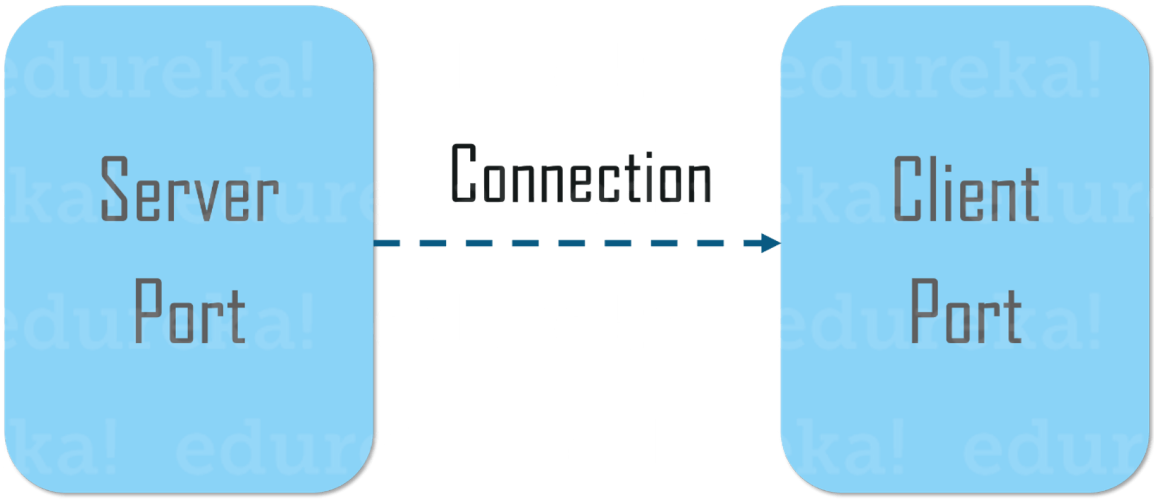
Networks

Socket Programming
Assignment #1



Github Repo : https://github.com/khadijaAssem/Sockets

Introduction

Using sockets to implement a simple web client that communicates with a web server using a restricted subset of HTTP commands (GET and POST).

In GET requests: The client sends a GET request that contains headers of the desired hostname and port number and the required files to get (TXT, HTML, IMG). Once the server receives the request it processes it and sends the response containing the data requested and the client reads it and saves it.

In POST request: The client sends a POST request that contains the headers of the desired hostname and port number and the required file to be posted data. Once the server receives the request it processes it and reads the data of the POST file and saves it.

Overall organization

There are two main files: client_runner.cpp and server_runner.cpp for both client_runner and server_runner there are client.cpp and server.cpp that use utility files that include all extra needed functions to read from file or write in a file and do some extra processing on messages.

I also added two make files to build and compile the client and servers and generate the executables for both server and client.

For client:

Client_runner.cpp: It is the main function that it is used to run the client and sends the required arguments to the client.

```
int main(int argc, char *argv[]) {
    client c;

    char PORT[] = "3490"; // set default value of PORT
    char HOSTNAME[] = "localhost"; // set default value of HOSTNAME
    char COMMANDFILE[] = "commands.txt"; // set default value of COMMANDFILE

    if (argc < 4) {
        printf("WILL USE HOSTNAME: %s\nPORTNUMBER: %s\nCOMMANDFILE: %s\n", HOSTNAME, PORT, COMMANDFILE);
        printf("IF YOU WANT TO CHANGE ONE OF THE DEFAULT RUN THE FOLLOWING\n");
        printf("./my_client hostname portNumber commands\n");
        printf("THANKS :)\n--------------------------------------------------------------------------------\n");
    }

    if (argc == 4) {
        strcpy(COMMANDFILE, argv[3]); // get command file from user
        strcpy(PORT, argv[2]); // get port number from user
        strcpy(HOSTNAME, argv[1]); // get host name from user
    }

    c.run(HOSTNAME, PORT, COMMANDFILE);
}
```

