Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Alec Barber

Aayush Madaan

3D3

# Project 1

# Simple HTTP Client and Server

# Report

## High Level Design of Project

### Web-server

The web-server initialises by taking in up to 3 arguments, the URL, the port and the file directory in which to operate. The file directory input should be in the following format, "~/Documents/nextFolder/NextFolder". A HTTPresponse object is then created and the establishServer method is called. All other functionality of the server comes about by this method.

The establishServer method then calls functions to set up all sockets and set up addresses. The server then waits for a connection. Once a connection with a client is established, a message is received. This message is then parsed to get the file name and location. A method is then called to find the file. If it exists then it is passed and returned with the correct heading. If it is not found then a 404 header is returned. If something along route happens abnormally, then 400 is returned.

All methods were declared using inline. It was found through experimentation that this actually reduces file size, as every method is only ever called once. This would also reduce overhead when running the program.

### Web-client

The web-client I established by taking one URL argument (for example 134.226.44.157:4000/index.txt) . This URL encapsulates info about the destination IP, the port and the name and location of the required file. The web-client parses through this information and opens a connection with the server. The web-client then sends a GET command through the requestFile method. The client then waits for a response and closes the connection.

The header of the file is parsed. If 200 OK was sent, the header is removed from the file, and then the file is saved locally. If 404 or 400 is received, then an error is sent to terminal outlying what the error was.

This process is repeated for every argument passed into the web-client when launching.

