Riley Bruce

Part 1.

(a) & (b)

Server Code

```
def handle_receive(client_socket, stop_event, client_id, session):
    while not stop_event.is_set():
        try:
            message = client_socket.recv(1024)
            if not message:
                print("\nClient disconnected.")
                stop_event.set()
                break
            message = message.decode()
            if message.startswith("SEND_FILE"):
                # Client wants to send a file
                parts = message.split()
                if len(parts) != 2:
                    print("Invalid SEND_FILE command from client.")
                    continue
                filename = parts[1]
                # Send acknowledgment to client
                client_socket.sendall("READY_TO_RECEIVE_FILE".encode())
                # Receive file size
                # Receive file size (10 bytes)
                file_size_data = b''
                while len(file_size_data) < 10:
                    chunk = client_socket.recv(10 - len(file_size_data))
                    if not chunk:
                        print("Connection closed while reading file size.")
                        stop_event.set()
                        session.app.exit()
                        return
                    file_size_data += chunk

                file_size = int(file_size_data.decode())
                # Receive file data
                file_data = b''
                while len(file_data) < file_size:
                    data = client_socket.recv(1024)
                    file_data += data
                # Display the file content
                print("\nReceived file content from client:")
                print(file_data.decode())
                # Save the file locally
                with open('received_' + filename, 'wb') as file:
```

In this section, the server handles any incoming messages from the client. Both regular text messages and the file messages are handled here. Using the SEND_FILE command followed by the name of the file, the client can request the server to append “This is an added line from the server.” to the end and send it back. The file size is

retrieved by sending a 10-byte string from the client and then the data is retrieved by looping over that length.

```
            # Save the file locally
            with open('received_' + filename, 'wb') as file:
                file.write(file_data)
            # Append a line to the file
            with open('received_' + filename, 'a') as file:
                file.write('\nThis is an added line from the server.')
            # Read the modified file
            with open('received_' + filename, 'rb') as file:
                modified_file_data = file.read()
            modified_file_size = len(modified_file_data)
            # Notify client that modified file is being sent
            client_socket.sendall("SENDING_MODIFIED_FILE".encode())
            # Send modified file size
            client_socket.sendall(str(modified_file_size).encode())
            # Send modified file data
            client_socket.sendall(modified_file_data)
            print(f"Sent modified file 'received_{filename}' back to client.")
        else:
            print(f"\nClient says: {message}")
            if message == f"Bye from Client {client_id}":
                print("Termination message received from client.")
                stop_event.set()
                session.app.exit()
                break
    except ConnectionResetError:
        print("\nConnection was reset by the client.")
        break
    except socket.timeout:
        continue
    except Exception as e:
        print(f"\nError receiving message: {e}")
        stop_event.set()
        session.app.exit()
        break
```

This section handles the writing/sending of the modified file back to the client as well as the general text message receipt and some error handling.

```
def handle_send(client_socket, stop_event, server_id, session):
    with patch_stdout():
        while not stop_event.is_set():
            try:
                response = session.prompt()
                if stop_event.is_set():
                    break
                client_socket.sendall(response.encode())
                if response == f"Bye from Server {server_id}":
                    print("Termination message sent.")
                    stop_event.set()
                    break
            except EOFError:
                break
            except Exception as e:
                print(f"\nError sending message: {e}")
                stop_event.set()
                break

    # Do not shutdown or close the socket here
```

Here the server handles any messages sent back to the clients, ensuring that input is not blocking for fluid messaging.

```python
def server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((SERVER_NAME, 12000))
    server_socket.listen(1)
    print("Server is listening on port 12000...")

    client_socket, address = server_socket.accept()
    print(f"Connected to client at {address}")

    # Receive client ID
    client_id = client_socket.recv(1024).decode()
    print(f'Client says: Hello from Client {client_id}')

    server_id = input('Enter your user ID: ')
    client_socket.sendall(server_id.encode())
    print(f"Sent server ID: {server_id}")

    client_socket.settimeout(1.0)

    session = PromptSession()
    stop_event = threading.Event()

    # Create threads for sending and receiving messages
    receive_thread = threading.Thread(target=handle_receive, args=(client_socket, stop_event, client_id, session))
    send_thread = threading.Thread(target=handle_send, args=(client_socket, stop_event, server_id, session))

    receive_thread.start()
    send_thread.start()

    # Wait for both threads to finish
    send_thread.join()
    receive_thread.join()

    # Close the client socket and server socket
    client_socket.close()
    server_socket.close()
    print("Server connection closed.")
```

Here is the main server function. Using the socket library, the server waits for an incoming client connection on port 12000. A server ID is set via input to allow for connection termination from the server side. I used the PromptSession class from the prompt_toolkit library to handle asynchronous input/output, so that messages wouldn't be interrupted or blocked by other users or the server. Finally, I used the threading library to run the sending/receiving functions in separate threads, so that users could message the server concurrently.

