PROGRAMMING ASSIGNMENT – 2

Due: 10/16/2016 11:55 pm

NOTE: This assignment is NOT a group assignment, but an individual assignment.

Objective:

To create a HTTP-based web server that handles multiple simultaneous requests from users.

Foreword:

By this point of time, you should know the basics of C and socket data transmission through C/C++. This programming assignment is considerably going to be considerably harder compared to your first PA. You would have to begin early, else you wouldn't be able to complete it on time. Please take time to thoroughly read and understand the document. Only concepts, directions and objectives will be provided in this document. The assignment will teach you the basics of network programming, client/server message structures, and issues in building high performance servers. You must work *individually* on this assignment; we will check for plagiarism.

Background:

In this assignment, you will learn the basics of socket programming for TCP connections in C: how to create a socket, bind it to a specific address and port, as well as send and receive a HTTP packet. You will also learn some basics of HTTP header format. You will first develop a web server that handles one HTTP request at a time. Your web server should accept and parse the HTTP request, get the requested file from the server's file system, create an HTTP response message consisting of the requested file preceded by header lines, and then send the response directly to the client. If the requested file is not present in the server, the server should send an HTTP "404 Not Found" message back to the client. Similar to the 404 Not Found error, you will have to handle other error codes in the 4XX and 5XX series. They have been listed in detail towards the end of the document. Please handle all the errors that you have been asked to handle. You will then extend the functionality of the server to accept multiple requests from clients (either simultaneously from different clients or multiple parallel requests from the same client). The web server would have to honor every request and send the correct response message to the corresponding requests.

Q.) What is a web-server?

A.) A web server is a program that receives requests from a client and sends the processed result back to the client as a response to the command. A client is usually a user trying to access the server using a web browser (chrome, Firefox).

The HTTP request consists of three substrings – request method, request URL, and request version. All three parts should be separated by one white space.

The request method should be capital letters like “GET”, “HEAD”, and “POST”.

The request URL is a set of words which are separated by “/” and the server should treat the URL as a relative path of the current document root directory.

The request version follows the rule like “HTTP/x,y” where x and y are numbers.

Here is an example to make things more clear:

If you type an URL like http://www.w3.org/Protocols/rfc1945/rfc1945.txt in a web browser, the web browser will send an HTTP GET command to the server http://www.w3.org/ after establishing a TCP connection with port 80 on the server. In this scenario, the format for the HTTP GET command received at the web server is:

```
GET /Protocols/rfc1945/rfc1945.txt HTTP/1.1
```

Based on the rules, we can identify three substrings like the following:

Request Method: GET

Request URI: /Protocols/rfc1945/rfc1945.txt

Request Version: HTTP/1.1

Deliverables:

Usage of pre-written C or C++ includes/libraries for HTTP server or socket connection or pipelining other than standard system modules is not permitted.

Your server should start without any arguments: ./webServer

Your server should be running in a Forever running loop once it has started. You are expected to exit out of the server gracefully (Either finish all the pending requests, timeouts and then exit or just exit out bluntly closing the listen socket properly though) when pressing the escape sequence (Ctrl + C or any key(s) of your choice).

In this assignment, the web server will have a **configuration file** which will be in the same directory as the server file. This directory will have another directory inside called the **document root directory**. The document root directory would contain files (and subfolders, if any) with the extension ".html" ".htm", ".txt", ".jpg", ".gif", and ".png" etc. These are the minimum requirements and you can choose to add support to other file formats (There won't be any additional efforts if you get one case (say .jpg) to work). When the web-server receives a HTTP request from a client (from a web-browser such as chrome or telnet) for a particular file, it should open the file from this document root and then send the file back to the client with proper header information. Header information typically contains the required headers which should be sent along with the file so that the web-browser can understand the HTTP request being successful and the result being received. In all cases, the names of files in the client's request URL should be interpreted in the context of the document root directory. If the server interprets the request (HTTP GET) and the requested file exists on the specified location, the server needs to reply with an appropriate status code followed by **Content-Type** and **Content-Length** headers. The file content should be separated by a **carriage-return-line-feed** from the last header. For example, when the client requests an existing ".html" file on the server, the server replies with a 200 status code.

