NETWORKING WITH WINSOCK

1. Project Overview

This is a project that uses Winsock to implement a simple file-sharing system between a client and a server. The server provides a list of files available for download, and the client can request and download files from the server. The client can download multiple chunks of a file simultaneously using worker sockets for multi-threaded file transfer.

The project also utilize the multithreaded programming with C++ now only for downloading file, but also apply this concept for realtime update in the user interface.

This report consist of:

- **Overview**: Brief description of the project.
- **Functions and Prototypes**: List of functions and prototypes used in the program.
- **Detailed Function and Prototype List**: Detailed list of functions categorized by files and purposes.
- **Build and Run**: Instructions for building and running the program.
- **Client UI**: Description of the client user interface.
- **Client Protocol**: Protocol for client-server communication.
- **Server Protocol**: Protocol for server-client communication.

2. Functions and Prototypes

First we will see which function are used in the program, these are the function used in both server, and client code.

2.1. Libraries and Preprocessor Directives

`#include <signal.h>`

- **Description**: Includes functions and macros for signal handling, which allows you to define custom behavior for signals like `SIGINT` (e.g., Ctrl+C).
- **Usage**: Used to handle asynchronous events like interrupts or termination requests in programs.

`#include <winsock2.h>`

- **Description**: Provides APIs for Windows Sockets programming, enabling network communication such as TCP and UDP connections.
- **Usage**: Required for socket programming on Windows.

`#pragma comment(lib, "ws2_32.lib")`

- **Description**: Links the `ws2_32.lib` library at compile-time, which is necessary for Winsock functions like `socket`, `bind`, and `listen`.

2.2. Winsock Initialization and Cleanup

`WSADATA wsa`

- **Description**: A structure used by the `WSAStartup` function to store details about the Windows Sockets implementation.

`WSAStartup(MAKEWORD(2, 2), &wsa)`

- **Description**: Initializes Winsock. The `MAKEWORD(2, 2)` specifies version 2.2 of Winsock.
- **Usage**: Must be called before any other Winsock functions.

`WSACleanup()`

- **Description**: Terminates the use of Winsock. Cleans up resources allocated by Winsock functions.
- **Usage**: Always call this when Winsock-related operations are complete.

2.3. Socket Programming

`SOCKET server_socket, client_socket`

- **Description**: Variables of type `SOCKET`, used to represent the server and client sockets.

`sockaddr_in server{}, client{}`

- **Description**: Structures used to store address information for the server and client.
- **Fields**:
  - `sin_family`: Specifies the address family (e.g., `AF_INET` for IPv4).
  - `sin_addr.s_addr`: Stores the IP address.
  - `sin_port`: Specifies the port number.

`server.sin_family = AF_INET;`

- **Description**: Specifies that the socket will use the IPv4 address family.

`server.sin_addr.s_addr = inet_addr(ip.c_str());`

- **Description**: Converts an IP address from a string to a network byte order and assigns it to `server.sin_addr.s_addr`.

`server.sin_port = htons(port);`

- **Description**: Converts the port number to network byte order and assigns it to `server.sin_port`.

`listen(server_socket, MAX_PENDING)`

- **Description**: Prepares the socket to accept incoming connections.
- **Parameters**:
  - `server_socket`: The socket to listen on.
  - `MAX_PENDING`: Maximum number of pending connections.

`accept(server_socket, reinterpret_cast<sockaddr *>(&client), &client_length)`

- **Description**: Accepts an incoming connection on `server_socket`.
- **Parameters**:
  - `server_socket`: The listening socket.
  - `reinterpret_cast<sockaddr *>(&client)`: A pointer to a `sockaddr` structure for client information.
  - `client_length`: The size of the `client` structure.

`send(client_socket, data, sizeof(short_message), 0)`

- **Description**: Sends data to the connected client.
- **Parameters**:
  - `client_socket`: The socket connected to the client.
  - `data`: The data to send.
  - `sizeof(short_message)`: Size of the data.
  - `0`: Flags for the send operation.

`closesocket(so)`

- **Description**: Closes a socket and frees its resources.

2.4. Signal Handling

`signal(SIGINT, exit_on_signal);`

- **Description**: Sets up a signal handler to call `exit_on_signal` when the program receives a `SIGINT` signal (e.g., Ctrl+C).