Client Code

```
def handle_receive(client_socket, stop_event, server_id, session):
    while not stop_event.is_set():
        try:
            response = client_socket.recv(1024)
            if not response:
                print("\nServer disconnected.")
                stop_event.set()
                break
            response = response.decode()
            if response.startswith("READY_TO_RECEIVE_FILE"):
                continue
            elif response.startswith("SENDING_MODIFIED_FILE"):
                # Server is sending back the modified file
                # Receive the file size
                modified_file_size_data = client_socket.recv(1024).decode()
                modified_file_size = int(modified_file_size_data)
                # Receive the file data
                modified_file_data = b''
                while len(modified_file_data) < modified_file_size:
                    data = client_socket.recv(1024)
                    modified_file_data += data
                # Display the modified file content
                print("\nReceived modified file content:")
                print(modified_file_data.decode())
            else:
                print(f"\nServer says: {response}")
                if response == f"Bye from Server {server_id}":
                    print("Termination message received from server.")
                    stop_event.set()
                    session.app.exit()
                    break
        except ConnectionResetError:
            print("\nConnection was reset by the server.")
            break
        except socket.timeout:
            continue
        except Exception as e:
            print(f"\nError receiving message: {e}")
            stop_event.set()
            session.app.exit()
            break
```

On the client side, the code is like the server only initiating the connection and generally requesting things of the server. The client still needs to receive messages from the server, so that's where this function comes in. This function handles printing server

responses, as well as handling notifications from the server to confirm that it is ready to receive the file and that it is sending the modified file back. When it receives the notification that the file is returning, it prints to the screen.

```
def handle_send(client_socket, stop_event, client_id, session):
    with patch_stdout():
        while not stop_event.is_set():
            try:
                message = session.prompt()
                if stop_event.is_set():
                    break
                if message.startswith("SEND_FILE"):
                    # Extract filename
                    parts = message.split()
                    if len(parts) != 2:
                        print("Invalid SEND_FILE command. Usage: SEND_FILE filename")
                        continue
                    filename = parts[1]
                    if not os.path.exists(filename):
                        print(f"File '{filename}' does not exist.")
                        continue
                    # Notify server about file transfer
                    client_socket.sendall(message.encode())
                    # Wait for server's response (handled in handle_receive)
                    # After server sends 'READY_TO_RECEIVE_FILE', proceed to send the file
                    # Open the file and send the data
                    with open(filename, 'rb') as file:
                        file_data = file.read()
                    file_size = len(file_data)
                    file_size_str = str(file_size).zfill(10)
                    # Send file size
                    client_socket.sendall(file_size_str.encode())
                    # Send file data
                    client_socket.sendall(file_data)
                    print(f"Sent file '{filename}' to server.")
                else:
                    client_socket.sendall(message.encode())
                    if message == f"Bye from Client {client_id}":
                        print("Termination message sent.")
                        stop_event.set()
                        break
            except EOFError:
                break
            except Exception as e:
                print(f"\nError sending message: {e}")
                stop_event.set()
```

This is the sending function which handles sending the text file and sending regular messages. The SEND_FILE command triggers the reading and sending of the text file. First the size is determined and padded to ensure a fixed length of 10 bytes, so that the size can be sent separate from the actual data. Not doing this caused the text data to be concatenated to the size, which threw errors.

```python
def client():
    server_name = 'localhost'
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server_name, 12000))
    print("Connected to the server.")

    client_id = input('Enter your user ID: ')
    client_socket.sendall(client_id.encode())
    print(f"Sent client ID: {client_id}")

    server_id = client_socket.recv(1024).decode()
    print(f"Hello from Server {server_id}.")

    client_socket.settimeout(1.0)

    session = PromptSession()
    stop_event = threading.Event()

    # Create threads for sending and receiving messages
    receive_thread = threading.Thread(target=handle_receive, args=(client_socket, stop_event, server_id, session))
    send_thread = threading.Thread(target=handle_send, args=(client_socket, stop_event, client_id, session))

    receive_thread.start()
    send_thread.start()

    # Wait for both threads to finish
    send_thread.join()
    receive_thread.join()

    # Close the socket
    client_socket.close()
    print("Client connection closed.")
```

Here we have the main client function. This initiates the connection using the server name (in this case localhost since I'm running it locally) and sets the ID for the client. This is then shared with the server and a timeout is set to ensure clean termination (in the case of blocking input). Like the server, separate threads are used to handle sending and receiving messages concurrently.

