Software Design Document

Simple Http Server

Authors:

Scott Melero

Korbie Sevilla

Program Structure

Data Design

- #Define a buffer size of 4096
- Sockets to manage the connection.
- httpObject struct to keep track of the http request pieces
    - Char for the *Method, filename, httpversion, header, data buffer*
    - Ints for *content_length, statuscode, filename file descriptor*

Pseudocode Outline

- The server is run with an address to bind to and a port number to listen on.
    - The address pointer is run through a function,. getaddr(), to convert it into an unsigned long int. This function also checks if the given address is valid.
    - The port number is also specified at the command line and is converted into an int. If there is no port number specified, default to 80.

- Create a new struct “sockaddr_in server_addr” to hold information about the server.

- Create the server socket(AF_INET, SOCK_STREAM, 0). If there are any errors creating the socket, print them.

- Bind() the server address to this newly opened socket. And Listen() on the sockets for any incoming connections.

- In an infinite loop, create a new client socket by calling accept() on the server socket, with the client address and address length. This is so that we can keep accepting client requests until the server is closed, or runs needs to exit.

- After the client connection has been accepted and curl has sent the request to the server, we start our 3 step execution: Read the request, process the request, and construct a response. Each step will take the clients socket, as well as an instance of our httpObject struct we will call 'message'.

- **read_http_response()** Needs to parse our header into the httpObject struct.
    - In a loop, keep recv'ing so we can get the whole request. Since our header will not exceed 4kb, we can read the entire header into a buffer of size 4096 bytes. In this loop we need to:
        - Detect end of file by using strstr() to check if our buff has the newline sequence. If so, we can start parsing our message by using sscanf() to pass the request data into our messages struct.
        - We also check to see if our header contains a content length. This means that we are doing a put request, and will use this int later on when we are creating the new file.
        - break
    - After we have parsed the header, Remove the / from the file path. Loop through the message[0].filename char and shift each bit back one. During this loop, we run isalnum() on each character to determine the string's legality. If the string is formatted incorrectly, return a status of 400 Bad Request.

- **process_request()** will figure out what actions need to be taken based off of the values in the message struct.
    - Use strncmp() on the message[0].method to find out what kind of request we got. If it is not one of GET or PUT, return a status of 400 Bad Request.
    - If GET:
        - This means that we need to print the contents of filename to the client socket, and return the number of bytes that were sent.
        - Use open() to make a new file descriptor.
            - If there are any errors, check to see which error we received by comparing errno to various open() error codes. (2 = file

not found, 13 = permission denied, all others = internal server error)
        - On successful open, use fstat() to find and set the content length. And set the status code to 200 OK.
        - Keep the fd open so we can print it in the next step of this process
    - If PUT:
        - Use open() to create a new file if it doesn't exist, or truncate it if it does.
            - If there are any errors, check to see which error we received by comparing errno to various open() error codes. (2 = file not found, 13 = permission denied, all others = internal server error)
        - On a successful open, set the status code to 201 preemptively. In a loop, keep recv'ing the data sent from the client and write() it into the fd that we opened. Make an int to keep track of the total bytes read during the loop, and keep recv'ing while byteCount is less than the content length sent by the client.
            - If any any point during the loop, recv returns a 0 and byteCount < content_length, this means that the connection was closed before we could finish out PUT. in this case, return a status of 500 Internal Server Error, set the content_length to byteCount, and exit the loop.

- **contruct_http_resposne()** will send() the formatted response back to the client socket.
    - We need to format a response based on the results of process_request. We do this by checking the Status code that we saved into the message struct.
        - If 200, Print the "200 OK" message along with the content length and data that the client requested. We do this by send()'ing, to the client socket, the data read in from the fd we opened. Close the fd and return
        - If 201, then we were able to successfully PUT a file on the server, and we need to let the client know that the file was created. We send the messages "201 Created" and a content length of 0.
        - If 400, the server received a bad request. This could be a bad method, an illegal file name, etc. If this is the case, we need to send the client "400 Bad Request" and a content length of 0.

