Name:	Dy, Sealtiel B.	Section:	S13
Name:	Lasala, Kyle Carlo C.	Section:	S13
Name:	Manlises, Maria Monica	Section:	S13

1.0 Machine Project – File Exchange System

OVERVIEW

The program is a simple File Exchange System with additional messaging features created using Python. A graphical user interface (GUI) was also created for the client for ease of use. An input field accepts commands which are then verified by the client application prior to being sent to the server for additional verification before execution. The client sees system messages through the console of the GUI to know if an error occurred or if the command was successfully executed.

SERVER APPLICATION

A class was created for the server application. It has its own variables and methods.

Importing Libraries (server.py)

Necessary modules such as the socket, threading, datetime, and os modules are imperative to be imported. These will be used in the server application.

```
# Import libraries
from socket import *
import threading
from datetime import datetime
import os
```

Server Class Initialization (server.py)

The server class needs to be initialized with its variables including the port number, socket, dictionary of clients, and dictionary of commands. Here, the socket was also initialized and bound to the localhost or ‘127.0.0.1’ and the port number (12345).

```
class Server:
    # Initialize the server
    def __init__(self):
        # Set port to 12345
        self.serverPort = 12345
        # Create a server socket and bind it to the localhost and port
        self.serverSocket = socket(AF_INET, SOCK_STREAM)
        self.serverSocket.bind(('127.0.0.1', self.serverPort))
        # Start listening for connections
        self.serverSocket.listen()
        # Initialize a dictionary to store clients
        self.clients = {} # key: handle, values: (socket, address)
        # Define the dictionary of commands with description, usage, and method to call
        self.commands = {
            "/leave": {
                "desc": "Disconnect from the server application",
                "usage": "/leave",
                "call": self.disconnect_client
            },
            "/register": {
                "desc": "Register a unique handle or alias",
                "usage": "/register <handle>",
                "call": self.register_handle
            },
            "/store":{
                "desc": "Send file to server",
                "usage": "/store <handle> <filename>",
                "call": self.store_file
            },
            "/dir": {
                "desc": "Request directory file list from a server",
                "usage": "/dir",
                "call": self.get_dir
            },
            "/get":{
                "desc": "Fetch a file from a server",
                "usage": "/get <filename>",
                "call": self.get_file
            },
            "/pm": {
                "desc": "Message another client connected to the server",
                "usage": "/pm <client_name> <message>",
                "call": self.message
            },
            "/all": {
                "desc": "Message clients connected to the server",
                "usage": "/all <message>",
                "call": self.message_all
            }
        }
```

During the initialization, the server also already starts actively listening for incoming connection requests. When there is an incoming connection, the server allows the client to connect two sockets, one general socket, and one messaging socket specifically for handling the additional messaging features. The server also creates a thread to handle the client through the ‘handle_client’ method. Any IO errors are caught and printed.

```
# Keep listening for connections
        while True:
            print('CSNETWK Web Server is ready to serve...')

            try:
                # Accept an incoming connection
                consoleSocket, addr = self.serverSocket.accept()
                messageSocket, _ = self.serverSocket.accept()
                # Create a thread to handle the client
                thread = threading.Thread(target=self.handle_client, args=(consoleSocket,
messageSocket, addr))
                thread.start()
                # Display number of active connections
                print(f"[ACTIVE CONNECTIONS] {threading.active_count() - 1}")

            except IOError as e:
                # Catch IO Errors
                print(f"Error: {e}")
```

Handle Client Connections (server.py)

This method allows for the server to receive commands from the client. Additional parameters are appended for certain commands. This method also calls the respective server methods associated with the received commands. Errors are caught and printed.

```
# Method to handle client connections
    def handle_client(self, connectionSocket, secondSocket, addr):
        print(f"[NEW CONNECTION] {addr} connected.")
        while True:
            try:
                # Receive data from the client
                data = connectionSocket.recv(1024).decode('utf-8')
                # Disconnect if empty string is received
                if not data:
                    print(f"[DISCONNECTED] {addr} disconnected.")
                    break

                # Split the received data to be saved as a list
                command = data.split(' ')
                if isinstance(command, str):
                    args = [command]
                elif isinstance(command, list):
                    args = command

                # If /register, append the client socket and address for later
                if args[0] == '/register':
                    args.append(connectionSocket)
                    args.append(secondSocket)
                    args.append(addr)

                # If /store or /get, append the client socket for later
                if args[0] == '/store' or args[0] == '/get':
                    args.append(connectionSocket)

                # Call the corresponding method based on the args
                # Store the response
                res = self.commands[args[0]]["call"](args)

                # Send the response to the client socket
                connectionSocket.send(res.encode('utf-8'))

            except Exception as e:
                # Print error
                print(f"Error: {str(e)}")
                break
```

Register Client Handle (server.py)

This method is called when the client sends a request to register their handle. It first checks if the requested handle already exists. If the handle is unique, the server saves the client, including its socket, messaging socket, and address to the dictionary of clients with the handle as the key. A “Welcome” response is sent back to the client to indicate successful registration. If any errors are caught, they are printed and returned as a response to the client.

```
# Method to register the alias of the client
    def register_handle(self, params):
        try:
            # Check if the handle is already existing
            if params[1] in self.clients.keys():
                raise Exception("Registration failed. Handle or alias already exists.")
            # Saving the client to the dictionary of clients
            self.clients[params[1]] = {"socket" : params[2], "message_socket" : params[3],
"address" : params[4]}
            # Return a response to indicate successful registration
            return f"Welcome {params[1]}!"
        except Exception as e:
            # Print error
            errorMsg = f"{e}"
            print("Error:", errorMsg)
            return errorMsg
```

Disconnect Client Connections (server.py)

This method is called when the client sends a request to disconnect. It first checks if the client is registered in order to delete it from the dictionary of clients. The actual socket disconnection happens on the client side. The server then returns a response to indicate successful disconnection. If any errors are caught, they are printed and returned as a response to the client.

```
# Method to disconnect the client from the server
    def disconnect_client(self, params):
        try:
            # Delete the client from the list of clients if registered
            if params[1] is not None and params[1] in self.clients.keys():
                del self.clients[params[1]]
            # Return a response to indicate successful disconnection
            return "Connection closed. Thank you!"
        except Exception as e:
            # Print error
            errorMsg = f"{e}"
            print("Error:", errorMsg)
            return errorMsg
```

Store File in The Server (server.py)

This method implements the storing of the file sent by the client to the server, specifically in the 'server_files' folder. It first creates the said folder if it doesn't exist yet. It also includes the logic of appending a number to the filename if the name of the file to be stored already exists in the folder. For example, if 'pikachu.txt' already exists in the folder, if a client uploads another file named 'pikachu.txt', it instead gets saved as 'pikachu-1.txt'. This method creates the new file in the 'server_files' folder and writes the 1024-bit chunks it receives from the socket making sure not to include the end of file (EOF) tag. It also returns a response indicating either successful storage or an encountered error.

```
# Method to store a file sent by the client to the server
    def store_file(self, params):
        try:
            # Save client socket and filename as variables
            connectionSocket = params[3]
            filename = params[2]
            # Get the current directory and files inside
            dir = os.path.dirname(os.path.abspath(__file__))
            dir_files = os.listdir(dir)

            # Create the folder 'server_files' to save the files
            if 'server_files' not in dir_files:
                os.mkdir(dir + '/server_files')

            # Get the list of files in the 'server_files' folder
            dir_files = os.listdir(dir + '\\server_files')

            # Logic for appending -n to the nth version of the file with the same name
            if filename in dir_files:
                i = 1
                actual_file = filename.split('.')
                # Add a number to the end of the filename
                if isinstance(actual_file, list):
                    filename = ''
                    for j in range(len(actual_file) - 1):
                        filename += actual_file[j]
                        if j != len(actual_file) - 2:
                            filename += '.'

                    while (f'{filename}-{i}.{actual_file[-1]}') in dir_files:
                        i += 1
                    filename = f'{filename}-{i}.{actual_file[-1]}'
                else:
                    filename = actual_file
                    while (f'{filename}-{i}') in dir_files:
                        i += 1
                    filename = f'{filename}-{i}'

            # Writing the file data to the newly created file
            with open(dir + '\\server_files\\' + filename, 'wb') as file:
                while True:
                    # Get 1024 bits of data at a time
                    data = connectionSocket.recv(1024)
                    # If last chunk of data, write but excluding the EOF tag
                    if b"<<EOF>>" in data:
                        file.write(data[:-7])
                        break
                    # Write to the file
                    file.write(data)
                # Close the file
                file.close()

            # Return a response indicating successful storage of file in server
            msg = f"{params[1]} <{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}>: Uploaded
{filename}"
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"{e}"
            print("Error:", errorMsg)
            return errorMsg
```

Get Directory of Files in The Server (server.py)

This method gets the directory of files stored in the server. It first gets the path to the ‘server_files’ folder and returns a response message containing the list of files. It can also return an error message if one is encountered.

```
# Method to get the directory of files stored in the server
    def get_dir(self, params):
        try:
            # Getting the path to the 'server_files' folder
            dir = os.path.dirname(os.path.abspath(__file__)) + '\\server_files'
            # Getting the list of files in the directory
            dir_files = os.listdir(dir)

            # Preparing the message to be returned as response
            msg = "----------------------\n|| SERVER DIRECTORY ||\n"
            # Iterating through the files and adding the filenames to the response
            for i in range(len(dir_files)):
                msg += dir_files[i] + ('\n- ' if i != len(dir_files) - 1 else '')
            msg += "\n----------------------\n"
            # Return the response containing the files and indicating successful directory
getting
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"{e}"
            print("Error:", errorMsg)
            return errorMsg
```