2.5. Multithreading

`std::thread render_ui(call_render_ui, ref(ui));`

- **Description**: Creates a thread to execute the `call_render_ui` function, passing a reference to `ui`.

`render_ui.join();`

- **Description**: Waits for the `render_ui` thread to complete before proceeding.

2.6. Utility Functions and Conversions

`reinterpret_cast<char*>`

- **Description**: A C++ casting operator that converts one type into another. In this case, it is used to cast data types into `char*` for socket programming purposes.

`std::cerr << message << ": " << WSAGetLastError() << std::endl;`

- **Description**: Outputs an error message to the standard error stream. The `WSAGetLastError()` function retrieves the last error code set by a Winsock function.

Overall Categories

1. **Libraries and Initialization**: `#include`, `#pragma`, `WSAStartup`, `WSACleanup`.
2. **Socket Programming**: Sockets, `sockaddr_in`, `listen`, `accept`, `send`, `closesocket`.
3. **Error Handling**: `WSAGetLastError`, `std::cerr`.
4. **Signal Handling**: `signal`.
5. **Multithreading**: `std::thread`, `join`.

3. Detailed Function and Prototype List

This is the list of functions used in the program, categorized by their respective files and purposes.

3.1. `./client/recieve_file.hpp`

3.1.1. `get_download`

```
bool get_download(string filename, string ipAddress, unsigned short port,
                  unsigned long long offset, unsigned long long size,
                  ofstream& lout, clientUI& ui);
```

- **Description**: Retrieves a file from the server, manages workers, combines file chunks, and writes the data to disk.
- **Parameters**:

- `filename` : The name of the file to download.
- `ipAddress` : Server IP address.
- `port` : Server port.
- `offset` : Offset in the file to start downloading.
- `size` : Size of the file chunk to download.
- `lout` : Output stream for logs.
- `ui` : Reference to a user interface object.

3.1.2. `recieve_file`

```
void recieve_file(SOCKET server, const string& filename, const string& rename, ofstream& lout);
```

- **Description**: Receives a small file (e.g., a file list) from the server without worker control.
- **Parameters**:
  - `server` : Server socket.
  - `filename` : Name of the file to receive.
  - `rename` : Optional new name for the received file.
  - `lout` : Output stream for logs.

3.1.3. `get_file_list`

```
void get_file_list(SOCKET server, ofstream& lout, clientUI& ui);
```

- **Description**: Retrieves and formats the list of files available on the server.

3.1.4. `handle_each_file`

```
void handle_each_file(SOCKET server, string serverIp, unsigned short serverPort,
                      map<string, long long>& filelist, string name,
                      ofstream& lout, clientUI& ui);
```

- **Description**: Validates user input, sets up messaging with the server, and handles the file protocol.

3.1.5. `handle_download`

```
void handle_download(SOCKET server, string serverIp, unsigned short serverPort,
                     vector<string>& downloaded_files, ofstream& lout, clientUI& ui);
```

- **Description**: Scans `input.txt` for new files to download and manages the download process by calling helper functions like `handle_each_file`.

3.1.6. `Worker::run`

```
void Worker::run();
```

- **Description**: Establishes a connection between a worker thread and the server.

3.1.7. `Worker::get_file`

```
void Worker::get_file(string filename, unsigned long long offset,
                      unsigned long long len, unsigned long long filesize,
                      int part, unsigned long long& progress, ofstream& lout);
```

- **Description**: Manages the protocol for receiving file chunks, writes to temporary files, and tracks progress.

3.2. `./server/serve_list.hpp`

3.2.1. `serve_file`

```
void serve_file(SOCKET client_socket, const string& filename, const string& showname);
```

- **Description**: Sends the specified file to the client.
- **Parameters**:
  - `client_socket` : Client's socket.
  - `filename` : Actual file name.

- `showname` : Name to display for the client.

3.2.2. `get_available_file`

```
map<string, unsigned long long> get_available_file();
```

- **Description**: Uses the filesystem to retrieve a list of files available on the server.

3.2.3. `serve_list`

```
void serve_list(SOCKET client_socket);
```

- **Description**: Sends the list of available files to the client.

3.2.4. `check_file_to_download`

```
void check_file_to_download(SOCKET client_socket);
```

- **Description**: Checks if a file requested by the client is available on the server.

3.2.5. `serve_chunk`

```
void serve_chunk(SOCKET client_socket);
```

- **Description**: Sends a chunk of a file to the client.

