

A4

Due Dec 6, 2023 by 4p.m.

Points 1

Updates:

Nov 28th:

- toronto.bmp has been more explicitly added to the page
- Clarification of bad request response meaning, to use the function `bad_request_response` (error 400, not 500 or 404, 303 or 200)

Table of Contents

- [Learning Objectives](#)
- [Introduction](#)
- [Setup and starter code](#)
 - [Lab 10 and port number setup](#)
- [Part 1: Parsing the first line of an HTTP request](#)
 - [Understanding the start line of an HTTP request](#)
- [Part 2: String responses](#)
- [Part 3: Running an image filter](#)
 - [Input validation](#)
- [Part 4: Uploading files](#)
 - [Buffering the data](#)
- [Submission](#)

In this assignment, you will build a small webserver that allows users to run the image filters you created in Assignment 3 on custom images using only their web browser.

Learning Objectives

At the end of this assignment students will be able to

- write a program that communicates with other processes across a network using sockets
- parse parts of HTTP requests
- write a program that creates new processes
- read, write, and manipulate plaintext and binary data
- read and add to a medium-sized C program

Introduction

A web server is a program that uses sockets to listen for HTTP *requests*, and constructs an appropriate HTTP *response*. The response often contains a HTML document that the browser renders on the screen, although you will also see other types of responses in this assignment.

Take a look at `main.html` in the starter code to see an example of data that the server will send back to the client when the client sends a request. You can see how your web browser will render this data by using the `Open File` option in your web browser (E.g. Chrome, Firefox) to open `main.html`. Note that when you open the file like this, the dropdown menu beside "Select an image" is empty. That's because the server fills in this image list based on the current contents of the images directory using a bit of JavaScript.

For this assignment, we are simplifying (and hard-coding) some aspects of a web server and the HTTP protocol. In particular, your the server only considers the first line of the HTTP request that contains the type of request (GET or POST), and the *resource* requested.

NOTE 1: We strongly recommend that you only use Chrome or Firefox to test your work on this assignment. We tested the starter code on these two browsers.

NOTE 2: It is important that you kill `image_server` program (especially on the lab machines) when you are not working on it. Although we have tried to make the code safe, we cannot guarantee that there are no security holes in this application.

Setup and starter code

Do a `git pull` in your repository; you will see the starter code under the new `a4/` directory.

Like previous assignments, there's quite a lot of provided code to read through and understand. Here is an overview of the starter code to help you get started.

Source code:

- `image_server.c` is the main program. It contains the central `handle_client` function, and a `main` to actually run the program.
- `request.h`, `request.c`: functions to handle the parsing of requests. `request.h` also defines the key structs used to store data for these requests.
- `response.h`, `response.c`: functions to handle how the server should respond to requests (i.e., what data the server should write back to the client).
- `socket.h` and `socket.c` contain the basic server socket code from lecture. You shouldn't need to change these files.
- `Makefile`: a sample makefile. Running `make` will create and populate the `images/` and `filters/`

directories required by the server, in addition to compiling the `image_server` executable.

Other files:

- `copy`: this is an executable in case you did not complete Assignment 3 successfully. You'll be able to use this executable to test your work on the Teaching Lab servers, but it likely won't run on your machine. Of course, you can ignore this file completely and simply use your own version from Assignment 3.
- `dog.bmp`: a sample bitmap file.
- `main.html`: an HTML file that acts as the "welcome" page for the server. Users can use this page to submit other types of request to the server.

NOTE: Be careful which files you commit and push to your repo. In particular, you should not commit any executable files, image files, or .o files to your repo.

Lab 10 and port number setup

We strongly recommend working on this assignment after completing [Lab 10 \(https://q.utoronto.ca/courses/309775/assignments/1142836\)](https://q.utoronto.ca/courses/309775/assignments/1142836). You can use any code you wrote on Lab 10 for this assignment.

The first thing you should do is **change the PORT number in the Makefile** in the same way as Lab 10; this will help you avoid port conflicts with other students on the Teaching Lab machines.

Part 1: Parsing the first line of an HTTP request

When you type a URL into your browser, it will send a request to the server. The URL to request that the `main.html` page be displayed is `http://localhost:<port>/main.html` if you are running the web browser on the same machine as the server, and if you replace `<port>` with the port number that your server is listening on.