Get a File from the Server (server.py)

This method facilitates the sending of the file contents of a requested file to the client who requested it. It first checks if a ‘server_files’ folder exists before checking if the requested file is inside said folder. Afterwards, it opens the file for reading and sends the contents in 1024 bits at a time to the socket followed by the EOF tag. It returns a response indicating whether the file was successfully fetched or not as it also is wary of IO errors.

```
# Method for a client to download a file from the server
    def get_file(self, params):
        try:
            # Save client socket and filename as variables for later
            connectionSocket = params[2]
            filename = params[1]

            # Get the current directory and files inside
            dir = os.path.dirname(os.path.abspath(__file__))
            dir_files = os.listdir(dir)

            # Check if the 'server_files' folder exists
            if 'server_files' not in dir_files:
                raise Exception("No files in the server")

            # Change directory to be inside 'server_files' folder
            dir += '\\server_files'
            dir_files = os.listdir(dir)

            # Check if file is in the 'server_files' folder
            if filename not in dir_files:
                raise Exception("File not found in the server")

            # Open the file for reading
            with open(dir + '\\' + filename, 'rb') as file:
                # Read the initial 1024 bits
                file_data = file.read(1024)
                while file_data:
                    # Send the chunks to the socket and continue reading
                    connectionSocket.send(file_data)
                    file_data = file.read(1024)
                # Send the EOF tag to the socket
                connectionSocket.send(b"<<EOF>>")
                # Close the file
                file.close()

            # Return a response indicating successful download from server
            msg = f"File received from Server: {filename}"
            print(msg)
            return msg

        except IOError:
            # Catch any IO errors
            errorMsg = "Error: IO Error."
            print(errorMsg)
            return errorMsg
        except Exception as e:
            # Print other Errors
            errorMsg = f"{e}"
            print("Error:", errorMsg)
            return errorMsg
```

Unicast Message (server.py)

This method facilitates the sending of a message from one registered client to another registered client. It makes use of the 'message_socket' of clients instead of the general socket. It first checks if the destination client exists and it also prevents the client from messaging themselves. Afterwards, it simply sends the message to the messaging socket of the destination or receiving client. This method returns a response indicating successful message sending or an error message if one were to occur.

```
# Method to allow client to message another client through unicast
    def message(self, params):
        try:
            # Save the source and destination client's handles
            source_name = params[1]
            dest_name = params[2]
            # Save the message, allowing messages with spaces by connecting the words
            msg = ' '.join(params[3:])

            # Check if the destination client exists
            if dest_name not in self.clients.keys():
                raise Exception("Client not found in the server.")
            # Check if the client is trying to message themselves
            if source_name == dest_name:
                raise Exception("Cannot message yourself.")
            # Send the message to the messaging socket of the destination client
            dest_socket = self.clients[dest_name]["message_socket"]
            dest_socket.send(f"{source_name}: {msg}".encode('utf-8'))

            # Return a response indicating successful sending of the message
            msg = f"Message sent to {dest_name}."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"{e}"
            print("Error:", errorMsg)
            return errorMsg
```

Broadcast Message (server.py)

This method facilitates the sending of a message from one registered client to all other registered clients. It makes use of the ‘message_socket’ of clients instead of the general socket. It loops through all registered clients and also prevents the client from messaging themselves. Afterwards, it simply sends the message to the messaging sockets of the destination or receiving clients. This method returns a response indicating successful message sending or an error message if one were to occur.

```
# Method to allow client to broadcast a message to all other registeredclients
    def message_all(self, params):
        try:
            # Save the source client's handle
            source_name = params[1]
            # Save the message, allowing messages with spaces by connecting the words
            msg = ' '.join(params[2:])

            # Send the message to the messaging sockets of all other clients
            for dest_name in self.clients.keys():
                if dest_name != source_name:
                    dest_socket = self.clients[dest_name]["message_socket"]
                    dest_socket.send(f"{source_name} (to all): {msg}".encode('utf-8'))

            # Return a response indicating successful broadcasting of the message
            msg = f"Message sent to all other clients."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"{e}"
            print("Error:", errorMsg)
            return errorMsg
```

Server Instantiation (server.py)

When server.py is ran, an instance of the class Server() is created.

```
server = Server()
```

CLIENT APPLICATION

Importing Libraries (client.py)

Necessary modules such as the socket, os, and threading modules were imported for the client application. The tkinter module was also imported for the Graphical User Interface.

```
# Import libraries
import socket
import tkinter as tk
from tkinter import scrolledtext, Entry, Button
import os
import threading
```

Client Class Initialization (client.py)

The client class needs to be initialized along with its variables. The client handle, socket, and messaging socket are initialized to None. Note that the messaging socket was created specifically for the additional messaging features of the application. The dictionary of commands was also defined here, detailing each command's description, usage, and the method that it will call. During initialization, the GUI is also set up using tkinter. It includes the GUI for the console and for the chat room, as well as the input area and send button.

```
class Client:
    # Initialize the server
    def __init__(self):
        # Initialize the client's handle
        self.handle = None
        # Initialize the client's socket
        self.socket = None
        # Initialize a separate socket for messaging
        self.message_socket = None
        # Define the dictionary of commands with description, usage, and method to call
        self.commands = {
            "/join": {
                "desc": "Connect to the server application",
                "usage": "/join <server_ip_add> <port>",
                "call": self.connect_to_server
            },
            "/leave": {
                "desc": "Disconnect from the server application",
                "usage": "/leave",
                "call": self.disconnect_from_server
            },
            "/register": {
                "desc": "Register a unique handle or alias",
                "usage": "/register <handle>",
                "call": self.register_handle
            },
            "/store": {
                "desc": "Send file to server",
                "usage": "/store <filename>",
                "call": self.send_file_to_server
            },
            "/dir": {
                "desc": "Request directory file list from a server",
                "usage": "/dir",
                "call": self.request_directory_list
            },
            "/get": {
                "desc": "Fetch a file from a server",
                "usage": "/get <filename>",
                "call": self.fetch_file_from_server
            },
            "/pm": {
                "desc": "Message another registered client connected to the server",
                "usage": "/pm <client_name> <message>",
                "call": self.message
            },
            "/all": {
                "desc": "Message other registered clients connected to the server",
                "usage": "/all <message>",
                "call": self.message_all
            },
            "/?": {
                "desc": "List of commands",
                "usage": "/?",
                "call": self.show_commands
            }
        }
# Tkinter GUI setup
        self.root = tk.Tk()
        self.root.title("File Exchange System")

        # GUI for the console
        self.output_text = scrolledtext.ScrolledText(self.root, width=70, height=20,
state='disabled')
        self.output_text.grid(row=0, column=0, padx=10, pady=10)

        # GUI for the chat room
        self.chat_room = scrolledtext.ScrolledText(self.root, width=50, height=20,
state='disabled')
        self.chat_room.grid(row=0, column=1, padx=10, pady=10)

        # GUI for the command entry and send button
        self.input_entry = Entry(self.root, width=70)
        self.input_entry.grid(row=1, columnspan=2, pady=5)
        self.send_button = Button(self.root, text="Send", command=self.send_user_input)
        self.send_button.grid(row=2, columnspan=2, pady=(5, 10))
        self.send_button.bind(func=self.send_user_input, sequence="<Return>")
```

Send User Input (client.py)

This method sends the user input from the GUI input entry to the server. It first gets the input and checks if the command is valid by calling the check_command method. It then calls the method associated with that command which then starts processing the request on the client side before it is sent to the server side and back. After processing the request, this method also updates the GUI with the response, whether it is successful or not.

```
# Sends user input to the server
    def send_user_input(self, event=None):
        # Get the input from the console
        command = self.input_entry.get()

        try:
            # Check if the command is valid and separate the command arguments
            isValid, args = self.check_command(command)
            # If the command is valid, call the method associated with the command
            if isValid:
                res = self.commands[args[0]]["call"](args) # Call the command's method

                # Display the response to GUI's console
                self.output_text.configure(state='normal')
                self.output_text.insert(tk.END, f"{res}\n")
                self.output_text.configure(state='disabled')

                # Clear the input field if the command is valid
                if 'Error:' not in res:
                    self.input_entry.delete(0, tk.END)
            else:
                raise Exception("Command not found.")

        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            self.output_text.configure(state='normal')
            self.output_text.insert(tk.END, f"{errorMsg}\n")
            self.output_text.configure(state='disabled')
```

Check Command (client.py)

This method, which is called in the send_user_input method checks whether a given client command is valid. It first checks that the command sent contains parameters. It then splits it in order to check if the first parameter is a valid command by cross-checking with the client's dictionary of commands.

```
# Method to check if the command is valid
    def check_command(self, params):
        # Check if there are parameters
        if len(params) == 0:
            return False, None
        # Split the received parameters into arguments
        args = params.split(" ")
        # If there is only 1 parameter, place it in a list
        if isinstance(args, str):
            args = [args]

        # Check if the first argument is a valid command
        # Return the arguments passed
        if args[0] in self.commands.keys():
            return True, args

        # Returns False if command is invalid
        return False, None
```

