Assignment 4 (Simple HTTP Client-Server)
Due: February 12, 2023, 2 pm

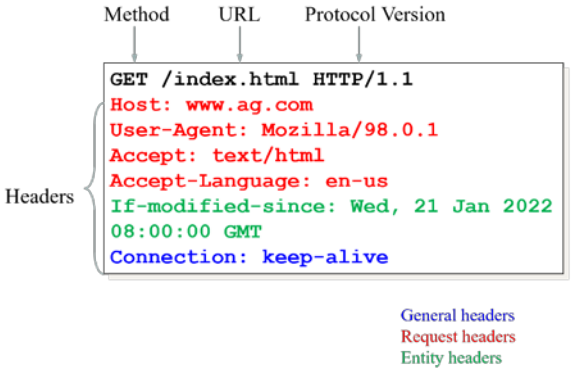
In this assignment, you will write a simplified TCP concurrent Hyper Text Transfer Protocol (HTTP) client and server, namely, *MyOwnBrowser* and *MyOwnHTTP*.

Basic HTTP is a text-based protocol, where a client sends a text command and receives a text response. The general format of a HTTP request/response is given below:

Request/Response line
General headers
Request/Response headers
Entity headers
One blank line
Entity body

where the Request/Response line contains the command, the headers are text based headers of the form "<header name>=value" with one header field in each line, and the Entity body is the actual content if applicable (for example the content of the actual file downloaded). Header can be of variable size and the one blank line before the Entity body says where headers end and the content may begin. The general headers are headers that are applicable to both requests and responses, the request headers are applicable to requests only and response headers are applicable to the responses only, and the entity headers are applicable to the actual content sent/received if any. Each response also sends a well-defined status code. Each header field appears in a separate line, with the format "field name>: <comma separated field vaues>".

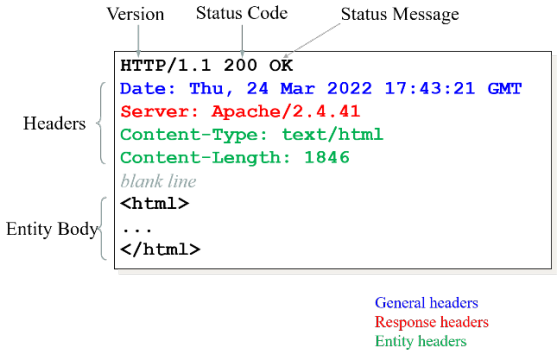
As an example, consider the following HTTP request.



Line 1 is the command line with command GET, red lines are request headers, green lines are entity headers, and the blue line is the general header. Note that they need not be in order. This command is for getting the file www.ag.com/index.html following the protocol HTTP version 1.1, it is issued from a Mozilla browser with the shown version no., the request will accept text files with extension .html in language US English. The content is accepted only if it is modified after the given date. Finally, the last

header field says to keep the http connection open after getting the response (maybe because another request will be sent soon).

Similarly, the following is an example of a response received.



It says the request was satisfied successfully (status code 200) on the date/time shown, the webserver was an Apache server with the version shown, the content type is html and has 1846 bytes The entity body part is the actual content of the html file.

In this assignment, you are supposed to implement a basic HTTP client and server. A major part of this assignment is to read RFCs (Request For Comments) for HTTP 1.1 (the version we will use), which are definitive documents that define a protocols syntax and semantics. HTTP supports a large number of headers to enable a wide range of functionalities. Don't get scared, you will implement only a very very small part of it. But you will implement it exactly as per the HTTP specification so that your client can connect to any http server and vice-versa to support at least the basic things you implement (well, you will not have authentication, so you will be able to connect only with servers with http:// and not with https:// usually).

To find details of HTTP commands, headers, and their values, start with RFC 9110 (just search in google for it). It obsoletes and modifies a set of earlier HTTP RFCs (RFC 7230-7235). There is a very old RFC, RFC 2616, that, though obsolete, you can also start with as it is an easy read and many things stay the same. The RFCs are detailed, you will not understand many things in it, and you do not need to. But you will get what commands, requests, responses, headers field etc. are, and their syntax. You can also look up textbooks and other internet sources in addition (for some, you will probably have to, do ficus'ed google search for the particular command/header if needed). Once you understand them, it is just a matter of forming the messages with the right syntax and transferring; you already know the other details like sending and receiving variable length messages etc. Please spend some time on this.

Your client and server should support the two basic HTTP commands only, GET and PUT. The GET and PUT commands are for downloading/replacing document(s) from/to server. You need to learn the exact format for the commands and the header fields you are asked to implement.

In this assignment, we will assume that we will download and upload only html, text, and pdf files.

HTTP Client (*MyOwnBrowser*):

- *MyOwnBrowser* is executed in the client machine. If successful, it waits on "*MyOwnBrowser*>" prompt for command.