The Status Code "200" indicates that the requested file exists and the server is going to send the processed data (the requested file) back to the client. The string next to Status Code "200" conveys the information that the requested document follows in this reply. Typically, it is 200 OK.

```
HTTP/1.1 200 OK
Content-Type: <> #Tells about the type of content and the formatting of <file contents>
Content-Length: <> #Numeric length value of the no. of bytes of <file contents>

<file contents>
```

The server will open and retrieve the contents of the file which was requested in the request url. If the file is of type "text/html"; it will parse the html for embedded links (such as images, css style sheet, javascripts) and then make separate connections to the web server to retrieve the embedded files. For example, if a web page contains 4 images, a total of five separate connections will be made to the web server to retrieve the html and the four image files. HTTP/1.0 and HTTP/1.1 protocols are what you will be supporting first. HTTP/1.0 is described in RFC 1945. The client browser will do the multiple requests automatically and nothing needs to be done in the server other than sending correct responses.

Next, add simple HTTP/1.1 support to your web server: support persistent connections and pipelining of client requests. Pipelining is an extra credit task. It is optional. You will need to add

a logic to your web server to determine when it will close a "persistent" connection - say 10 seconds. That is, after the results of a single request are returned (e.g., index.html), the server should by default leave the connection open for some period of time, allowing the client to reuse that connection socket (print your socket descriptor integer id to observe this) to make subsequent requests. This timeout value **needs to be configurable in the server**. This entire portion is not needed if you are not doing the Pipelining extra credit.

If the client sends a HTTP/1.0 request, the server must respond back with a HTTP/1.0 protocol in its reply. Ensure that this is consistently met in your server program.

Configuration File:

As mentioned earlier, the Web server configuration file named "**ws.conf**" stores the initial parameters of the server which must be read by the server when the server starts running. Also, the Content-Type associated with a filename extension is stored in this file. An example of "ws.conf" file is as follows:

```
#listening port number
ListenPort 9999
#document root
DocumentRoot "/home/saha2618/www"
#Default web page
DirectoryIndex index.html
#content-Type
ContentType    .html    text/html
ContentType    .htm     text/html
ContentType    .txt     text/plain
ContentType    .png     image/png
ContentType    .gif     image/gif
ContentType    .jpg     image/jpeg
ContentType    .jpeg    image/jpeg
#connection timeout – not needed if not doing the extra credit
KeepaliveTime 10
```

This file should be used to modify all configuration parameters of the web server. The server should read from this file to get all the required configuration for its operation in the very beginning of its execution. If you change any values in this file, the server has to be restarted to notice the changes.

Default Page:

If the Request URL is the directory itself, the web server tries to find a default web page such as "index.html" or "index.htm" on the requested directory. What this means is that when no file is requested in the URL and just the directory is requested (Example: GET / HTTP/1.1 or GET /inside/ HTTP/1.1), then a default web-page should be displayed. This should be named either "index.html" and it should be present in the corresponding directory of the Request Url. The default web page and document root directory should be searched in the server configuration file present above. The server can accept and search for the exact file listed on the DirectoryIndex. Remember this case only applies when a folder is requested by the client without providing any specific filename.

Handling Multiple Connections:

When the client receives a web page that contains a lot of embedded pictures, it repeatedly requests an HTTP GET message for each object to the server. In such a case the server should be capable of serving multiple request at same point of time. This can be done using the following ways:

- A multi-threaded approach that will spawn a new thread for each incoming connection. That is, once the server accepts a connection, it will spawn a thread to parse the request, transmit the file, etc.
- A multi-process approach that maintains a worker pool of active processes to hand requests off to and from the main server. This approach is largely appropriate because of its portability. It does face increased context-switch overhead relative to a multi-threaded approach. You can search for fork() calls.