Connect To Server (client.py)

This method allows the client to connect to the server and is called using the '/join' command. It first checks for the correct number of parameters as per the usage directions. It also checks that the third parameter, port number, is numerical. It also verifies that the client has no existing connection socket. Afterwards, it creates both the general and messaging sockets which are connected to the server host and port. This method is also wary of connection errors, connection reset errors, and other exceptions that may occur or be raised.

```
# Method to connect to the server
    def connect_to_server(self, params):
        try:
            # Check that there are exactly 3 parameters and that the third param is numerical
            if len(params) != 3 or not params[2].isdigit():
                raise Exception("Command parameters do not match or is not allowed.")
            # Check that there is already an existing socket
            if self.socket is not None:
                raise Exception("Connection to the Server is already established.")

            # Save the server host name and port number
            server_host = params[1]
            server_port = int(params[2])
            # Initialize the general socket and messaging socket
            self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.message_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            # Connect both the general socket and messaging socket
            try:
                self.socket.connect((server_host, server_port))
                self.message_socket.connect((server_host, server_port))
            except:
                raise ConnectionError
            # Return a response indicating successful connection
            msg = "Connection to the File Exchange Server is successful!"
            print(msg)
            return msg
        except ConnectionResetError:
            # Check if connection to server has been reset
            errorMsg = f"Error: Connection to the Server has been reset."
            print(errorMsg)
            self.output_text.configure(state='normal')
            self.output_text.insert(tk.END, f"{errorMsg}\n")
            self.output_text.configure(state='disabled')
        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and
Port Number."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Disconnect From Server (client.py)

This method allows the client to disconnect from the server and is called using the '/leave' command. It first checks for the correct number of parameters and that the client is connected to the server. It then sends a disconnect request to the server, waiting for its response. If the command is successfully processed in the server, this method will close the sockets and set them to None. The client's handle will also be removed and a response indicating that the client was successfully disconnected will be returned.

```
# Method to disconnect from the server
    def disconnect_from_server(self, params):
        try:
            # Check that there is exactly 1 parameters
            if len(params) != 1:
                raise Exception("Command parameters do not match or is not allowed.")
            # Check that there is an existing socket
            if self.socket is None:
                raise Exception("Disconnection failed. Please connect to the server first.")

            # Send the disconnect command to the server
            self.socket.send(f'/leave {self.handle}'.encode('utf-8'))
            # Receive the server's response
            res = self.socket.recv(1024).decode('utf-8')

            # Check if the server has successfully disconnected
            if res == "Connection closed. Thank you!":
                # Close the sockets and set them to None
                self.socket.close()
                self.message_socket.close()
                self.socket = None
                self.message_socket = None
                self.handle = None
                return res
            else:
                raise Exception("Error: Disconnection failed.")

        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and
Port Number."
            print(msg)
            return msg

        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Register Handle

This method allows the client to register a handle and is called using the '/register' command. It first checks for the correct number of parameters and that the client is not yet registered but is already connected to the server. The method then sends the register command to the server, waiting for a response. If the command is successfully processed in the server, this method will also return a response indicating that the client was successfully registered. The method also catches connection errors and other exceptions.

```
# Method to register a handle
    def register_handle(self, params):
        try:
            # Check that there are exactly 2 parameters
            if len(params) != 2:
                raise Exception("Command parameters do not match or are not allowed.")
            # Check that the client is not yet registered
            if self.handle is not None:
                raise Exception("Already registered.")
            # Check that there is an existing socket
            if self.socket is None:
                raise Exception("Registration failed. Please connect to the server first.")

            # Save the user handle
            user_handle = params[1]
            # Send the register command to the server
            self.socket.send(f'/register {user_handle}'.encode('utf-8'))
            # Receive the server's response
            res = self.socket.recv(1024).decode('utf-8')
            # Check if the server has successfully registered the handle
            if res == f"Welcome {user_handle}!":
                self.handle = user_handle
                # Start a thread to listen for messages
                threading.Thread(target=self.receive).start()
                return res
            else:
                raise Exception(res)

        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and
Port Number."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Send File to Server (client.py)

This method allows the client to send a file to the server using the ‘/store’ command. It first checks for the correct number of parameters and that the client is already connected and registered. It then gets the current directory to check if the file to be stored is available for sending. It then sends the command to store to the server and sends 1024-bit chunks of the file for the server to receive. It ends the file contents with an EOF tag. It then awaits the server’s response, returning the status of the process. This method catches connection errors, IO errors, and other exceptions.

```
# Method to send a file to the server
    def send_file_to_server(self, params):
        try:
            # Check that there is exactly 2 parameters
            if len(params) != 2:
                raise Exception("Command parameters do not match or are not allowed.")
            # Check that there is an existing socket
            if self.socket is None:
                raise Exception("Cannot send file. Please connect to the server first.")
            # Check that the client is already registered
```

```python
            if self.handle is None:
                raise Exception("Cannot send file. Please register first.")

            # Save the filename
            filename = params[1]
            # Get the current directory and files inside
            dir = os.path.dirname(os.path.abspath(__file__))
            dir_files = os.listdir(dir)

            # Check that requested file is in the directory
            if filename not in dir_files:
                raise Exception("File not found.")

            # Send the storing command to the server
            self.socket.send(f"/store {self.handle} {filename}".encode('utf-8'))

            # Reading the file
            with open(dir + '\\' + filename, 'rb') as file:
                # Read the initial 1024 bits
                file_data = file.read(1024)
                while file_data:
                    # Send the chunks to the socket and continue reading
                    self.socket.send(file_data)
                    file_data = file.read(1024)
                # Send tht EOF tag to the socket
                self.socket.send(b'<<EOF>>')
                # Close the file
                file.close()

            # Receive the server's response
            res = self.socket.recv(1024).decode('utf-8')

            # Check if response from server is OK and print as follows
            if "Uploaded" in res:
                print(res)
                return res
            else:
                raise Exception(res)

        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and
Port Number."
            print(msg)
            return msg
        except IOError:
            # Catch any IO errors
            errorMsg = "Error: File not found."
            print(errorMsg)
            return errorMsg
        except Exception as e:
            # Print other errors
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Request Directory List (client.py)

This method requests the list of files stored in the server using the '/dir' command. It checks for the correct number of parameters and that the client is already connected and registered. It then sends the request to the server, awaiting a response. This displays the directory if successful, otherwise displaying errors such as connection errors or other exceptions found.

```
# Method to request the directory list from the server
    def request_directory_list(self, params):
        try:
            # Check that the number of parameters is exactly 1
            if len(params) != 1:
                raise Exception("Command parameters do not match or are not allowed.")
            # Check that there is an existing socket
            if self.socket is None:
                raise Exception("Cannot check file directory. Please connect to the server
first.")
            # Check that the client is already registered
            if self.handle is None:
                raise Exception("Cannot check file directory. Please register to the server
first.")
            # Send the directory command to the server
            self.socket.send(("/dir").encode('utf-8'))
            # Receive the response from the server
            res = self.socket.recv(1024).decode('utf-8')
            # Check if the server has successfully sent the directory list
            if "SERVER DIRECTORY" in res:
                print(res)
                return res
            else:
                raise Exception(res)
        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and
Port Number."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Fetch File From Server (client.py)

This method allows the client to fetch a file from the server using the ‘/get’ command. It first checks that the number of parameters is correct and that the client is connected and registered. It then sends the command to the server for processing. In case there already exists a file with the same filename in the client’s directory, a number is appended at the end of the file name. Based on the server’s response, the client will know if it is okay to proceed receiving the file contents and do so in 1024-bit chunks. It ensures the EOF tag is not included in the file contents and returns a message indicating the successful fetching of file. It also catches connection errors and other exceptions.

```
# Method to fetch a file from the server
    def fetch_file_from_server(self, params):
        try:
            # Check that the number of parameters is exactly 2
            if len(params) != 2:
                raise Exception("Command parameters do not match or are not allowed.")
            # Check that there is an existing socket
            if self.socket is None:
                raise Exception("Cannot fetch file. Please connect to the server first.")
            # Check that the client is already registeref
            if self.handle is None:
                raise Exception("Cannot fetch file. Please register first.")