- Three command-line operators are supported: GET, PUT and QUIT. The GET and PUT commands are for downloading/replacing document(s) from/to server. After every GET or PUT command, the prompt returns to *MyOwnBrowser*>. On entering QUIT, the program exits from *MyOwnBrowser*> prompt.
- The GET command implements a simple version of the corresponding GET command in HTTP:
    - Opens a TCP connection with the http server.
    - Download/retrieve document from the http server.
    - Format: *MyOwnBrowser*> GET {URL}
    - URL specifies the url of a web document. The default port number is 80, you do not need to mention the port if it is 80, otherwise you need to specify the port in the format shown below. Example:
        i. Example: *MyBrowser*> GET http://cse.iitkgp.ac.in/~agupta/networks/index.html
        ii. *MyBrowser*> GET http://10.98.78.2/docs/a1.pdf:8080

      Since many of you may not know yet how to get the IP address given a DNS name, you may assume that the IP address form (as shown in (ii)) will be given, so you can parse out the IP from it to connect.
    - On receiving the document from the http server, the client opens the document. You should try to open each file type by using appropriate applications. You have to support the following document types and applications: *Adobe Acrobat* for PDF files; *Mozilla/Chrome* for HTML page, any app you want for *jpeg* files, *gedit* for all others, to have a real feel (use fork and exec commands learnt in OS).
    - Closes the connection after receive.
- The PUT command implements a simple version of the corresponding PUT command in HTTP and is used to upload/replace document to the server.
    - Opens a TCP connection with the http server.
    - Uploads/replaces a document at the http server.
    - Format: *MyOwnBrowser*> PUT {URL} <filename>
    - URL specifies the url of a web document. The default port number is 80. Example:
        i. *MyOwnBrowser*> PUT http://cse.iitkgp.ac.in/~agupta/compsyslab assignment.pdf
        ii. *MyOwnBrowser*> PUT http://10.98.78.2/docs:8080 biodata.txt
    - Closes the connection after receiving the response.

  Note that opening and closing a connection for each GET/PUT is not necessary in practice, but we will do so for simplicity.

HTTP Server (*MyHTTP*):

MyOwnHTTP receives the "GET" or "PUT" requests from *MyOwnBrowser*. It then parses the input stream, extracts the fields, and then sends back appropriate response. In case of error, appropriate error numbers and messages are returned to the client.

MyOwnHTTP maintains an Access Log (AccessLog.txt) which records every client accesses. Format of every record/line of the AccessLog.txt: <Date(ddmmyy)>:<Time(hhmmss)>:<Client IP>:<Client Port>:<GET/PUT>:<URL>

All messages should be in proper HTTP format. Following header fields should be supported for this assignment. Other fields are to be ignored (note that ignored does not mean they will not occur in any message; just that they need not be handled if encountered). Some of these fields have multiple possible

values, you need to handle only the values for each and take action as per details given below (sub-bullets are header fields and sub-sub-bullets are the values to be handled if there are multiple values possible). For some, no action is specified; it means you should read the RFC's and figure out the correct action to take. Some fields will get in both GET and PUT commands, some in only one, you need to figure out. In short, before you write any code, read the relevant parts of the RFCs well.

- Request Message (some are relevant for both GET and PUT, some for only GET and some for only PUT. You need to think and put the appropriate headers depending on the request type. You need to put all headers that apply.)
  - Host
  - Connection
    - close
  - Date
  - Accept:
    - text/html if the file asked in the url has extension .html
    - application/pdf if the file asked in the url has extension .pdf
    - image/jpeg if the file asked for has extension .jpg
    - text/* for anything else
  - Accept-Language: en-us preferred, otherwise just English (find out how to specify this)
  - If-Modified-Since: should always put current time – 2 days
  - Content-language: set to en-us always
  - Content-length
  - Content-type: fill in as per the content type as specified by Accept header field values earlier
- Response Message (same comment as request messages)
  - Expires: always set to current time + 3 days
  - Cache-control: should set to no-store always
  - Content-language: set to en-us
  - Content-length
  - Content-type: fill in as per the content type as specified by header field values earlier
  - Last modified

HTPP responses also contain a status code and a message for each code. You will need to send/handle only the following status codes in your response messages.

- Status Codes: 200, 400, 403, 404

You should read what the fields and the status codes mean and send/interpret them accordingly. The client should print an appropriate message for the status codes defined above. For any other status codes, which must be some other error, the client should print a generic message such as "Unknown error" along with the code received.

Timeout to receive a response to be used in client is 3 seconds, after which client should print a message, close the connection, and come back to the prompt to wait for the next GET/PUT command..

Both the client and the server should print out all the requests and responses received (method + all header fields, but no entity body).

You should submit two C files, the *MyBrowser.c* and *MyHTTP.c*.