Client.cpp: It is the main client class that contains the overall logic of the client to construct messages and send them to the server.It is initiated when the client_runner calls it's function run with the required arguments.

```
int run(char HOSTNAME[], char PORT[], char COMMANDFILE[]);
```

For server:

Server_runner.cpp: It is the main function that it is used to run the server and sends the required arguments to the server.

```
int main(int argc, char *argv[]) {
    server s;
    // All ports below 1024 are RESERVED (unless you're the superuser)! You can have any port number above that, right up to
    char PORT[] = "3490"; // set default value of PORT

    if (argc < 2) {
        printf("WILL USE PORTNUMBER: %s\n", PORT);
        printf("IF YOU WANT TO CHANGE THE DEFAULT RUN THE FOLLOWING\n");
        printf("./my_server portNumber\n");
        printf("THANKS :)\n--------------------------------------------------------------------------------\n");
    }
    if (argc >= 2){
        strcpy(PORT, argv[1]); // get port number from user
    }

    s.run(PORT);
}
```

Server.cpp: It is the main server class that contains the overall logic of the server to receive messages and send responses to the client.It is initiated when the server_runner calls it's function run with the required arguments.

```
int run(char PORT[]);
```

For utilities:

It contains the important definitions used by both server and client.

```
#define BACKLOG 10   // how many pending connections queue will hold
#define MAXDATASIZE 1000000 // max number of bytes we can get at once
#define MAXFILEPATHSIZE 50 // max number of bytes we can get at once
#define MAXPORTNUMBSIZE 20 // max number of bytes we can get at once
#define MAXHOSTNAMESIZE 50 // max number of bytes we can get at once

#define GET_REQUEST "GET"
#define POST_REQUEST "POST"

#define OK_RESPONSE "HTTP/1.1 200 OK\r\n"
#define NOTFOUND_RESPONSE "HTTP/1.1 404 Not Found\r\n"
#define CONTENT_LENGTH "Content-Length: "
#define ENDREQUEST "\r\n"

#define TIMEOUT 5000000
```

Contains the message data structure used by both server and client.

```
struct messege_content {

    char request[8]; // GET or Post
    char file_path[MAXFILEPATHSIZE];
    char host_name[MAXHOSTNAMESIZE];
    char port_number[MAXPORTNUMBSIZE];
    char request_msg[MAXDATASIZE]; // Whole request messege

};
```

Contains the functions used by both server and client.

```
class utilities
{
    public:
    void save_data_to_path(char *, char *);
    std::string read_data_from_path(char *);
    struct messege_content request_processing(char[]);

};
```

Important Functions

For client:

```
int client::run(char HOSTNAME[], char PORT[], char COMMANDFILE[])
```

The function that does the whole logic of the client and uses some utility functions from the utilities class for some sub-tasks.

1. Process the commands file and create the message requests
2. Create socket for the client
3. Connects with the server on the servers socket
4. Start sending the requests to the server and waits for 2 seconds between these requests (for debugging to be more easier)

```
char *client::send_recieve_routine(struct messege_content cmd, int sockfd)
```

It performs the task of sending the request to the server and waits for response and processes this response by parsing it and sending it to a utility function to write the extracted data to a local file.

For server:

```
int server::run(char PORT[])
```

The function that does the whole logic of the server and uses some utility functions from the utilities class for some sub-tasks.

1. Create socket for the server
2. Bind to the server socket
3. Start listening on the socket so incoming connections will wait on the queue waiting to be accepted by the server
4. The main accept loop to create socket_fd for each connection with new client and send them to a new thread to receive requests and send responses.

```
void *thread_handler(void *arg)
```

The thread handler routine that performs requests receiving and responses sending and it uses the utilities functions to save the received data from the client on it's storage.