            # Save the filename
            filename = params[1]
            # Send the get file command to the server
            self.socket.send(f"/get {filename}".encode('utf-8'))
            # Get the current directory and files inside
            dir = os.path.dirname(os.path.abspath(__file__))
            dir_files = os.listdir(dir)

            # Logic for appending -n to the nth version of the file with the same name
            if filename in dir_files:
                i = 1
                actual_file = filename.split('.')
                # Add a number to the end of the filename
                if isinstance(actual_file, list):
                    filename = ''
                    for j in range(len(actual_file) - 1):
                        filename += actual_file[j]
                        if j != len(actual_file) - 2:
                            filename += '.'

                    while (f'{filename}-{i}.{actual_file[-1]}') in dir_files:
                        i += 1
                    filename = f'{filename}-{i}.{actual_file[-1]}'
                else:
                    filename = actual_file
                    while (f'{filename}-{i}') in dir_files:
                        i += 1
                    filename = f'{filename}-{i}'

            proceed = self.socket.recv(1024).decode('utf-8')
            if proceed != "Proceed to receive.":
                raise Exception(proceed)

            # Writing the file data to the newly created file
            with open(dir + '\\' + filename, 'wb') as file:
                while True:
                    # Get 1024 bits of data at a time
                    file_data = self.socket.recv(1024)
                    # If last chunk of data, write but excluding the EOF tag
                    if b'<<EOF>>' in file_data:
                        break
                    # Write to the file
                    file.write(file_data)
                # Close the file
                file.close()

            # Receive the server's response
            res = self.socket.recv(1024).decode('utf-8')

            # Check if the server has successfully sent the file
            if "File received from Server: " in res:
                print(f"File received from Server: {filename}")
                return f"File received from Server: {filename}"
            else:
                raise Exception(res)
        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and
Port Number."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Show Commands (client.py)

This method allows the client to see the list of commands in the command directory through the '/?' command.

```
# Method to show the list of commands
    def show_commands(self, params):
        # Print the list of commands with formatting
        msg = msg = "-----------------------\n|| COMMANDS ||\n"
        for command in self.commands:
            msg += f"{self.commands[command]['usage']} - {self.commands[command]['desc']}\n"
        msg += "\n----------------------\n"
        print(msg)
        return msg
```

Unicast Message (client.py)

This method allows the client to communicate with another specified registered client using the '/dm' command. It first checks that the number of parameters is correct, and that the client is connected and registered. It gets the destination client name from the command arguments and sends this along with the sending client's handle and the message to the server for processing. Based on the server's response, this method returns an indication of whether the messaging was successful. It also catches connection errors and other exceptions.

```
# Method for the client to message another client
    def message(self, params):
        try:
            # Check that there are at least 3 parameters
            if len(params) < 3:
                raise Exception("Command parameters do not match or are not allowed.")
            # Check that there is an existing socket
            if self.socket is None:
                raise Exception("Cannot send message. Please connect to the server first.")
            # Check that the client is already registered
            if self.handle is None:
                raise Exception("Cannot send message. Please register first.")

            # Save the destination client's handle
            destination_client_name = params[1]
            # Save the message, allowing messages with spaces by connecting the words
            message = " ".join(params[2:])
            # Send the message command to the server
            self.socket.send(f"/pm {self.handle} {destination_client_name}
{message}".encode('utf-8'))
```

```
            # Receive the server's response
            res = self.socket.recv(1024).decode('utf-8')

            # Check if the server has successfully sent the message
            if "Message sent" in res:
                print(res)
                return res
            else:
                raise Exception(res)

        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and 
Port Number."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Broadcast Message (client.py)

This method allows the client to broadcast a message to all other registered clients using the '/all' command. It first checks that the number of parameters is correct, and that the client is connected and registered. It the sending client's handle and the message to the server for processing. Based on the server's response, this method returns an indication of whether the messaging was successful. It also catches connection errors and other exceptions.

```
# Method for the client to message all other clients
    def message_all(self, params):
        try:
            # Check that there are at least 2 parameters
            if len(params) < 2:
                raise Exception("Command parameters do not match or are not allowed.")
            # Check that there is an existing socket
            if self.socket is None:
                raise Exception("Cannot send message. Please connect to the server first.")
            # Check that there is an existing handle
            if self.handle is None:
                raise Exception("Cannot send message. Please register first.")

            # Save the message, allowing messages with spaces by connecting the words
            message = " ".join(params[1:])
            # Send the message all command to the server
            self.socket.send(f"/all {self.handle} {message}".encode('utf-8'))

            # Receive the server's response
            res = self.socket.recv(1024).decode('utf-8')
            # Check if the server has successfully sent the message
            if res == "Message sent to all other clients.":
                print(res)
                return res
            else:
                raise Exception(res)
        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and
Port Number."
            print(msg)
            return msg
        except Exception as e:
            # Print error
            errorMsg = f"Error: {e}"
            print(errorMsg)
            return errorMsg
```

Receive Messages (client.py)

This method allows the client to receive messages from other clients through the server. The client application receives the messages through the message_socket which is continuously listening. The messages received are displayed in the GUI in a separate tab from the console messages. This method catches connection errors.

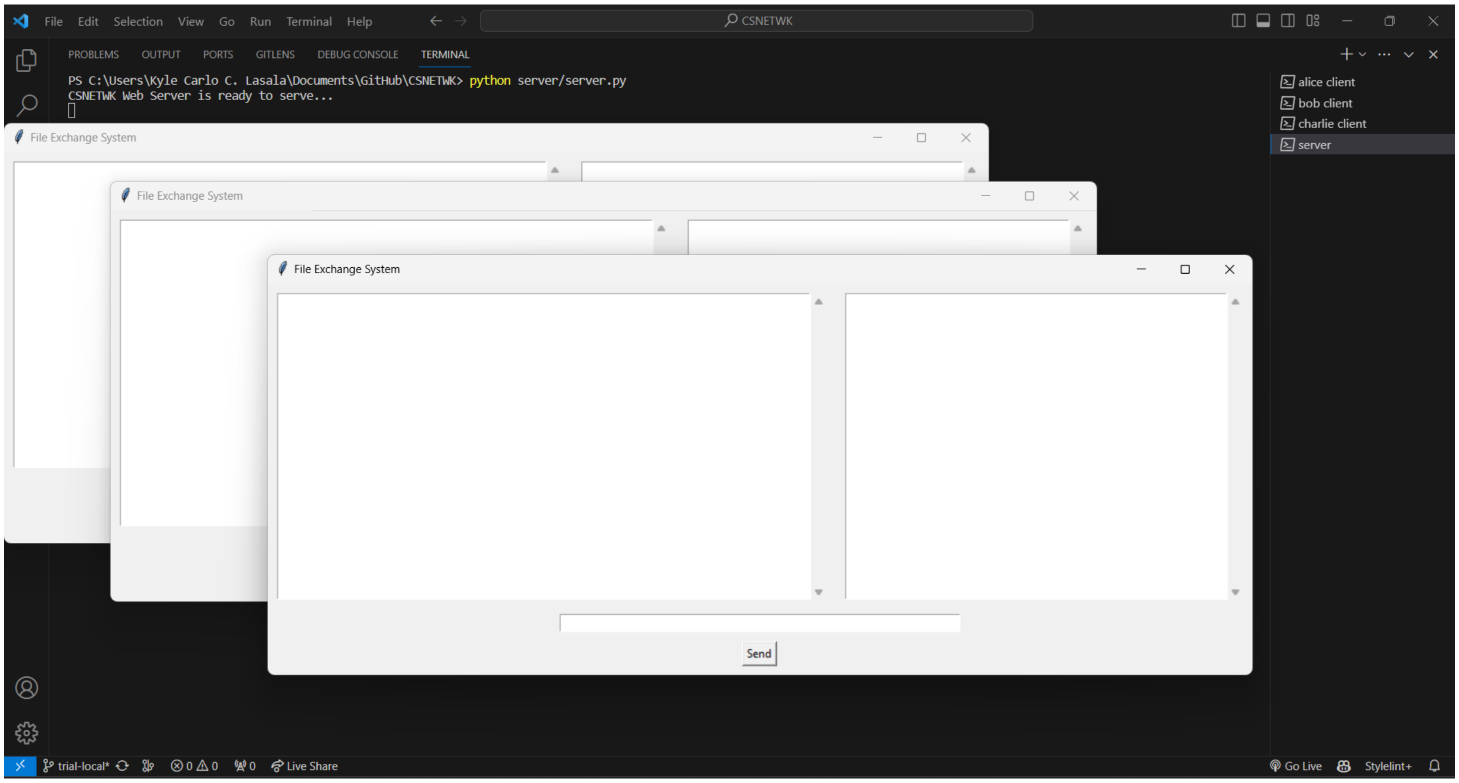
```
# Method for the client to receive messages
    def receive(self):
        # Continuously receive messages from the server
        try:
            while True:
                message = self.message_socket.recv(1024).decode('utf-8')

