Bootcamp Week 3: Simple HTTP server

This week, we will modify our simple echo server of last week to build a simple HTTP server, serving HTML web pages. While your previous client and server were exchanging simple text, we will now make them exchange HTTP requests and responses.

HTTP server specification

Your HTTP web server must parse the data received from the client and construct a HTTP request. It must then generate a suitable HTTP response and send it back to the client over the socket. Before you start, skim through HTTP Made Really Easy to understand the HTTP specifications. This HTTP Tutorial is also a useful reference. You need not implement the complete HTTP specification, but only a subset. Below are some specifications of what all aspects of HTTP you must handle.

1. It is enough if you handle only HTTP GET requests for simple HTML files for now. The URL specified in the HTTP GET request must be read from the local filesystem and returned in the HTTP response. A URL can resemble the path of a directory, e.g., `/dir_a/sub_dir_b/`, or that of a file, e.g., `/dir_a/sub_dir_b/index.html`. If the URL is that of a directory, we will be looking for an `index.html` file in that directory and serve that. You can use any starting root folder for your HTML files. We have provided a sample set of HTML files for you to use in your testing, which are stored in the directory `html_files`. You can use this as the root directory for your server's HTML files.

2. The communication between the client and the server will be through TCP protocol. If you do not have root permissions on your machine to open a socket on port 80, you can use a higher port number like 8080 for the server to listen on. It is enough to support HTTP 1.0 for now, in which the server closes the TCP connection immediately after responding to the client’s request. That is, your server worker thread can close the connection once it sends a HTTP response.

3. The HTTP response returned by the server should return the status code of 200 in its initial line if the requested file is found and can be successfully returned. The server must return a response with status code 404 if the requested resource cannot be found. When returning the 404 error code, this error message must be wrapped in simple HTML headers for the browser to display it correctly.

4. It is not required for the server to parse any of the headers in the HTTP request. For a first implementation, you can also skip returning any of the HTTP headers in the response file, and only return the status of the response followed by the actual HTML content. Once this basic version works, you should try to support the HTTP headers of Date, Content-Type, and Content-Length in the HTTP response. Note that there is a end-of-line sequence (\r\n) present after every header line, and an extra end of line present after the headers and before the start of the body. There is no need for an end-of-line sequence after the message body. To fill the headers, you may use the `stat` function in the C library to get information about a file like its size. You can use functions like `time` to get the current time. It is also fine to return dummy values in these headers for an initial implementation.

If you complete building your server with these simple specifications, you can proceed to support more HTTP functionality as well, e.g., parse more headers, or add support for more content types. (This part is optional.)

Skeleton code

Recall that the HTTP server must read data from the socket, parse the received data to form a HTTP request structure from it, generate a suitable HTTP response structure, convert this response back into a string, and write it into the socket. We have provided some skeleton code for you to get started with the HTTP processing at the server. This skeleton code is only provided as a hint, and you may choose to ignore it and write all the HTTP processing by yourself as well.

The file `http_server.h` defines data structures to store the HTTP request and response at the server. You can add more fields or modify the data structures in any way you want. We have provided you with some starting code to parse HTTP requests and generate HTTP responses in `http_server.cpp`, which you can complete based on the specification given above. In this file, you must fill in code to extract the HTTP request from a string (received on the socket) in the constructor function of the HTTP request. You must fill in code to generate a HTTP response from a received HTTP request in the function `handle_request`. You must fill in code in the function `get_string` to convert the HTTP response structure into a string to write into the socket.

You must now integrate this HTTP parsing into your simple echo server built in the previous week. The worker thread of your server that is handling the client must read data from the client socket, create a HTTP request object from it, call the function `handle_request` to generate a HTTP response, call the function `get_string` on this object to convert the response to a string, and write it into the client socket before closing the connection.

For your server code to compile correctly, you will need to include `http_server.h` in the server file, so that the HTTP related functions and data structures are accessible. You also need to compile the server with the `http_server.cpp` to generate your final executable. You can use a Makefile to compile your server correctly.

Testing your HTTP server

Open your favourite web browser (e.g., Mozilla Firefox), and type `http://localhost:8080/<filename\>` (with a suitable port number and filename as per your implementation) in the address bar to request for the particular file from the server. If your HTTP server works properly, then you should be able to see a corresponding webpage (if the requested URL exists) or a 404 message (if the requested URL doesn't exist).

You can also test your server using the wget command-line tool which is used to download files from the web. If you use the command `wget http://localhost:8080/<filename\> -O <output_file>`, then the file at the requested url should be downloaded and saved in the output_file. Test your server thoroughly before proceeding to the next exercise.

Submission

Push your simple HTTP web server into your GitHub repository.