3.3. `message.hpp`

Message Structs

```
struct short_message {
    int len;
    char content[SHORT_MESSAGE_LEN];
};

struct data_message {
    int len;
    char content[DATA_LEN];
};

struct start_file_transfer {
    unsigned long long file_size;
    int len;
    char filename[SHORT_MESSAGE_LEN];
};

struct start_chunk_transfer {
    unsigned long long file_size;
    unsigned long long offset;
    unsigned long long offset_lenght;
    int len;
    char filename[SHORT_MESSAGE_LEN];
};
```

- **Description**: Fixed-size message frames used for communication between the client and server. Each struct includes a buffer and length field to manage the message content.

3.3.1. `get_content_short`

```
string get_content_short(const short_message& mess);
```

- **Description**: Extracts the content of a `short_message` .

3.3.2. `get_content`

```
string get_content(char ch[], int len);
```

- **Description**: Extracts a string from a character buffer and length.

3.3.3. `make_short_message`

```
short_message make_short_message(const string& s);
```

- **Description**: Converts a string into a `short_message`.

3.3.4. Template Functions

```
template <typename T>
bool copy_buffer_to_message(char* buffer, int size, T& target);

template <typename T>
int send(T& data, SOCKET& server, const string& error_message);

template <typename T>
int recv(T& data, SOCKET& server, const string& error_message);
```

- **Description**: Templated functions for sending and receiving structured messages.

3.3.5. `is_valid_message`

```
bool is_valid_message(const string& message);
```

- **Description**: Checks if a message string is valid.

3.3. `./client/file_manipulate.hpp`

3.3.1. `compare_file_set`

```
bool compare_file_set(const vector<string>& list, const string& filename);
```

- **Description**: Compares a file against a list to check if it is already downloaded.

3.3.2. `check_download_file`

```
bool check_download_file(const string& filename);
```

- **Description**: Checks if a specific file has already been downloaded.

3.3.3. `get_filename_size`

```
bool get_filename_size(string& name, unsigned long long& size, ifstream& fin);
```

- **Description**: Extracts the name and size of a file from an input stream.

This formatting organizes the functions, parameters, and descriptions in a clear and readable manner. Let me know if you need further refinements!

4. Build and Run

4.1. Build:

```
cd ./TCP-Redo
./build all

// or ./build client
// or ./build server
```

4.2. Prepare:

Before using the server, you need to put the files you want to share in the `./server/Files` folder. The server will read the files in this folder and send the list to the client. If the folder `Files` does not exist, you need to create it and put the files in it.

About the client, the file will auto download to the `./client/Files` folder. If the folder does not exist the program will create it.