                if ":" not in message:
                    message = "[DISCONNECTED]."
                    print(message)
                    break
                # Display the message to the GUI's console
                self.chat_room.configure(state='normal')
                self.chat_room.insert(tk.END, f"{message}\n")
                self.chat_room.configure(state='disabled')
                print(message)
        except ConnectionError:
            # Check for possible connection errors
            # Set the sockets to None for safety
            self.socket = None
            self.message_socket = None
            # Print error
            msg = f"Error: Connection to the Server has failed! Please check IP Address and 
Port Number."
            print(msg)
            return msg
```

Client Instantiation (client.py)

When the client.py file is ran, an instance of a client class is created and an instance of the GUI is opened for the user to interact with.

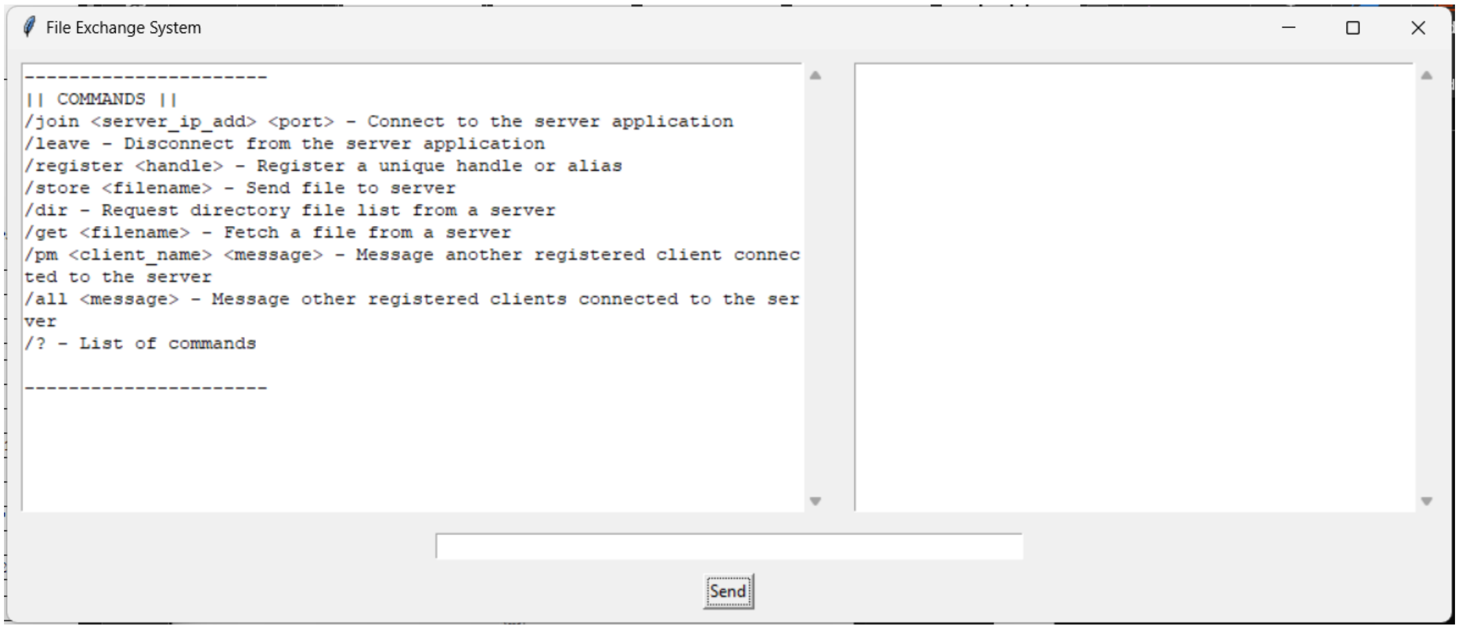
```
# Main method
if __name__ == "__main__":
    # Instantiate the client
    client = Client()
    # Start the GUI
    client.root.mainloop()
```

2.0 Screenshots



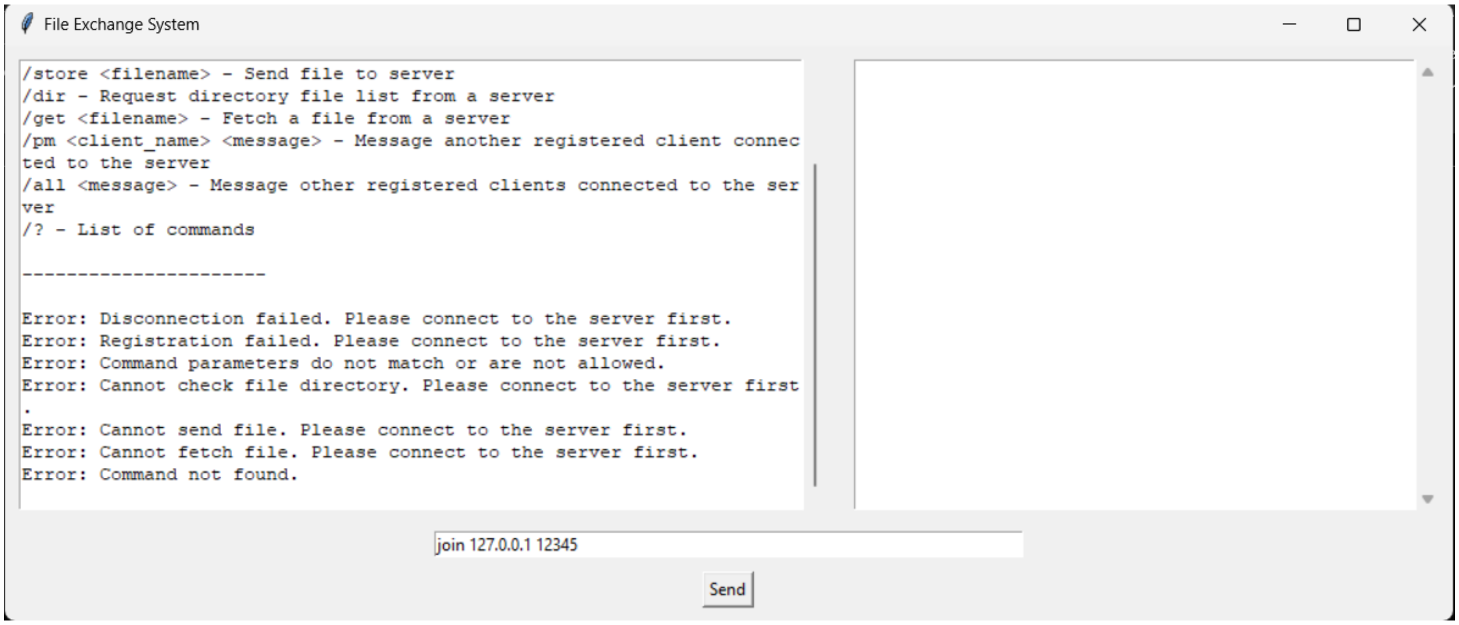
```
PS C:\Users\Kyle Carlo C. Lasala\Documents\GitHub\CSNETWK> python server/server.py
CSNETWK Web Server is ready to serve...
```

Figure 1 Running the server and 3 instances of clients (Alice, Bob, Charlie)



```
----------------------
|| COMMANDS ||
/join <server_ip_add> <port> - Connect to the server application
/leave - Disconnect from the server application
/register <handle> - Register a unique handle or alias
/store <filename> - Send file to server
/dir - Request directory file list from a server
/get <filename> - Fetch a file from a server
/pm <client_name> <message> - Message another registered client connec
ted to the server
/all <message> - Message other registered clients connected to the ser
ver
/? - List of commands

----------------------
```

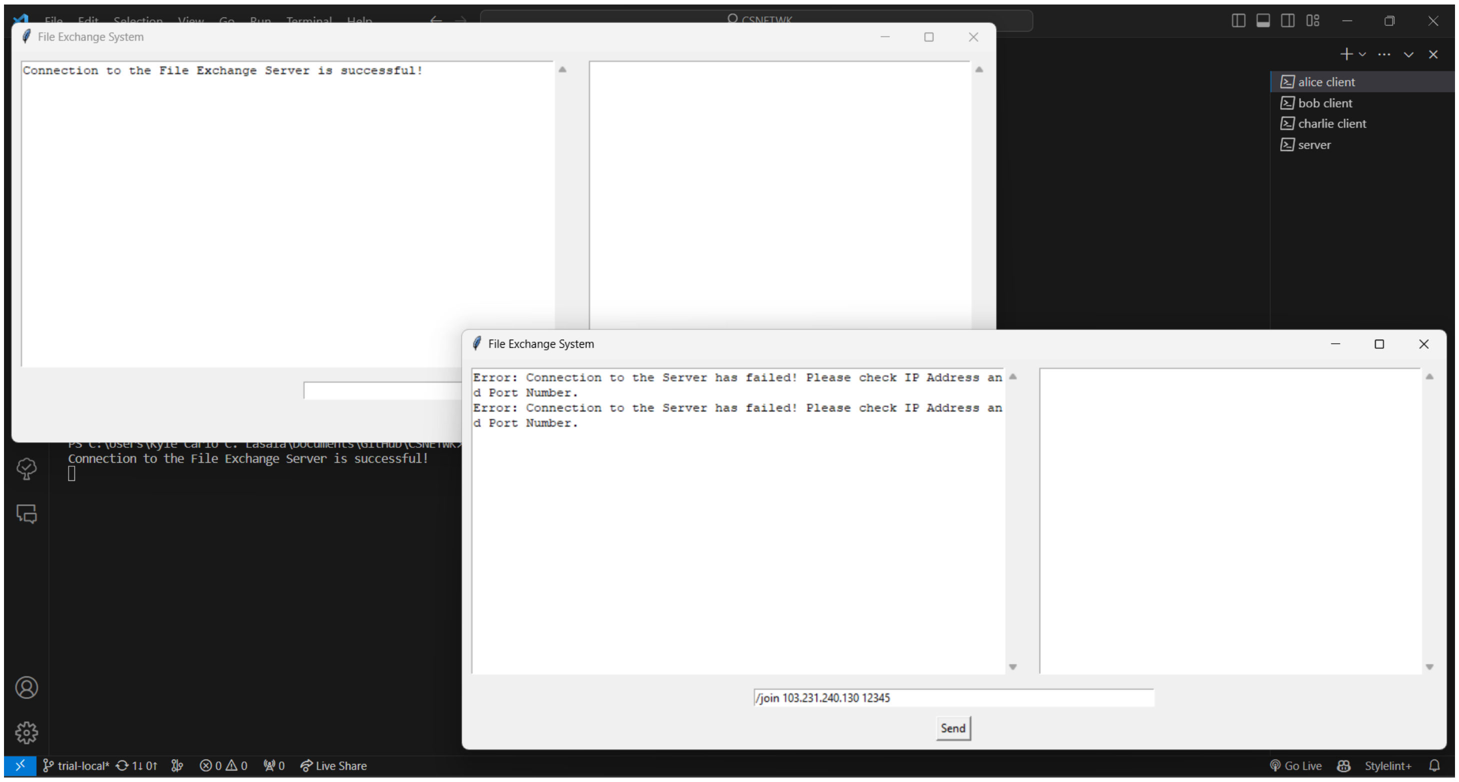
Figure 2 Using '/?' to check the list of commands