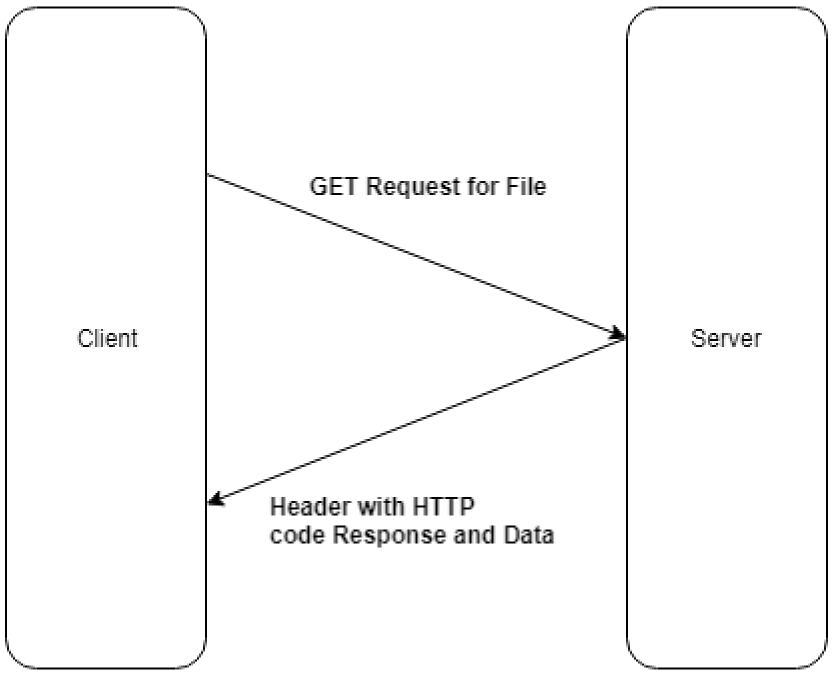

Fig. 1 Server-Client Communications

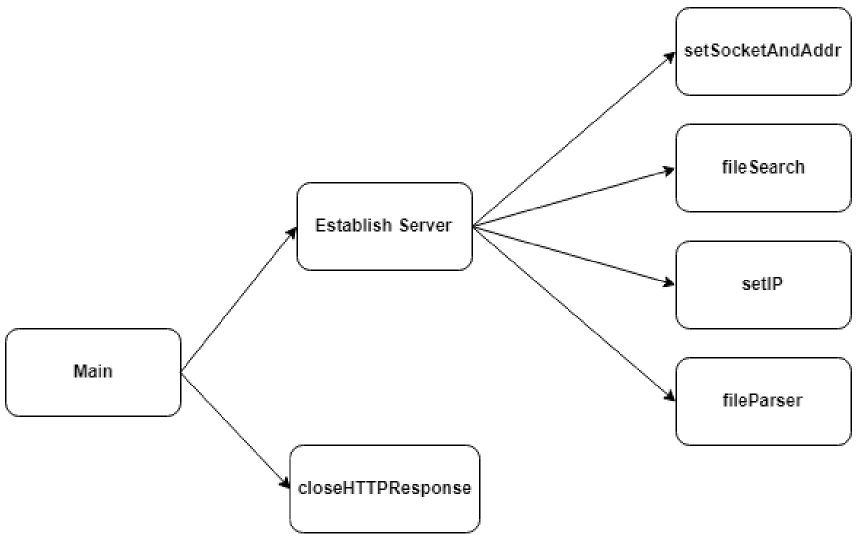

Fig. 2 Methods modules for web-server

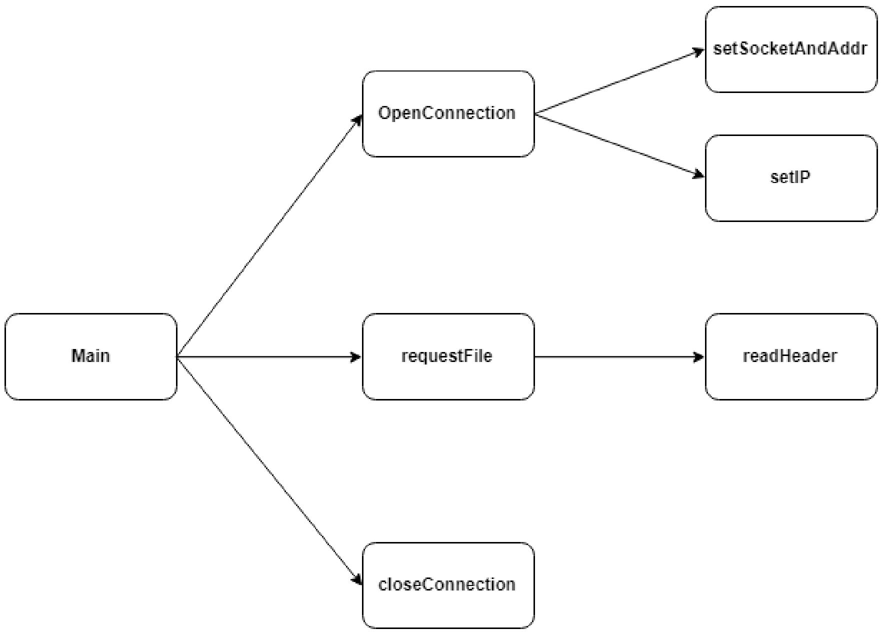

Fig. 3 Methods modules for web-client

**Problems Confronted**

Many problems were confronted in the development of this program.

1.) A basic send function was first built that sent and received a function between the server and client, all the code was in two main functions. The first major task was to find a way of incapsulating both these blocks of code in two classes.
2.) Finding out how to use file directories took a lot of time. It was eventually found that using ../. would let the server run from the parent directory and thus allow the client to ask for a bigger selection of files. Another method is using ~/ and then all the directories below your main folder.
3.) Another problem encountered, was building multiple parsers. A parser was needed to write and decode and remove headers from files. And also for decoding the URL input to the client into the URL, file requested and the port. This took a bit of time to implement well, with suitable robustness (taking into account if there was a mistake in typing the URL such as wrong format, no port or missing a /).

4.) A problem that was encountered at the beginning of the project for a long time, was when the server did not shutdown correctly, the socket would be left open, which meant that no other connection could be established until it had been closed. This was resolved by adding in closeSocket method calls whenever an error occurred, before returning.
5.) One problem that was confronted by not fixed, was that the client is not able to receive image files. This is because the image file needs to be encoded server side first and then decoded back from a text string to an image again. Unfortunately, I did not have time to implement these functions, but I believe it would be very doable given a reasonable amount of time.

## Additional Instructions on how to build Project

To build the code on a Linux machine, make sure all the cpp files and make file are present. Type make into your terminal when in the correct directory. If you have previously made the file, and would like to remake it, then use: "make -B all". This forces a rebuild of all files.

To run the server, type ./web-server [URL] [PORT] [File_Directory]

All inputs for the server are optional, and will default to localhost 4000 ".".

Once the server is running, use a different terminal to launch the web-client.

To run the client, type ./web-client [URL]

The URL default if not entered is localhost:4000/.

The server will return the file requested or return a 400 or 404 error.

No other libraries were used.

One outstanding bug in the program, is that when the client requests a file which is located within a folder which has a space character in it, the web-server may become confused and may not look at any characters after the space. This can be avoided by using folders without spaces, which is usually considered a good practice.

I have also included two txt files for testing, index.txt and taleOfTwoCities.txt.

## Testing the Code

The code was tested exhaustively, using every edge case possible. Some tests reformed are as follows:

1.) URL entered for client incomplete or missing characters (tested every scenario)
2.) Leaving document segment on URL blank (localhost:4000/)
3.) Entering multiple URLs at one to the web-client
4.) Running multiple clients at once (upto 10) to see if server can deal with more than one client
5.) Making an error in the URL as in 1.) but on only one of a set or entered URLs. This resulted in the web-client just skipping that particular request and outputting an error.
6.) Multiple files were tested (index.txt and taleOfTwoCities.txt), both were found and saved correctly.
7.) The server was set up on a parent directory and then the client then made a request as follows http//:localhost:4000/someFolder/index.txt .

## Extra Testing

1.) A server was set up on my current working URL on one computer and a server connected to the URL on another computer in the Linux lab and successfully received the requested file.
2.) A server was set up using the computers IP again, and then Firefox was used to connect to the server and display the contents of the .txt file. This was achieved by typing http//:134.226.44.157:4000/index.txt into Firefox. There was a bug with this test that was not resolved. The browser would attempt to display every character in the array, even if they were NULL characters. This meant that all the text was displayed correctly, but then followed by a number of '?' unknown characters. I tried to fix this issue by using EOF file characters and setting all characters to 0, but to no avail. Multiple computers in the Linux lab also managed to connect to the server in this way at the same time.
3.) The web-client was used to connect to popular domains such as google.ie and bbc.co.uk to request a file. Although I did not successfully download any files, I did manage to connect and get responses saying that the file was either not found, or some other redirection request responses, which my client is not built to deal with yet.

## Team Contribution

| | | | |
|---|---|---|---|
| Alec Barber: | 15321255 | 90% | Wrote Code and Report |
| Aayush Madaan: | ******** | 10% | Did research on code |

## Misc. Notes

This implementation of a web-server/ web-client does not use threading or any other asynchronous communication. I was going to implement threading, but decided not to as for the task at hand, decided that it was not necessary. The client opens a non-persistent connection which is closed as soon as the server responds. This means that the connection will never be open for more than a moment. The listen function incorporates a backlog, so if more than one client attempts to connect at the same time, they will be queued, this was tested with up to 10 clients successfully.
By not implementing threading, I have simplified the code with no real loss to performance.