignment 1
Andi Zhao

1. Goals:

The goal of this assignment is to implement a simple single-threaded HTTP server that handles HTTP GET requests, HTTP PUT requests, and HTTP HEAD requests. The server works with localhost to any valid port number. On a PUT request, the server should create a file with the proper contents on the hard drive. GET requests should return the file contents to the client and HEAD requests should return only the header to the client.

2. General flow of my server:

The main function will set up the server, making a series of calls including socket, setsockopt bind, listen..
I set my listen queue to 5
As the server should always be up, it infinitely listens for connections and accepts a connection when it comes, assuming the queue is not full.
when my server gets a connection, it makes a first pass at the client socket to get the header, then passes the header buffer into a string parser, which will set the file name, file length, and request type from the buffer.
With the information from the parser, I can now determine which file on the server to access from file name and which method to use from request type. The server then calls the appropriate function (GET,PUT,HEAD).
When the function returns, the server is done with the request and goes back to listening

3. Design

There are three main parts to this design. The first part is the standard server set up in which a server socket will bind to a specified port and listen indefinitely for client connections

The second part is the heading parser which will read the header from the client socket and extract the necessary information.

With that information, the server will call the appropriate handler function (GET,PUT,HEAD) to handle the client request. Afterwards, the client connection is closed and the server goes back to listening for connections.

2.1 Main function design:

inputs : port number
Output: None (-1) if program fails at any point.
Variables: char* requestType, char* fileName, size_t fileSize
Set up server socket with socket()
Set socket options with setsockopt()
Bind to port bind()

Listen for connections listen()

2.2 Parsing the Header:

Inputs: char* fileName, char* requestType, size_t length, int clientSockd
Output: assigns proper values to fileName, requestType and length
Variables char* tokenized, int count
Char* buffer ---> read(clientSockd)
Tokenize the buffer using strtok
tokenized -> strtok(buffer,"/r/n")
While tokenize is not null:
    Assign values to inputs using sscanf
    if(there is a content length header)
        Assign length to the number that follows
    If there is another content length header, throw 400
    Move to next line

2.3 Use cases:

2.3.1 Client sends a GET request:

Inputs: int clientSockd, fileName
Output contents of fileName
Variables uint8_t* buffer
Open fileName
    If fileName can not be opened:
        Throw an error
        return
    Send a 200 OK message to client
    Read from fileName
        Write contents read to clientSockd
            If write fails
            Throw an error and close the file
            return

2.3.2 client sends a PUT request:

Input int clientSockd, fileName
Outputs: create a file named fileName with the same contents on disk
Open fileName, if it does not exist, create a new file named fileName with write permissions
If it exists
    delete(fileName) and open a new one with permissions
Read from clientSockd to get the HTTP body.
Write contents to fileName
Return 201 created message and content length on completion

2.3.3 Client sends a HEAD request:

Input: int clientSockd, fileName
Output: sends the content length of file to clientSockd
Open fileName
Use fstat to get the content-length
Write 200 OK message followed by content-length to client

4. Helper functions:

3.1 Validate Header:

Reads data from the socket and checks for “/r/n/r/n”, indicating end of header

```
Int count =0
While target not found and count < 4096:
        Read from socket and add to buffer
        Parse for target strstr
        count+= read bytes
        If found:
                TargetFound->true
```

3.2 validate fileName

```
*called after eliminating the character of fileName if necessary
If length of fileName > 27:
        Return -1;
For all characters in fileName:
        If character is not valid
                Return -1
Return 0;
```

5. Error Handling:

This server should be relatively robust and able to throw appropriate errors when necessary

The error codes that are implemented are:

404 not found(if file is not found)

403 Forbidden (if fileName is not supported)

400 Bad Request (malformed syntax)

500 Server Error

At any point that read or write returns -1, an error should be thrown and the function will return

If at any point the client does not send data within 5 seconds, the server should send a timed out message

If the header is greater than 4096 bytes, the header length is out of range and thus will throw a 400 error.

If the input for the port number is not a number or less than 1025, server will throw an error

If the fileName does not exist, a 404 error will be thrown for read.

If the fileName is not valid by length > 27 or contains invalid characters, a 403 error is thrown

My server does have capabilities to throw a 500 error, but as bad headers including valid POST or FETCH requests are considered a 400 error according to Daniel, my server does not actually throw a 500 error.