NP Assignment1 Report

Using C/C++ to Setup a Simple Web Server

資工三 406410035 秦紫頤

Setup

Components

In the main assignment1 folder there:

- 5 folders: datastore, image, server, doc, css
- 2 files: index.html, NP_hw1_report.pdf

And the description of all the files is below:

- **datastore**: This folder will store the data POST from the browser so if you haven't done anything yet it should be empty. Every time you POST from the browser it should be stored in here.
- **images**: There is only one image **"avatar.png"** in this folder. Which will be shown in my HTML file when opened in the browser
- **server**: There are 2 files in this folder: server.cpp, makefile.
    - **server.cpp**: This is my c++ webserver which can process simple GET and POST request
    - **makefile**: This file is for generating my c++ webserver executable file
- **doc:** There is only one file **"myresume.pdf"** in this folder. Cause in my program there is one clickable link which should direct to this pdf file.
- **css:** Instead here is my **"custom.css"** for my web page which style my webpage a bit.
- **index.html**: This is the HTML file I use to show on the browser
- **NP_hw1_report.pdf**: This assignment's report

Prerequisite

Before execution you first need to check if you have:

1. gcc/g++
2. Firefox

Also you need to go into "./server/server.cpp" and find the line "string wholefilePath=" change the path to the path this project directory is in.
Ex: string wholefilePath="[your path of this project]/NP_2019/assignment1/datastore/"+myFileName;

Execution

1. In the terminal, navigate to the **"server"** directory
2. **make** -> now an executable file **"server"** will be generated in the same directory
3. **./server** -> start the webserver
4. Open your preferred browser and enter **0.0.0.0:8000**

5. **“index.html”** will be shown on this web page, you can also see the whole GET request packet in the terminal
6. Upload a file to the form on the web page
7. The message you entered will be stored in a file in the folder **“datastore”**,
8. **^c** -> shutdown the webserver

Method Description

I split the whole assignment1 into 5 parts. The detail of them is as follow:

Part1: Write a simple web server that can show your HTML on your browser

1. Write a simple HTML file
    a. Write a Hello World! In your index.html
    b. Test with **python -m http.server**
    c. Open Chrome and type **localhost:8000**
    d. It should show **“Hello World!”** on the screen
2. Define the web dev filetype extensions structure
3. Change the directory to the folder that contains my HTML file
    a. Folder name
    b. chdir()
    c. Return error if can’t change the directory
4. Create socket
    a. Use **socket()** to get an available file descriptor
    b. If the return value is less than 0 this means an error occurs
5. Setup the network connection
    a. Create a basic IP protocol
    b. Set the address family (domain) to **IPv4**
    c. Set the internet address to **INADDR_ANY** (**0.0.0.0** means any address for binding)
    d. Set the port to **8000**
6. Create and start listening to network
    a. Bind the server to the listen file descriptor using **bind()**
    b. Listening using **listen()** -> the socket that will be used to accept the incoming connection request
    c. Initializes the active file descriptor set to have zero bits for all file descriptors
    d. Put the listenfd into the active file descriptor set
7. Patrol all the socket in fd_set
    a. Use **select()** to monitor all the socketfd, waiting until one or more socket becomes ready
    b. Accept the connection that is ready
    c. Put the accept connection into active fd_set

Part2: Handle the connection from the client socket

1. Read the request from the browser

   a. Read the request to the buffer by **read()**
   b. Set the end of the request to 0
   c. Delete the newline symbol **\r\n**
2. Process **GET request**
   a. See if the buffer (packet) I read has **"GET"** or **"get"**
   b. Parse filename and HTTP/1.1 using whitespace
3. Read index.html (or any other GET request file)
   a. Avoid user going to upper directory
   b. Copy GET request to the buffer
4. Recognize the filetype request by the client and open with the browser
   a. Recognize the file type by the ending of the file name
   b. Open the file
   c. Send message (response) back to the browser -> **OK and the content type**
   d. Read the file content to the client web browser

Part3: Execute the request from the client

1. Last bit of code
   a. Go to the main function
   b. Handle the respond connection by the function we defined in **Part2**
   c. If success then close the connection and clear active fd_set
2. Test on your browser
   a. make
   b. ./server
   c. Open your browser (**firefox**) and type **0.0.0.0:8000**
   d. You can now see **"Hello World!"** on the screen
3. Add an image in index.html

Part4: Upload file from the browser to the HTTP server

1. Add an upload form in index.html
2. Try to get the packet from the browser when POSTING
   a. What does your packet look like for now (header, content)
   b. Is your packet empty?
   c. Where should you start parsing from the packet to store to your webserver (the actual place that stores your content)
3. Get content name
   a. Locate your content from the whole packet
   b. Your content name (the file you upload) should be in **Content-Disposition** that line
4. Get content type
5. Write the content to file
   a. Specify the location the POST file should be saved in the webserver
   b. Create a file in your web server and the name of the file will be the content name I get at **Step3**
   c. Read the content from the packet to your web server until reaching the boundary

d. Close the file and return

6. Send response

Part5: Beautify your webpage

Add some more **html+css+bootstrap+javascript** will make your index.html looks better

Results

The results of the purely going to the web page are like the following image. **Figure (a)** shows the website