4.3. Run server:

```
// from TCP
cd ./server
./server.exe <ip> <port>
```

4.4. Run client:

```
// from TCP
cd ./client
./client.exe <ip> <port>
```

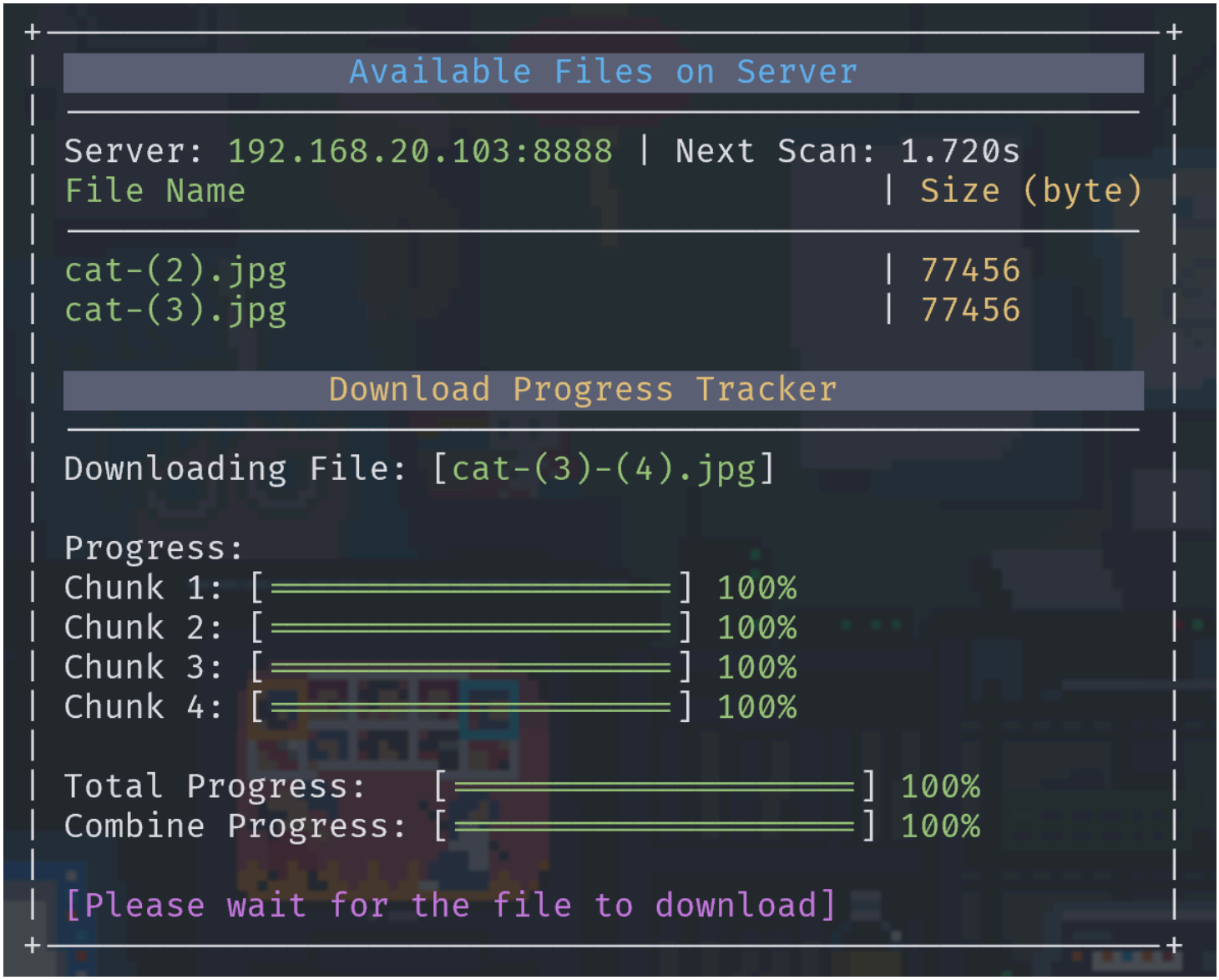
5. Client UI

The UI is a simple Terminal UI, it will show the progress of the download and the download speed. Also the time remaining for next time it scans the `input.txt` .