For utilities:

```
void save_data_to_path(char *, char *);
```

It saves the passed data to a file on a local directory with the given path.

```
std::string read_data_from_path(char *);
```

It reads data from the passed path and returns it.

```
struct messege_content request_processing(char[]);
```

Used by client to transfer the given command from this form :

```
// client_get file-path host-name (port-number)
// client_post file-path host-name (port-number)
```

To the data structure message_content to be able to be used by the client to send the message.

Data Structures

```
struct messege_content {

    char request[8]; // GET or Post
    char file_path[MAXFILEPATHSIZE];
    char host_name[MAXHOSTNAMESIZE];
    char port_number[MAXPORTNUMBSIZE];
    char request_msg[MAXDATASIZE]; // Whole request messege

};
```

A structure to save important info about the request message weather it is GET or POST and the file path associated with the request and hostname and the port number to be sent to.

Sample Request/ Response

Sample request received by the server from the client

```
Server: received from client on socket 4: 'GET request'
GET /index.html HTTP/1.1
Host: localhost:2000
```

Sample request received by the client from the server

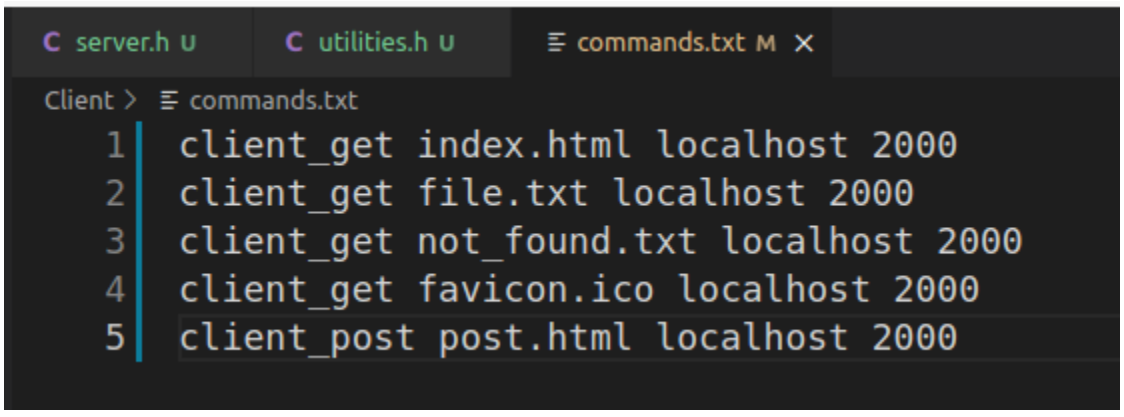
```
client: received from server on socket 3: 118 'HTTP/1.1 200 OK
Content-Length: 78

<!DOCTYPE html>
<html>
<div>
    <p>
        INDEX
    </p>
</div>
</html>

'
```

Sample Run

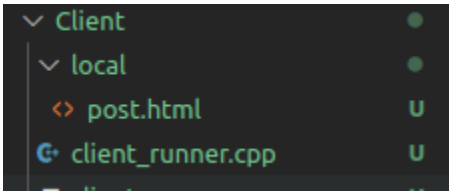
Server initially running and waiting for clients to serve:

```
khadija@khadija-HP-Pavilion-x360-Convertible-14-cd1xxx:~/Desktop/Sockets/Server$ ./my_server 2000
Successful bind at: 0.0.0.0
server: waiting for connections...
```

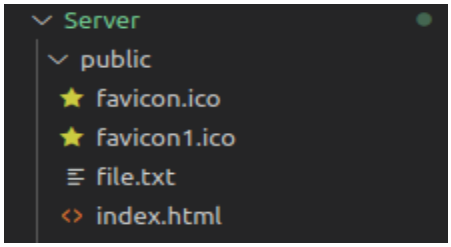
Running the client with command file:



Local directory at client initially:



Server directory before execution:



Initially client running and processing commands:

```
khadija@khadija-HP-Pavilion-x360-Convertible-14-cd1xxx:~/Desktop/Sockets/Client$ ./my_client localhost 2000 commands.txt
Client will send
client_get index.html localhost 2000
client_get file.txt localhost 2000
client_get not_found.txt localhost 2000
client_get favicon.ico localhost 2000
client_post post.html localhost 2000
```

Client console after running and closing upon reaching end of command file:

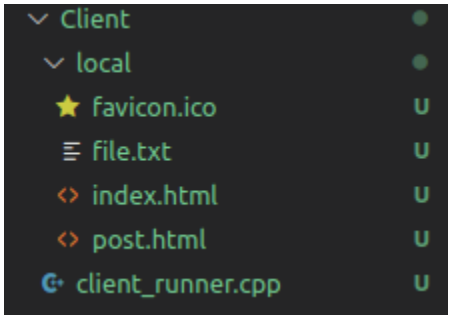
```
Client: successful connect at: 0.0.0.0
client: connecting to 127.0.0.1
Sending GET request
Sent
client: received from server on socket 3: 118 'HTTP/1.1 200 OK
Content-Length: 78

<!DOCTYPE html>
<html>
<div>
    <p>
        INDEX
    </p>
</div>
</html>

'
Writitng to path local/index.html
------------------------------------------------------------
Sending GET request
Sent
client: received from server on socket 3: 59 'HTTP/1.1 200 OK
Content-Length: 19

This is a txt file

'
Writitng to path local/file.txt
------------------------------------------------------------
Sending GET request
Sent
client: received from server on socket 3: 24 'HTTP/1.1 404 Not Found
'
------------------------------------------------------------
Sending GET request
Sent
client: received from server on socket 3: 695 'HTTP/1.1 200 OK
Content-Length: 654

◆PNG
◆
'
Writitng to path local/favicon.ico
------------------------------------------------------------
Sending POST request
Sent
client: received from server on socket 3: 17 'HTTP/1.1 200 OK
'
------------------------------------------------------------
```

Server console after receiving and responding to the client responses:

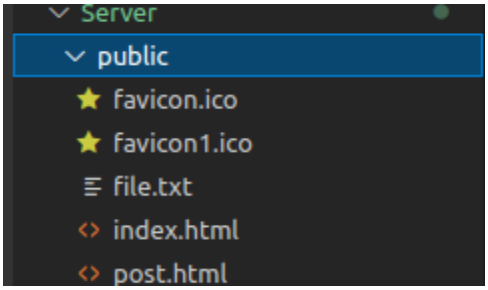
```
khadija@khadija-HP-Pavilion-x360-Convertible-14-cd1xxx:~/Desktop/Sockets/Server$ ./my_server 2000
Successful bind at: 0.0.0.0
server: waiting for connections...
server: got connection from 127.0.0.1
Server: received from client on socket 4: 'GET request'
GET /index.html HTTP/1.1
Host: localhost:2000


Reading from path public/index.html
------------------------------------------------------------------------
Server: received from client on socket 4: 'GET request'
GET /file.txt HTTP/1.1
Host: localhost:2000


Reading from path public/file.txt
------------------------------------------------------------------------
Server: received from client on socket 4: 'GET request'
GET /not_found.txt HTTP/1.1
Host: localhost:2000


Reading from path public/not_found.txt
------------------------------------------------------------------------
Server: received from client on socket 4: 'GET request'
GET /favicon.ico HTTP/1.1
Host: localhost:2000


Reading from path public/favicon.ico
------------------------------------------------------------------------
Server: received from client on socket 4: 'POST request'
POST /post.html HTTP/1.1
Host: localhost:2000
Content-Length: 43

<html> <div> <p> POSTING </p></div></html>


Writitng to path public/post.html
------------------------------------------------------------------------
CLOSING CONNECTION WITH CLIENT SINCE TIMEOUT
```

Note that the server closed the connection with the client once the client is done with it's requests since the server waited for 5 seconds and found no more requests to serve with this client so it killed the thread with this client and kept open for more clients in case any came it will create a thread for it and start serving it.