If you want to run the server on `wolf.teach.cs.toronto.edu` and the web browser on your own laptop or desktop, you can use the URL `http://wolf.teach.cs.toronto.edu:<port>/main.html`.

This causes your browser to send a request to the server that looks something like the following: (Note that all lines use the CRLF line endings ("`\r\n`").)

```
GET /main.html HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:56.0) Gecko/20100101 Firefox/56.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

For most of this assignment, we only care about the first line of the HTTP request and will ignore all of the other data in the HTTP request.

Understanding the *start line* of an HTTP request

The image server we are building can respond to different kinds of requests: an initial one to load the main HTML page, a request to upload a bitmap file, and a request to run a filter.

This data is encoded in the first line (called *start line*) of an HTTP request in the format


```
<method> <target> HTTP/1.1
```

where:

- `<method>` is either `GET` or `POST`
- `<target>` consists of a forward slash (`/`) and resource name (e.g., `main.html`).
 - If the method is `GET`, the resource name can be *optionally* followed immediately by a `?` and then zero or more name-value pairs (called *query parameters*), each separated by `&`.
 - Each name-value pair is written in the form `<name>=<value>`; name and values are always string.
- `<target>` does not have any spaces. Moreover, the resource name, and query parameter names and values all do not contain `?`, `&`, and `=`.

Some examples of HTTP request start lines are:

```
GET / HTTP/1.1
GET /main.html HTTP/1.1
GET /my-resource HTTP/1.1
POST /david HTTP/1.1
GET /my-resource?name1=value1&name2=value2 HTTP/1.1
```

(Note: this is a simplified version of the full format of this line. Read more about the full format [here](https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html)  (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html>).

Your first task is to complete the implementation of `handle_client` up to and including the parsing of the first line of the HTTP request according to the format above, including the implementation of `parse_req_start_line` and some helper functions in `request.c`. Note that even though the client buffer is not null-terminated automatically, all values stored in the `ReqData` struct must be null-terminated strings.

We've included a function `log_request` that you can use to check the values of these strings. To debug your work, you can run your server and then visit the url `localhost:<port>/main.html?name1=value1&name2=value2` in a web browser of your choice (where you should replace `<port>` with your port number, configured above). Even though you won't see anything in the web browser, if you go to your running server, you should see the following output from `log_request`:

```
Request parsed: [GET] [/main.html]
name1 -> value1
name2 -> value2
```

Try out different URLs to ensure your parsing is working properly.

Part 2: String responses

Now that you have parsed the request start line, you can begin writing the code that will allow the server to respond to different requests.

Your next task is to add to the implementation of `handle_client` according to its comments so that it does the following:

- When given a `GET` request with resource name `MAIN_HTML` (this is a defined constant), it renders the provided `main.html` page. Any query parameters and the rest of the HTTP request are ignored.
- On any other request, it renders the provided "Not Found" string.

Note that you aren't responsible for writing any of the actual response text yourself; instead, read the provided starter code carefully to understand the functions we've provided, and call them appropriately to complete this part.

To check your work, you should try visiting `localhost:<port>/main.html` and `localhost:<port>/random-page.html` in your web browser. In the first case, you should see a plain-looking webpage with the title "CSC209: Image Filter Server"; in the second, you should see the message "Page not found".

Note: the provided code illustrates two types of responses: `main.html` is rendered as an HTML response, while the "Page not found" message is rendered as a plain text response. Your web browser probably displays them quite differently; cool, eh?

Part 3: Running an image filter

The `main.html` page has two different forms the user may use to interact with the server. The first form allows the user to select an image filter executable and a file name. When the user presses the "Run filter" button, a `GET` request is sent with target with resource name `/image-filter`, and two query parameters `filter`, whose value is the name of the filter to apply, and `image`, whose value is the filename of the bitmap image to process. For example:

```
GET /image-filter?filter=copy&image=dog.bmp
```

Your task here is to implement the `image_filter_response` function and call it appropriately in `handle_client` when given a request. Once you have this working, you should be able to submit a request by pressing the "Run filter" button and download the filtered bitmap image!