Pipelining: (Extra Credit: 10 points for CSCI4273/5273 and ECEN 5023 students)

As mentioned in the previous paragraphs, the client browser that supports persistent connections may pipeline its requests (i.e., send multiple requests without waiting for each response). The server must send its responses to those requests in the same order that the requests were received. So, if there is a "Connection: Keep-alive" header in the request then a timer has to start pertaining to the socket which had received this request. The server will keep that socket connection alive till the timeout value ends. The server should close the socket connection only after the timeout period is over if no other requests are received on that particular socket. If any subsequent requests arrive on that socket before the timeout period (essentially pipelined requests), the server should reset the timeout period for that socket back to the timeout value and continue to wait to receive further requests. This timer will have a value of what was read in the configuration file. If there was no "Connection: Keep-alive" in the request header, then the socket will be closed immediately; the timer will never start for this

case. For example, if the server received this client request and the message is something like this:

```
GET /index.html HTTP/1.1
Host: localhost
Connection: Keep-alive
```

The response for this request should be something like the following:

```
HTTP/1.1 200 OK
Content-Type: <>
Content-Length: <>
Connection: Keep-alive
<file contents>
```

Note:

If there is no “Connection: Keep-alive” header in the request message, the server should close the socket upon responding and reply with “**Connection: Close**” header line. If there is a “Connection: close” header in the request message, the server should again close the close upon responding and reply with the same “**Connection: Close**” header line.

Example Scenario:

Consider a request is received by the server with the Keep-alive header. At this point a timer is triggered. Now if another request from the same client socket is received after 2 seconds of the first request, the server will process the new request and reset the timer be reset to 10 seconds again. If no other request is received by the web-server within the 10 second timeout period, then the socket connection is closed by the web-server.

Error Handling:

When the HTTP request results in an error then the web-server should respond to the client with an error code. In this assignment, the following status codes should be used:

1. The response Status Code “400” represents that the request message was in a wrong format and nothing can be responded by the server except stating the reason for that error. It sends the client one of following messages depending on the reason:

```
HTTP/1.1 400 Bad Request
<other-headers>
<html><body>400 Bad Request Reason: Invalid Method :<<request method>></body></html>
or
<html><body>400 Bad Request Reason: Invalid URL: <<requested url>></body></html>
or
<html><body>400 Bad Request Reason: Invalid HTTP-Version: <<req version>></body></html>
```

2. The response Status Code “404” informs the client that the requested URL doesn’t exist (file does not exist in the document root). It results in the following message.

```
HTTP/1.1 404 Not Found
<other-headers>
<html><body>404 Not Found Reason URL does not exist :<<requested url>></body></html>
```

3. The response Status Code “501” represents that the requested file type is not supported by the server and thus it cannot be delivered to the client. This is also the appropriate response when the server does not recognize the request method/HTTP version and is not capable of supporting it for any resource. It results in the following message:

```
HTTP/1.1 501 Not Implemented
<other-headers>
<html><body>501 Not Implemented <<error type>>: <<requested data>></body></html>
```

4. All the other error messages can be treated as the “500 Internal Server Error” indicating that the server experiences unexpected system errors; it results in the following messages.

```
HTTP/1.1 500 Internal Server Error: cannot allocate memory
```

These messages in the exact format as shown above should be sent back to the client if any of the above error occurs.

Additional Error handling checklist for your server program:

- Port numbers below 1024 should be rejected by the webserver with an error message.
- Unsupported methods and Unsupported HTTP versions should be gracefully handled.
- Malformed requests should be handled with 400 Bad Request error. Reason is optional.
- If ws.conf is not found, the web server shouldn't start.
- The server should gracefully exit upon pressing the Escape sequence or Ctrl+C key combination.

Supported Methods:

Only **GET** Method needs to be handled in your program. Other methods such as POST, HEAD, DELETE, OPTIONS etc.) need not be handled. Please display an error message 501 Not Implemented when receiving methods which are not supported.

Extra Credit:

POST method support (10 points for CSCI4273/5273 and ECEN 5023 students):

You can run this command either through the browser or through an app (extension) or through a telnet client. When you send the POST request, you should handle it the same way you have handled GET and return the same web page as you return for the GET request (In this case: the requested POST URL). But in this reply webpage, you should have an added section with a header "<h1>Post Data</h1>" followed by a <pre> tag containing the POST data. The POST data is everything in the POST request following the first blank line. This added information must be embedded within your <html>/<body> tags; basically it should be visible on the client browser.

