**CS3D3: PROJECT 1 REPORT**

Aniket Agarwal 17317437

**1 MES-REQ-RES Implementation**

The http\_message class is an abstract class that is extended by http\_request and http\_response classes. These classes are being used to handle the encoding and the decoding of the Http Messages. The classes are defined in mes-req-res.h and mes-req-res.cpp. The http\_request class contains several functions. Functions of the http\_message class are:-

1. decode\_first\_line (): An abstract function that decodes first line of http messages.
2. encode () : An abstract function to encode the http messages.
3. encode\_header() : Encodes the http messages as a vector of chars.
4. decode\_header\_line() : Given a vector of bytes, decodes the header line and store in a map return 0 if success and -1 if error.
5. set\_header() : Sets a header
6. set\_payload() : Sets a payload
7. set\_version() : Sets the http version
8. get\_version() : Returns the http version either 1.0 or 1.1
9. get\_payload() : Returns the payload as a vector of bytes
10. get\_header() : Returns the value of the header as specified by the key

The http\_request class inherited the http\_message class has the following functions:

1. decode\_first\_line() : It decodes first line of http messages.
2. encode() : It encodes the http messages as a vector of chars.
3. set\_url() : Sets the URL of the http request
4. set\_method() : Sets the method of the http request
5. get\_url() : Returns the URL of the http request
6. get\_method() : Returns the method of the http request

The http\_response class inherited the http\_message class has the following functions:

1. decode\_first\_line() : It decodes the first line of the http messages.
2. encode() : It encodes the http request as a vector of chars.
3. set\_status() : Sets the status of the http response
4. set\_description() : Sets the status description of the http response
5. get\_status() : Returns the status of the http request
6. get\_description() : Returns the status description of the http response

The difference between the http\_request and the http\_response classes is in their implementation of the function decode\_first\_line (), due to the differences in the first line of each type of http message. The most important function in the http\_message class is the decode\_header\_line function, which takes a string or a stream of bytes and extracts the header and all its values. These are then placed in a map and that map headers to values.

These classes are being used to implement the http messages in my web server and web client. The server and the client build the mes-req-res from the scratch from the stream of bytes with decode. The messages are also set as a stream of bytes after calling the encode function.

**2 Web Server**

It is a basic web server that supports HTTP 1.0 with timeouts and works as follows:

**Listening for Connections:** The server creates a socket that listens to the hostname and the port that has been specified in the field in the arguments. If no hostname and port are specified then the server listens to localhost at port 4000. The server then serves files from the directory that has been specified in the argument list, otherwise it serves from the current directory.

**Accepting Connections:** When the web server receives a new connection, it creates a child process with the fork() system call. The child process then handles serving the requested file. Because of these parallel processes, the web server has the capability to handle the concurrent connections. The server supports three different status codes: 200 OK, 400 Bad Request, 404 Not Found. If the server does not receive data from the client connection for five seconds and the whole HTTP request has not yet been transmitted then the server will time out and send a response with the 408 Request Timeout status.

**Parsing Request:** The first line of the request is parsed to get the requested file information. The remaining headers are decoded line by line the method which is basically the same as the one used in the client, which is described below.

**Sending Response:** After parsing every line of the request, the server then attempts to send the requested file. The URL of the request is parsed to find the first singular slash. Everything after the first singular ‘/’ indicates the file requested. The server attempts to open the file, and sends a 404 Not Found response if the file could not be opened. After opening the file, the server sends a 200 OK response and then the file if possible. After the response has been sent, the connection is closed.

**3 Web Client**

The client side works as follows:

**Parsing URL:** The client first parses the URL argument to obtain the host name, port number and the requested file name. Using these parameters I created a TCP socket to connect to the server. The http GET request message is always the same no matter the server, with only the URL and host name being changed depending on the use.

**Read Response from Server:** The client reads the response from the socket using the “recv” function, storing the values in a 4096 byte buffer. I decode the response data into two sections the headers and the payload which are then stored in an http\_response object. These two sections are then parsed using the following:

1. Each header is separated by a CRLF (\r\n) and I utilize this property to properly decode them. Initially I parsed only the first header which contains status information about the request by processing data from the beginning of the buffer until the first CRLF.

From there I moved the iterator forward by two bytes so that it is located at the beginning of the next header, move the second iterator to the next CR and then decode the line. I repeat this process until I decode all headers, using “recv” each time I reach the current buffer data.

* If the end of the buffer is reached before the header portion of the message is completely received, then the first iterator is set back to the beginning of the buffer.
* If the “recv" function does not receive any information for more than 5 seconds, it times out and closes the socket.
* The client implements some logic so that if there are two consecutive CRLF bytes, it knows that any more data afterward will be a part of the payload.

1. After obtaining the entire HTTP Response header fields, the client looks through the headers of the message; If the status of the response is `404' or `400,' the client is informed of the error, nothing is downloaded, the socket is closed, and the program finishes. If the status is a success, the client will use the file name and create a file with that name using ofstream. A special case is implemented to change the file name of `/' into `index.html.'
2. I create a payload buffer of 256KB and read from the socket in 256KB chunks and write to the file. I loop this using “recv" to read 256KB at a time and write it to the file. If the “recv" call returns 0, this means the socket has been closed and the whole file has been sent. If the”recv" call does not receive any data for five seconds before the entire message has been received, the client will timeout and close the socket.

The socket is then closed and the program finishes.

**4 PROBLEMS ENCOUNTERED**

For the web server, the biggest issue that I encountered was handling edge cases when parsing the URL of the request. I had a lot of bugs (corrupted downloading, file not being created, etc.) stemming from an incorrectly parsed URL. Because the request may not necessarily have http:// at the beginning, I had decided to parse the URL by searching for the first singular slash and keeping everything after it. I ran into issues when the first singular slash was the last character or if the slash was not present. I first tried to use sscanf to find the first slash; however, I ran into bugs when the slash was at the end of the string. I eventually implemented it iteratively with for loop and some if statements.

Another problem I faced was with handling large files. Originally I thought that HTTP/1.0 required the Content-Length header, and I used that information to create the size of my payload buffer. However, for large files (such as 100MB or larger), this would create an enormous vector of chars. As a result, the kernel would kill my client for using too much memory. To solve this issue, I decided to read from the socket and write to the file in chunks of 256KB. I determined that the file was finished sending then the “recv” call returned 0, meaning that the socket was closed.

Also, since I were unable to complete it on time my program is not fully tested.

**5 TESTING**

Firstly I built my web server, I tested it with the web browser. I connected by forwarding some private IP addresses and had my server listen to that IP address and port number. Then I tested it with text files and html files. I expanded it to include jpg, png, and pdfs as well. I defaulted my content type to octet-stream so that just in case the file was not recognised by the client it could still be downloaded by the client.

I tested my client by creating different types of files (.html, .txt, .jpg, .png, .pdf, .gif) in my server directory and then downloading them from different directories and sub-directories.

**6 REFERENCES**

1. <https://msdn.microsoft.com/en-us/library/windows/desktop/ms737530(v=vs.85).aspx>
2. Also referred from stack exchange, geeksforgeeks, github and many other random sites that I could find or that showed us what I were looking for. I don’t remember exactly all the sites.
3. Also referred from sample codes given in the assignment.