5.1. Ascii Art

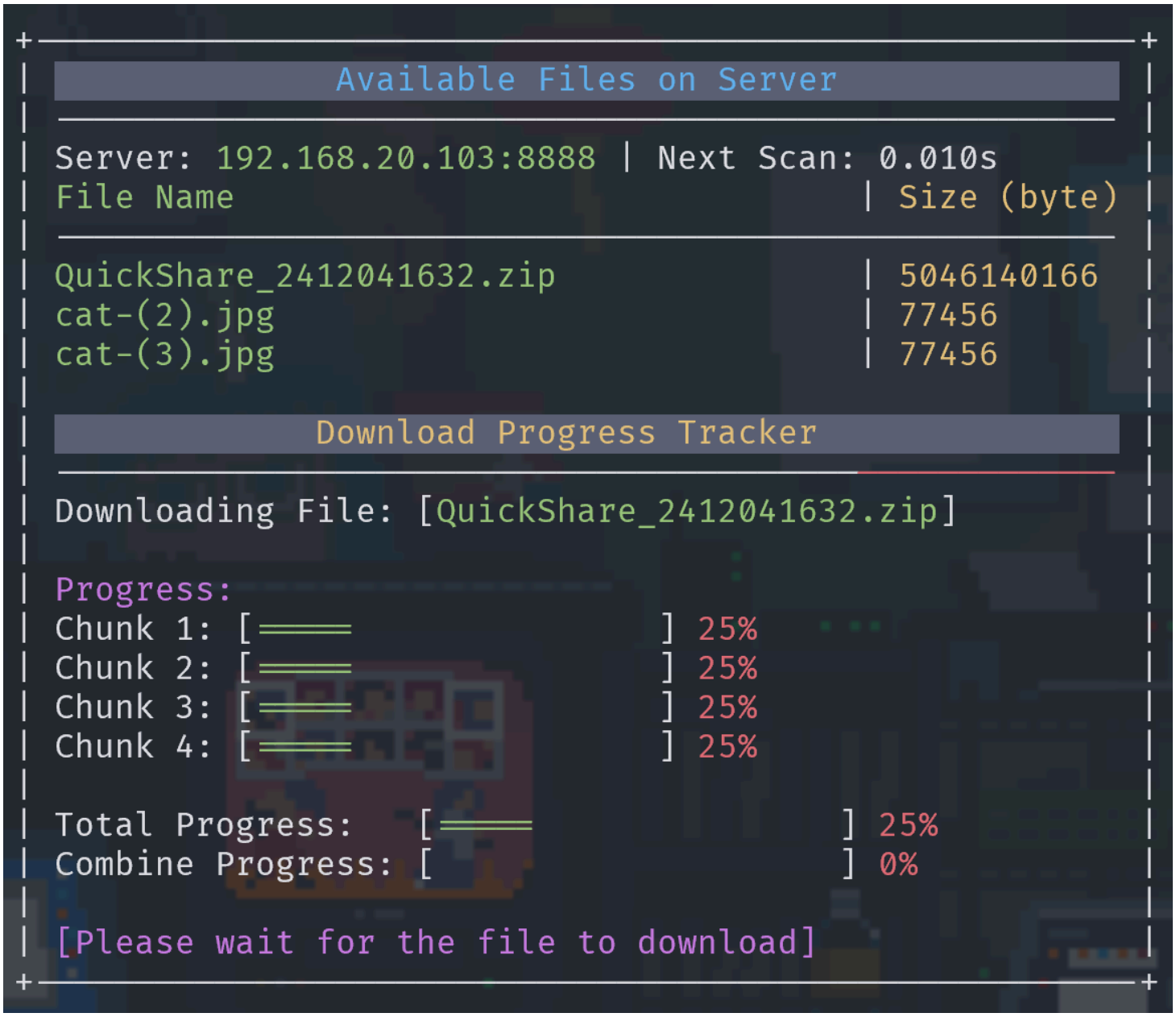
```
+-------------------------------------------------------+
|               Available Files on Server               |
| ----------------------------------------------------- |
| Server: 192.168.20.103:8888 | Next Scan: 2.670s       |
| File Name                                | Size (byte) |
| ----------------------------------------------------- |
| QuickShare_2412041632.zip                | 5046140166  |
| cat     -   (2).jpg                      | 77456       |
| cat-(3).jpg                              | 77456       |
|                                                       |
|               Download Progress Tracker               |
| ----------------------------------------------------- |
| Downloading File: [QuickShare_2412041632-(3).zip]     |
|                                                       |
| Progress: 678.64 MB | 4.69 GB                         |
| Chunk 1: [===                ] 13%                    |
| Chunk 2: [===                ] 14%                    |
| Chunk 3: [===                ] 14%                    |
| Chunk 4: [===                ] 13%                    |
|                                                       |
| Total Progress:   [===                ] 14%           |
| Combine Progress: [                   ] 0%            |
|                                                       |
| [Please wait for the file to download]                |
+-------------------------------------------------------+
```

5.2. Image demo



When the client start the program, set says using the command `./client 192.168.20.103 8888` , it first fetch the available files from the server, shows to the user. then it will start downloading the files. The files will be saved in the `./client/Files` folder.

When the file is dowloaded, notify will tell the user can press ctrl + c to exit the program. While it download, it will tell the user wait for the program to download.



Example of the program running downloading big files.

6. Client Protocol

6.1. Connect to the Server

Establish a TCP connection to the server.

6.2. Get the List of Files from the Server

Function: `get_file_list(...)`

Steps:

1. **Send a "GET_LIST" Request**
   The client requests the server to provide a list of available files.
2. **Receive File Information**
   - Call `receive_file(...)` to retrieve the file list.
   - Steps in `receive_file(...)` :
     - Receive a file transfer packet containing confirmation and file information.
     - Send an "OK" message back to the server.
     - Open the `ready.txt` file on the client's computer to store the list.
     - Receive and write file data in a `while` loop.
     - Close the file after successfully receiving and writing the content.