(c)

Updated Server Code

```
def broadcast_message(message, sender_socket, group_name, groups):
    with groups_lock:
        for client in groups[group_name]:
            if client != sender_socket:
                try:
                    client.sendall(message.encode('utf-8'))
                except Exception as e:
                    print(f"Error broadcasting to a client: {e}")
```

This function handles the sending of messages to more than one user, but only within the specified group.

```
else:
    print(f"\n[{group_name}] {client_id} says: {message}")
    # Broadcast the message to other clients in the same group
    broadcast_message(f"{client_id}: {message}", client_socket, group_name, groups)
```

This was the only section that was updated in the handle_receive function of the server. This simply routes the incoming messages to the correct group using the previous function.

```python
def handle_client(client_socket, address, groups):
    session = PromptSession()
    client_id = ''
    server_id = 'Server'

    try:
        # Receive client ID
        client_id = client_socket.recv(1024).decode('utf-8')
        print(f'Client says: Hello from Client {client_id}')
        # Send server ID
        client_socket.sendall(server_id.encode('utf-8'))
        print(f"Sent server ID to Client {client_id}.")

        # Receive group name from client
        group_name = client_socket.recv(1024).decode('utf-8')
        print(f"Client {client_id} wants to join group '{group_name}'.")

        # Add client to the group
        with groups_lock:
            if group_name not in groups:
                groups[group_name] = []
            groups[group_name].append(client_socket)

        # Set a timeout on the socket
        client_socket.settimeout(1.0)

        stop_event = threading.Event()

        # Create threads for sending and receiving messages
        receive_thread = threading.Thread(target=handle_receive, args=(client_socket, stop_event, client_id, session, group_name, groups))
        send_thread = threading.Thread(target=handle_send, args=(client_socket, stop_event, client_id, session))

        receive_thread.start()
        send_thread.start()

        # Wait for both threads to finish
        receive_thread.join()
        send_thread.join()
    except Exception as e:
        print(f"Error with client {client_id}: {e}")
```

This function handles the incoming client connections and assigns their client and group IDs. Everything else is practically the same as before, spawning separate threads for sending/receiving and setting a timeout/stop event to ensure clean termination.

```python
def server():
    SERVER_NAME = 'localhost'
    SERVER_PORT = 12000
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((SERVER_NAME, SERVER_PORT))
    server_socket.listen(5)  # Allow up to 5 pending connections
    print(f"Server is listening on port {SERVER_PORT}...")
    client_threads = []
    groups = {}
    global groups_lock
    groups_lock = threading.Lock()

    try:
        while True:
            client_socket, address = server_socket.accept()
            print(f"Connected to client at {address}")

            # Start a new thread to handle the client
            client_thread = threading.Thread(target=handle_client, args=(client_socket, address, groups))
            client_thread.start()
            client_threads.append(client_thread)
    except KeyboardInterrupt:
        print("\nServer shutting down.")
    finally:
        # Signal all threads to stop
        for thread in client_threads:
            thread.join()
        server_socket.close()
        print("Server connection closed.")
```

Finally, the main server function was updated to loop over the different incoming client connections and spawn a separate thread for each one, using the previous function as the target function.

Updated Client Code

```python
# Send group name to server
group_name = input('Enter the group name you want to join: ')
client_socket.sendall(group_name.encode('utf-8'))
print(f"Requested to join group '{group_name}'.")
```

The only additional thing here is the request for group name selection.

Part 2.

Server Code

```
def handle_client(client_socket, address):
    print(f"[+] New connection from {address}")
    while True:
        try:
            # Receive the calculation request from the client
            data = client_socket.recv(1024).decode('utf-8')
            if not data:
                # Client disconnected
                print(f"[-] Connection closed by {address}")
                break
            if data.lower() == 'exit':
                print(f"[-] Client {address} requested to close the connection.")
                break
            print(f"[{address}] Received expression: {data}")

            # Evaluate the expression safely
            result = evaluate_expression(data)
            # Send the result back to the client
            client_socket.sendall(str(result).encode('utf-8'))
        except Exception as e:
            print(f"[!] Error with client {address}: {e}")
            break
    client_socket.close()
```

