Assignment 1 Design Document

Chris Sclipei

CruzID: csclipei

CSE 130, Fall 2019

1. **Goal**

   The goal for this assignment was to implement an HTTP server to emulate the interactions between a server and a client using simple requests like PUT and GET. We would copy the contents of a file on the client side and store it on the server with a 27-character ASCII name prompted by the client. Similarly given the name of the file we could retrieve the content of a file on the server side and print it out to the *stdout* on the on the client side followed by a response code.

2. **Assumption**

   For this assignment, I am assuming that *curl* will handle the client side so I don't have to produce it myself. I am also under the assumption that the user will input proper PUT and GET requests and that the files being sent or retrieved are text files. I will also make the assumption that the only two acceptable arguments for picking an address is either a valid IP or localhost. I will also respond to the requests with the following responses:

   - 200 – OK
   - 201 – Created
   - 400 – Bad Request
   - 403 – Forbidden
   - 404 – Not Found
   - 500 – Internal Server Error

   In addition to the response codes, headers are expected to be less than 4 KiB but the content/data can be much longer.

3. **Approach**

   For this assignment, I will first start off by using the starter code that provided for us by the TA and some of the resources online. From there, I will look up how the commands, PUT and GET, work and figure out how to send and receive data from the client to the server and vice versa. Going off the rubric where it states that the name of the file needs to be exactly 27 ASCII characters long with characters in the range of:

   i. A-Z
   ii. a-z
   iii. 0-9

iv. (_) and (-)

I can handle by using a regular expression to cover the cases in which it doesn't match the required name prompted by the user. But before I do that, I'll have to parse the content header to retrieve the necessary requests and then the name of the file so I can write it to a file on the server side or print it to the *stdout* on the client side. Then following the request make from the client, the server will have to send back a response based on the request made, which will be handled by the 6 responses listed in *Assumptions* section. I will also need to handle the case in which a user specifies an IP address and a port to ensure that connection between the server and client is valid to allow for data transfer.

4. **Design**

To start, I will need to use the following system calls to establish a connection between the client and server:

- *Socket*
- *Bind*
- *Listen*
- *Accept*
- *Connect*

The user also has the option of specifying an IP address and port when calling on *httpserver*, which will prompt the server to listen on that port with the desired IP address, which will be used as data in the *sockaddr_in* data structure. If no port is specified, it will default to port 80, which will need to be used with administrative privileges. Then using *curl*, I will issue either a PUT or GET request which will then be received using the *recv* call by the server. The server will then load the content header into the buffer, which will then be parsed using *sscanf* to get the request and name of the file for the data to be put in on the server side. Using *regcomp* and *regexec*, I can use a regular expression to make the file name satisfied the requirements. Following a PUT request, I will use another *recv* call to retrieve the content in the file issued to be sent to the server and write it on the server side's directory into the file name specified by the header. Following a GET request, I will use *read* to read form the desired file specified in the content header and then use *send* to write to the *stdout* on the client side. I will also include the content length of the file by using *read* to return the number of bytes read from the file. Following this outline I will have the following order for calls:

i. Check the arguments the user inputted to match IP address and port number to connect client and server.
ii. Keep the server open in order to enable multiple requests without closing the server.
iii. Use *recv* to retrieve the content header in order to parse the request
    1. If GET then
       *open* file specified by header

*read* file from file name listed in the content header
*write* to *stdout* the content length of the file on the server side.
*send* back to the client to be printed to *stdout*
*send* back response based on completion or error

2. If PUT then
   check to see if file exists
   if true: truncate it to overwrite existing data
   if not: create and write contents in it
   *recv* to get the content in the file to write to
   *send* back a response code to the client signaling completion or error

iv. Close socket and open a new one.

5. **Pseudocode**

Here's some pseudocode for code that appears after the skeleton code that was given to us.

*recv(socket, buffer, size of buffer)*
*sscanf(buffer)* to parse requests and protocol
*regexec(*regular expression to check against any issues)
**if** *regexec fails* **then**
    return error code
**end if**
compare parsed header and check for which request is given
*open(fd* of the file name requested from the client)
**if** *GET* **then**
    *read(fd* given by the client)
    *send(*content back to the client)
    *send(*response code)
**end if**
**if** *PUT* **then**
    *recv(*rest of socket data for the content)
    *write(fd* that was specified by the client)
    *send(*response code back)
**end if**