6.3. Download Files

Function: `handle_download(...)`

Steps:

1. **Scan the Input File (`input.txt`)**
   Check if any new line is added, then read the file list to identify the names of files to download.
2. **Download Each File**
   For each new file added in `input.txt` , perform the following:
   - Call `handle_each_file(...)` for individual file download.

6.4. Handle Each File

Function: `handle_each_file(...)`

Steps:

1. **Initiate File Download**
   - Send a `DOWNLOAD_FILE` request to notify the server that the client will use a worker socket for file transfer.

- Receive acknowledgment (`ACK`) from the server.

2. **Request Specific File**
   - Send the file name and size to the server for confirmation.
   - Wait for the server to send an acceptance response.
   - Transition to the `get_download(...)` function to begin downloading the file using worker threads.

6.5. Download File with Multi-Threading

Function: `get_download(...)`

Steps:

1. **Create Worker Threads**
   - Initialize 4 worker sockets, each connecting to the server on a different socket from the main client.
   - Create 4 threads, with each thread running `worker.get_file(...)` to download a chunk of the file.
2. **Update UI and Combine File Chunks**
   - Continuously update the UI to reflect the download progress.
   - After all threads have completed, combine the 4 file chunks into the final file.

6.6. Worker File Download

Function: `worker.get_file(...)`

Steps:

1. **Request File Chunk**
   - Send a `WORKER_GET_CHUNK` request to the server to initiate chunk retrieval.
2. **Confirm Transfer Details**
   - Receive an "OK" from the server.
   - Request specific details: file name, file size, and chunk information.
3. **Receive and Write File Chunk**
   - Open the target file for writing.
   - Send an "OK" message to the server to confirm readiness.
   - Begin streaming and receiving the file chunk.

4. **Close the File**
    - After the transfer is complete, close the file to finalize the process.

7. Server Protocol

7.1. Initialize Server

1. **Setup Winsock Library**
    - Use `WSAStartup` to initialize Winsock.
    - Create a socket with `socket(AF_INET, SOCK_STREAM, 0)`.
2. **Bind and Listen**
    - Bind the socket to the provided IP and port.
    - Start listening for incoming connections with `listen()`.
3. **Accept Client Connections**
    - Use `accept()` to handle incoming connections.
    - For each client, spawn a new thread to manage its session via the `handle_client()` function.

7.2. Handle Client Requests

The `handle_client()` function manages communication with a client socket.

7.2.1 Welcome Message

- Send a short message to the client upon connection:
    - "Welcome to the server! You are client [client_id]."

7.2.2 Process Client Commands

- The server receives client messages via `recv()`. These messages are parsed into commands (e.g., `GET_LIST`, `DOWNLOAD_FILE`, `WORKER_GET_CHUNK`, `QUIT`).

Supported Commands:

1. **`GET_LIST`**
    - Calls `serve_list()` to:

    - Retrieve a list of files available on the server.
    - Send the list to the client in a structured message.
2. **`DOWNLOAD_FILE`**
    - Calls `check_file_to_download()` to:
        - Validate the requested file.
        - Prepare for transfer and coordinate with worker sockets for multi-threaded file download.
3. **`WORKER_GET_CHUNK`**
    - Calls `serve_chunk()` to:
        - Serve a specific chunk of the file to a worker socket.
        - Handle chunk indexing and ensure data integrity.
4. **`QUIT`**
    - Closes the client connection upon receiving this command.

7.3. Serve File Chunks (Worker Sockets)

Worker sockets are used to serve file chunks for multi-threaded downloads. This process involves:

- **Command**: `WORKER_GET_CHUNK` .
- **Steps**:
    i. Validate the request.
    ii. Identify the requested file chunk.
    iii. Stream the file chunk to the worker socket.
    iv. Wait for acknowledgment from the worker before continuing.

7.4. Cleanup

1. **Close Client Socket**
    - After handling all client requests or receiving the `QUIT` command, close the client socket using `closesocket()` .
2. **Server Shutdown**
    - Clean up Winsock resources with `WSACleanup()` if the server is terminated.

To do

- [x] complete 4 worker socket
- [x] find a way for worker to update progress to master
- [x] complete UI
- [x] report the protocols and idea
- [ ] clean the server log
- [x] progress bar for concating files
- [x] read small size chunk -> utilize the most of cache -> currently use << rbuff
- [x] add total byte downloaded.
- [x] dowload file with space in name.
- [ ] add list of function used in the program into the report