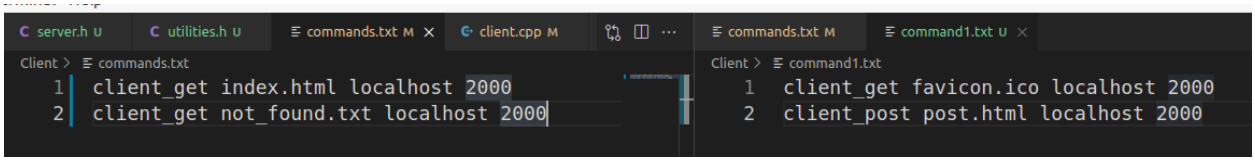
Client local directory after execution

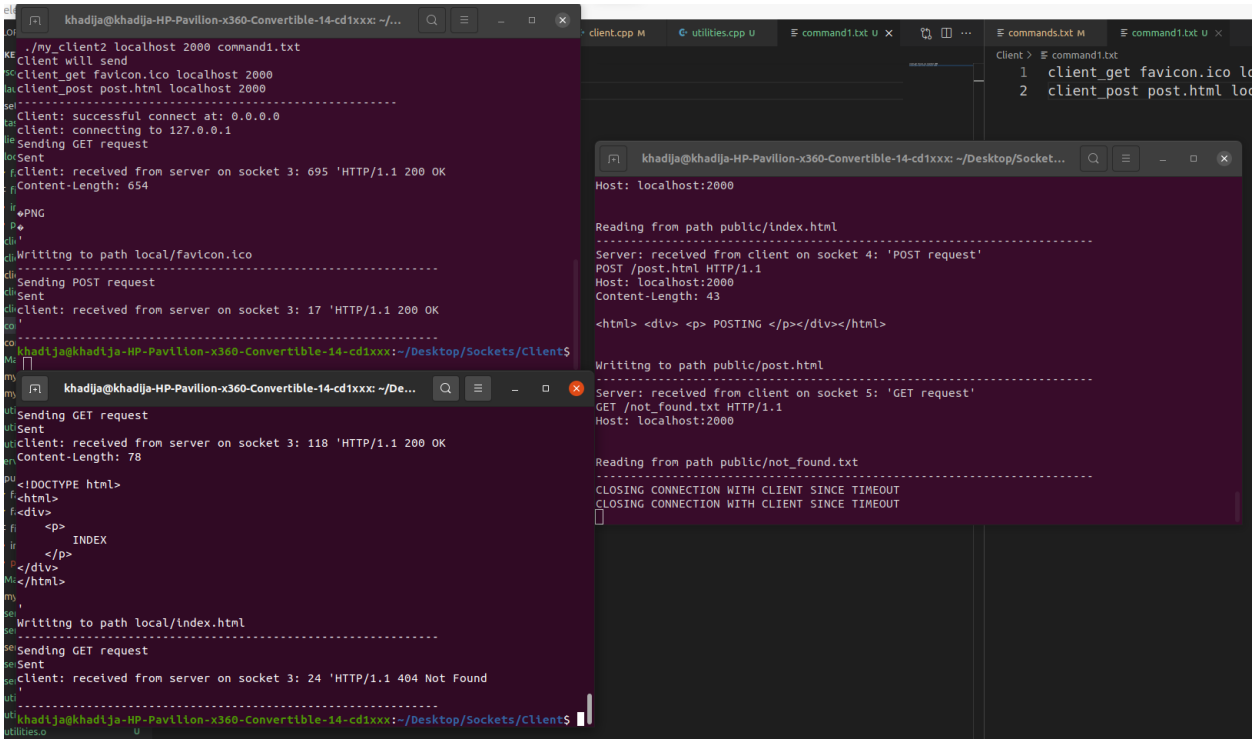


Server public directory after execution



Second running two clients with different commands files





Socket_fd for different clients on the server side are different.

Third running on website