```
/store <filename> - Send file to server
/dir - Request directory file list from a server
/get <filename> - Fetch a file from a server
/pm <client_name> <message> - Message another registered client connec
ted to the server
/all <message> - Message other registered clients connected to the ser
ver
/? - List of commands

----------------------

Error: Disconnection failed. Please connect to the server first.
Error: Registration failed. Please connect to the server first.
Error: Command parameters do not match or are not allowed.
Error: Cannot check file directory. Please connect to the server first
.
Error: Cannot send file. Please connect to the server first.
Error: Cannot fetch file. Please connect to the server first.
Error: Command not found.
```

```
join 127.0.0.1 12345
```

Figure 3 Testing different commands if user hasn't connected to the server



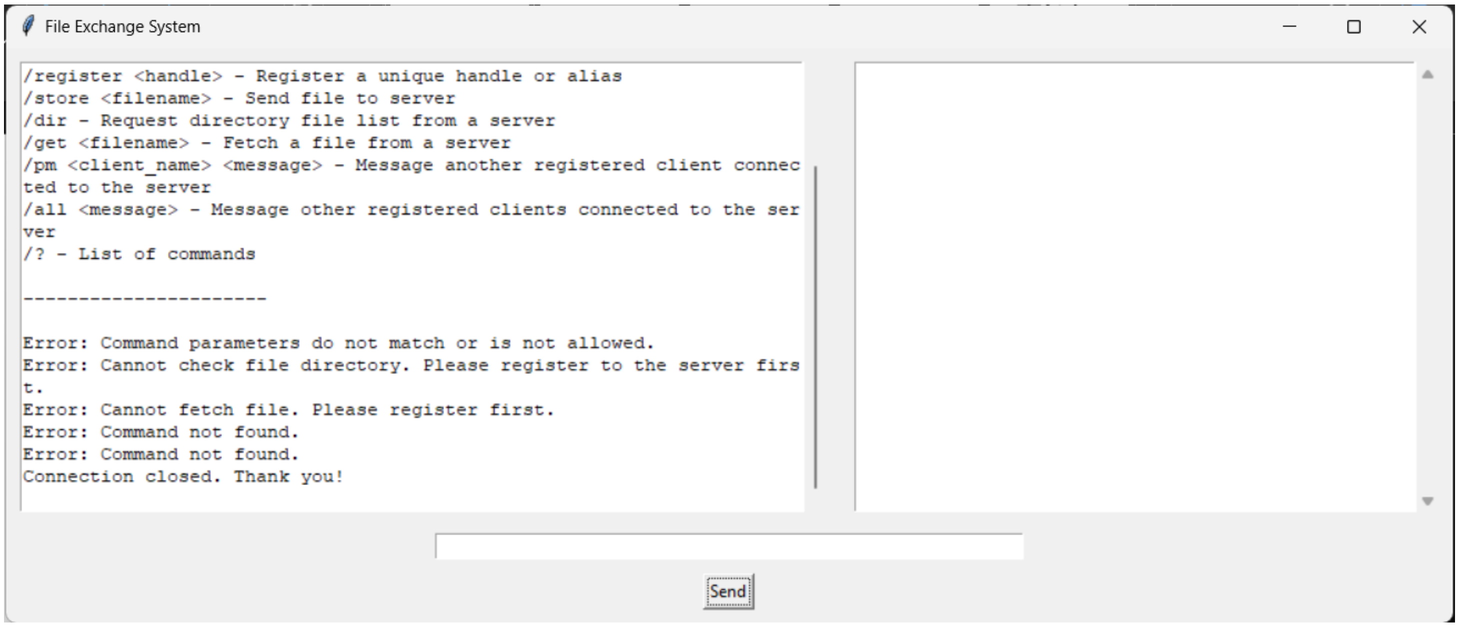
```
Connection to the File Exchange Server is successful!
```

```
Connection to the File Exchange Server is successful!
```

```
Error: Connection to the Server has failed! Please check IP Address an
d Port Number.
Error: Connection to the Server has failed! Please check IP Address an
d Port Number.
```

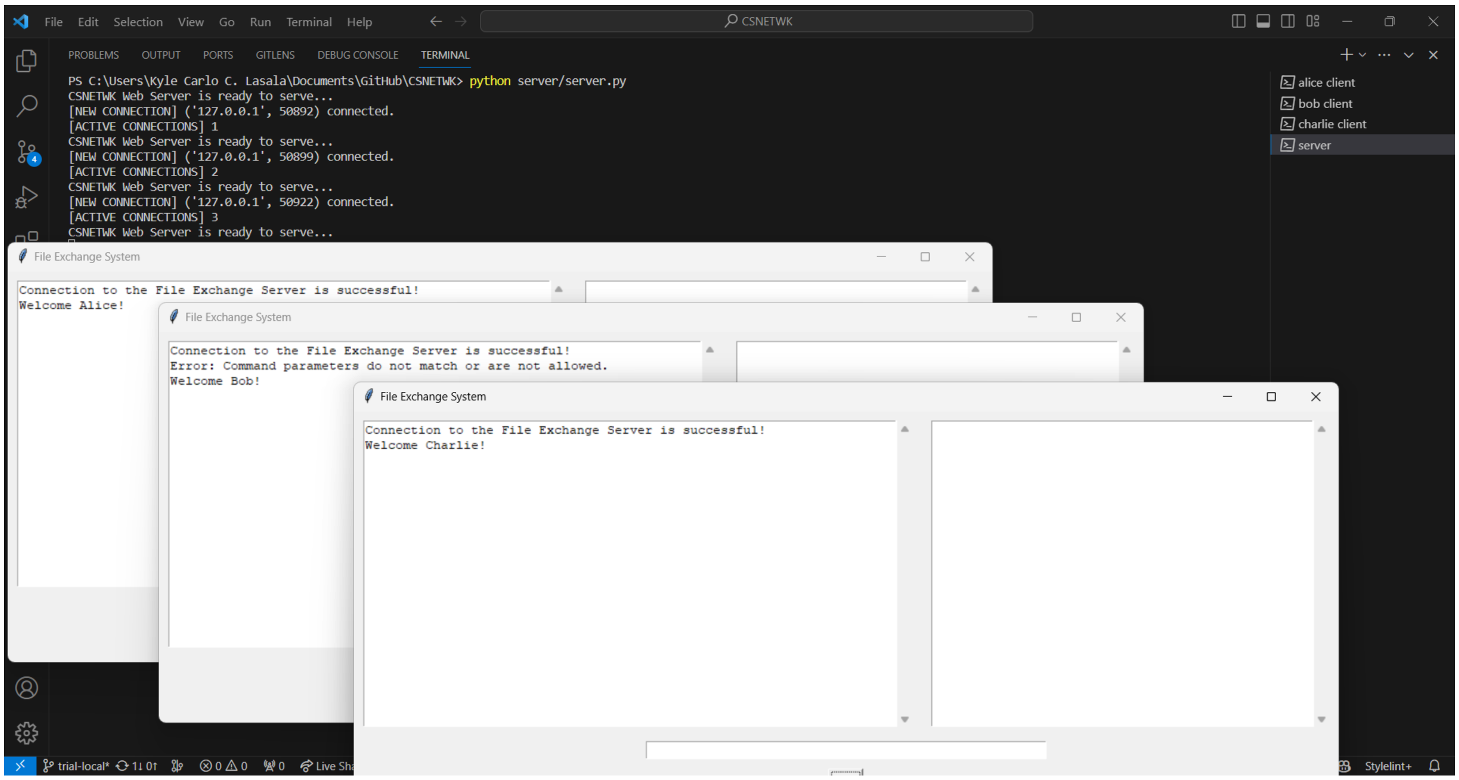
```
/join 103.231.240.130 12345
```

Figure 4 User successfully connects to the server



```
/register <handle> - Register a unique handle or alias
/store <filename> - Send file to server
/dir - Request directory file list from a server
/get <filename> - Fetch a file from a server
/pm <client_name> <message> - Message another registered client connec
ted to the server
/all <message> - Message other registered clients connected to the ser
ver
/? - List of commands