        - If 403, permission was denied when opening or writing to a file, so we send the client “403 Forbidden” and a content length of 0.
        - If 404, then filename could not be found. In this case, then we need to send the client “404 File not found” and a content length of 0.
        - If 500, then there was an internal server error. I.e the server did not complete a put, or if opening the file did not result in a 403 or 404. In this case, We need to send the client “500 Internal Server error” and a content length of 0.

- After curl has received an adequate response, then we can close the client socket, and get ready to receive the next request.

System Components

- *Main()*
    - Gets shit started
- *struct httpObject:*
    - Keep track of all components necessary for an HTTP message.
- *read_http_response():*
    - Read the response from the client and parse out the data. Fills out the pieces of our httpObject struct, as well as inspect the file in question.
- *execute_get():*
    - Perform the GET request for the server. Fills in the appropriate struct variables and sets the proper status code based on how the operation went.
- *execute_put()*:
    - Perform the PUT request for the server. Creates or truncates a new file with the data sent by the client. Fills in the appropriate struct variables and sets the proper status code based on how the operation went.
- *process_request():*
    - Depending on the parsed request from the client, determine if the request is for GET or PUT and will call *execute_get()* or *execute_put()* as needed. If the method is neither, then return a status of 400.
- *contruct_http_resposne():*
    - Based on the results of the processing, construct and send() the appropriate response to the client socket.

Description of the User Interface

- Server terminal:
    - The user will need to interact with this program through a terminal by establishing a server connection via “./httpserver address port# “. The address must be specified, and an error will be returned if the given arg is not valid.
    - The port number is optional. If it is not defined, the server will default to port 80. You need to execute the code as a super user in order to run on port 80 “sudo ./httpserver address port# “
- Client terminal:
    - Once a server is established, the use will then be able to request a file/application from the server. All files names **MUST** be 10 character ascii strings containing only the char’s [0-9] & [a/A - b/B]. To perform an op on the server, the client can run the following for their requests:
        - curl -T file http://localhost:8080/filename, to PUT a file onto the server directory
        - curl http://localhost:8080/filename, to GET a file that is in the server directory.
    - The server will send back a response to the client terminal containing a status code and content length.

Testing

Classes of tests

- Start with these curl commands for testing:
    - * curl -T t1 http://localhost:8080/0123456789 -v, for **PUT**
    - * curl http://localhost:8080/0123456789 -v, for **GET**
- Simulate the following status codes:
    - 200 (OK): Returned from successful GET.
        - Run the curl command to get a valid file on the server and wait for the server to send the content back
    - 201 (Created): Returned after a successful PUT.

        - Run the curl command to PUT a valid file on the server and wait for the server to respond. Check to see if the file was successfully created and overwritten.
    - 400 (Bad Request): Improper request/bad file
        - Run curl to a HEAD response. Since this is not a GET or a PUT, the request is bad and ther server should respond as such.
        - Run a curl GET or PUT with a file that has bad characters in it and/or does not have a strlen() of 10.
    - 403 (Forbidden): permission of file action denied.
        - Create a file that does not give the server read and/or write permissions. Run a curl command that operates on the forbidden file. Since the server will not be able to interact with the file, it should respond accordingly.
    - 404 (Not Found): file cannot be found.
        - Operate on a file that does not exist. Run curl on a file that does not exist on the server directory. Since the server cannot find the specified file, it should respond as such.
    - 500 (Internal Server Error): server is closed before request is complete.
        - Something went wrong with the server, and the operation cannot be completed.
            - When we open a file, check to see if it is a 403 or 404. If not, then return 500.
            - Close the server in the middle of a PUT, before it has had a chance to recv/write the rest of the bytes.

Expected software response

- The server will be able to parse simple HTTP headers. Essentially the client sends a request to the server that either asks to send a file from the client to server (PUT) or obtain a file from server to the client (GET). If anything goes wrong with the operation, we need to capture that as a status code and send the appropriate response.

Performance bounds

- The server is single threaded and can only handle one client at a time as per the asg specification