This function waits for a message from the connected client and when received, evaluates the input, unless the input is ‘exit’ in which case it exits.

```python
def evaluate_expression(expression):
    try:
        # Allowed functions and constants
        allowed_names = {
            'abs': abs,
            'round': round,
            'min': min,
            'max': max,
            '__builtins__': {},
        }
        # Evaluate the expression safely
        result = eval(expression, {"__builtins__": None}, allowed_names)
        return result
    except Exception as e:
        return f"Error: {e}"

def start_server(host='localhost', port=65432):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((host, port))
    server.listen(5)
    print(f"[+] Server listening on {host}:{port}")

    try:
        while True:
            client_socket, address = server.accept()
            client_handler = threading.Thread(
                target=handle_client,
                args=(client_socket, address)
            )
            client_handler.start()
    except KeyboardInterrupt:
        print("\n[!] Server shutting down.")
    finally:
        server.close()
```

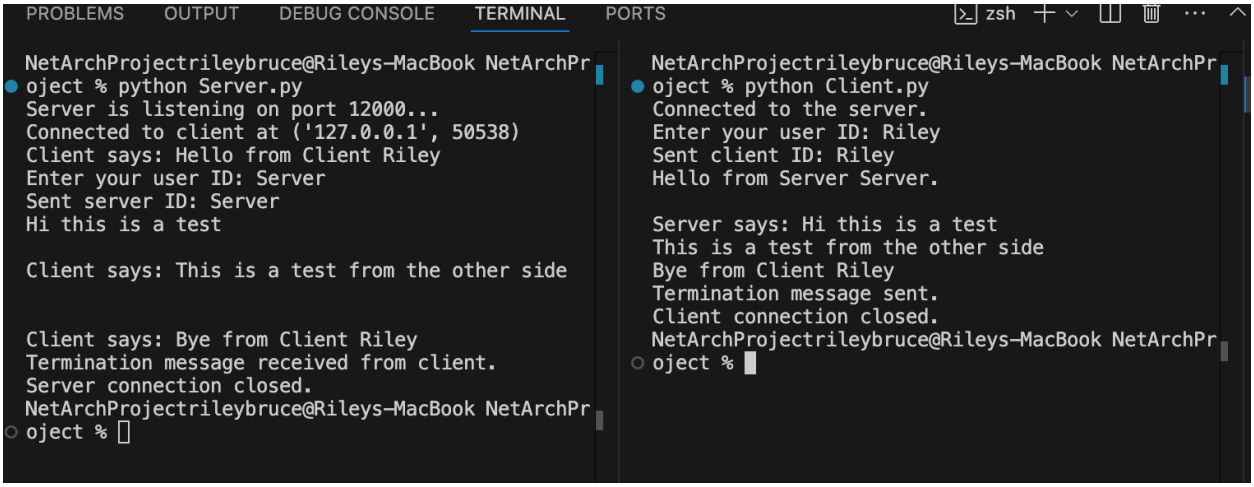
The first function here is what evaluates the mathematical expressions sent to the server. It uses the built-in math functions in python to evaluate the incoming messages. The second function is the main server function that allows for multiple users to connect (up to 5). This is done, again, by using threads and the handle_client function as the target.