----------------------

Error: Command parameters do not match or is not allowed.
Error: Cannot check file directory. Please register to the server firs
t.
Error: Cannot fetch file. Please register first.
Error: Command not found.
Error: Command not found.
Connection closed. Thank you!
```

Figure 5 Testing different commands if user has connected to the server but is unregistered



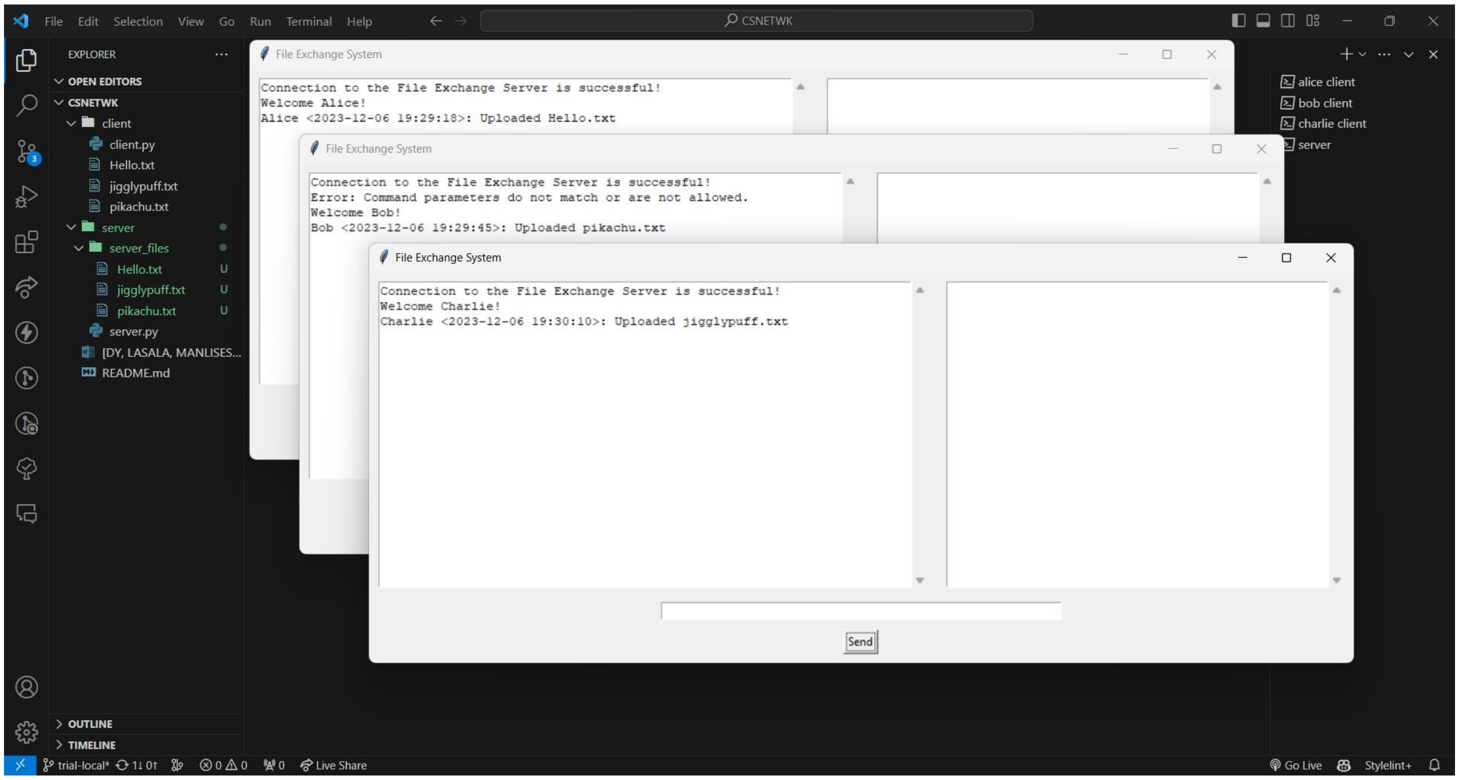
```
PS C:\Users\Kyle Carlo C. Lasala\Documents\GitHub\CSNETWK> python server/server.py
CSNETWK Web Server is ready to serve...
[NEW CONNECTION] ('127.0.0.1', 50892) connected.
[ACTIVE CONNECTIONS] 1
CSNETWK Web Server is ready to serve...
[NEW CONNECTION] ('127.0.0.1', 50899) connected.
[ACTIVE CONNECTIONS] 2
CSNETWK Web Server is ready to serve...
[NEW CONNECTION] ('127.0.0.1', 50922) connected.
[ACTIVE CONNECTIONS] 3
CSNETWK Web Server is ready to serve...
```

```
Connection to the File Exchange Server is successful!
Welcome Alice!
```

```
Connection to the File Exchange Server is successful!
Error: Command parameters do not match or are not allowed.
Welcome Bob!
```

```
Connection to the File Exchange Server is successful!
Welcome Charlie!
```

Figure 6 All 3 users successfully registering

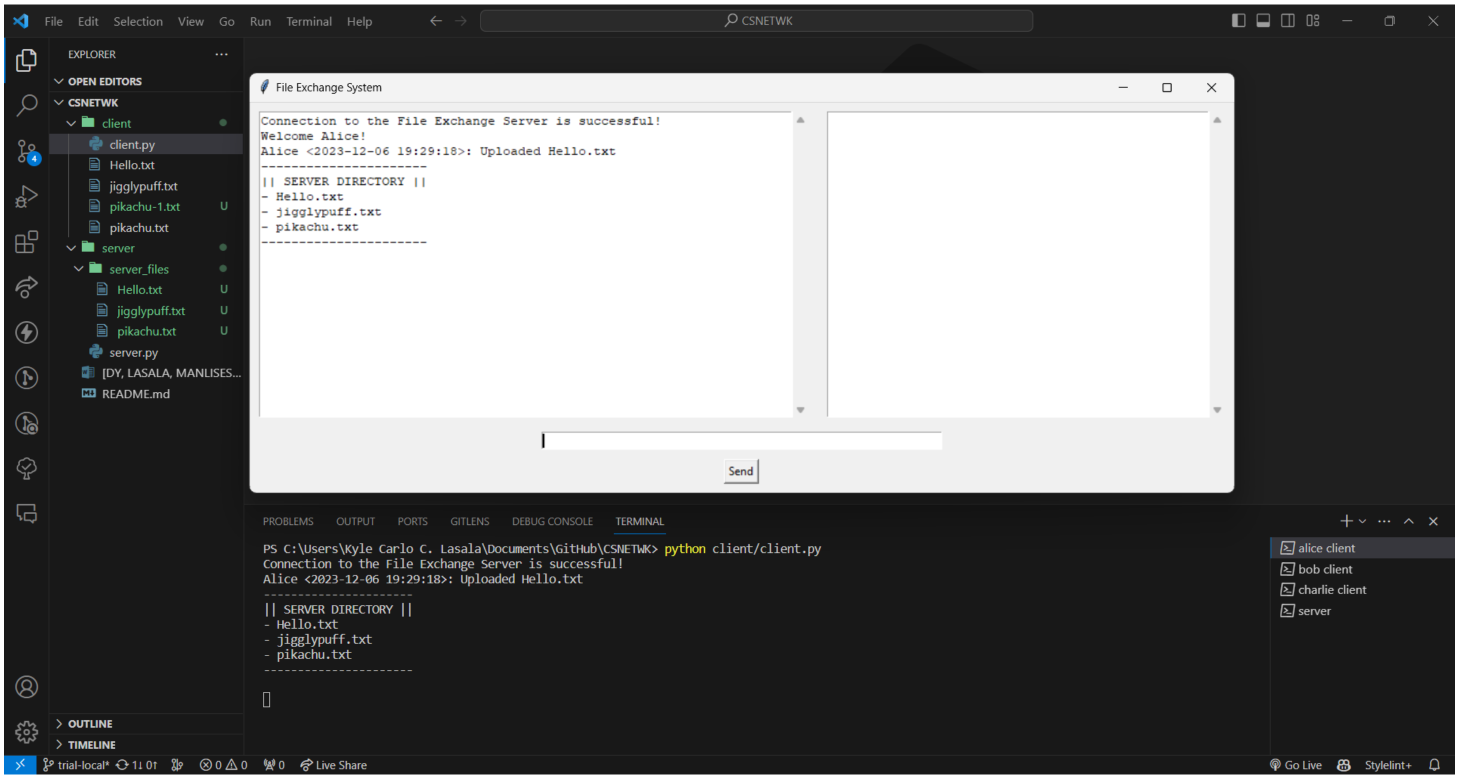


```
Connection to the File Exchange Server is successful!
Welcome Alice!
Alice <2023-12-06 19:29:18>: Uploaded Hello.txt
```

```
Connection to the File Exchange Server is successful!
Error: Command parameters do not match or are not allowed.
Welcome Bob!
Bob <2023-12-06 19:29:45>: Uploaded pikachu.txt
```