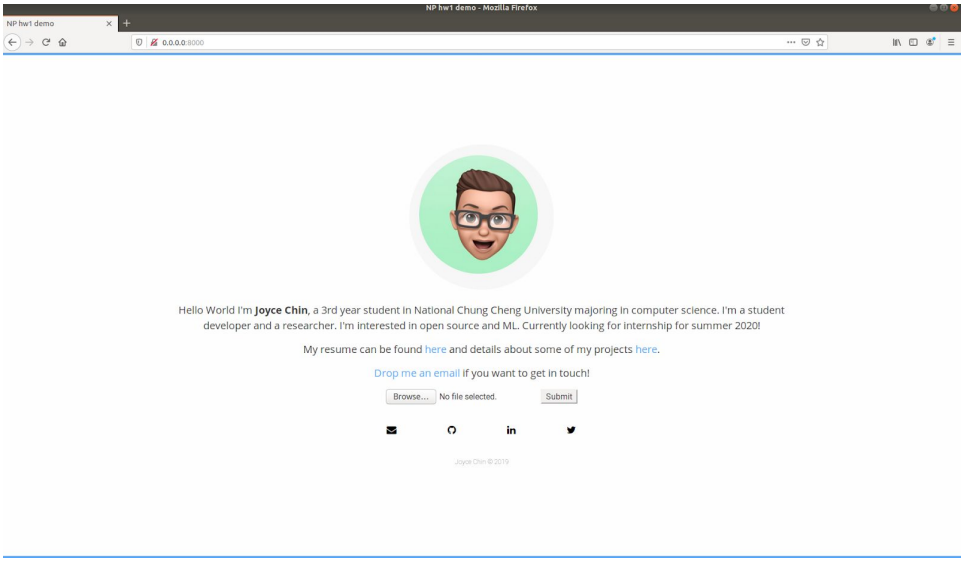

Figure (a)

If upload any file it should send a post request to the webserver
Figure (b) shows the POST request in the inspect element window
Figure (c) shows the image I post save to my webserver (in **"datastore"** folder)

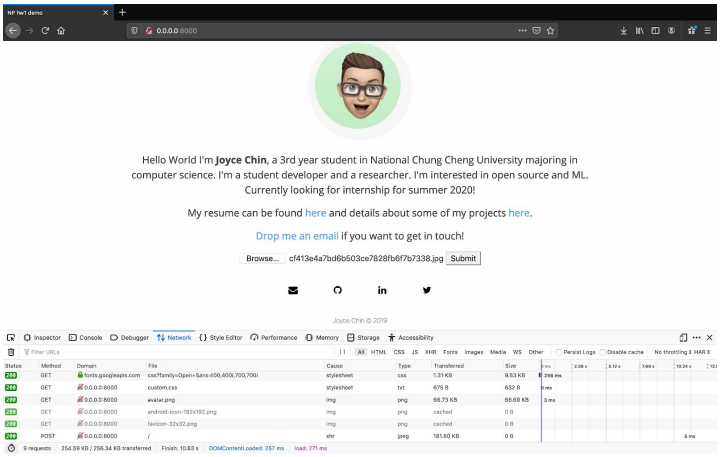

Figure (b)

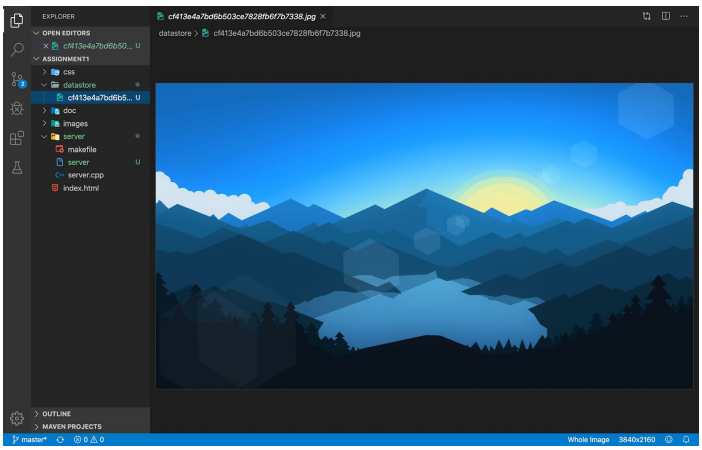

Figure (c)

Discussion

There is something I find pretty weird when POSTing. If the file I upload has the file size range from **0 KB to 500 KB** than the file mostly can successfully be stored in my webserver. The file in my webserver will look the same as the original file. But if the file is **larger than 500KB**, the file can still be stored in my webserver, but the image often distorts a lot. I am certain that I have written all the bits to the webserver cause I have printed out the information on the screen but the actual file size I get is far less than the original file size. I think when encountering large files, basic **read()** and **write()** is not enough. I should use other methods to process large files but I have no clue how as for now.

Problems and Difficulties

When I'm doing POSTing I have encountered a problem. At first, I chose Chrome as my browser to test. The packet I received from the browser has header only and no body. But when I'm using inspect element to check the packet, I can see the body. I thought maybe I didn't handle the post form in index.html well so I search for another way to implement the form but still, the body of the packet didn't show up in my server program. I think there is no problem with the process I have done to the packet cause I print the packet information before doing anything. The information I print just don't include the body of the packet. So the only problem should be Chrome. Chrome didn't send the POST request properly. I decided to switch to Firefox as my browser. And the body of the packet successfully sends back to my server.