Client Code

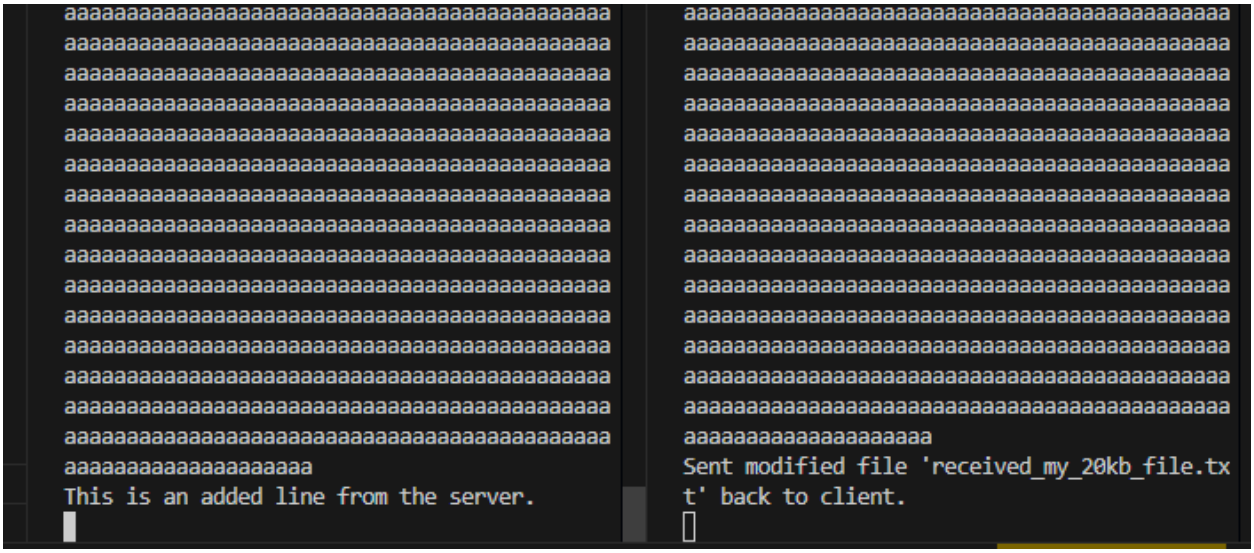
```
def start_client(host='localhost', port=65432):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client.connect((host, port))
        print(f"[+] Connected to server at {host}:{port}")
        print("Enter mathematical expressions to calculate or 'exit' to quit.")
        while True:
            # Get user input
            expression = input("Enter expression: ")
            if expression.lower() == 'exit':
                client.sendall(expression.encode('utf-8'))
                print("[-] Exiting.")
                break
            # Send the expression to the server
            client.sendall(expression.encode('utf-8'))
            # Receive the result from the server
            result = client.recv(1024).decode('utf-8')
            print(f"Result: {result}")
    except ConnectionRefusedError:
        print("[!] Server is not available.")
    except Exception as e:
        print(f"[!] An error occurred: {e}")
    finally:
        client.close()
```

Finally, we have the client code for the calculator app. It uses the server information known to start a connection and then waits for input from the user. If the user enters ‘exit’ the program exits.

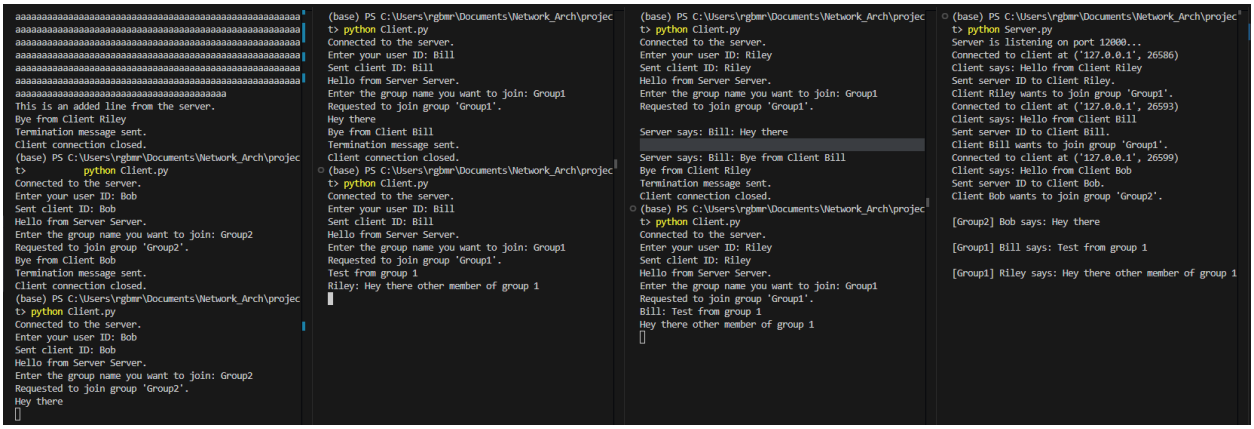
Sample output from part (a)



(b)

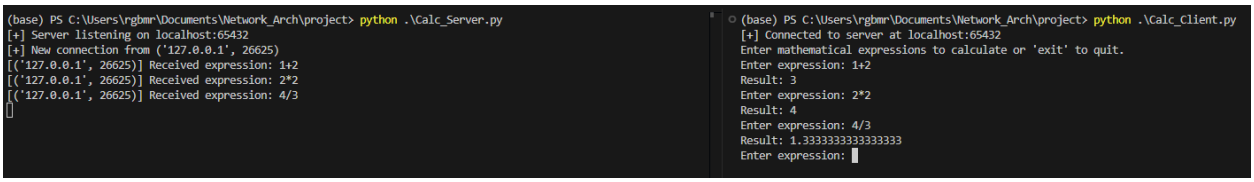


(c)



Riley Bruce

Part 2.



Link to Github with all code: https://github.com/Riley94/NetArchProject