```
Connection to the File Exchange Server is successful!
Welcome Charlie!
Charlie <2023-12-06 19:30:10>: Uploaded jigglypuff.txt
```

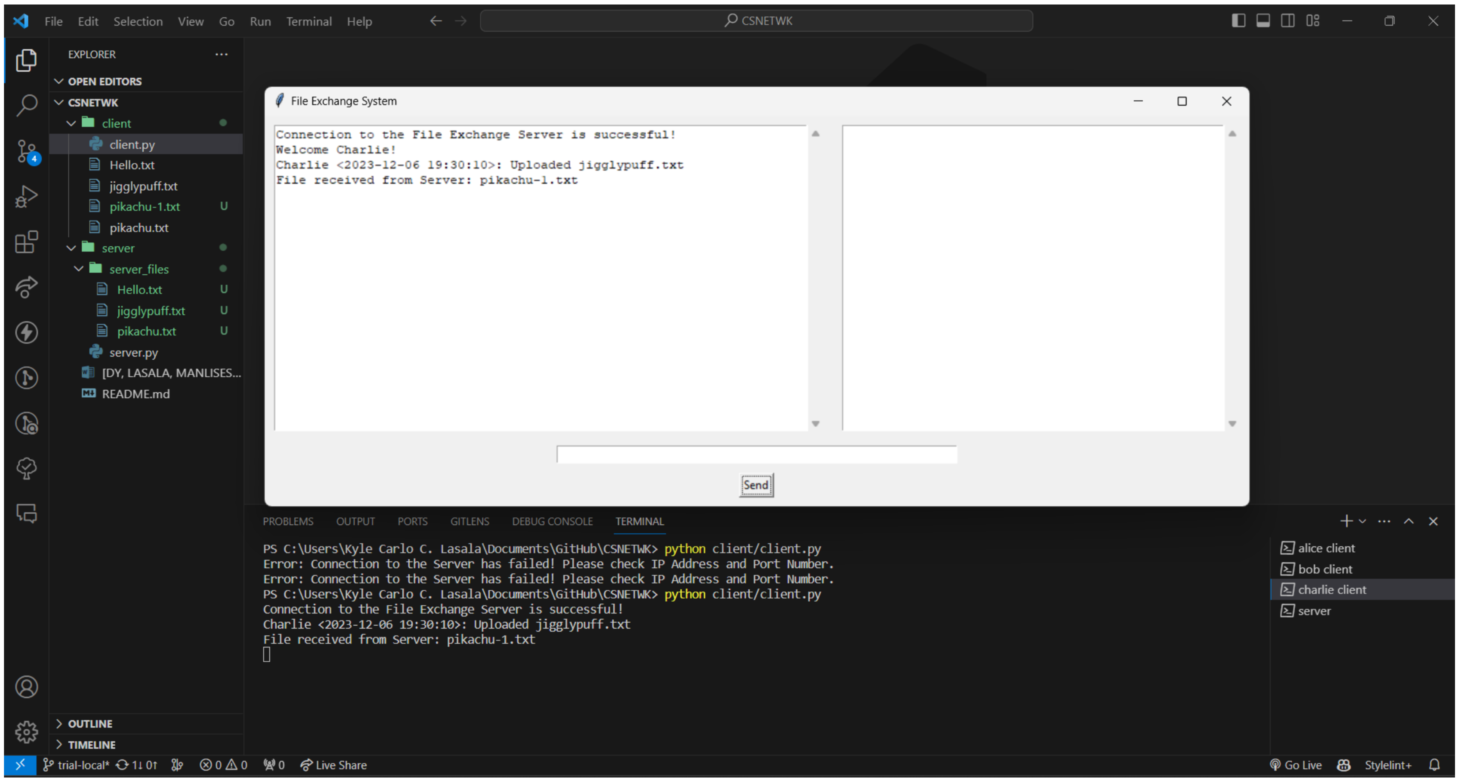
Figure 7 All 3 users successfully storing files



```
Connection to the File Exchange Server is successful!
Welcome Alice!
Alice <2023-12-06 19:29:18>: Uploaded Hello.txt
----------------------
|| SERVER DIRECTORY ||
- Hello.txt
- jigglypuff.txt
- pikachu.txt
----------------------
```

```
PS C:\Users\Kyle Carlo C. Lasala\Documents\GitHub\CSNETWK> python client/client.py
Connection to the File Exchange Server is successful!
Alice <2023-12-06 19:29:18>: Uploaded Hello.txt
---------------------
|| SERVER DIRECTORY ||
- Hello.txt
- jigglypuff.txt
- pikachu.txt
---------------------
```

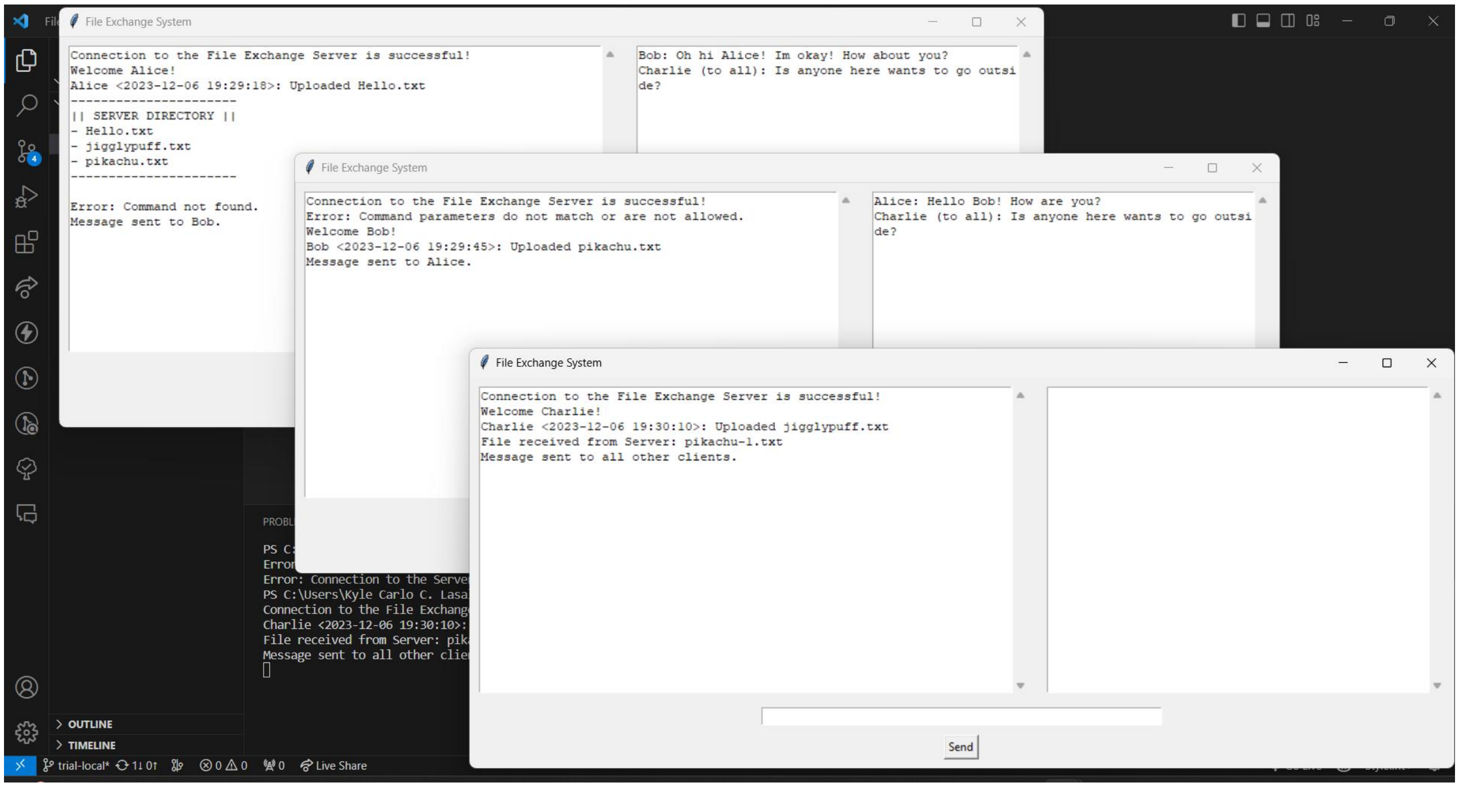
Figure 8 Requesting for the directory list from the server



```
Connection to the File Exchange Server is successful!
Welcome Charlie!
Charlie <2023-12-06 19:30:10>: Uploaded jigglypuff.txt
File received from Server: pikachu-1.txt
```

```
PS C:\Users\Kyle Carlo C. Lasala\Documents\GitHub\CSNETWK> python client/client.py
Error: Connection to the Server has failed! Please check IP Address and Port Number.
Error: Connection to the Server has failed! Please check IP Address and Port Number.
PS C:\Users\Kyle Carlo C. Lasala\Documents\GitHub\CSNETWK> python client/client.py
Connection to the File Exchange Server is successful!
Charlie <2023-12-06 19:30:10>: Uploaded jigglypuff.txt
File received from Server: pikachu-1.txt
```

Figure 9 Successfully fetching a file from the server



```
Connection to the File Exchange Server is successful!
Welcome Alice!
Alice <2023-12-06 19:29:18>: Uploaded Hello.txt
----------------------
|| SERVER DIRECTORY ||
- Hello.txt
- jigglypuff.txt
- pikachu.txt
----------------------

Error: Command not found.
Message sent to Bob.
```

```
Bob: Oh hi Alice! Im okay! How about you?
Charlie (to all): Is anyone here wants to go outsi
de?
```

```
Connection to the File Exchange Server is successful!
Error: Command parameters do not match or are not allowed.
Welcome Bob!
Bob <2023-12-06 19:29:45>: Uploaded pikachu.txt
Message sent to Alice.
```

```
Alice: Hello Bob! How are you?
Charlie (to all): Is anyone here wants to go outsi
de?
```

```
Connection to the File Exchange Server is successful!
Welcome Charlie!
Charlie <2023-12-06 19:30:10>: Uploaded jigglypuff.txt
File received from Server: pikachu-1.txt
Message sent to all other clients.
```

```
PS C:
Error
Error: Connection to the Serve
PS C:\Users\Kyle Carlo C. Lasa
Connection to the File Exchang
Charlie <2023-12-06 19:30:10>:
File received from Server: pik
Message sent to all other clie
```

Figure 10 Successful unicast and broadcast messaging features