Other methods can remain unsupported through the 501 Not Implemented error code handling.

Earn criteria: Should be able to see the POST data in the server's response along with the requested URL's contents.

Submission Requirements:

Testing the programs are a **compulsory** requirement. You can make functional mistakes, but crashes or hangs are absolutely not accepted. You should also follow the functional paradigm (main() is not your only function). Your code must be organized and clear with comments/explanations for your functions and arguments.

1. Please turn in one zip/tar.gz (**NO .rar or .7z**) file containing the following:
    - web server C/CPP file
    - ws.conf file
    - Entire contents of the document root folder
    - README (.txt)
      The readme file must explain what you have done and how to run it. The documentation does not have to be long, but does have to be very clear. The code that you turn in for this programming assignment must be your own original work and must compile properly under a GCC compiler. Mention how you execute your program and which language you have used in that file.

2. Include **short comments** in your web server code and explain the complex stuffs you have done in code. This will be extremely useful for you when you read your code for interviews.
3. The web server should handle all the error conditions checklist listed above.
4. The code should serve all the file formats mentioned way above and the client should receive these files when requested. You can choose to support as many formats as you want, but remember to update the configuration file as that serves as the bookkeeper of the supported formats.
5. Simultaneous requests would be sent to the server. Both ways: through 2+ clients sending messages to server at the same time; or one client sending multiple requests at the same time. Your server should handle simultaneous requests.
6. Pipeline support should be present in the code with a configurable timer (per ws.conf) if that extra credit is attempted.
7. Usage of any libraries for HTTP server is not accepted.

Testing your Web server:

In order to check the correctness of the server, you have to test your server with any of commercial Web browsers such as Chrome or FireFox or Internet Explorer. It is very helpful if you test the server after making a sample web page that contains some pictures and text.

You can send a request to http://localhost:9999/index.html from your browser and check for response. The response should be ideally displayed on your web browser. If nothing is visible, something is wrong.

Testing pipelining on the Webserver:

In order to check for pipelining on the server, you can run the following command on a shell.

Command to be typed on linux terminal:

```
(echo -en "GET /index.html HTTP/1.1\r\nHost: localhost\r\nConnection: Keep-alive\r\n\r\nGET /index.html HTTP/1.1\r\nHost: localhost\r\n\r\n"; sleep 10) | telnet 127.0.0.1 9999
```

For the equivalent on windows, write a client.c file and include only the above message within double-quotes in a single client socket send.

Idle Response:

```
Trying 127.0.0.1:9999...
Connected to localhost.
Escape character is '^]'.
HTTP/1.1 200 OK
Date: Sun, 27 Oct 2013 17:51:58 GMT
Content-Length: 62
Connection: Keep-Alive
Content-Type: text/html

<html>
    <body>
        <h1>test</h1>
    </body>
</html>
HTTP/1.1 200 OK
Date: Sun, 27 Oct 2013 17:51:58 GMT
Content-Length: 62
Connection: Close
Content-Type: text/html

<html>
    <body>
        <h1>test</h1>
    </body>
</html>
```

Helpful Links:

1. **https://www.jmarshall.com/easy/http/** : HTTP Well explained (One of the BEST reads!)
2. http://www.garshol.priv.no/download/text/http-tut.html : This is a tutorial on understanding HTTP protocol
3. http://www.cs.dartmouth.edu/~campbell/cs50/socketprogramming.html Socket Programming - TCP echo client and server with fork().
4. http://www.binarytides.com/server-client-example-c-sockets-linux/ TCP echo client and server with threads.
5. https://vcansimplify.wordpress.com/2013/03/14/c-socket-tutorial-echo-server/ - This is a code for a simple echo server between a server and client.
6. http://www.binarytides.com/server-client-example-c-sockets-linux/ - Socket programming and multithreading in C

FAQ:

I get an "Address already in use" error when I try and start my server. What is going on?

First make sure you don't have another copy of the server running. If you have recently aborted a server process on the same port number, it can take a while before the OS considers the port free. So you may need to switch to a new number unless you want to wait. This problem will also happen if you have aborted from your server without properly closing the socket that it was bound onto.