Design Document

Author: Miguel Avalos

Student ID: 1704078

Project Description

httpserver is a program that creates an http server that receives requests from clients and attempts to complete their request. After the attempt, the server sends a response to the client containing the code indicating the result and possibly data related to the request. There are three types of requests: PUT to store data into a file in the server, GET to retrieve a file back to the client, and HEAD retrieving file size.

Program logic

httpserver will wait for a request from a client. Once it receives one, the request will be parsed. Then error checking will start. The server will respond with a 400 error, close the connection, and will listen for a new connection if one of the following happens: if the request is missing information required for the type of request, fails formatting requirements, or has a body when it shouldn't have. 501 error response will be sent if the request isn't HEAD, PUT, or GET. For a PUT request, it will check if the file is invalid by checking if there already exists a file but lacks permissions to overwrite it or if there exists a directory with the same name. Both will respond with a 403 error . For a GET or HEAD request, it will also check if a file is invalid like the PUT request does, but they have the extra requirement that the file needs to exist in the directory. If this requirement is failed, then the server will respond with a 404 error. If the request passes this error checking, then it will likely execute the request successfully with a 200 or 201 code response (with attached data if necessary). Unless the last request had a 400 error, it will check if the client has sent other requests on the same connection; If so, it will process them the same way as above. Once the client closes the connection, the server will close the connection too and listen for a new one. If the connection is closed by the client at any time during this process, the server will not send a response, the connection will be closed, and will listen for a new connection.

Data Structures

A character array will be used as a buffer to store data received from a client and data from files that will be sent to the client.

Functions

- int filerror(int option) 403 404 500
  This function is responsible for determining what error occurred while interacting with a file and returning an integer value that will be used as a code for the response to the

client. The caller must call this function using the errno(3) return value as the argument for **option**.

  - If the program lacks the permissions to access the file, then 403 will be returned.
  - If the file doesn't exist, then 404 will be returned.
  - For any other error, 500 will be returned.

- void err_response(int code, int conn_fd, bool body)
  This function is responsible for sending http responses to the client if an error occurs. If **body** is true, a body will be sent containing a brief description of the error.
  The responses vary by code:
  - 400 - Bad Request
  - 403 - Forbidden
  - 404 - Not Found
  - 500 - Internal Server Error
  - 501 - Not Implemented

- void discardbody(int conn_fd, int length)
  This function is responsible for discarding **length** bytes of data that are reported by the caller to be part of the body of a request that is not needed.
  If the **length** is less than 1 the function will return. Otherwise, it continues to discard the data up to length or when eof is reached.

- bool process_request(int conn_fd)
  This function is responsible for parsing the request for the client and calling the necessary function to complete the request if no errors were found with the request. A return value of true indicates that the connection needs to be closed. If the connection is closed by the client at any time during this process (including all function calls), the server will not send a response, the connection will be closed, and will listen for a new connection.

\

  1. Attempt to get the first line into a buffer of 4001 bytes (one space is dedicated to '\0' character, receive byte by byte to find a newline character that ends the first line.
  2. While reading all data before the body, search the remaining request data for the content-length component.
  3. Let variable bad be false.
  4. Scan for command and /filepath in the buffer. If they are not detected, set bad to true.
  5. (If bad isn't already true) If filepath is not exactly 15 characters or not all of the characters are alphanumeric, set bad to true.
  6. Match command to head, put, or get. If none match, respond with a 501 error with a body, discard the remaining body and return true.
  7. Depending on the type of request:
     a. For PUT requests:
        i. Respond with a 400 error with a body and return true if:
           1. bad is true

            2. content-length's value: does not exist, is negative or contains a non-digit.
        ii. Otherwise, call the putfile function with the filepath**,** content-length's value, and **conn_fd**. If error occurs with request, discard all body contents and send an appropriate error response.
    b. For HEAD requests:
        i. Respond with a 400 error without a body and return true if:
            1. Bad is true
            2. Context-length's value exists
        ii. Otherwise, call headfile function with the filepath and **conn_fd**. If error occurs with request, send an appropriate error response.
    c. For GET requests:
        i. Respond with a 400 error with a body and return true if:
            1. bad is true
            2. Context-length's value exists
        ii. Otherwise, call getfile function with the filepath and **conn_fd**. If error occurs with request, send an appropriate error response.
8. Return false to caller if this point is reached

- void handle_connection(int connfd)
  This is responsible for handling the connection to the client.
    1. Will Call process_request to handle the request data sent unless one the following happens:
        a. (Does not apply for the first request of established connection) The last request has a 400 error
        b. the connection was closed by client
        c. If there is no more data that will be received from the client
    2. Close the connection with the client and the server reverts back to listening for a new connection.

These functions are called after the request head is processed, the body of the request is still unread.

- int headfile (char* filepath, int conn_fd)
  This function is responsible for sending the file size in response to head request if no errors are encountered.

    1. This function will first check if there are errors with the file path provided before proceeding in reading and will return an error code if there is an error detected:
        a. 403 - Lacking permission to access the file
        b. 404 - The file does not exist
        c. 403 - The path refers to a directory
        d. 500 - Other errors
    2. (with a 200 code)The response header will be sent until the Context-Length's value.
    3. The file size will be measured then sent to the client.
    4. Any remaining "\r\n" will be sent.

5. Return 200 - indicating success to the caller

- int getfile(char* filepath, int conn_fd)
  This function is responsible for sending data from a file on the server to a client as a response to a get request if no errors are encountered.

  1. Will call headfile(**filepath**, **conn_fd**) to handle error checking and sending a header to the client
  2. All file data will be sent to the client as the response's body
  3. Return 200 - indicating success to the caller.

- int putfile(char* filepath, int length, int conn_fd)
  This function is responsible for receiving data from the client to be saved into a file if no errors are encountered. If the file exists and can be accessed, then the file will be overwritten.
  1. This function will first check if there are errors with the file path provided before proceeding in reading and will return an error code if there is an error detected:
     a. 403 - Lacking permission to access the file
     b. 403 - The path refers to a directory
     c. 500 - Other errors
  2. Will check if the connection was closed by client
  3. The file object will be opened or created
  4. Data received from the client will be read and written to the file object until **length** bytes are processed or the connection is closed by the client.
     a. Return 0 - indicating closed connection
  5. Return: 200 - indicating success to caller or 201 - indicating success with file creation to the caller.

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 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)
  If the program detects that the client has disconnected prematurely, then it will not execute the request and return to listening for a new connection. This extra concern was not present in my implementation of shoulders because all data that was accessed were locally stored. Whereas in this assignment, we are receiving data through a network and that introduces possible connectivity problems including connection lost.

  .
- 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?

My design distinguishes between these types of data by using the blank line right before the body. When parsing the request, I look for “\r\n\r\n”. “\r\n” is used to indicate the end of a line. The first “\r\n” is from the line before and the following “\r\n” is the blank line that appears before the body of the request.